Preproceedings of the 26nd Symposium on Implementation and Application of Functional Languages (IFL 2014)

Sam Tobin-Hochstadt *(editor)*

Contents

Preface

The 26th Symposium on Implementation and Application of Functional Languages (IFL 2014) takes place at Northeastern University in Boston, USA from October 1 to 3, 2014. It represents the return of IFL to the USA for the third time. IFL 2014 is hosted by the Programming Research Lab at Northeastern University. At the time of writing, the symposium had 47 registered participants from Denmark, the Netherlands, Norway, Germany, France, Hungary, the United Kingdom and the United States of America.

The goal of the IFL symposia is to bring together researchers actively engaged in the implemen- tation and application of functional and function-based programming languages. It is a venue for researchers to present and discuss new ideas and concepts, works in progress, and publication-ripe results.

Following the IFL tradition, there is a post-symposium review process to produce formal proceedings which will be published by the ACM in the International Conference Proceedings Series. All participants in IFL 2014 were invited to submit either a draft paper or an extended abstract describing work to be presented at the symposium. The submissions were screened by the program committee chair to make sure they are within the scope of IFL. Submissions appearing in the draft proceedings are *not* peer-reviewed publications. After the symposium, authors have the opportunity to incorporate the feedback from discussions at the symposium into their paper and may submit a revised full article for the formal review process. These revised submissions will be reviewed by the program committee using prevailing academic standards.

The IFL 2014 program consists of 31 presentations and one invited talk. The contributions in this volume are ordered according to the intended schedule of presentation. In order to make IFL 2014 as accessible as possible, we have not insisted on any particular style or length for the submissions. Such rules only apply to the version submitted for post-symposium reviewing.

As is usual for IFL, the program last three days with a social event and an invited talk. The invited talk will be given by Niko Matsakis of Mozilla Research, who will discuss the role of ownership in the type system of Rust, a progamming language designed for low-level systems programming in a memory-safe fashion. The social event takes place on October 2 and consists of two parts: a trip through the city and harbor of Boston on a duck boat, and, in the evening, a banquet dinner in downtown Copley Square.

We are grateful to many people for their help in preparating for IFL 2014. Most significantly, Asumu Takikawa of Northeastern University served as local arrangements chair, and without his efforts this event would not have been possible.

Additionally, the staff of the College of Computer and Information Science, particularly Nicole Bekerian and Doreen Hodgkin, helped make IFL a success. Our student volunteers also play an important role in the smooth running of the event.

Special thanks are due to Rinus Plasmeijer, last year's chair and the head of the IFL steering committee, for advice and experience that improved IFL 2014.

Reactive Web Applications with Dynamic Dataflow in F#

Anton Tayanovskyy Simon Fowler Loïc Denuzière Adam Granicz
IntelliFactory, http://www.intellifactory.com
{anton.tayanovskyy, simon.fowler, loic.denuziere, granicz.adam}@intellifactory.com

Abstract

Modern web applications depend heavily on data which may change over the course of the application's execution: this may be in response to input from a user, information received from a server, or DOM events, for example.

Much recent work has been carried out with the hope of improving upon the current callback-driven model: in particular, approaches such as functional reactive programming and data binding have proven to be promising models for the creation of reactive web-based user interfaces.

In this paper, we present a framework, `UI.Next`, for the creation of reactive web applications in the functional-first language F#, using the WebSharper web framework. We provide an elegant abstraction to integrate a dataflow layer built on the notion of a dynamic dataflow graph—a dataflow graph which may vary with time—with a DOM frontend, allowing updates to be automatically propagated when data changes. Additionally, we provide an interface for the specification of declarative animations, and show how the framework can ease the implementation of existing functional web abstractions such as Flowlets [3] and Piglets [11].

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Data-flow languages; Applicative (functional) languages

Keywords Dataflow; Web Programming; Functional Programming; Graphical User Interfaces; F#

1. Introduction and Background

Modern web applications depend hugely on data which changes during the course of the application's execution. A common approach to handling this in JavaScript is through the use of callbacks, meaning that whenever a piece of data changes, a callback is executed which performs the appropriate updates.

While this suffices for smaller applications, callbacks become unwieldy as an application grows: inversion of control reduces the ability to reason about applications, often resulting in concurrency issues such as race conditions. Applications also become increasingly difficult to structure, as view modification code becomes intertwined with application logic, decreasing modularity.

The *reactive programming* paradigm provides a promising solution to this problem: instead of relying on state which is mutated by callbacks, reactive approaches work by defining interdependent variables, and rely on a *dataflow graph* to propagate changes to dependent elements. Research into *functional* reactive programming [12] (FRP) builds upon this by using concepts from functional programming to model time-varying data.

FRP systems are based around the notion of a *Signal* or *Behaviour*—a value which varies with time—and an *Event*, which can be thought of as either a *discrete* occurrence such as a mouse click, or a *predicate* event which occurs exactly when a signal satisfies a set of predicates.

Functional Reactive Programming provides a very expressive and clear semantics, with powerful higher-order combinators to work with continuously-varying values. Although the theory is clear, implementing such systems poses several challenges such as space leaks, in particular with regard to higher-order stream operations: the canonical example of this is that by allowing a signal of signals, it therefore becomes necessary to record the *entire history* of the signal after its creation in order to allow future signals to depend on previous values. This naturally results in a memory leak, as memory usage must grow linearly with execution time.

To overcome this issue, different approaches have been taken by different systems. In particular, *Arrowised FRP* [16] disallows signals from being treated as first-class altogether, relying instead on a set of primitive signals and a set of stream combinators to manipulate these, while the *Elm* language [10] prohibits the creation of higher-order signals directly in the type system. Real-time FRP [32] and event-driven FRP [33] systems further restrict operations on streams to those which can be implemented in a manner that will yield acceptable real-time behaviour.

The use of higher-order signals is, however, natural when creating graphical applications. By ruling out higher-order signals, the underlying dataflow graph remains *static*, meaning that it cannot change during the course of the application's execution. In practice, this means that it is difficult to create applications with multiple sub-pages, such as in the single-page application (SPA) paradigm.

Our alternative solution consists of a dataflow layer which interacts with a DOM frontend, using F# and the WebSharper [1] functional web framework. We introduce reactive variables, or `Var`s, and `View`s, read-only projection of `Var`s within the dataflow graph. A key difference between this and the traditional FRP paradigm is that the system we propose only makes use of the *latest* available value in the system, but as a result supports *monadic* combinators to support *dynamic* composition of `View`s.

We designed our framework, `UI.Next`, taking into account the following key principles:

Modularity: The dataflow graph should be defined separately to the DOM representation, meaning that it should be possible to display the same data in different ways on the view layer in a similar way to the Model-View-Controller architecture. It should

[1] `http://www.websharper.com`

also be possible to perform different transformations to the same node in the dataflow graph.

Leak-Freedom: A primary design decision was to prevent space leaks. Traditional pure monadic FRP systems permit the inclusion of space leaks by allowing higher-order event stream combinators, whereas other dataflow systems often keep strong links between nodes of the dataflow graph and as a result require manual unsubscription from data sources. Our solution does not keep strong links between nodes in the dataflow graph, and as a result prevents this class of leaks. space leaks are avoided by working purely with the latest value of reactive variables.

Preservation of DOM Node Identity: DOM nodes consist of more than is described in the DOM tree. State such as whether an element is currently in focus, or the current text in an input box, is preserved upon a DOM update, and only subtrees which have been explicitly marked as time-varying will change on an update.

Composability and Ease-of-Integration: Elements in the DOM representation compose easily due to a monoidal interface, and we introduce an elegant embedding abstraction to allow time-varying DOM fragments to be integrated with the remainder of the DOM tree representation.

1.1 Contributions

- An implementation of a dynamic dataflow system which is amenable to garbage collection by not retaining strong links between nodes in the dataflow graph through the use of an approach inspired by Concurrent ML [28] (Section 2).
- A reactive DOM frontend for the WebSharper web framework, allowing DOM nodes to depend on dataflow nodes and update automatically (Section 3).
- A declarative animation API which integrates with the DOM frontend, and can be driven by the dataflow system (Section 4).
- Implementations of functional abstractions such as Flowlets [3] and Piglets [11] using `UI.Next`, showing how the framework can ease the implementation of such abstractions, and how these abstractions can be used to build larger applications (Section 5).
- Example applications making use of the framework (Section 6).

Source code for the framework can be found at `http://www.bitbucket.org/IntelliFactory/websharper.ui.next`, and a website containing samples and their associated source code can be found at `http://intellifactory.github.io/websharper.ui.next`.

2. Dataflow Layer

The dataflow layer exists to model data dependencies and consequently to perform change propagation. The layer is specified completely separately from the reactive DOM layer, and as such may be treated as a render-agnostic data model.

The dataflow layer consists primarily of two primitives: reactive variables, `Var`s, and reactive views, `View`s.

A `Var` is a time-varying variable, and can be thought of as very similar to a standard F# `ref` cell. The difference, however, is that a `Var` may be observed by `View`s: changes to a `Var` therefore update any dependent `View`s in order to trigger change propagation through the remainder of the dataflow graph.

2.1 `Var`s

`Var`s are parameterised over a particular type. The actions that may be performed on `Var`s are straightforward: they may be created, their value may be set, and they may be updated using the current value.

One additional operation, `SetFinal`, marks the value as finalised, meaning that no more writes to that variable are permitted. This is included in order to prevent a class of memory leaks: if it is known that a value does not change after a certain point, then `SetFinal` may be used to optimise accesses to the variable.

The operations that may be performed on `Var`s are detailed in Listing 1.

Listing 1. Basic operations on Vars

```
type Var =
  static member Create : 'T -> Var<'T>
  static member Get : Var<'T> -> 'T
  static member Set : Var<'T> -> 'T -> unit
  static member SetFinal : Var<'T> -> 'T -> unit
  static member Update : Var<'T> -> ('T -> 'T) ->
      unit
```

2.2 `Views`

A `View` provides a way of observing a `Var` as it changes. More specifically, a `View` can be thought of as a node in the dataflow graph which is dependent on a data source (`Var`), or one or more other dataflow nodes (`View`).

The power of `View`s comes as a result of the implementation of applicative and monadic combinators, allowing multiple views to be combined: these operations are shown in Listing 2.

Listing 2. Operations on Views

```
type View =
  static member Const : 'T -> View<'T>
  static member FromVar : Var<'T> -> View<'T>
  static member Sink : ('T -> unit) -> View<'T> ->
      unit
  static member Map : ('A -> 'B) -> View<'A> -> View
      <'B>
  static member MapAsync : ('A -> Async<'B>) -> View
      <'A> -> View<'B>
  static member Map2 : ('A -> 'B -> 'C) -> View<'A>
      -> View<'B> -> View<'C>
  static member Apply : View<'A -> 'B> -> View<'A>
      -> View<'B>
  static member Join : View<View<'T>> -> View<'T>
  static member Bind : ('A -> View<'B>) -> View<'A>
      -> View<'B>
```

A `View` can be created from a `Var` using the `FromVar` function. Additionally, it is possible to create a `View` of a constant value using the `Const` function.

The `Sink` function acts as an *imperative observer* of the `View` – that is, the possibly side-effecting callback function of type `('T -> unit)` is executed whenever the value being observed changes. This function is crucial in the implementation of the reactive DOM layer described in Section 3.

The remaining abstractions are ubiquitous in the functional domain: `Map` allows a function to be applied to the new value of an observed `Var` whenever it changes, yielding another view. In terms of the dataflow graph, this results in an additional node which depends on the original view. `MapAsync` is a helper function which facilitates asynchronous calls as supported within F# and WebSharper.

The applicative combinators `Map2` and `Apply`, as first exposited by McBride and Paterson [21], allow for *static composition* of `View`s. Using these combinators, it is possible to apply functions of arbitrary arity to nodes within the dataflow graph.

Finally, the monadic combinators `Join` and `Bind` allow *dynamic composition* of graphs – that is, a dataflow graph may consist of nodes which are themselves time-varying dataflow graphs, allowing the graph to change during the course of execution. Although this dynamism is natural in GUI programming, most implementations of FRP systems do not support this for efficiency reasons as outlined in Section 7.2.

2.3 Models and Collections

When working with collections which may change with time, it is often better to work with higher-level *models* than simple `Var`s and `View`s. A `Model<'I, 'M>` represents a mutable model, providing a projection between a mutable type `'M` (such as an F# `ResizeArray`) and an immutable type `'I` (such as an F# `list`). This proves very useful when rendering a collection, for example.

A specialisation of the `Model` type is the `ListModel`, which internally represents a time-varying collection as a `ResizeArray` but allows the model to be viewed as a `list`. Additionally, several operations to modify the collection are provided: these are shown in Listing 3.

Listing 3. Operations on a ListModel

```
type ListModel<'Key,'T> with
  member Add : 'T -> unit
  member Remove : 'T -> unit

type ListModel with
  static member Create<'Key,'T when 'Key : equality>
       : ('T -> 'Key) -> seq<'T> -> ListModel<'Key,'
      T>
  static member FromSeq<'T when 'T : equality> : seq
      <'T> -> ListModel<'T,'T>
  static member View : ListModel<'Key,'T> -> View<
      seq<'T>>
```

A `ListModel` is created using a function which derives a key to be used for equality testing, and a sequence of elements. Addition and removal of elements can be performed with the `Add` and `Remove` functions, but more importantly it is possible to obtain a `View` of the collection using the `ListModel.View` function.

2.4 Implementation

2.4.1 `Vars`

The implementation of a `Var` is shown in Listing 4. A value of type `Var<'T>`, where `'T` is a polymorphic type variable, consists of mutable value field, a flag to specify whether or not the `Var` has been set as final, and a method by which any dependent views may be notified that the variable has been updated. This is implemented as a `Snap`, discussed in Section 2.4.2

Listing 4. Implementation of a Var

```
type Var<'T> =
  {
    mutable Const : bool
    mutable Current : 'T
    mutable Snap : Snap<'T>
  }
```

2.4.2 `Snaps`

The implementation of the dataflow layer depends largely on the notion of a `Snap`: an observable snapshot of a value.

The IVar Abstraction

At its core, a `Snap` is based on the notion of an *immutable variable*, or IVar [28]. An IVar is created as an empty cell, which can be written to only once: multiple writes to an IVar are not permitted. Attempting to read from a 'full' IVar will immediately yield the value contained in the cell, whereas attempting to read from an 'empty' IVar will result in the thread blocking until such a variable becomes available. This is shown in Figure 1.

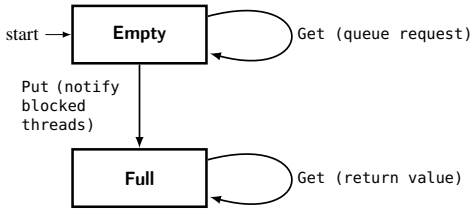


Figure 1. State Transition Diagram for an IVar

The IVar abstraction heavily inspires the method by which change propagation in the graph is handled. In this sense, there are no explicit links within the dataflow graph: that is, edges in the dataflow graph are not represented using concrete links – instead, dependent nodes can be thought of as attempting to retrieve a value from an IVar indicating obsoleteness. If the value is not obsolete, indicated in the IVar model as trying to retrieve a value from an empty cell, then the requests are queued[2]. As soon as the value in the dataflow node has been updated, meaning that it should be propagated through the graph, then all threads are notified with the latest value and continue execution.

`Snap` Implementation

While the IVar abstraction encapsulates the essence of a `Snap`, in reality the implementation is slightly more complex. A `Snap` can be thought of as a state machine consisting of four separate states:

Ready: A `Snap` containing an up-to-date value, and a list of threads to notify when the value becomes obsolete.

Waiting: A `Snap` without a current value. Contains a list of threads to notify when the value becomes available, and a list of threads to notify should the `Snap` become obsolete prior to receiving a value.

Forever: A snap in the `Forever` state indicates that it contains a value that will never change. This is an optimisation as it prevents nodes waiting for the `Snap` to become obsolete when this will never be the case.

Obsolete: A snap in the `Obsolete` state indicates that the snap contains obsolete information.

The state transition diagram for a `Snap` is shown in Figure 2.

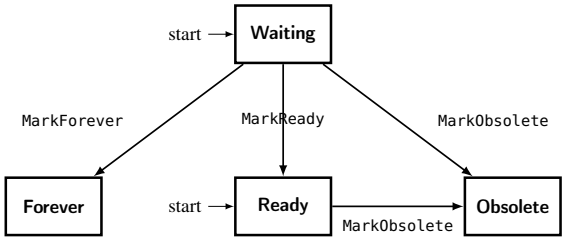


Figure 2. State Transition Diagram for a `Snap`

`Snaps` can be modified by four operations. These are:

[2] Native JavaScript is single-threaded, but we make use of the F# asynchronous workflow capabilities on the client by using a custom scheduler.

MarkForever: Updates the `Snap` with a value, transitioning to the `Forever` state to indicate that the value will never change.

MarkObsolete: Marks the `Snap` as obsolete, notifying all threads that are waiting for an updated value.

MarkReady: Marks the `Snap` as containing a new, up-to-date value, notifying all threads that are waiting for the initial value.

MarkDone: Marks the `Snap` as containing a value. If the `Snap` has been marked as constant, then transitions to the `Forever` state, otherwise transitions to the `Ready` state.

Additionally, `Snap`s support a variety of applicative and monadic combinators in order to implement the operations provided by `Views`: to implement `Map2` for example, a `Snap` must be created which is marked as obsolete as soon as either of the two dependent `Snap`s becomes obsolete.

In order to react to lifecycle events and trigger change propagation through the dataflow graph, the `When` eliminator function is used.

```
val When : Snap<'T> -> ready: ('T -> unit) ->
     obsolete: (unit -> unit) -> unit
```

The `When` function takes a snap and two callbacks: `ready`, which is invoked when a value becomes available, and `obsolete`, which is invoked when the `Snap` becomes obsolete.

2.5 Change Propagation

As discussed in Section 2.4.1, a `Var` consists of a current value, and a `Snap` which is used to drive change propagation. When the value of a `Var` is updated, the current `Snap` is marked as obsolete and replaced by a new `Snap` in the `Ready` state.

At its core, a `View` consists of a function `observe` to return a `Snap` of the current value.

```
type View<'T> =
    | V of (unit -> Snap<'T>)
```

The simplest `View` directly observes a single `Var`: this simply accesses the current `Snap` associated with that `Var`, updating whenever the `Snap` becomes obsolete.

Listing 5 shows the pattern of creating views which depend on other views. The `CreateLazy` function takes as its argument an observation function of type `(unit -> Snap<'A>)`, which is a function returning a `Snap` representing the latest value of the dependent dataflow nodes. This is created lazily for efficiency.

Listing 5. View Implementation

```
static member CreateLazy observe =
  let cur = ref None
  let obs () =
    match !cur with
    | Some sn when not (Snap.IsObsolete sn) -> sn
    | _ ->
      let sn = observe ()
      cur := Some sn
      sn
  V obs

static member Map fn (V observe) =
  View.CreateLazy (fun () ->
    observe () |> Snap.Map fn)

static member Map2 fn (V o1) (V o2) =
  View.CreateLazy (fun () ->
    let s1 = o1 ()
    let s2 = o2 ()
    Snap.Map2 fn s1 s2)
```

The implementations of `Snap.Map` and `Snap.Map2` are shown in Listing 6. We omit some optimisations for brevity.

Listing 6. Snap Combinator Implementation

```
let Map fn sn =
  let res = Create ()
  When sn (fn >> MarkDone res sn) (fun () ->
      MarkObsolete res)
  res

let Map2 fn sn1 sn2 =
  let res = Create ()
  let v1 = ref None; let v2 = ref None
  let obs () =
      v1 := None; v2 := None
      MarkObsolete res
  let cont () =
      match !v1, !v2 with
      | Some x, Some y ->
        MarkReady res (fn x y)
      | _ -> ()
  When sn1 (fun x -> v1 := Some x; cont ()) obs
  When sn2 (fun y -> v2 := Some y; cont ()) obs
  res
```

The `Snap.Map2` function takes a dependent `Snap sn` and a function `fn` to apply to the value of `sn` when it becomes available. Firstly, an empty `Snap` is created. This is passed to the `When` eliminator along with two callbacks: the first, called when `sn` is ready, marks `res` as ready, containing the result of `fn` applied to the value of `sn`. The second, called when `sn` is obsolete, marks `res` as obsolete.

The `Snap.Map2` function applies a function to multiple arguments, which can in turn be used to implement applicative combinators. In order to do this, a `Snap res` and two mutable reference cells, `v1` and `v2`, are used. When either of the dependent `Snaps` `sn1` or `sn2` update, the corresponding reference cell is updated and the continuation function `cont` is called. If both of the reference cells contain values, then the continuation function marks `res` ready, containing the result of `fn` applied to `sn1` and `sn2`. If either of the dependent `Snaps` become obsolete, then `res` is marked as obsolete. This avoids glitches, which are intermediate states present during the course of change propagation, and avoids such intermediate states being observed by the reactive DOM layer.

3. Reactive DOM Layer

The reactive DOM layer allows data models described using the dataflow backend to be used to create reactive web applications which update automatically as a result of change propagation within the dataflow graph. In addition to providing a set of reactive input controls which depend on and modify `Vars`, the DOM layer provides combinators allowing dynamic DOM fragments to be directly composed with static fragments.

The simplest example of this is a text label which mirrors the contents of an input text box. This is shown in Listing 7.

Listing 7. A label mirroring the contents of an input box

```
let rvText = Var.Create ""
let inputField = Doc.Input [] rvText
let label = Doc.TextView rvText.View
Div0 [
    inputField
    label
]
```

We begin by declaring a variable `rvText` of type `Var<string>`, which is a reactive variable to hold the contents of the input

box. Secondly, we create an input box which is associated with `rvText`, meaning that whenever the contents of the input field changes, `rvText` will be updated accordingly. Next, we create a label using `Doc.TextView`, which we associate with a view of `rvText`. Finally, we can place these components inside a `<div>` tag using the `Div0` function.

3.1 Monoidal Interface

A key design decision that was made in implementing the reactive DOM layer was the decision to use a *monoidal interface* for both DOM elements and DOM attributes. As the API is purely generative, meaning that it does not permit the deconstruction of nodes, we believe the use of a monoidal interface is an appropriate choice for DOM combinators as it does not differentiate between the absence of a node, a single node, or a list of nodes. Previous iterations of DOM node combinators within WebSharper did not use such an interface, and therefore often required explicit `yield` expressions within node lists.

All DOM elements in the reactive DOM layer are of type `Doc`, which represents either an empty DOM node, a single DOM node, or multiple DOM nodes. To form a monoid, `Doc` supports the operations shown in Listing 8. The same operations are supported by reactive attributes, of type `Attr`.

Listing 8. Monoidal operations on Doc

```
static member Empty : Doc
static member Append : Doc -> Doc -> Doc
static member Concat : seq<Doc> -> Doc
```

Here, `Empty` is the neutral identity element, which represents an empty DOM tree. `Append` is an associative binary operation, which combines two DOM subtrees: more precisely, the two DOM subtrees become sibling nodes, and the second subtree is rendered after the first.

In accordance with the monoid laws, appending an element to the empty `Doc`, and appending the empty `Doc` to the element does not change the element. Concatenation is implemented as a fold over a sequence of `Doc`s, using `Doc.Empty` as the initial element.

3.2 Reactive Elements and Attributes

Reactive elements are created using the `Doc.Element` function, which takes as its arguments a tag name, a sequence of attributes, and a sequence of child elements. As discussed in section 3.1, these sequences are concatenated.

```
static member Element : name: string -> seq<Attr> ->
      seq<Doc> -> Doc
```

Reactive attributes can be static, dynamic, or animated. Static attributes correspond to simple key-value pairs, as found in traditional DOM applications, whereas dynamic attributes are instead backed by a `View<string>`. We defer discussion of animation attributes to Section 4.

```
static member Create :
  name: string -> value: string -> Attr
static member Dynamic :
  name: string -> value: View<string> -> Attr
static member Animated :
  name: string ->
  Trans<'T> ->
  view: View<'T> ->
  value: ('T -> string) -> Attr
```

3.3 Embedding Reactive Views

Arguably the most important function within the Reactive DOM layer is the `Doc.EmbedView` function:

```
static member EmbedView : View<Doc> -> Doc
```

Semantically, this allows us to embed a *time-varying* DOM fragment into a larger DOM tree. This is the key to creating reactive DOM applications using the dataflow layer: by using `View.Map` to map a rendering function onto a variable, for example, we can create a value of type `View<Doc>` to be embedded using `EmbedView`.

By way of example, consider rendering an item in a to-do list, where the item should be rendered with a strikethrough if the task has been completed. We begin by defining a simple type, with a reactive variable of type `Var<bool>` which is set to true if the task has been completed.

```
type TodoItem =
    { Done : Var<bool>
      TodoText : string }
```

It would then be possible to render such an item as shown in Listing 9. Note that here, `Del0` is a notational shorthand for an HTML `<del>` element without any attributes, and `Doc.TextNode` creates a DOM text node.

We also make use of the F# construct `|>`, pronounced 'pipe', which signifies reverse function application.

```
let (|>) x f = f x
```

Listing 9. Embedding Reactive Views

```
View.FromVar todo.Done
|> View.Map (fun isDone ->
    if isDone
        then Del0 [ Doc.TextNode todo.TodoText ]
        else Doc.TextNode todo.TodoText)
|> Doc.EmbedView
```

We use this pattern extensively when developing applications using `UI.Next`.

3.4 Implementation

`Doc`

We store an in-memory representation of DOM trees, propagating these to the DOM when necessary. At the outermost layer, a `Doc` consists of information about its associated subtree, and a unit view which is used to propagate updates upwards through a tree. This is shown in Listing 10.

Listing 10. Implementation of the Doc type

```
type Doc =
    { DocNode : DocNode; Updates : View<unit> }
```

A `DocNode` is a node in the in-memory skeleton DOM representation. Defined as an algebraic data type, a `DocNode` may represent the concatenation of two `Doc`s as a result of an `Append` operation, a `DOM` element, an embedding of a reactive DOM subtree, a DOM text node, or an empty node.

When creating nodes, the `Updates` view is combined with any dependent sub-views using the monadic and applicative combinators discussed in Section 2.2. For example, the `Updates` view of an `AppendDoc` is dependent on the `Updates` views of both sub-nodes, and as such is constructed using the `Doc.Map2` combinator, as shown in Listing 11. Note that the `Docs.Mk` is simply a constructor function for a `Doc` record, taking a `DocNode` and an `Updates` view as its arguments. The `||>` operator is similar to `|>`, but instead takes a tuple of arguments to pass to a function.

```
let (||>) (a, b) f = f a b
```

Listing 11. Construction of an `AppendDoc` `DocNode`

```
static member Append a b =
    (a.Updates, b.Updates)
    ||> View.Map2 (fun () () -> ())
    |> Docs.Mk (AppendDoc (a.DocNode, b.DocNode))
```

`EmbedView`

As discussed in Section 3.3, the `EmbedView` allows a time-varying DOM segment to be embedded within the DOM tree, with any updates in this segment being reflected within the DOM. The implementation of this is based on the idea of 'dirty-checking', as employed by many reactive DOM libraries such as Facebook React [1].

The `DocNode` representation of a time-varying DOM node is a `DocEmbedNode`, shown in Listing 12.

Listing 12. The `DocEmbedNode` type

```
type DocEmbedNode =
    { mutable Current : DocNode
      mutable Dirty : bool }
```

The record has two mutable fields: the `Current` field represents the current value of the embedded view, and the `Dirty` field is set to true if the `View<Doc>` has changed, indicating that the DOM subtree should be updated.

The implementation of the `EmbedView` function is shown in Listing 13.

Listing 13. Implementation of the `EmbedView` function

```
static member EmbedView view =
    let node = Docs.CreateEmbedNode ()
    view
    |> View.Bind (fun doc ->
        Docs.UpdateEmbedNode node doc.DocNode
        doc.Updates)
    |> View.Map ignore
    |> Docs.Mk (EmbedDoc node)
```

The `EmbedView` function works by creating a new entry in the dataflow graph, depending on the time-varying DOM segment. Conceptually, this can be thought of as a `View<View<Doc>>`, which would not be permissible in many FRP systems. Here, the monadic `Bind` operation provided by the dynamic dataflow layer is crucial in allowing us to observe not only changes *within* the `Doc` subtree (using `doc.Updates`), but changes *to* the `Doc` itself: when either change occurs, the `DocEmbedNode` is marked as dirty, and the update is propagated upwards through the tree.

Synchronisation

As previously discussed, any updates in the DOM representation are propagated *upwards* through the tree representation. In order to trigger a DOM update, we use the `Sink` imperative observer function discussed in Section 2.2, which triggers a function whenever a `View` (in this case the `Updates` view of the root node in the `Doc` tree) changes.

Synchronisation between the virtual DOM skeleton and the physical DOM representation is performed using an $\mathcal{O}(n)$ traversal of the virtual DOM tree, in a similar way to existing libraries. While at first this may seem prohibitive for a responsive web application, such an approach has been proven by libraries such as *React* to yield acceptable performance since such traversals are generally not computationally expensive, and there tend to be few physical DOM changes.

For an element node, the synchronisation algorithm recursively checks whether any child nodes have been marked as dirty. In the case of `EmbedNode`s, it is not only necessary to check whether the `EmbedNode` itself is dirty but also whether the current subtree value represented by the `EmbedNode` is dirty: this ensures that both global (entire subtree changes) and local (changes within the subtree) changes have been taken into account.

An important consideration when implementing the synchronisation algorithm was the preservation of node *identity* – that is, the internal state associated with an element such as the current input in a text box, and whether the element is in focus. For this reason, when updating the children of a node, simply removing and reinserting all children of an element marked dirty is not a viable solution: instead we associate a *key* with each item, which is used for equality checking, and perform a set difference operation to calculate the nodes to be removed.

4. Declarative Animation

Animation is increasingly used in modern web applications, especially when visualising data, when processing user input, or advancing state within a control flow.

In the context of web applications, animations are typically implemented as an interpolation between attribute values over time. Such animations must be composable in order for different animations to run either sequentially or concurrently, must support the specification of interpolation strategies for a given type, and *easing* functions, which specify how quickly the animation progresses at different points during the animation.

Native CSS provides animation functionality which can interpolate values, apply easing functions, and apply animations both sequentially and concurrently through the use of keyframes. While this is sufficient and intuitive for simple applications, the approach founders when animations depend explicitly on dynamic data and cannot be determined statically.

The D3 library [4] provides more powerful animation functionality. In particular, the library enables animations to depend directly on data sets, for animations to be delayed, and for the specification of *transitions*—animations which are triggered when a node is added, changed, or removed from the DOM.

D3 is an extremely powerful library, and its use has led to some very impressive animated visualisations. The API, however, does not lend itself particularly well to a functional, statically typed language: in particular, animations are generally constructed using *selections*, using function chaining to add animations and transitions. This results in animations being declared in a more imperative style.

The declarative animation library in `UI.Next` allows animations (an interpolation of a value over time) and transitions to be specified separately. These are then integrated with the reactive DOM layer in one of two ways: more commonly, they can be attached directly to elements as attributes and therefore react directly to changes within the dataflow graph, but they may also be scheduled imperatively.

An animation is defined using the `Anim<'T>` type, where the `'T` type parameter defines the type of value to be interpolated during the animation. As shown in Listing 14, an `Anim<'T>` type is internally represented as a function `Compute`, mapping a normalised time (a value between 0 and 1 denoting progress through the animation) to a value, and the duration of the animation.

Listing 14. Implementation of the `Anim<'T>` type

```
type Anim<'T> =
  { Compute : Time -> 'T; Duration : Time }
```

An animation can be constructed using the `Anim.Simple` or `Anim.Delayed` functions: `Anim.Simple` can be seen as a delayed

animation with a delay of 0. This takes as its arguments an interpolation strategy, an easing function, the duration of the animation, the delay of the animation in milliseconds, and the start and end values.

```
static member Anim.Simple :
  Interpolation<'T> ->
  Easing ->
  duration: Time ->
  delay: Time ->
  startValue: 'T ->
  endValue: 'T ->
  Anim<'T>
```

To describe collections of animations, we once again make use of a monoidal interface: in this case, the semantics of monoid concatenation are that the animations play concurrently as opposed to sequentially. Collections of animations are represented by the `Anim` type and supports the monoidal `Empty`, `Append` and `Concat` operations, as well as a function `Pack` to lift an `Anim<unit>` type into a singleton animation collection.

```
static member Append : Anim -> Anim -> Anim
static member Concat : seq<Anim> -> Anim
static member Empty : Anim
static member Pack : Anim<unit> -> Anim
```

Concatenation of a list of animations involves creating a new animation with the length of the longest constituent animation, and a compute function which 'prolongs' shorter animations by not performing further interpolation after the end of the original animation.

Transitions are specified using the `Trans` type. Functions for creating and modify `Trans` types are shown in Listing 4.

```
static member Create : ('T -> 'T -> Anim<'T>) ->
    Trans<'T>
static member Trivial : unit -> Trans<'T>
static member Change : ('T -> 'T -> Anim<'T>) ->
    Trans<'T> -> Trans<'T>
static member Enter : ('T -> Anim<'T>) -> Trans<'T>
    -> Trans<'T>
static member Exit : ('T -> Anim<'T>) -> Trans<'T>
    -> Trans<'T>
```

A transition can either be created with the `Trivial` function, meaning that no animation occurs on changes, or with an animation. Enter and exit transitions, which occur when a node is added or removed from the DOM tree respectively, can be specified using the `Enter` and `Exit` functions. Upon a DOM update, a set intersection is performed between the nodes that have enter and exit transitions and the nodes which have been added and removed respectively, and these are concatenated and played as an animation collection.

An animation is embedded within the reactive DOM layer as an attribute through the `Attr.Animated` function:

```
static member Animated : name: string -> Trans<'
    T> -> view: View<'T> -> value: ('T -> string
    ) -> Attr
```

This function takes the name ofthe attribute to animate, a transition, a view of a value upon which the animation depends (for example, an item's rank in an ordered list), and a projection function from that value to a string, in such a way that it may be embedded into the DOM.

5. Functional Web Abstractions

Functional programming and static type systems can ease web programming by facilitating the implementation of functional web abstractions. In particular, Formlets [8] provide a structured means, based on the notion of an applicative functor [21], of retrieving input from a user. Using Formlets, it is possible to define a statically-typed *model* of the input (for example as a record), and populate this model with input gained from form controls.

Previous work has built upon Formlets in various ways. By extending Formlets with a monadic interface in addition to an applicative one, *sequences* of Formlets called Flowlets [3] can be created, where each stage in the flow can depend on previously-submitted input. Additionally, Formlets by default do not allow any flexibility in how forms are rendered: this is addressed by a Pluggable GUI-let, or Piglet [11].

`UI.Next` greatly simplifies the implementation of Piglets and Flowlet-style combinators. In this section, we discuss the implementation of these abstractions, and how their implementation has been eased using the framework. Additionally, we discuss a method by which pages and application state may be synchronised with the current URL, to allow easier sharing of locations within single-page applications.

5.1 Flowlets

The Flowlet-style combinators we have implemented are shown in Listing 15. It is important to note that this is not a direct implementation of Flowlets: in particular, we do not build forms using static, applicative composition, instead allowing each stage of the flow to handle the retrieval and processing of user input. The primary objective of these combinators, however, is to allow applications with a linear control flow to be constructed in a simple, intuitive fashion.

Listing 15. Flowlet Combinators

```
type Flow =
  // Definition
  static member Define : (('A -> unit) -> Doc) ->
      Flow<'A>
  static member Static : Doc -> Flow<unit>
  // Mapping
  static member Map : ('A -> 'B) -> Flow<'A> -> Flow
      <'B>
  // Monadic Combinators
  static member Bind : Flow<'A> -> ('A -> Flow<'B>)
      -> Flow<'B>
  static member Return : 'A -> Flow<'A>
  // Rendering function
  static member Embed : Flow<'A> -> Doc
  // Helper function
  static member Do : FlowBuilder
```

Pages in the flow are defined using the `Define` function. To define a page, we require function takes a callback of type `('A -> unit)`, used to pass the resulting value to subsequent stages of the flow, and renders the page as a `Doc`. It is also possible to define a static page which does not progress the flow by using the `Static` function.

Internally, a `Flow<'T>` is represented as a singleton record containing one member, a function `Render` which takes as its arguments a `Var` to be used for rendering the flow and a continuation function `('T -> unit)`, resulting in a unit value.

Listing 16. Implementation of the `Flow` type

```
type Flow<'T> =
  { Render : Var<Doc> -> ('T -> unit) -> unit }
```

To combine multiple stages of a flow, the `Bind` function is used. This is shown in Listing 17.

Listing 17. Implementation of the `Flow.Bind` function

```
static member Bind m k  =
    { Render = fun var cont -> m.Render var (fun r
        -> (k r).Render var cont) }
```

The `Bind` function is implemented by creating a new `Flow` record from a flow `m` of type `Flow<'A>` and a continuation function `('A -> Flow<'B>)`. The newly-created flow renders `m` to the `Var var`, with the value `r` returned by that particular stage of the flow applied to the continuation function `k` to construct the next stage in the flow.

The eliminator function for `Flow` types, `Embed`, is defined in Listing 18.

Listing 18. Implementation of `Flow.Embed`

```
static member Embed fl =
    let v = Var.Create Doc.Empty
    fl.Render v ignore
    Doc.EmbedView (View.FromVar v)
```

We begin by creating a `Var v` used to contain the current page rendering, initially consisting of the empty document. The flow is 'executed' by invoking the `Render` function with the variable, which is updated by the bind operation, and finally by embedding the resulting `View`.

Through the use of *computation expressions* [24], it is possible to specify flows in a manner analogous to do-notation in Haskell. Consider the following example flow:

1. A user is asked for a name and address, which is used to create a `Person` record.
2. The user is then asked to specify whether they wish to specify a phone number or e-mail address.
3. Based on the previous answer, the user is asked for either a phone number or an e-mail address.
4. The user is shown the data that they entered.

Such a flow would be described by the computation expression shown in Listing 19.

Listing 19. A flow described as a computation expression

```
let ExampleFlow () =
    Flow.Do {
        let! person = personFlowlet
        let! ct = contactTypeFlowlet
        let! contactDetails = contactFlowlet ct
        return! Flow.Static (finalPage person
            contactDetails)
    }
    |> Flow.Embed
```

This format provides a simple and expressive way of describing applications with a linear control flow.

5.2 Piglets

Formlets [8] allow user input to be retrieved in a type-safe fashion through the use of applicative functors to aid static composition. In spite of their advantages including type-safety, composability, and formal definition, formlets suffer from a lack of *modularity*: that is, the rendering of a formlet is tightly coupled to its data model. In order to change the ordering of components within the formlet, for example, it is necessary to modify the underlying data model.

Piglets [11] alleviate these concerns by separating the model and the view. Piglets consist of two separate components: a *stream* composed of values of components within the Piglets, and a *view builder function* which is provided with the values in the stream, and returns a rendering of the form. This is shown in Listing 20.

Listing 20. Structure of a Piglet

```
type Piglet<'a, 'v> =
    { stream: Stream<'a>; viewBuilder: 'v }
```

In order to create a Piglet, the `Yield` function is used. The argument to this can be a function, in which case static composition can be achieved through the use of the applicative-style composition operator ⊗. Both operations are shown in Listing 21.

Listing 21. Piglet Construction and Composition Functions

```
val Yield :
  'a -> Piglet<'a, (Stream<'a> -> 'b) -> 'b>

val ⊗:
  Piglet<'a -> 'b, 'v1 -> 'v2> ->
  Piglet<'a, 'v2 -> 'v3> ->
  Piglet<'b, 'v1 -> 'v3> ->
```

We are currently working towards an implementation of Piglets using the `UI.Next` framework. In particular, `UI.Next` replaces the `Stream` with primitives from the dataflow layer, as shown in Listing 22.

Listing 22. Piglets using UI.Next primitives

```
type Piglet<'a, 'v> =
    { read : View<Result<'a>> ; render : 'v }

val Yield : 'a -> Piglet<'a, (Var<'a> -> 'v) -> 'v>
```

In particular, a Piglet implemented using `UI.Next` consists of a `View` containing the current state of the form, and a rendering function. Creating a Piglet using `Yield` creates a `Var` which is used within the rendering function, and the implementation of ⊗ uses `View.Map2` to create a dependent `View`.

The original `Stream` implementation relied on manual subscription and pushing of values, whereas this is all handled by dataflow combinators in the `UI.Next` implementation. Replacing much of this imperative-style logic with functional combinators results in more concise, understandable, and readable code. Use of `UI.Next` primitives also avoids the need to specify explicit disposal functions, as was the case with Streams.

5.3 Sites with Multiple Pages

This work focuses on facilitating the creation of *single-page applications*: applications which run in a single page in the browser. Such applications often consist of multiple sub-pages, using JavaScript to transition between them.

Using our approach, implementing such functionality is simple and idiomatic. We begin by declaring a data type which describes the different pages in the site, and rendering functions (producing a `Doc`) for each:

```
type IFLPage = | Home | CallForPapers | Registration
     | Submission |

let renderHome v =
    Div0 [
        H10 [txt "Home"]
        ...
    ]

let renderCFP v = ...
```

We then create a `Var<IFLPage>`, representing the current page. Using this, we may then create a `View`, and map the appropriate rendering function, resulting in a view of the rendering of the current page. This may then be embedded into a page using `EmbedView`; navigation between pages is possible by changing the previously-created `Var`.

```
let v = Var.Create Home
View.FromVar v
|> View.Map (fun pg ->
    match pg with
    | Home -> renderHome v
    | CallForPapers -> renderCFP v
    | Registration -> renderRegistration v
    | Submission-> renderSubmission v)
|> Doc.EmbedView
```

6. Examples

In this section, we present two examples using the framework: the first of which showcases reactive animation and the treatment of object identity, and the second of which describes a form rendered using the Piglets implementation backed by `UI.Next`.

6.1 Object Constancy

Object Constancy is a technique for allowing an object representing a particular datum to be tracked through an animation. In particular, consider the case where the underlying data does not change, but the user controls filtering criteria: changes in such criteria may add new data to the visualisation, remove currently-displayed data, and assuming sorting criteria remains constant, change the ordering of the data.

In such a case, the objects representing the data remaining in the visualisation should not be removed and re-added, but instead should transition to their new positions. Such an example is discussed by Bostock [5], using the D3 [4] library.

The example described by Bostock [5] displays the percentage of the population in a particular age bracket for a number of different states, where 10 states are displayed. The percentages for each state are displayed in descending order. To recreate this example using our declarative animation framework, we begin by defining a data model using F# records.

```
type AgeBracket = | AgeBracket of string
type State = | State of string
type DataSet = {
      Brackets : AgeBracket []
      Population : AgeBracket -> State -> int
      States : State []
  }
type StateView = {
      MaxValue : double
      Position : int
      State : string
      Total : int
      Value : double
  }
```

Here, `AgeBracket` and `State` are representations of age brackets and states respectively, and `DataSet` is a representation of the entire data set as read in from an external data source. The `StateView` record specifies details about how a state should be displayed based on other visible items: `MaxValue` specifies the maximum percentage, `Position` specifies the rank of the item in the visible set, `State` specifies the name of the state, `Total` specifies the total number of items within the set and `Value` specifies the percentage value of the item.

```
let SimpleAnimation x y =
    Anim.Simple Interpolation.Double Easing.
        CubicInOut
        300.0 x y
let SimpleTransition =
    Trans.Create SimpleAnimation
let InOutTransition =
    SimpleTransition
    |> Trans.Enter (fun y -> SimpleAnimation Height
        y)
    |> Trans.Exit (fun y -> SimpleAnimation y Height
        )
```

Using this, it is possible to define an animation lasting for 300ms between 2 given values. With the animation, we can then create two transitions: an unconditional transition `SimpleTransition`, and a transition `InOutTransition` which is triggered when a DOM entry is added (`Enter`) and removed (`Exit`).

The `Enter` and `Exit` transitions interpolate the `y` co-ordinate of a bar between the bottom of the SVG graphic (`Height`) and a given position. In particular, upon entry, the element will transition from the origin position to the desired position; and will transition back to the origin position on exit.

We now specify a rendering function taking a `View<StateView>` and returning a `Doc` to be embedded within the tree. We elide some of the function in the interest of brevity, but you can find the complete source of the example online[3].

```
let Render (state: View<StateView>) =
  let anim name kind (proj: StateView -> double) =
      Attr.Animated name kind (View.Map proj state)
          string
  // Projection functions
  let x st = Width * st.Value / st.MaxValue
  let y st = Height * double st.Position / double st
      .Total
  let h st = Height / double st.Total - 2.

  S.G [Attr.Style "fill" "steelblue"] [
    S.Rect [
        "x" ==> "0"
        anim "y" InOutTransition y
        anim "width" SimpleTransition x
        anim "height" SimpleTransition h
    ] []
  ]
```

The helper function `anim` takes the name of the attribute to animate, the transition to use, and a projection from the state view to the value to use within the transition. We then specify three projection functions: one for the width of the bar, based on the value as a proportion of the maximum value in the set; one for the Y-position of the bar, and and one for the height of the bar. These may then be specified as attributes of the object to animate.

Finally, we create a selection box to allow the user to modify the age bracket, which in turn modifies the current list of `StateView`s. To implement object constancy, a key which uniquely identifies the data is required [14]. In the case of `StateView`, this is `State`: we use this when embedding the current set of visible elements using the `ConvertSeqBy` function, which is a memoising conversion function useful for preserving node identity. It is then possible to embed this into the DOM using `EmbedView`.

```
S.Svg ["width" ==> string Width; "height" ==> string
      Height] [
    shownData
```

[3] `https://github.com/intellifactory/websharper.ui.next/blob/master/src/ObjectConstancy.fs`

```
    |> View.ConvertSeqBy (fun s -> s.State) Render
    |> View.Map Doc.Concat
    |> Doc.EmbedView
]
```

6.2 Reactive Piglets

In this simple example, we define a form which consists of three fields: a first name, a last name, and a type of pet. We begin by defining a data model.

```
type Pet = | Cat | Dog | Piglet
type Person = { firstName: string; lastName: string;
     pet : Pet }
let Pets = [ Cat ; Dog ; Piglet ]
let showPet = function
  | Cat -> "Cat" | Dog -> "Dog" | Piglet -> "Piglet"
```

The next step is to use the `Return` and `Yield` operation to construct a Piglet. We also use the `Validation.Is` function to add validation to the form: should either the first or last name be empty, the failure will be propagated and displayed upon submission. The `WithSubmit` function adds a `Submitter` type, which can be used to snapshot the state of the form stream when a submission button is pressed. This can then be used as part of an AJAX call to a server, or to display errors.

```
let Person init =
    Piglet.Return (fun f l p -> { firstName = f;
        lastName = l ; pet = p})
    <*> (Piglet.Yield init.firstName
        |> Validation.Is Validation.NotEmpty "Please
            enter your first name.")
    <*> (Piglet.Yield init.lastName
        |> Validation.Is Validation.NotEmpty "Please
            enter your last name.")
    <*> (Piglet.Yield init.pet)
    |> Piglet.WithSubmit
```

Finally, we define a view for the Piglet. The `Render` function is provided with `Var`s for the fields in the Piglet, which are in turn used with the built-in form components in `UI.Next`.

```
let radioButtons (v: Var<Pet>) = ...
let Person init =
 ViewModel.Person init
 |> Piglet.Render (fun first last pet submit ->
     Doc.Concat [
      Div0 [Doc.TextNode "First Name: " ; Doc.Input
          [] first]
      Div0 [Doc.TextNode "Last Name: " ; Doc.Input
          [] last]
      radioButtons pet
      Div0 [Doc.Button "Submit" [] submit.Trigger]
      Div0 [Doc.TextView (submit.Output |>
       View.Map (function
       | Success u ->
          "Person: " + u.firstName + " " +
          u.lastName + ", Pet: " + (showPet u.pet)
       | Failure errs ->
          List.fold (fun out (str: ErrorMessage) ->
          out + " " + str.Message) "" errs)) ] ])
```

7. Related Work

7.1 Dataflow Systems

Synchronous dataflow languages originated as a means of specifying, designing, and implementing real-time systems such as those used within hardware. Languages such as ESTEREL [2], LUSTRE [13], and Lucid Synchrone [27] can compile programs to transition systems, in such a way that they may be formally verified using techniques such model checking. Such approaches are generally limited to the hardware domain, as the languages do not support features required in more general programs such as recursion or dynamic memory allocation.

REACTIVEML [20] is an extension of OCaml embedding the synchronous dataflow paradigm, providing primitives such as `signal` and `await` to express dataflow within the language. Cooper and Krishnamurthi [9] extend the Scheme programming language with dynamic dataflow to create a system, FrTime, which works by modifying the Scheme evaluator.

Scala.React [19] embeds the dataflow paradigm into Scala, introducing an imperative dataflow language, time-varying values, and event streams. The implementation is driven by a scheduler which proceeds in discrete steps, known as *propagation turns*, and the graph is constructed using weak pointers. A similar technique is used within OCaml React [6].

7.2 Functional Reactive Programming

Functional Reactive Programming [12, 15, 31] has served a large inspiration for the dataflow-based model for reactive user interfaces that we have described. FRP systems are based around primitive abstractions modelling time-varying data, referred to as `Behaviours` or `Signals`, and `Events` which occur either as a response to events such as user input, or when a signal satisfies a set of predicates.

The semantics of FRP are extremely attractive and clear: time-varying values are simple to transform and reason about, and event streams provide a method by which interaction can modify these. Despite their mathematical simplicity, the implementation of FRP semantics is notoriously difficult. In particular, early implementations of FRP such as `Fran` [12] remained very true to FRP semantics, at the cost of introducing memory leaks: in order to fully implement the FRP semantics (which allowed signals to depend on any past, present or future value), it was necessary to store every signal value, regardless of whether or not it would be used. This in turn led to memory usage growing linearly with time.

Subsequent approaches favour less expressive forms of FRP to provide better runtime guarantees. Real-time FRP [32] only allows signals to be used in ways which can be implemented efficiently, and can be reasoned about when developing real-time systems. Event-driven FRP [33] takes this further by only propagating changes as a result of a discrete *event*.

Arrowised FRP [16, 23] disallows signals to be treated as first-class values instead providing only *transformers* or *combinators* on primitive signals. Manipulating signals in this way is eased through the use of the Arrow abstraction [17]. Such an approach, although less expressive than purely-monadic FRP approaches such as `Fran`, are far more efficient and practical. The issue with arrowised FRP as it pertains to GUI programming is that it cannot adequately express *dynamic* dataflow graphs, as it becomes impossible to specify monadic combinators on time-varying values. Since our implementation of signals (`Views`), works purely on the latest available value, it *is* possible to specify monadic combinators, meaning that dynamic composition is possible.

The *Elm* programming language [10] is a language providing first-class FRP primitives with the goal of easing the creation of responsive GUIs. Elm implements static signal composition operators such as `lift` which work on the latest value in the signal, equivalent to `View.Map` in our dataflow layer, and $\texttt{lift}_n$, which is equivalent to `View.Map2` and `View.Apply`. In addition, Elm provides a construct `foldp` to perform transformations based on previous signal values. The asynchronous capabilities of Elm are mirrored in `UI.Next` through the use of the `MapAsync` function, which is supplemented by the RPC functionality supported by

WebSharper. In order to prevent space leaks, the creation of higher-order signals is prohibited by the type system. Such an approach is a good solution to the problem, but is not feasible when working within an existing ML type system. Instead, we forego the ability to perform time-dependent transformations as a primitive operation within the reactive layer, instead postulating that such functionality may be attained either using simple single-layer callbacks, or an approach based on concurrent processes such as Concurrent ML [28].

More theoretical recent work [18**?**] focuses on languages implementing FRP semantics, including higher-order signals, while guaranteeing leak-freedom. In particular, the approach described by **?**] divides expressions into those which may be evaluated immediately, and those which depend on future values and whose evaluation must be delayed. In order to prevent space leaks, obsolete behaviour values are aggressively deleted. The approach relies on a specialised type system and an explicit notion of time being exposed to the programmer, which limits its applicability to our problem domain.

7.3 Reactive DOM Libraries

Facebook React [1] is a library which, in a similar way to our approach, allows developers to construct DOM nodes programmatically. This process is facilitated through `JSX`, an HTML-like markup language with facilities for property-based data binding. The key concept behind React is the use of an automated 'diff' algorithm, driven by a global notion of time instead of a dataflow system: as a result, DOM updates are batched for efficiency. We decided to use a dataflow-backed system instead of purely a diff algorithm to avoid losing control over DOM node identity. Our approach uses some aspects of React, such as dirty-checking, but this is localised to DOM fragments which have been specifically embedded.

Flapjax [22] is a dataflow-backed programming language providing full FRP functionality which can also be used as a JavaScript library. As the library does not prohibit higher-order signals, it is possible to introduce space leaks as previously discussed.

7.4 Functional Web Programming

Functional programming has been found to be very applicable to the web programming domain. In particular, functional programming and the static type systems associated with many functional programming languages allow for the development of many powerful web abstractions to ease the structuring and development of web applications.

WebSharper takes inspiration from Links [7], a language which allows client, server, and database code to be written in a single source language, thus mitigating the impedance mismatch problem. F# functions are compiled to JavaScript aided by quotations [30], and AJAX calls are easily represented using F# asynchronous workflows.

Formlets [8] are an abstraction for retrieving typed user input from HTML forms, which have been extended by Denuzière et al. [11] to enable the specification of customised rendering functions. Flowlets [3] augment Formlets with a monadic interface to enable the construction of multiple dependent formlets.

The *i*nteractive *Data*, or `iData` abstraction [25] is an edit-driven approach to type-safe forms: edits to input fields trigger computations, with previous state being restored in the case of invalid data being entered. This is taken further by the `iTasks` workflow management system [26] which makes use of multiple high-level combinators such as recursion, sequence, and choice.

8. Conclusion

In this paper, we have presented a framework in F#, `UI.Next`, facilitating the creation of reactive DOM applications backed by a dynamic dataflow graph. Guided by previous work on functional reactive programming and dataflow systems, our framework consists of a dataflow layer consisting of `Var`s, representing time-varying variables, and `View`s, read-only representations of `Var`s in a dataflow graph. Our dataflow representation is modular as it is decoupled from the DOM layer, and amenable to garbage collection by not allowing higher-order event streams or keeping strong links between dataflow nodes. While inspired by functional reactive programming, we make several simplifications which facilitate the implementation of higher-order monadic operations on `View`s to allow dynamic dataflow graph composition, in turn supporting common GUI programming patterns.

The DOM layer uses a monoidal interface to aid composability, and through the use of the `EmbedView` function, allows time-varying DOM elements to be directly embedded into a larger tree. Updates to the in-browser DOM are performed only when necessary and build on the well-founded notion of dirty-checking, minimising needless node generation both as an efficiency measure and to preserve node identity. Such an approach within a strongly, statically-typed language has proven extremely useful in aiding the implementation of several functional web abstractions, such as Piglets and Flowlets.

Additionally, we have presented an interface for declarative animation based on the dataflow graph, which integrates directly into the reactive DOM layer as reactive attributes, and can be backed directly by reactive attributes. This enables the creation of rich, data-backed animations using a statically-typed, declarative interface.

8.1 Future Work

The current implementation of `UI.Next` is freely available for experimentation. Future work will be centred around effectively integrating event streams within the dataflow layer to aid handling of user interactions. We envisage that usage of the concurrent programming paradigm as in Concurrent ML [28] or Hopac [4] will prove to be a promising future direction for this purpose.

We are currently investigating how to further integrate the reactive layer with plain HTML through the use of an F# type provider [29]. A more ambitious goal involves implementing the dataflow layer in a distributed setting with updates to a `Var` on a server being propagated automatically to clients. We are additionally currently working on a frontend implementation for the Windows Presentation Foundation, to allow the dynamic dataflow backend to be used within traditional desktop applications.

Other planned work includes further efficiency benchmarking and optimisation: while we currently implement some optimisations to minimise physical DOM accesses, further optimisation is possible.

Strongly, statically-typed languages have been shown to aid the development of web applications by better allowing applications to be structured, and decreasing debugging time by detecting errors earlier in the development process. We hope that continued research into functional web programming will allow web developers to fully take advantage of these advances.

References

[1] React | A JavaScript Library for Building User Interfaces. `http://facebook.github.io/react/`, 2014.

[2] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992. ISSN 01676423. .

[3] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing Reactive GUIs in F# using WebSharper. In *Implementation and Application of Functional Languages*, pages 203–216. Springer, 2011.

[4] `https://github.com/VesaKarvonen/Hopac`

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-Driven Documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[5] Mike Bostock. Object Constancy. `http://bost.ocks.org/mike/constancy/`, 2012.

[6] Daniel Bünzli. React / Erratique. `http://erratique.ch/software/react`, 2010.

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In FrankS de Boer, MarcelloM Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer Berlin Heidelberg, 2007. .

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The Essence of Form Abstraction. In *Programming Languages and Systems*, pages 205–220. Springer, 2008.

[9] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin Heidelberg, 2006. .

[10] Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM. .

[11] Loïc Denuzière, Ernesto Rodriguez, and Adam Granicz. Piglets to the Rescue. In Rinus Plasmeijer, editor, *Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages (IFL '13)*, 2013.

[12] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. ISSN 0018-9219. .

[14] Jeffrey Heer and Michael Bostock. Declarative Language Design for Interactive Visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1149–1156, 2010.

[15] Paul Hudak. Functional Reactive Programming. In Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, chapter 1, page 1. Springer Berlin Heidelberg, Berlin, Heidelberg, March 1999. ISBN 978-3-540-65699-9. .

[16] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[17] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000. ISSN 01676423. .

[18] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order Functional Reactive Programming in Bounded Space. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 45–58, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. .

[19] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the Observer Pattern. Technical Report EPFL-REPORT-148043, 2010.

[20] Louis Mandel and Marc Pouzet. ReactiveML: A Reactive Extension to ML. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 82–93, New York, NY, USA, 2005. ACM. ISBN 1-59593-090-6. .

[21] Conor McBride and Ross Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18(01):1–13, May 2007. .

[22] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM. .

[23] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. .

[24] Tomas Petricek and Don Syme. The f# Computation Expression Zoo. In *Practical Aspects of Declarative Languages*, pages 33–48. Springer, 2014.

[25] Rinus Plasmeijer and Peter Achten. iData for the World Wide Web âĂŞ Programming Interconnected Web Forms. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 242–258. Springer Berlin Heidelberg, 2006. .

[26] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 141–152, New York, NY, USA, 2007. ACM. .

[27] Marc Pouzet. Lucid Synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006.

[28] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[29] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, and Others. Strongly-typed language support for internet-scale information sources. Technical report, Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

[30] Don Syme, Adam Granicz, and Antonio Cisternino. Language-Oriented Programming: Advanced Techniques. In *Expert F# 3.0*, pages 477–501. Springer, 2012.

[31] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from First Principles. *SIGPLAN Not.*, 35(5):242–252, May 2000. ISSN 0362-1340. .

[32] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, volume 36 of *ICFP '01*, pages 146–156, New York, NY, USA, October 2001. ACM. ISBN 1-58113-415-0. .

[33] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-Driven FRP. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 155–172. Springer Berlin Heidelberg, 2002. .

Blank Canvas and the remote-monad design pattern

A Foreign Function Interface to the JavaScript Canvas API

Extended Abstract

Andrew Gill Aleksander Eskilson Ryan Scott James Stanton

Information and Telecommunication Technology Center
The University of Kansas
{andygill,aeskilson,ryanscott,jstanton}@ittc.ku.edu

Abstract

JavaScript is the de-facto assembly language of the internet. Browsers offer an array of powerful rendering and event processing services, including a simple 2D canvas. Blank Canvas is Haskell DSL that provides a Foreign Function Interface to the JavaScript canvas API and the JavaScript event API. With this capability, Haskell programmers can draw pictures on the browsers, and access input from the keyboard and mouse. At the University of Kansas, we use the blank-canvas package for teaching Haskell, where it provides a more interesting I/O experience than stdio.

We investigate the use of the remote-monad design pattern, using Blank Canvas as our driving example. After explaining the design pattern, and constructing the basic remote capability, we critically assess the feasibility of our straightforward approach, and explore improvements.

1. Introduction

Blank Canvas is a Haskell binding to the complete HTML5 Canvas API. Blank Canvas allows Haskell users to write, in Haskell, interactive images onto their web browsers. Blank Canvas gives the user a single full-window canvas, and provides many well-documented functions for rendering images.

As a first example and in order to give a feel for the library, consider drawing a single red line onto the canvas. In Haskell, using Blank Canvas we can write the following.

```
send context $ do                    -- ❶
        moveTo(50,50)                -- ❷
        lineTo(200,100)
        lineWidth 10
        strokeStyle "red"
        stroke()                     -- ❸
```


First, the `send` command (❶) sends a monadic list of commands to a (graphics) context. Second, the list of commands (❷) operates on this context in an imperative manner. Finally, the `stroke()` commands (❸) actually draws the red line. At this point, the screen looks like

In JavaScript, the same actions can be performed using an almost identical code fragment.

```
-- JavaScript
context.moveTo(50,50);
context.lineTo(200,100);
context.lineWidth = 10;
context.strokeStyle = "red";
context.stroke();
```

Blank Canvas has packaged the JavaScript API as a small Domain Specific Language in Haskell, and allows Haskell users to access the canvas. At the University of Kansas, we make extensive use of this API. Students find it easy to understand, and complete medium-sized projects, usually games, using Blank Canvas as the primary IO mechanism. In the graduate FP class, we also undertake an FRP exercise [1, 8], which uses Blank Canvas to render shapes onto the canvas. Using the Blank Canvas API, we also have developed slide presentation software, and an internal animation framework. Finally, the popular `diagrams` package [11, 12] has been ported to use blank Canvas as a back end [7].

The central issue, and the subject of the full paper, is quantifying the costs associated with having code execute outside the Haskell runtime system, and remotely running monadic code. The browser, running JavaScript, is typically a separately executing process from a Haskell program. Thus, we have two extreme solutions to our API implementation, sending each command over a network connection piecemeal, or compiling the entire Haskell program and runtime system into JavaScript. We investigate a middle ground between sending commands piecemeal, and compiling wholesale to JavaScript, using a design pattern.

2. Remote-monad DSL Pattern

Haskell has no standard graphics library. Instead, a rich Foreign Function Interface (FFI) capability is used to tunnel to C, and onwards to established libraries, such as OpenGL. There are three conceptual problems to be solved in crossing to non-native C (and C++) libraries, such as OpenGL:

- First, control flow needs to flow to the correct C function. Given the lowest level of the GHC runtime system is written in C, this is straightforward. Callbacks, from C to Haskell can also be arranged.
- Second, the data structures that are arguments and results of calls to (and from) C need to be coerced into the correct format. C strings are not the same as Haskell strings.
- Third, the abstractions of OpenGL may not be idiomatic Haskell abstractions. For example, many APIs assume OO-style class inheritance. This can be simulated in Haskell, but raises an obfuscation barrier.

Any time control flow leaves the eco-system, all three of these concerns come to into play. All three are well handled in the Haskell FFI for C. There is a way of directly promoting a C function into Haskell-land, there is a good support for marshalling data structures, in C structures, as well an automatic memory management support, and Haskell abstraction capabilities are used to build more Haskell-centric APIs on top of the FFI capability. Calling C functions directly from Haskell is cheap. However, we want to investigate another FFI with a different tradeoff, where the call is remote and expensive, and understand what abstractions can be used.

The **remote-monad pattern** is our name for the transmission of a (fixed) set of commands to a remote site, for execution. In its most basic form, we have a `send` command, a remote location identifier, and a single command.

```
send :: Name -> RemoteCommand a -> IO a
remote :: Name
readRemoteFile :: String -> RemoteCommand String
example :: IO ()
example = do
     txt <- send remote (readFile "foo")
     print txt
```

The idea is that the `send` command reifies the `RemoteCommand`, sends it to the remote location, runs it, accepts the response, transports it back to the original `send`, and returns the remotely generated value. This pattern can implemented using the first two of the three requirements above of a an FFI interface, as described above. First, a remote function is called (control is moved to the remote site), and back. Second, the pattern takes care of the necessary data conversation conventions, both in transport, and on the remote site.

The remote-monad command has many manifestations. At KU, we have used it for Blank Canvas, but also for Sunroof (sending whole JavaScript programs), and using Kansas Lava [2] to talk to remote peripherals. Furthermore, The pattern appears in many different places. If we interpret "remote" to mean different environment, the run function for many well know monads can be considered a `send`. Software transactional memories (`atomically` [3]), the ST monad (`runST` [5]) and IO (`forkIO` [4]) can also ben considered close instances of the remote-monad pattern, where a monad is executed in a different context.

The remote-monad patterns has two laws:

```
send (return a) = return a              (1)
send m1 >>= send m2 = send (m1 >>= m2) (2) [*]
```

The first law states that a `send` has no effect except the remote commands. The second law, which has a pre-condition of non-interference[*], states that remote commands preserve ordering, and can be split and joined into different sized packets. The pre-condition is interesting: it is possible to have the result of two `send`'s be the same as a single `send`, yet the observable effects be different, for example a screen update is done between the two send commands.

3. Blank Canvas

Blank Canvas is a small library at around 1500 lines of Haskell. At the heart of the library is the remote-monad, and the `send` command. There is quite a bit of careful construction, however, to make everything work.

The packet principles are:

- Where possible, everything in a `send`-packet should be sent to be executed together.
- The breaks between packets should be deterministic and statically/syntactically determinable.
- Packets are not combined between different calls to `send`.

The command principles are:

- Anything that returns `()` is asynchronous, and may be combined with the next monadic command, or send instantly.
- Anything that does not return `()` is synchronous, and requires a round-trip to the server.

The `Canvas` data type has a small number of constructors. The four main constructor are:

```
data Canvas :: * -> * where
  Method :: Method                        -> Canvas ()
  Query  :: Query a                       -> Canvas a
  Bind   :: Canvas a -> (a -> Canvas b) -> Canvas b
  Return :: a                             -> Canvas a
```

The choice of constructors follow the principles carefully. `Method` is used for asynchronous drawing commands, while `Query` is used for commands that need a round trip. `Bind` and `Return` form the monad for Canvas, allowing monadic reification [9, 10].

4. Benchmarking Blank Canvas

A key question is the cost of using the remote-monad design pattern. At first glance, it would seem prohibitive. The current version of Blank Canvas (0.5) uses Haskell Strings internally, transliterating each command to a String, and combines intra-`send` commands, where possible. Absolutely every command needs translated then sent over a (typically local) network.

We have measured Blank Canvas on a small number of benchmarks, and compared to native JavaScript. We have two classes of benchmarks: "display" benchmarks, that simply render to the HTML5 canvas, and "query" benchmarks, that the inner loop of the benchmark invokes some from of query that requires a round-trip from server, to client, back to the server.

Blank Canvas works on almost any modern, HTML5 compliant browser. Figure 1 gives our initial results. We have tested each benchmark on recent versions of Firefox and Chrome, on both Linux and OSX, to gain a crude overall benchmark for how much using the remote-monad design pattern costs. The Haskell tests were run 100 times using criterion [6], the JavaScript tests were averaged over 100 runs.

		Linux						OSX					
		Firefox			Chrome			Firefox			Chrome		
	Benchmark	Haskell	JS	Ratio	Haskell	JS	Ratio	Haskell	JS	Ratio	Haskell	JS	Ratio
Display	Bezier	6.90	4.09	1.69	4.03	1.71	2.36	11.56	3.23	3.58	8.51	0.55	15.47
	CirclesRandSz	138.64	105.45	1.31	71.15	25.07	2.84	68.77	46.26	1.49	66.97	12.84	5.22
	CirclesUniSz	106.90	75.41	1.42	62.43	15.28	4.09	71.19	31.32	2.27	67.52	12.54	5.38
	FillText	57.95	48.33	1.20	4.99	1.80	2.77	7.81	5.22	1.50	5.07	1.29	3.93
	StaticAsteroids	365.10	121.71	3.00	309.59	14.92	20.75	197.92	30.49	6.49	201.21	8.07	24.93
	Image	214.63	21.87	9.81	421.41	57.41	7.34	596.29	209.74	2.84	657.68	75.82	8.67
Query	IsPointInPath	22.31	0.49	45.53	27.73	0.26	106.65	33.72	0.73	46.19	74.71	0.37	201.91
	MeasureText	184.18	50.56	3.64	160.76	2.04	78.80	265.22	5.92	44.80	320.49	1.40	228.92
	Rave	58.30	20.50	2.84	38.66	1.71	22.61	62.18	10.98	5.66	115.43	0.58	199.02

Table 1. Benchmarking Blank Canvas vs. Native JavaScript. (times in milliseconds)

The display benchmarks are:

- Bezier – drawing 1000 bezier curves.
- CirclesRandomSize – 1000 filled in circles of random sizes.
- CirclesUniformSize – 1000 filled in circles of a uniform size.
- FillText – 50 words
- StaticAsteroids – 1000 wire polygons.
- Image – 100 images of a cat, drawn at different sizes.

What can be seen is that the relative performance varies widely, depending on browser and benchmark, but on average, the cost of using Haskell, and the Blank Canvas API is between approximately 2 and 25, and typically less than 5. This is surprising and encouraging! We were expecting a larger overhead. We can also see the importance of a testing with different environments.

The query benchmarks are:

- IsPointInPath – Draw 1000 rectangles and and points; the points' color depends on if the point is inside the rectangle.
- MeasureText – measure the width of 100 words.
- Rave – gradient bars.

Here, as expecting, the cost is much higher. However, again, the result is encouraging. The places where the overhead is especially high are where a specific browser does an especially good job of optimization. Further, as has been pointed out by Jeffrey Rosenbluth, a number of our queries simply allocate a numbered resource, and this unique number generation can be done on the server, allowing a command rather than query to be used.

5. Related Work

The final paper will contain a detailed related work section, including various JavaScript-based Haskell compilers, and other approaches to the FFI problem.

6. Conclusion

This short extended abstract has introduced the remote-monad design pattern, and shown its use in a full scale case-study for accessing the HTML5 Canvas JavaScript API. The cost was not prohibitive, and the API is useful in practice.

Acknowledgments

We would like the thank Jeffrey Rosenbluth, for writing `diagrams-canvas`, and helping with the implementation of Blank Canvas, and Justin Dawson, for working on an earlier version of the asteroid benchmark. This material is based upon work supported by the National Science Foundation under Grant No. 1117569 and Grant No. 1350901.

References

[1] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL `http://conal.net/papers/icfp97/`.

[2] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp. Types and associated type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. *Higher-Order and Symbolic Computation*, pages 1–20, 2013. ISSN 1388-3690. . URL `http://dx.doi.org/10.1007/s10990-013-9098-7`.

[3] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[4] S. L. P. Jones, A. D. Gordon, and S. Finne. Concurrent haskell. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 295–308, 1996.

[5] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. *ACM SIGPLAN Notices*, 29(6):24–35, 1994.

[6] B. O'Sullivan. `http://hackage.haskell.org/package/criterion`, 2014.

[7] J. Rosenbluth. `http://hackage.haskell.org/package/diagrams-canvas`, 2014.

[8] N. Sculthorpe and A. Gill. `http://hackage.haskell.org/package/yampa-canvas`, 2014.

[9] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298. ACM, 2013. URL `http://dl.acm.org/citation.cfm?doid=2500365.2500602`.

[10] J. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *International Conference on Functional Programming*, pages 299–304. ACM, 2013.

[11] B. Yorgey. `http://hackage.haskell.org/package/diagrams`, 2014.

[12] B. A. Yorgey. Monoids: theme and variations (functional pearl). In *Haskell Syposium*. ACM, 2012.

Project H: Programming R in Haskell

PRELIMINARY DRAFT

Mathieu Boespflug Facundo Domínguez
Alexander Vershilov
Tweag I/O

Allen Brown
Amgen

Abstract

A standard method for augmenting the "native" set of libraries available within any given programming environment is to extend this set via a foreign function interface provided by the programming language. In this way, by exporting the functionality of external libraries via *binding modules*, one is able to reuse libraries without having to reimplement them in the language *du jour*.

However, *a priori* bindings of entire system libraries is a tedious process that quickly creates an unbearable maintenance burden. We demonstrate an alternative to monolithic and imposing binding modules, even to make use of libraries implemented in a special-purpose, dynamically typed, interpreted language. As a case study, we present H, an R-to-Haskell interoperability solution making it possible to program all of R, including all library packages on CRAN, from Haskell, a general-purpose, statically typed, compiled language. We demonstrate how to do so efficiently, without marshalling costs when crossing language boundaries and with static guarantees of well-formation of expressions and safe acquisition of foreign language resources.

Keywords R, Haskell, foreign function interface, quasiquotation, language embedding, memory regions

1. Introduction

The success or failure in the industry of a programming language within a particular problem domain is often predicated upon the availability of a sufficiently plethoric set of good quality libraries relevant to the domain. Libraries enable code reuse, which ultimately leads to shorter development cycles. Yet business and regulatory constraints may impose orthogonal requirements that not all programming languages are able to satisfy.

Case in point: at Amgen, we operate within a stringent regulatory environment that requires us to establish high confidence as to the correctness of the software that we create. In life sciences, it is crucial that we aggressively minimize the risk that any bug in our code, which could lead to numerical, logical or modelling errors with tragic consequences, goes undetected.

We present a method to make available any foreign library without the overheads typically associated with more traditional approaches. Our goal is to allow for the seamless integration of R with Haskell — invoking R functions on Haskell data and *vice versa*.

Foreign Function Interfaces The complexity of modern software environments makes it all but essential to interoperate software components implemented in different programming languages. Most high-level programming languages today include a *foreign function interface (FFI)*, which allows interfacing with lower-level programming languages to get access to existing system and/or purpose-specific libraries [2, 9]. An FFI allows the programmer to give enough information to the compiler of the host language to figure out how to *invoke* a foreign function included as part of a foreign library, and how to *marshal* arguments to the function in a form that the foreign function expects. This information is typically given as a set of bindings, one for each function, as in the example below:

```
{-# LANGUAGE ForeignFunctionInterface #-}
module Example1 (getTime) where
import Foreign
import Foreign.C

#include <time.h>

data TimeSpec = TimeSpec
  {seconds     :: Int64
  ,nanoseconds :: Int32
  }

foreign import ccall "clock_gettime"
  c_clock_gettime :: ClockId → Ptr TimeSpec → IO CInt

getTime :: ClockId → IO TimeSpec
getTime cid = alloca $ λts. do
  throwErrnoIfMinus1_ "getTime" $
    c_clock_gettime cid ts
  peek ts
```

In the above, `c_clock_gettime` is a binding to the `clock_gettime()` C function. The API conventions of C functions are often quite different from that of the host language, so that it is convenient to export the wrapper function `getTime` rather than the binding directly. The wrapper function takes care of converting from C representations of arguments to values of user defined data types (performed by the `peek` function, not shown), as well as mapping any foreign language error condition to a host language exception.

Binding generators These bindings are tedious and error prone to write, verbose, hard to read and a pain to maintain as the API of the underlying library shifts over time. To ease the pain, over the years,

binding generators have appeared [1], in the form of pre-processors that can parse C header files and automate the construction of binding wrapper functions and argument marshalling. However, these tools:

1. do not alleviate the need for the programmer to repeat in the host language the type of the foreign function;
2. add yet more complexity to the compilation pipeline;
3. being textual pre-processors, generate code that is hard to debug;
4. are necessarily limited in terms of the fragments of the source language they understand and the types they can handle, or repeat the complexity of the compiler to parse the source code.

Point (1) above is particularly problematic, because function signatures in many foreign libraries have a knack for evolving over time, meaning that bindings invariably lag behind the upstream foreign libraries in terms of both the versions they support, and the number of functions they bind to.

Moreover, such binding generators are language specific, since they rely on intimate knowledge of the foreign language in which the foreign functions are available. In our case, the foreign language is R, which none of the existing binding generators support. We would have to implement our own binding generator to alleviate some of the burden of working with an FFI. But even with such a tool in hand, the tedium of writing bindings for all standard library functions of R, let alone all functions in all CRAN packages, is but a mildly exciting prospect. One would need to define a monolithic set of bindings (i.e. a *binding module*), for *each* R package. Because we cannot anticipate exactly which functions a user will need, we would have little recourse but to make these bindings as exhaustive as possible.

Rather than *bind* all of R, the alternative is to *embed* all of R. Noting that GHC flavoured Haskell is a capable meta-programming environment, the idea is to define code generators which, at each call site, generates code to invoke the right R function and pass arguments to it using the calling convention that it expects. In this way, there is no need for *a priori* bindings to all functions. Instead, it is the code generator that produces code spelling out to the compiler exactly how to perform the R function call – no binding necessary.

It just so happens that the source language for these code generators is R itself. In this way, users of H may express invocation of an R function using the full set of syntactical conveniences that R provides (named arguments, variadic functions, *etc.*), or indeed write arbitrary R expressions. R has its own equivalent to `clock_gettime()`, called `Sys.time()`. With an embedding of R in this fashion, calling it is as simple as:

```
printCurrentTime = do
  now ← [r| Sys.time() ]
  putStrLn ("The current time is: " ++ fromSEXP now)
```

The key syntactical device here is *quasiquotes* [7], which allow mixing code fragments with different syntax in the same source file — anything within an `[r| ... ]` pair of brackets is to be understood as R syntax.

Contributions In this paper, we advocate for a novel approach to programming with foreign libraries, and illustrate this approach with the first complete, high-performance tool to access all of R from a statically typed, compiled language. We highlight the difficulties of mixing and matching two garbage collected languages that know nothing about each other, and how to surmount them by bringing together existing techniques in the literature for safe memory management [6]. Finally, we show how to allow optionally ascribing precise types to R functions, as a form of compiler-checked documentation and to offer better safety guarantees.

Outline The paper is organized as follows. We will first walk through typical uses of H, before presenting its overall architecture (Section 2). We delve into a number of special topics in later sections, covering how to represent foreign values efficiently in a way that still allows for pattern matching (Section 3.1), optional static typing of dynamically typed foreign values (Section 3.2), creating R values from Haskell (Section 3.3) and efficient memory management in the presence of two separately managed heaps with objects pointing to arbitrary other objects in either heaps (Section 3.4). We conclude with a discussion of the overheads of cross language communication (Section 4) and an overview of related work (Section 5).

2. H walkthrough and overall architecture

2.1 Foreign values

Foreign functions act on *values*, for which presumably these foreign functions know the representation in order to compute with them. In the specific case of R, *all* values share a common structure. Internally, R represents every entity that it manipulates, be they scalars, vectors, uninterpreted term expressions, functions or external resources such as sockets, as pointers to a `SEXPREC` structure, defined in C as follows:

```
typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
        struct primsxp_struct primsxp;
        struct symsxp_struct symsxp;
        struct listsxp_struct listsxp;
        struct envsxp_struct envsxp;
        struct closxp_struct closxp;
        struct promsxp_struct promsxp;
    } u;
} SEXPREC, *SEXP;
```

Each variant of the union struct corresponds to a different form of value. However, no matter the form, all values at least share the same header (called `SEXPREC_HEADER`). The type of pointers to `SEXPREC`s is abbreviated as `SEXP`. In order to invoke functions defined in R then, we simply need a way to construct the right `SEXPREC` representing that invocation, and then have R interpret that invocation. We will cover how to do so in Section 2.2, but for now we do need to define in Haskell what a `SEXP` is:

```
data SEXPREC
type SEXP = Ptr SEXPREC
```

2.2 Interoperating scripting languages

R source code is organized as a set of *scripts*, which are loaded one by one into the R interpreter. Each statement in a each script is evaluated in-order and affects the global environment maintained by the R interpreter, which maps symbols to values. In its simplest form, H is an *interactive environment* much like R, with a global environment altered by the in-order evaluation of statements.

The central and most general mechanism by which H allows interoperating with R is quasiquotation. A *quasiquote* is a partial R script — that is, a script with "holes" in it that stand in for as of yet undetermined portions. An example quasiquote in Haskell of an R snippet is:

```
[r| function(x) x + 1 ]
```

This quasiquote is *ground*, in that it does not contain any holes (called *antiquotes*), but one can also antiquote inside a quasiquote:

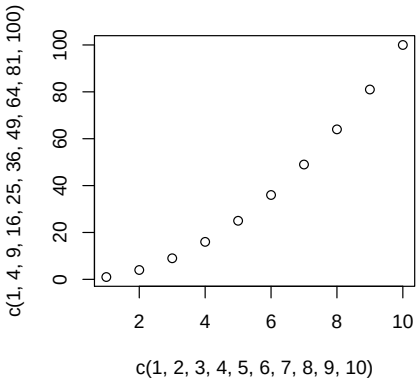


Figure 1. Output of ⟦r| `plot(xs_hs, ys_hs)` |⟧. The data is generated from Haskell. R draws the plot.

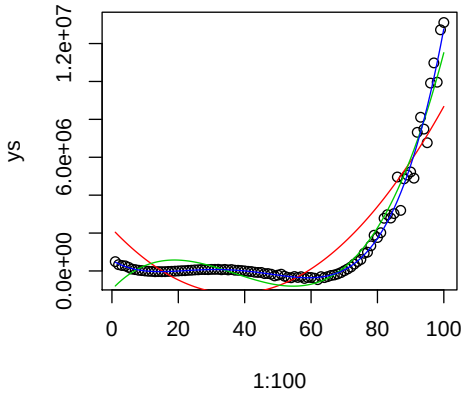


Figure 2. Fitting polynomial models of increasing degree ($n = \{2, 3, 4\}$) to a set of points in Haskell. R fits the models.

```
let y = mkSEXP 1
in ⟦r| function(x) x + y_hs |⟧
```

By convention, any symbol with a `_hs` suffix is treated specially (see Section 2.5). It is interpreted as a reference to a Haskell variable defined somewhere in the ambient source code. That is, any occurrence of a symbol of the form `x_hs` does not denote a variable of the object language — it is an antiquote referring to variable x in the host language. Given any quasiquote, it is possible to obtain a full R script, with no holes in it, by *splicing* the value of the Haskell variables into the quasiquote, in place of the antiquotes.

At a high-level, H is a desugarer for quasiquotes implemented on top of a Haskell interactive environment, such as GHCi [4]. It defines how to translate a quasiquotation into a Haskell expression. Just as R includes an interactive environment, H includes an interactive environment, where the input is a sequence of Haskell expressions including quasiquoted R code snippets, such as in the following session, where we plot part of the quadratic function, directly from the Haskell interactive prompt:

```
H⟩ let xs = [1 .. 10] :: [Double]
H⟩ let ys = [x^2 | x ← xs]
H⟩ result ← ⟦r| as.character(ys_hs) |⟧
H⟩ H.print result
[1] "1"  "4"  "9"  "16"  "25"  "36"  "49"  "64" ...
H⟩ ⟦r| plot(xs_hs, ys_hs) |⟧
⟨graphic output (See Figure 1)⟩
```

Now say that we are given a set of random points, roughly fitted by some non-linear model. For the sake of example, we can use points generated at random along a non-linear curve by the following Haskell function:

```
import System.Random.MWC
import System.Random.MWC.Distributions

generate :: Int32 → IO Double
generate ix =
  withSystemRandom ∘ asGenIO $ λgen.
    let r = (x − 10) ∗ (x − 20) ∗ (x − 40) ∗ (x − 70)
            + 28 ∗ x ∗ (log x)
    in do v ← standard gen
    return $ r ∗ (1 + 0.15 ∗ v)
    where x = fromIntegral ix
```

As before, take a set of coordinates:

```
H⟩ ⟦r| xs <- c(1:100) |⟧
H⟩ ⟦r| ys <- mapply(generate_hsfun, xs) |⟧
```

`generate_hsfun` is a *function splice* — just like any other splice, except that the spliced value is higher-order, i.e. a function. R's `mapply()` applies the Haskell function to each element of `xs`, yielding the list `ys`.

Our goal is to ask R to compute estimates of the parameters of polynomial models of increasing degree, with models of higher degree having a higher chance of fitting our dataset well. The R standard library provides the `nls()` function to compute the non-linear least-squares estimate of the parameters of the model. For example, we can try to fit a model expressing the relation between `ys` to `xs` as a polynomial of degree 3:

```
H⟩ ⟦r| P3 <- ys ~ a3*xs**3 + a2*xs**2 + a1*xs + a0 |⟧
H⟩ ⟦r| initialPoints <- list(a0=1,a1=1,a2=1,a3=1) |⟧
H⟩ ⟦r| model3 <- nls(P3, start=initialPoints) |⟧
```

As the degree of the model increases, the residual sum-of-squares decreases, to the point where in the end we can find a polynomial that fits the dataset rather well, as depicted in Figure 2, produced with the following code:

```
⟦r| plot(xs,ys) |⟧
⟨graphic output (See Figure 2)⟩
⟦r| lines(xs,predict(model2), col = 2) |⟧
⟦r| lines(xs,predict(model3), col = 3) |⟧
⟦r| lines(xs,predict(model4), col = 4) |⟧
```

2.3 Scripting from compiled modules

While an interactive prompt is extremely useful for exploratory programming, writing a program as a sequence of inputs for a prompt is a very imperative style of programming with limited abstraction facilities. Fortuntaly, H is also a library. Importing the library brings in scope the necessary definitions in order to embed quasiquotes such as the above in modules of a compiled program.

Behind the scenes, the H library hosts an *embedded instance* of the R interpreter, available at runtime. As in the interactive environment, this embedded instance is stateful. It is possible to mutate the global environment maintained by the interpreter, say by introducing a new top-level definition. Therefore, interaction with this embedded instance must be sequential. In order to enforce sequential access to the interpreter, we introduce the R monad and make all code that ultimately calls into the interpreter *actions* of

the R monad. As a first approximation, the R monad is a simple wrapper around the IO monad (but see Section 3.4)[1].

$$\textbf{newtype}\ \mathsf{R}\ a = \mathsf{R}\ (\mathsf{IO}\ a)$$

$$\mathsf{withEmbeddedR} :: \mathsf{R}\ a \to \mathsf{IO}\ a$$

withEmbeddedR first spawns an embedded instance, runs the provided action, then finalizes the embedded instance. There is no other way to run the R monad.

2.4 Rationale

R is a very dynamic language, allowing many code modifications during runtime, such as rebinding of top-level definitions, super assignment (modifying bindings in parent scopes), (quasi-)quotation and evaluation of expressions constructed dynamically. The R programming language is also so-called "latently typed" - types are checked during execution of the code, not ahead of time. Many of these features are not compiler friendly.

Haskell, by contrast, is designed to be much easier to compile. This means that not all R constructs and primitives can be readily mapped to statically generated Haskell code with decent performance. In particular, top-level definitions in Haskell are never dynamically rebound at runtime: a known function call is hence often a direct jump, rather than incurring a dynamic lookup in a symbol table (the environment).

Much of the dynamic flavour of R likely stems from the fact that it is a scripting language. The content of a script is meant to be evaluated in sequential order in an interactive R prompt. The effect of loading multiple R scripts at the prompt is in general different depending on the order of the scripts.

Central to the design of Haskell, by contrast, is the notion of separately compilable units of code, called *modules*. Modules can be compiled separately and in any order (provided some amount of metadata about dependencies). Contrary to R scripts, the order in which modules are loaded into memory is non-deterministic.

For this reason, in keeping to a simple solution to interoperating with R, we choose to devolve as much processing of R code as possible to an embedded instance of the R interpreter and retain the notion of global environment that R provides. This global environment can readily be manipulated from an interactive environment such as GHCi [4]. In compiled modules, access to the environment as well as encapsulation of any effects can be mediated through a custom monad, the R monad presented in Section 2.3.

2.5 Runtime reconstruction of expressions

Quasiquotes are purely syntactic sugar that are expanded at compile time. They have no existence at runtime. A quasiquote stands for an R expression, which at runtime we therefore have to reconstruct. In other words, the expansion of a quasiquote is code that generates an R expression. For a ground quasiquote whose content is R expression S, we construct a Haskell expression E, such that

$$\texttt{R\_Parse}(S) = E.$$

This law falls out as a special case of a more general law about antiquotation: for any substitution σ instantiating each antiquoted variable in S with some SEXP, we should have that

$$\texttt{R\_Parse}(S)\sigma = E\sigma.$$

That is, the abstract syntax tree (AST) constructed at runtime (right hand side) should be equivalent to that returned by `R_parse()` at compile time (left hand side). The easiest way to ensure this property is to simply use the R parser itself to identify what AST we need to build. Fortunately, R does export its parser as a standalone function, making this possible. Note that we only call the parser at compile time — reconstructing the AST at runtime programmatically is much faster than parsing.

The upside of reusing R's parser when expanding quasiquotes is that we get support for all of R's syntax, for free, and avoid a potential source of bugs. The flipside is that we cannot reliably extend R's syntax with meta-syntactic constructs for antiquotation. We must fit within R's existing syntax. It is for this reason that antiquotation does not have a dedicated syntax, but instead usurps the syntax of regular R variables.

3. Special topics

3.1 A native view of foreign values

Programming across two languages typically involves a tradeoff: one can try shipping off an entire dataset and invoking a foreign function that does all the processing in one go, or keep as much of the logic in the host language and only call into foreign functions punctually. For example, mapping an R function `frobnicate()` over a list of elements might be done entirely in R, on the whole list at once,

$$\texttt{H}\rangle\ \ ys \leftarrow [\![\texttt{r|}\ \texttt{mapply(frobnicate, xs\_hs)}\ ]\!]$$

or elementwise, driven from Haskell,

$$\texttt{H}\rangle\ \ ys \leftarrow \mathsf{mapM}\ (\lambda x.\ [\![\texttt{r|}\ \texttt{frobnicate(x\_hs)}\ ]\!])\ xs$$

The latter style is often desirable — the more code can be kept in Haskell the safer, because more code can be type checked statically.

The bane of language interoperability is the perceived cost of crossing the border between one language to another during execution. Any significant overhead incurred in passing arguments to a foreign function and transferring control to it discourages tightly integrated programs where foreign functions are called frequently, such as in the last line above. Much of this cost is due to marshalling values from the native representation of data, to the foreign representation, and back.

By default, and in order to avoid having to pay marshalling and unmarshalling costs for each argument every time one invokes an internal R function, we represent R values in exactly the same way R does, as a pointer to a SEXPREC structure (defined in `R/Rinternals.h`). This choice has a downside, however: Haskell's pattern matching facilities are not immediately available, since only algebraic datatypes can be pattern matched.

HExp is R's SEXP (or `*SEXPREC`) structure represented as a (generalized) algebraic datatype. Each `SEXPREC` comes with a "type" tag the uniquely identifies the layout (one of `primsxp_struct`, `symsxp_struct`, *etc.* as seen in Section 2.1). See Figure 3 for an excerpt of the R documentation enumerating all possible type tags[2]. A simplified definition of HExp would go along the lines of Figure 4. Notice that for each tag in Figure 3, there is a corresponding constructor in Figure 4.

For the sake of efficiency, we do not use HExp as the basic datatype that all H generated code expects. That is, we do not use HExp as the universe of R expressions, merely as a *view*. We introduce the following *view function* to locally convert to a HExp, given a SEXP from R.

$$\mathsf{hexp} :: \mathsf{SEXP} \to \mathsf{HExp}$$

The fact that this conversion is local is crucial for good performance of the translated code. It means that conversion happens at each use site, and happens against values with a statically known form. Thus we expect that the view function can usually be inlined, and the

[1] This definition does not guarantee that R actions will only be executed when the associated R interpreter instance is extant. See Section 3.4 for a fix.

[2] In R 3.1.0, there are 23 possible tags.

NILSXP There is only one object of type `NILSXP`, `R_NilValue`, with no data.

SYMSXP Pointers to the `PRINTNAME` (a `CHARSXP`), `SYMVALUE` and `INTERNAL`. (If the symbol's value is a `.Internal` function, the last is a pointer to the appropriate `SEXPREC`.) Many symbols have the symbol value set to `R_UnboundValue`.

LISTSXP Pointers to the `CAR`, `CDR` (usually a `LISTSXP` or `NILSXP`) and `TAG` (a `SYMSXP` or `NILSXP`).

CHARSXP `LENGTH`, `TRUELENGTH` followed by a block of bytes (allowing for the nul terminator).

REALSXP `LENGTH`, `TRUELENGTH` followed by a block of C doubles.

...

Figure 3. Extract from the R documentation enumerating all the different forms that values can take.

short-lived HExp values that it creates is compiled away by code simplification rules applied by GHC. Notice how HExp as defined in Figure 4 is a *shallow view* — the fields of each constructor are untranslated SEXP's, not HExp's. In other words, a HExp value corresponds to the one-level unfolding of a SEXP as an algebraic datatype. The fact that HExp is not a recursive datatype is crucial for performance. It means that the hexp view function can be defined non-recursively, and hence is a candidate for inlining[3].

In this manner, we get the convenience of pattern matching that comes with a *bona fide* algebraic datatype, but without paying the penalty of allocating long-lived data structures that need to be converted to and from R internals every time we invoke internal R functions or C extension functions.

Using an algebraic datatype for viewing R internal functions further has the advantage that invariants about these structures can readily be checked and enforced, including invariants that R itself does not check for (e.g. that types that are special forms of the list type really do have the right number of elements). The algebraic type statically guarantees that no ill-formed type will ever be constructed on the Haskell side and passed to R.

We also define an inverse of the view function:

```
unhexp :: HExp → SEXP
```

3.2 "Types" for R

3.2.1 Of types, classes or forms

Haskell is a statically typed language, whereas R is a dynamically typed language. However, this apparent mismatch does not cause any particular problem in practice. This is because the distinction between "statically typed" languages and "dynamically typed" languages is largely artificial, stemming from the conflation of two distinct concepts: that of a *class* and that of a *type* [5].

The prototypical example of a type with multiple classes of values is that of complex numbers. There is *one* type of complex numbers, but *two* (interchangeable) classes of complex numbers: those in rectangular coordinates and those in polar coordinates. Both classes represent values of the same type. Harper further points out:

> Crucially, the distinction between the two classes of complex number is dynamic in that a given computation may result in a number of either class, according to convenience or convention. A program may test whether a complex number is in polar or rectangular form, and we can form data structures such as sets of complex numbers in which individual elements can be of either form.

[3] The GHC optimizer never inlines recursive functions.

```
data HExp
  = Nil                              -- NILSXP
  | Symbol SEXP SEXP SEXP            -- SYMSXP
  | List SEXP SEXP SEXP              -- LISTSXP
  | Char Int32 (Vector Word8)        -- CHARSXP
  | Real Int32 (Vector Double)       -- REALSXP
  | ...
```

Figure 4. Untyped HExp view.

Hence what R calls "types" are better thought of as "classes" in the above sense. They correspond to *variants* (or *constructors*) of a single type in the Haskell sense. R is really a unityped language.

We call the type of all the classes that exist in R the *universe* (See Section 3.1). Each variant of the union field in the `SEXPREC` structure defined in Section 2.1 corresponds to a class in the above sense. The `SEXPREC` structure *is* the universe.

Because "class" is already an overloaded term in both R and in Haskell, in the following we use the term *form* to refer to what the above calls a "class".

Some R functions expect a large number of arguments. It is not always clear what the usage of those functions is. It is all too easy to pass a value of the wrong form as an argument, or provide too many arguments, or too few. R itself cannot detect such conditions until runtime, nor is it practical to create a static analysis for R to detect them earlier, given the permissive semantics of the language. However, some information about the expected forms for arguments is given in R's documentation for practically every function in the standard library. It is often useful to encode that information from the documentation in machine checkable form, in such a way that the Haskell compiler can bring to bear its own existing static analyses to check for mismatches between formal parameters and actual arguments.

3.2.2 Form indexed values

To this end, in H we have *form indexed* SEXPs. The actual definition of a SEXP in H is:

```
newtype SEXP s (a :: SEXPTYPE)
  = SEXP (PTR SEXPREC)
```

The a parameter refers to the form of a SEXP (See Section 3.4 for the meaning of the s type parameter). In this way, a SEXP of form `REALSXP` (meaning a vector of reals), can be ascribed the type SEXP s R.Real, distinct from SEXP s R.Closure, the type of closures. These types are all of kind SEXPTYPE, a datatype promoted to a kind[4]. Each inhabitant of the SEXPTYPE kind corresponds to a type tag as seen in Figure 3.

When some given function is used frequently throughout the code, it is sometimes useful to introduce a wrapper for it in Haskell, ascribing to it a particular type. For example, the function that parses source files can be written as[5]:

```
import qualified Foreign.R.Type as R

parse :: SEXP s 'R.String    -- Filename of source
      → SEXP s 'R.Int        -- Number of expressions to parse
      → SEXP s 'R.String     -- Source text
```

[4] Using GHC's `-XDataKind` language extension.

[5] The ′ is an artifact of the `-XDataKinds` extension, useful for disambiguation. It is not strictly necessary, but it is good practice to use it.

```
    → R s (SEXP s 'R.Expr)
parse file n txt = [r| parse(file_hs, n_hs, txt_hs) ]
```

Now that we have a Haskell function for parsing R source files, with a Haskell type signature, the Haskell compiler can check that all calls to `parse()` are well-formed. We found this feature immensely useful to document in the source code itself how to call various R functions, without having to constantly look up this information in the R documentation.

Of course, while form indexing SEXP can in practice be a useful enough surrogate for a real type system, it does not replace a real type system. A reasonable property for any adequate type system is *type preservation*, also called *subject reduction*. That is, we ought to have that:

$$\text{If } \Gamma \vdash M : T \text{ and } M \Downarrow V \text{ then } \Gamma \vdash V : T$$

where M is an expression, T is a type and V is the value of M. The crude form of indexing presented here does not enjoy this property. In particular, given some arbitrary expression, in general the form of the value of this expression is unknown. We have the following type of SEXP's of unknown form:

```
data SomeSEXP s = ∀a.SomeSEXP (SEXP s a)
```

Because the form of a value is in general unknown, the type of eval is:

```
eval :: SEXP s a → R s (SomeSEXP s)
```

That is, for any SEXP of any form a, the result is a SEXP of some (unknown) form.

3.2.3 Casts and coercions

SEXP's of unknown form aren't terribly useful. For example, they cannot be passed as-is to the successor function on integers, defined as:

```
succ :: SEXP s 'R.Int → R s SomeSEXP
succ x = [r| x_hs + 1 ]
```

Therefore, H provides *casting functions*, which introduce a dynamic form check. The user is allowed to *coerce* the type in Haskell of a SEXP given that the dynamic check passes. cast is defined as:

```
cast :: SSEXPTYPE a → SomeSEXP s → SEXP s a
cast ty s
    | fromSing ty ≡ R.typeOf s = unsafeCoerce s
    | otherwise = error "cast: Dynamic type cast failed."
```

where SSEXPTYPE is a singleton type reflecting at the type level the value of the first argument of cast. The use of a singleton type here allows us to to write a precise specification for cast: that the type of the return value is not just any type, but uniquely determined by the value of the first argument ty.

Now, `[r| 1 + 1 ]` stands for the *value* of the R expression "`1 + 1`". That is,

```
two = [r| 1 + 1 ] :: SomeSEXP s
```

In order to compute the successor of two, we need to cast the result:

```
three :: R s SomeSEXP
three = succ (two `cast` R.Int)
```

3.3 R values are (usually) vectors

An idiosyncratic feature of R is that scalars and vectors are treated uniformly, and in fact *represented* uniformly. This means that provided an interface to manipulate vectors alone, we can handle all scalars as well as all vectors. H exports a library of vector manipulation routines, that mirrors the API of the standard `vector` package. The advantage of keeping data represented as R vectors throughout a program is that no marshalling or unmarshalling costs need be incurred when passing the data to an R function. Because we provide the exact same API as for any other (non-R) vector representations, it is just as easy to manipulate R vectors instead, throughout.

3.4 Memory management

One tricky aspect of bridging two languages with automatic memory management such as R and Haskell is that we must be careful that the garbage collectors (GC) of both languages see eye-to-eye. The embedded R instance manages objects in its own heap, separate from the heap that the GHC runtime manages. However, objects from one heap can reference objects in the other heap and the other way around. This can make garbage collection unsafe because neither GC has a global view of the object graph, only a partial view corresponding to the objects in the heaps of each GC.

3.4.1 Memory protection

Fortunately, R provides a mechanism to "protect" objects from garbage collection until they are unprotected. We can use this mechanism to prevent R's GC from deallocating objects that are still referenced by at least one object in the Haskell heap.

One particular difficulty with protection is that one must not forget to unprotect objects that have been protected, in order to avoid memory leaks. H uses "regions" for pinning an object in memory and guaranteeing unprotection when the control flow exits a region.

3.4.2 Memory regions

There is currently one global region for R values, but in the future H will have support for multiple (nested) regions. A region is opened with the runRegion action, which creates a new region and executes the given action in the scope of that region. All allocation of R values during the course of the execution of the given action will happen within this new region. All such values will remain protected (i.e. pinned in memory) within the region. Once the action returns, all allocated R values are marked as deallocatable garbage all at once.

```
runRegion :: (∀s.R s a) → IO a
```

Regions are transactions, in that protection prevails during the transaction and ceases after transaction closure.

3.4.3 Automatic memory management

Nested regions work well as a memory management discipline for simple scenarios when the lifetime of an object can easily be made to fit within nested scopes. For more complex scenarios, it is often much easier to let memory be managed completely automatically, though at the cost of some memory overhead and performance penalty. H provides a mechanism to attach finalizers to R values. This mechanism piggybacks Haskell's GC to notify R's GC when it is safe to deallocate a value.

```
automatic :: MonadR m ⇒ R.SEXP s a → m (R.SEXP G a)
```

In this way, values may be deallocated far earlier than reaching the end of a region: As soon as Haskell's GC recognizes a value to no longer be reachable, and if the R GC agrees, the value is prone to be deallocated. Because automatic values have a lifetime independent of the scope of the current region, they are tagged with the global region G (a type synonym for GlobalRegion).

For example:

```
do x ← [r| 1:1000 ]
   y ← [r| 2 ]
   return $ automatic [r| x_hs * y_hs ]
```

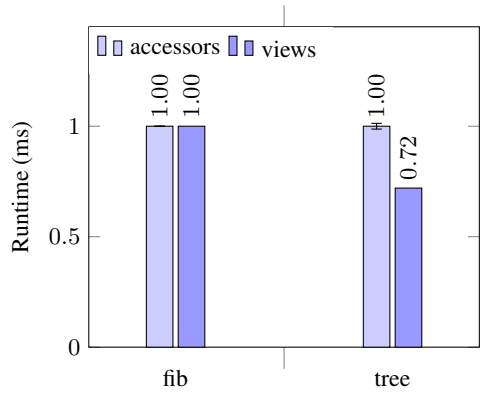


Figure 5. Normalized comparison of runtimes between views and accessor functions.

Automatic values can be mixed freely with other values.

4. Benchmarks

As explained in Section 3.1, there are essentially two ways to deconstruct a SEXP value returned from R: using accessor function provided by R's C extension API, or construct a view of the SEXP as an algebraic datatype and perform case analysis on that. The latter approach is more convenient, but potentially slower, because it nominally implies allocating a view data structure. However, as argued in Section 3.1, through careful engineering of the view, we expect the compiler to optimize away any extra allocation. In this section, we test this hypothesis experimentally.

To compare the performance cost of using a view function, we implement two micro benchmarks. The `fib` benchmark measures the performance of a naive implementation in Haskell of the Fibonacci series over R integers. Likewise, the `tree` benchmark concerns a binary tree traversal function implemented in Haskell, where the tree is represented using generic R vectors. The results[6] are provided in Figure 4. They were obtained using the Criterion benchmarking tool [10] using the default settings on a lightly loaded machine. Where the standard deviations are significant ($\not< 1\%$), we indicate them with error bars.

As expected, using a view does not significantly impact performance. In fact, as can be seen in the `tree` benchmark, views can even be faster than using accessor functions. The reason is that a view function can be completely inlined, yielding code that makes direct memory accesses to statically known offsets in memory, given a pointer to a SEXPREC structure. Accessors, on the other hand, are C functions that are opaque to the compiler — they cannot be inlined, meaning that the overhead of calling a C function, using the standard calling convention, must be incurred at each field access.

5. Related Work

TODO

6. Conclusion

Given modern extensions to the Haskell programming language, the language turns out to support surprisingly easy interoperability with other languages, with opt-in static guarantees on what arguments get passed to a function, but without the hassle of elaborate bindings. The enabling ingredient here is the existence of a quasiquotation mechanism, which makes it easy to express calls to foreign language functions in the foreign language itself, in a way that matches up the foreign documentation for that function. Many of the ideas in this paper can be transposed to any other functional language provided an equivalent quasiquotation mechanism exists (e.g. Camlp4 [8] or extension points in OCaml, or SML quotation [11]). In a multistage language but without quasiquotation, the strategy for avoiding marshalling costs would still apply, but constructing foreign calls would likely turn out rather less convenient.

Interacting with a latently typed language necessarily induces some number of runtime checks. While foreign functions can be selectively typed in the host language, this in practice requires a pervasive use of casts and coercions. Casts are runtime checks that have a cost, but moreover a cast failure is not very informative as to why it happened. A future direction could include integrating a notion of *blame* [3] into H, in order to better pinpoint the true cause of a dynamic check failure.

[6] Benchmarks available at `http://github.com/tweag/H`.

References

[1] M. M. Chakravarty. C ⟶ HASKELL, or yet another interfacing tool. In *Implementation of Functional Languages*, pages 131–148. Springer, 2000.

[2] M. M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. P. Jones, A. Reid, et al. The Haskell 98 foreign function interface 1.0. *An Addendum to the Haskell Report*.

[3] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, volume 37, pages 48–59. ACM, 2002.

[4] GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.3.

[5] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[6] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *ACM Sigplan Notices*, volume 44, pages 1–12. ACM, 2008.

[7] G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[8] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ml. In *Proc. 1994 Workshop on ML and its applications*, pages 70–78, 1994.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. The Definition of Standard ML, revised edition. *MIT Press*, 1(2):2–3, 1997.

[10] B. O'Sullivan. Criterion, 2014. URL `http://www.serpentine.com/criterion/`.

[11] K. Slind. Object language embedding in standard ml of new jersey. In *Proceedings of the Second ML Workshop, CMU SCS Technical Report. Carnegie Mellon University, Pittsburgh, Pennsylvania*, 1991.

Type-Directed Elaboration of Quasiquotations

A High-Level Syntax for Low-Level Reflection

David Raymond Christiansen
IT University of Copenhagen
drc@itu.dk

Abstract

Idris's reflection features allow Idris metaprograms to manipulate a representation of Idris's core language as a datatype, but these reflected terms were designed for ease of type checking and are therefore exceedingly verbose and tedious to work with. A simpler notation would make these programs both easier to read and easier to write. We describe a variation of quasiquotation that uses the language's compiler to translate high-level programs with holes into their corresponding reflected representation, both in pattern-matching and expression contexts. This provides a notation for reflected language that matches the notation used to write programs, allowing readable metaprograms.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords quasiquotation; proof automation; metaprogramming

1. Introduction

Idris [3] is a programming language with dependent types in the tradition of Agda [16] and Cayenne [1]. An important design goal for the Idris team is to enable the construction of embedded languages that can make strong guarantees about the safety of the programs written in them, rather than requiring users of these embedded languages to write proofs themselves. If this goal is to succeed, Idris will require good tools that library authors can use to automate the construction of proofs.

One such tool is *reflection*, in which an Idris program can construct a proof object by inspecting the AST of a the goal type and generating an AST for the proof term. This allows developers of proof automation to write functions that might otherwise be difficult, because eliminating types through pattern matching is unsound.

Idris has a well-defined core language, called TT, and all constructions in the high-level Idris language are given their semantics by defining their translation to TT. This translation process is referred to as *elaboration*. Terms in TT are exceedingly verbose: every binder has a fully explicit type annotation, every name is fully-qualified, and there are no implicit arguments. This verbosity drastically simplifies type checking. The intention is that one need only trust this simple type checker to be able to trust the rest of the system.

Because Idris reflection works directly with TT terms, it can quickly become overwhelming. Furthermore, the correspondence between high-level Idris terms and their corresponding TT terms are not always obvious to non-expert users of the language. What appears to be a simple function application at the level of Idris code might turn out to have very complicated type-level structure or non-trivial implicit arguments. In the normal course of programming, it is good to hide this complexity and allow the user to focus on her or his programming task, rather than being overwhelmed by minutiae. However, when writing metaprograms using reflection, this becomes an unfortunate trade-off, because the core language can be difficult to connect to user-visible terms. This difficulty is especially unpleasant when using Idris's *error reflection* [5], in which Idris code can be used to rewrite the compiler's error messages before they are presented to the user. In fact, it was practical experience with error reflection that motivated the work described in this paper.

We augment the high-level Idris language with *quasiquotations*, in which the Idris elaborator is invoked to transform high-level Idris into *reflected* TT terms using the same translation that produces TT terms for the type checker. Within quasiquoted terms, *antiquotations* allow other reflected terms to be spliced into the quotation. In a pattern context, antiquotations become patterns to be matched by the reflected term at the corresponding position. These quasiquotations allow the best of both worlds: high-level syntax for the uninteresting parts, with details filled in by type-directed elaboration, along with control over the details of term construction when and if it matters.

Contribution

The contributions of this paper are:

- a novel adaptation of quasiquotations to the context of dependently-typed programming with reflection that allows the use of high-level language syntax to construct and manipulate the corresponding terms in a core language;
- a description of an implementation technique for these quasiquotations in Idris, as an extension of the type-driven elaboration described in Brady's 2013 paper [3]; and
- demonstrations of the utility of these quasiquotations for proof automation and error message rewriting.

Furthermore, this paper can serve as a demonstration of how to extend a tactic-based elaborator to support a new high-level language feature.

2. Related Work

2.1 A Brief History of Quasiquotation

The notion of quasiquotation was invented by Quine in his 1940 book *Mathematical Logic* [12, pp. 33–37]. While ordinary quotations allow one to *mention* a phrase rather than *using* it, quasiquotations allow these quoted expressions to contain variables that stand for other expressions, just as mathematical expressions can contain variables that stand for values. In other words, a specific class of subexpression is treated as a use within a context that is mentioned. Quine used Greek letters to represent variables in quasiquotations.

The paradigmatic instance of quasiquotation in programming languages is that found in the Lisp family. Bawden's 1999 paper [2] summarizes the history and semantics of the quasiquotation mechanism found in both the Scheme family of languages and in Common Lisp. In the Lisp family, program code is represented in a uniform manner, using lists that contain either atomic data, such as symbols, strings, and numbers, or further lists. In Lisp parlance, these structures are referred to as "S-expressions". Because S-expressions are simply ordinary data, it makes sense to quote them, yielding a structure that can easily be manipulated. Additionally, most Lisps have a quasiquotation system, in which specially marked subexpressions of a quotation are evaluated, with the result substituted into the quotation. Unlike Quine's quasiquotation, the Lisp family of languages allow arbitrary expressions to be inserted into quasiquotations.

Languages outside of the Lisp family have also used quasiquotation to implement language extension. Because these languages' syntaxes do not have the regular format of Lisp S-expressions. The Camlp4 system [6] provides quasiquotation for the OCaml language, among other extensions. In Camlp4, quasiquotations consist of arbitrary strings that are transformed by a *quotation expander* to either a string representing valid concrete syntax or to an abstract syntax tree. These quotations support antiquotation, which invokes the parser to read an OCaml expression or pattern inside of the quotation. Template Haskell's quasiquotations [9] work on similar principles. Both systems fully expand all quotations at compile time, and both check that the generated code is well-typed.

The MetaML family of metaprogramming facilities [15], including MetaOCaml[17] and F# [14], implement a style of quotation in which the type of quoted expressions is parameterized over the type that would be inhabited by the the quoted expression if it were reified. These features are intended for use in staged computation. In addition to representing the types of the quoted expressions, these staging annotations feature static scope, so a quotation that contains a name contains the version of that name from the scope in which the quotation was generated.

Scala quasiquotations [13] are very much like Lisp quasiquotations. While their syntax resembles that of strings, this is a consequence of their implementation using Scala's string interpolators and they are in fact expanded to ASTs at compile time. The quasiquotations were initially intended to serve as an implementation technique for Scala macros [4], but they are also useful for both runtime code generation as well as generating program text. Scala macros closely resemble Lisp macros, in that they do not intend to allow arbitrary strings to be used as syntax, but instead implement transformations from one valid parse tree to another. Unlike Lisp, Scala programs that contain macros are type checked after macro expansion, and they are represented by a conventional AST that macros manipulate. Quasiquotations are a means of constructing and destructuring these trees using the syntax of the high-level Scala language.

Like Scala, C# is an object-oriented language with a notion of quotation [11]. In C#, quotation can be applied to an anonymous function by annotating it with the `Expression` type, which causes a datatype representing the function's AST to be generated instead of the function itself. However, this feature cannot properly be considered quasiquotation, as there is no mechanism for escaping the quotation and inserting a sub-tree that has been generated elsewhere.

2.2 Reflection, Proof Automation, and Tactic Languages

The ML language was originally developed as a metalanguage for the Edinburgh LCF proof assistant [8]. In fact, this is where the name ML is derived from. An abstract datatype was used to represent rules of inference in the underlying logic, and ML functions could then be used to construct these proofs. Higher-order functions could then be used to represent strategies for combining these functions. ML served as an expressive language for automating the construction of proofs.

Agda [16] has a notion of *reflection*, described by van der Walt and Swierstra. Agda reflection is a form a compile-time metaprogramming, where quoted terms are used to construct proof terms that are then reified and type checked at compile time. These terms are constructed through direct manipulation of the term AST, which is a simple untyped lambda calculus. Agda metaprograms can get access to reflected representations of the type that is expected at a particular source location as well as its lexical environment, and they can then use this information to construct a term matching the expected type. However, users of reflection in Agda must program with a notation matching the reflected term datatype, rather than with ordinary Agda syntax.

Coq is perhaps the best-known system that is designed to facilitate automating the construction of proofs. Early versions of Coq required that users extend the built-in collection of tactics using OCaml. LTac [7] is a domain-specific language for writing new tactics that works at a higher level of abstraction that OCaml. It provides facilities for pattern matching the syntax of arbitrary terms from Coq's term language Gallina, without these terms having been reduced to applications of constructors. Likewise, it can instantiate lower-level tactics and tacticals, which may contain Gallina terms, using portions of syntax extracted from the matched goals. Thus, LTac pattern matching can be considered a form of quasiquotation.

More recently, Ziliani et al. developed the MTac tactic language [19]. Like Agda's reflection mechanism and unlike LTac, MTac is implemented in Coq's term language, rather than being an external language. However, unlike Agda's reflection, MTac tactics use Coq's type system to classify the terms produced by tactics, and the type system can therefore catch errors in tactics. Due to the elimination restrictions and impredicativity of the `Prop` universe, one can pattern match over the structure of arbitrary terms in MTac, rather than just terms in canonical form. MTac required only minimal extensions to Coq, namely a primitive to run MTac tactics.

3. Reflection in Idris

Idris's reflection system is very similar to that of Agda. Elements of a datatype representing terms in a lambda calculus can be generated from the compiler's internal representation of `TT`, after which Idris programs can manipulate them or use them as input to procedures that generate new reflected terms. In addition to generating new terms, Idris allows the generation of tactic scripts through reflection, by providing a collection of base tactics as a datatype along with a primitive tactic that allows functions from an environment and a goal to a reflected tactic to be used as tactics themselves. Naturally, the tactic that applies an Idris function as a tactic is itself reflected.

Unlike Agda, the terms that are available through Idris's reflection mechanism are fully annotated with their types. Additionally, they include features of a development calculus in the style of McBride's OLEG [10], including special binding forms for holes

and guesses. This representation is more complicated and more accurate than Agda's, as it maintains typing information.

4. Idris Quasiquotations

TT is a minimalist dependently-typed λ-calculus with inductive-recursive families of types and operators defined by pattern matching. The full details of TT are available in Brady's 2013 article [3].

Our quasiquotations extend the Idris$^-$ language, which is a version of Idris in which purely syntactic transformations such as the translation of do-notation and idiom brackets to their underlying functions have been performed and user-defined syntax extensions have been expanded. We extend the expression language with three new productions:

$e, t ::=$ ...
- | `‘(` e `)` (quasiquotation of e)
- | `‘(` e `:` t `)` (quasiquotation of e with type t)
- | `~` e (antiquotation of e)

The parts of a term between a quotation but not within an antiquotation are said to be *quoted*. Antiquotations that are not quoted, and quoted quasiquotations, are static errors. The quoted regions of a term are elaborated in the same way as any other Idris expression. However, instead of being used directly, the elaborated TT terms are first reflected. Non-quoted regions are elaborated directly into reflected terms, which are inserted as usual.

Names occurring in the quoted portion of a term do not obey the typical lexical scoping rules of names in Idris. This is because quoted terms are intended to be used in places other than where they are constructed, and their reification site may have completely different bindings for the same names. Therefore, all names in the quoted portion are taken to refer to the global scope. Because antiquotations are ordinary terms, they obey the ordinary scoping rules of the language.

Idris supports type-driven disambiguation of overloaded names. This feature is used for everything from literal syntax for number- and list-like structures to providing consistent naming across related libraries. This is also used to allow "punning" between some types and their constructors. For instance, `()` represents both the unit type and its constructor in Idris, and `(Int, String)` can represent either a pair type or a pair of types. In ordinary Idris programs, all top-level definitions are required to have type annotations, so type information is available to aid in disambiguation. In quasiquoted terms, however, this information is not available. Thus, the second variant of quasiquotation above allows a goal type to be provided. Like quoted terms, it is elaborated in the global environment. Because the goal type does not occur in the final reflected term and simply exists as a shorthand to avoid explicitly annotating names, goal types may not contain antiquotations.

A particularly instructive example that demonstrates the need for goal types is the unit and product types. Following Haskell, Idris uses a "pun" on its notation for products. Both the unit type and its constructor are written `()`, and `(2, "two")` is an inhabitant of `(Int, String)`. Additionally, because a pair of types is a perfectly valid construct in a dependently-typed system, `(Int, String)` could represent either a pair of types or a pair type — respectively, a member of `(Type, Type)` or member of `Type`. In the context of a quasiquotation, defaulting rules would need to be used to disambiguate `()` and `(Int, String)`, and whichever version was not the default would become very difficult to use. The distinction between `‘(() : ())` and `‘(() : Type)`, however, is easy to see and easy to remember.

5. Elaboration

The Idris elaborator, described in detail in Brady's 2013 paper [3], uses proof tactics to translate desugared Idris to the core type theory TT. A full presentation of this process is far outside the scope of this paper; however, enough details are repeated to make the elaboration of quasiquotes understandable.

5.1 The Elaboration Monad

The Idris elaborator is built on top of a library for manipulating terms in type theory. The elaborator is defined inside of a monad with state that consists of a hole queue, a focused hole or guess, and a collection of unsolved unification problems. The hole queue contains goals that are yet to be solved - at the beginning of elaboration, it will contain a single hole, but later operations can introduce new holes. The focused hole represents the current goal. Additionally, the elaboration monad contains errors and error handling.

A number of meta-operations, or *tactics*, are defined in the elaboration monad. These tactics resemble the built-in proof tactics of a system like Coq. In this paper, we use the following subset of Brady's [3] meta-operations:

- CHECK, which type checks a complete term;
- CLAIM, which introduces a new hole with a particular type, placing it at the rear of the hole queue;
- GET, which binds the proof state to a variable;
- FILL, which adds a guess for the focused hole, solving the imposed unification constraints;
- NEWPROOF, which obliterates the proof state and establishes a new goal;
- PUT, which replaces the proof state with a new one;
- SOLVE, which causes a guess to be substituted for its hole;
- TERM, which returns the current term;
- UNFOCUS, which moves the focused hole to the end of the hole queue;

As a notational convention, we follow Brady [3] in letting the notation for names in the meta-language and names in the object language coincide, deferring to the reader to see which is being used. Names that occur in both contexts are metalanguage names referring to coinciding object language names. Additionally, unbound variables are taken to be fresh. When operations and their arguments occur under an arrow (e.g. $\text{CLAIM}\ \vec{a}$), it means that the operation is repeated on all the arguments in the sequence. This is similar to `mapM` in Haskell.

The meta-operations $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$, which run relative to a proof state, respectively elaborate expressions and patterns. These operations usually coincide; however, they treat unresolved free variables differently. Following $\mathcal{E}[\![\cdot]\!]$, unresolved holes or variables trigger an error, while unresolved names in patterns (that is, following $\mathcal{P}[\![\cdot]\!]$) are bound to pattern variables. Otherwise, constructors with implicit arguments (such as the length argument to the `(::)` case of `Vect`) would not be able to be pattern-matched.

Elaboration is type-directed, in the sense that the elaborator always has a goal type available and can make decisions based on this fact. However, sometimes the type will be either unknown or partially known. In these cases, unification constraints imposed by the elaboration of the term can cause the type to be solved.

In addition to the meta-operations described by Brady [3], we define four additional operations:

- ANYTHING, which introduces a hole whose type must be inferred;
- EXTRACTANTIQUOTES, which replaces antiquotations in a quasiquoted Idris$^-$ term with references to fresh names, returning the modified term and the mapping from these fresh names to their corresponding antiquotation terms;

- REFLECT, which returns a term corresponding to the reflection of its argument; and
- REFLECTP, which returns a pattern corresponding to the reflection of its argument.

The operation ANYTHING n can be defined as follows:

$$\begin{aligned}\text{ANYTHING } n \;=\; &\underline{\text{do}}\ \text{CLAIM } (n' \;:\; \texttt{Type})\\ &\quad\ \text{CLAIM } (n \;:\; n')\end{aligned}$$

This represents type inference because it hides the fresh name n' that is introduced for the type of n. Thus, the type must be later solved through unification with other elaborated terms. EXTRACTANTIQUOTES is a straightforward traversal of an Idris$^-$ term, replacing antiquotations with variables and accumulating a mapping from these fresh variables to the corresponding replaced subterms. The names alone are accessed by the operation $names$. REFLECT and REFLECTP each take a term and a collection of names of antiquotations (see Section 5.2) and return a quoted version of the term. Antiquotation names, however, are not quoted. Additionally, REFLECTP inserts universal patterns in certain cases — see Section 5.4

5.2 Elaborating Quasiquotations

We implement quasiquotations by extending the elaboration procedures for expressions and patterns, respectively $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$. Elaborating the quoted term proceeds through five steps, each of which is described in detail below:

1. Replace all antiquotations by fresh variables, keeping track of the antiquoted terms and their assigned names
2. Elaborate the resulting term in a fresh proof state, to avoid variable capture
3. If RHS, quote the term, leaving antiquotation variables free
4. If LHS, quote with strategically placed universal patterns for things like unused names
5. Restore local environment and elaborate antiquotations

Replace antiquotations We replace antiquotations with fresh variables because they will need to be treated differently than the rest of the term. Additionally, the expected types of the antiquotations must be inferable from the context in which they are found, because the quotations that will fill them provide no type information. We remember the association between the antiquoted terms and the names that they were replaced by so that the result of elaborating them can later be inserted.

Elaborate in a fresh proof state Quotations can occur in any Idris expression. However, names that are defined in quotations are resolved in the global scope, for reasons discussed in Section 4. Because the scopes of local variables are propagated using hole contexts in the proof state, it is sufficient to elaborate the quoted term in a fresh state. The replacement of antiquotations with references to fresh names means that there is no risk of elaborating the contents of the antiquotations too early. However, when the elaborator reaches these names, it will fail, because they are unknown. To fix this problem, we first use the ANYTHING meta-operation that was defined above to introduce holes for both these names and their types. Because this stage of elaboration occurs in term mode, rather than pattern mode, the elaboration will fail if the holes containing types don't get solved through unification.

Quote the term Quotation is the first step that differs between terms and patterns. In both cases, the term resulting from elaboration is quoted, with the names that were assigned to antiquotations left unquoted. However, if the term being elaborated is a pattern, then some aspects of the term are not quoted faithfully. See Section 5.4 for more information.

$$\begin{aligned}\mathcal{E}[\![\texttt{`(}\ e\ \texttt{)}]\!] \;=\; &\underline{\text{do}}\ (e', \vec{a}) \leftarrow \text{EXTRACTANTIQUOTES } e && (1)\\ &\quad st \leftarrow \text{GET} && (2)\\ &\quad \text{NEWPROOF } T\\ &\quad \text{CLAIM } (T \;:\; \texttt{Type})\\ &\quad \vec{\text{ANYTHING}}\ (names\ \vec{a})\\ &\quad \mathcal{E}[\![e']\!]\\ &\quad qt \leftarrow \text{TERM}\\ &\quad \text{CHECK } qt\\ &\quad \text{PUT } st\\ &\quad \vec{\text{CLAIM}}\ (names\ \vec{a} \;:\; \vec{\texttt{Term}}) && (3)\\ &\quad r \leftarrow \text{REFLECT } qt\ \vec{a}\\ &\quad \text{FILL } r\\ &\quad \text{SOLVE}\\ &\quad \vec{\text{ELABANTIQUOTE}}\ \vec{a} && (5)\end{aligned}$$

Figure 1. Elaboration of quasiquotations

Elaborate the antiquotations The quoted term from the previous step is ready to be spliced into the original hole. What remains is to solve the variables introduced for antiquotations in the previous step. This is done by first introducing each name as a hole expecting a quoted term, and then elaborating them straightforwardly into their respective holes.

Figure 1 describes this elaboration procedure in Brady's notation. The individual tactics that correspond to each of the steps 1–5 above are numbered. Antiquotations are replaced in the first line of the tactic script, using the previously-described operation EXTRACTANTIQUOTES (1). Then, the ordinary state monad operations GET and PUT are used to save and restore the original proof state. The region (2) bracketed by these operations corresponds to step 2 above — namely, elaboration of the quoted term in the global context, which is achieved using a fresh proof state introduced by NEWPROOF. Initially, the goal of the new proof is an unbound variable, but this variable is then bound as a hole expecting a type using the CLAIM meta-operation. The quoted term is provided with hole bindings for each of the fresh antiquotation names by the ANYTHING meta-operation. Then, the quoted term is elaborated into the main hole. If this process is successful, it will result in the hole T being filled out with a concrete type as well. The result of elaboration is saved in the variable qt, and then type checked one final time with CHECK to ensure that no errors occurred.

After the original proof state is restored with PUT, the actual quoting must be performed and the antiquotations must be spliced into the result (3). Each antiquotation name is now established as a hole of type `Term`, the datatype representing reflected terms, because the elaborated form must be a quotation. Now that the holes for the antiquotations are established, it is possible to insert the reflected term into the initial hole. The operation REFLECT is invoked, which quotes the term, leaving references to the antiquotation variables intact as references to the just-introduce holes. This quoted term is then filled in as a guess, and SOLVE is used to dispatch the proof obligation.

Finally, the antiquotations can be elaborated (5). This is done by focusing on their holes and elaborating the corresponding term into that hole. In the above script, this is represented by the tactic ELABANTIQUOTE, which can be defined as follows:

$$\begin{aligned}\text{ELABANTIQUOTE } (n, t) \;=\; &\underline{\text{do}}\ \text{FOCUS } n\\ &\quad\ \mathcal{E}[\![t]\!]\end{aligned}$$

$$
\begin{aligned}
&\mathcal{E}[\![\text{`(}\ e\ :\ t\ \text{)}]\!] = \underline{\text{do}} \\
&\qquad \vdots \\
&\qquad \textsc{Claim}\ (T\ :\ \texttt{Type}) \\
&\qquad \textsc{Focus}\ T \\
&\qquad \mathcal{E}[\![t]\!] \\
&\qquad \textsc{AnyThing}\ (\mathit{names}\ \vec{a}) \\
&\qquad \vdots
\end{aligned}
$$

Figure 2. Elaborating quasiquotations with goal types

A specific elaboration procedure for antiquotations is not necessary, because programs with antiquotations outside of quasiquotations are rejected prior to elaboration.

5.3 Elaborating Goal Types

Elaborating a quasiquotation with an explicit goal type is a straightforward extension of the procedure in the previous section. After introducing a hole for the type of the term that will be elaborated prior to the actual quotation, the goal type is elaborated into this hole. Because this is occurring immediately after the establishment of a fresh proof state, names in the goal type will be resolved in the global scope, as intended.

The formal procedure is largely identical, with only the small addition shown in Figure 2. Thus, the lines immediately before and immediately after are included to show where the additions have occurred. This seemingly-simple change has far-reaching effects, because type information is now available to the subsequent elaboration of e'. This type information can, for instance, enable implicit arguments to be solved due to unification constraints induced by the elaboration of t.

5.4 Elaborating Quasiquotation Patterns

Quasiquotations can also be used as patterns. Recall that the operation $\mathcal{P}[\![\cdot]\!]$ is a variation of $\mathcal{E}[\![\cdot]\!]$ that is used on the left-hand side of definitions in order to elaborate patterns. The primary difference is that $\mathcal{P}[\![\cdot]\!]$ does not fail when the elaborated term contains unknown variables. Instead, it inserts pattern variable bindings for these.

It is tempting, then, to simply use the pattern elaborator in the recursive elaboration clauses of the quasiquote elaboration procedures. However, this would not work. REFLECT would simply quote these new pattern variables, leading to terms that contain explicitly quoted fresh pattern variables. Pattern elaboration must instead invoke ordinary expression elaboration when generating the term to be quoted, but then use pattern elaboration for the antiquotations.

For practical reasons, pattern elaboration must use a specialized reflection procedure REFLECTP that introduces some universal patterns in strategic places. For example, ordinary non-dependent function types are represented in TT as dependent functions in which the bound name is not free in the type on the right hand side. These names are chosen by the compiler, and they are difficult to predict. Therefore, they are represented as universal patterns (_) rather than their names. Additionally, universe level variables and internal type and constructor tag values are replaced with universal patterns. There is no solid theoretical basis for the current selection of universal pattern addition heuristics. Rather, it is a result of experimentation with the system and writing practical programs.

Figure 3 demonstrates the formal procedure for elaboration of quasiquotation patterns. This procedure uses two variations on previously-seen meta-operations: REFLECTP, like REFLECT, is a traversal of the resulting tree structure that implements step 4 abvove, and ELABANTIQUOTEP is defined as follows:

$$
\begin{aligned}
&\mathcal{P}[\![\text{`(}\ e\ \text{)}]\!] = \underline{\text{do}}\ (e', \vec{a}) \leftarrow \textsc{ExtractAntiquotes}\ e && (1)\\
&\qquad st \leftarrow \textsc{Get} && (2)\\
&\qquad \textsc{NewProof}\ T \\
&\qquad \textsc{Claim}\ (T\ :\ \texttt{Type}) \\
&\qquad \textsc{AnyThing}\ (\mathit{names}\ \vec{a}) \\
&\qquad \mathcal{E}[\![e']\!] \\
&\qquad qt \leftarrow \textsc{Term} \\
&\qquad \textsc{Check}\ qt \\
&\qquad \textsc{Put}\ st \\
&\qquad \textsc{Claim}\ (\mathit{names}\ \vec{a}\ :\ \texttt{Term}) && (4)\\
&\qquad r \leftarrow \textsc{ReflectP}\ qt\ \vec{a} \\
&\qquad \textsc{Fill}\ r \\
&\qquad \textsc{Solve} \\
&\qquad \textsc{ElabAntiquoteP}\ \vec{a} && (5)
\end{aligned}
$$

Figure 3. Elaborating quasiquote patterns

$$
\begin{aligned}
&\textsc{ElabAntiquoteP}\ (n, t) = \underline{\text{do}}\ \textsc{Focus}\ n \\
&\qquad\qquad\qquad\qquad \mathcal{P}[\![t]\!]
\end{aligned}
$$

The modifications necessary to elaborate a quasiquotation pattern with a goal type are identical to the non-pattern case. In the real implementation, of course, quasiquote elaboration with or without goal types and in pattern mode or expression mode is handled by one code path, with conditionals expressing the four possibilities. They are presented as four separate procedures here for reasons of clarity.

6. Examples

This section demonstrates the usefulness of quasiquotations through a number of examples, showing how the high-level notation of Idris quasiquotation simplifies their expression and reduces the need for the user to comprehend all of the details of elaboration.

6.1 Custom Tactics

In Idris, a custom tactic is a function from a proof context and goal to a reflected tactic expression. Reflected tactics are represented by the `Tactic` datatype, which has constructors such as `Exact` for solving the goal with some proof term, `Refine` for applying a name to solve the goal, leaving holes for the remaining arguments, and `Skip` which does nothing, along with tactics such as `Seq` for sequential composition and `Try` to provide a fallback in case of errors. These tactics correspond to the elaborator tactics described in Section 5.

The native tactic `applyTactic` runs a custom tactic in the scope of the current proof. In other words, its argument should be an expression of type:

```
List (TTName, Binder TT) -> TT -> Tactic
```

This construction allows Idris to be its own metalanguage for purposes of proof automation.

6.1.1 Trivial Goals

When writing proofs, it may be the case that a particular goal is completely trivial. Either the goal type is one such as `()` or the equality type that have only a single constructor, or we have a premise available with precisely the type that we desire. Idris already has a built-in tactic to solve these kinds of goals, called `trivial`. However, this built-in tactic is not extensible with support for new trivial types.

Figure 4 demonstrates an implementation of a trivial tactic that uses our newly-introduced quasiquotations. The first case checks

```
triv : List (TTName, Binder TT) -> TT -> Tactic
triv ctxt `(() : Type) =
  Exact `(() : ())
triv ctxt `((=) {A=~A} {B=~B} ~x ~y) =
  Exact `(the ((=) {A=~A} {B=~B} ~x ~y) refl)
triv ((n, b)::ctxt) goal =
  if binderTy b == goal
    then Exact (P Bound n Erased)
    else triv ctxt goal
triv [] _ =
  Fail [TextPart "Decidedly nontrivial!"]
```

Figure 4. A tactic for trivial goals

```
rewrite_plusSuccRightSucc : TT -> Maybe Tactic
rewrite_plusSuccRightSucc `(plus ~n (S ~m)) =
  Just (Rewrite `(plusSuccRightSucc ~n ~m))
rewrite_plusSuccRightSucc _ = Nothing

rewrite_plusZeroRightNeutral : TT -> Maybe Tactic
rewrite_plusZeroRightNeutral `(plus ~n Z) =
  Just (Rewrite `(sym (plusZeroRightNeutral ~n)))
rewrite_plusZeroRightNeutral _ = Nothing
```

Figure 5. Rewriters for addition

whether the goal is the unit type. The goal annotation is necessary because of Idris's defaulting rules, which prioritize the unit constructor during disambiguation. The second case checks whether the goal is an identity type. The explicit provision of both `A` and `B` is necessary because Idris uses heterogeneous equality, and the elaborator is unable to guess what these types are. The third case provides for a traversal of the context, checking whether a proof is already available. Finally, the fourth case causes an error to be thrown if the proof was not trivial.

6.1.2 Simplifying Arithmetic Expressions

The function `plus` that implements natural number addition is defined by recursion on its first argument. This means that certain equalities that users may consider to be trivial, such as $n + Succ(m) = Succ(n + m)$, exist as lemmas in the library that must be explicitly applied. This process is entirely tedious and can be automated. However, a general-purpose search mechanism that attempted to use the entire standard library to rewrite equalities to something easily provable would very likely be too slow and fragile to use. This is an excellent use for a custom tactic.

Indeed, a family of such tactics can be defined using a simple combinator language. In this example, we define rewriters for arithmetic expressions involving addition, zero, and successors, but the approach can easily be extended to cover more equalities.

Let a *rewriter* be a function in `TT -> Maybe Tactic`. A rewriter, when passed a goal, should either return a tactic that simplifies the goal or `Nothing`. Figure 5 demonstrates two rewriters for addition. The first uses the library proof `plusSuccRightSucc`, which expresses the identity $n + Succ(m) = Succ(n + m)$. The second uses the proof `plusZeroRightNeutral`, which expresses that zero is a right-identity of addition. Quasiquotes provide a convenient notation for both pattern-matching the goal terms and constructing the proof objects to rewrite with. Without quasiquotes, the first example would be much longer, as can be seen in Figure 6.

It is important to point out that this is a particularly *easy* case to translate. The function is monomorphic, with no implicit arguments to be solved. The types in question are first-order, with no parameters or indices. In many realistic programs, especially those in which implicit arguments must be solved, the relationship between the term to be rewritten and its low-level reflected representation might be much more difficult to discern.

```
rewrite_plusSuccRightSucc : TT -> Maybe Tactic
rewrite_plusSuccRightSucc
  (App
    (App
      (P Ref (NS (UN "plus") ["Nat", "Prelude"]) _)
      n)
    (App
      (P (DCon 1 _)
         (NS (UN "S") ["Nat", "Prelude"])
         _)
      m)) =
  Just (Rewrite
          (App (App (P Ref
                       (NS (UN "plusSuccRightSucc")
                           ["Nat", "Prelude"])
                       _)
                    n)
               m))
rewrite_plusSuccRightSucc _ = Nothing
```

Figure 6. A rewriter, without quasiquotes

Returning to the rewriting library, we can define a few simple combinators:

```
(<||>) : (TT -> Maybe Tactic) ->
         (TT -> Maybe Tactic) ->
         TT -> Maybe Tactic
rewrite_eq : (TT -> Maybe Tactic) ->
             TT -> Maybe Tactic
rewrite_nat : (TT -> Maybe Tactic) ->
              TT -> Maybe Tactic
```

The `(<||>)` operator attempts to rewrite using its left-hand rewriter. If this fails, it will attempt to rewrite with its right-hand operator. The operators `rewrite_eq` and `rewrite_nat` recurse over the structure of the goal, attempting to apply rewrite rules at each step. They apply to equality types and natural number expressions, respectively.

It is possible to derive a rewriter for equalities of expressions involving natural numbers and addition as follows:

```
rewrite_eq
  (rewrite_nat
    (rewrite_plusSuccRightSucc <||>
     rewrite_plusZeroRightNeutral))
```

This rewriter can be used in a custom tactic to repeatedly rewrite until a normal form has been reached.

6.2 Error Reflection

As described in the introduction and in a previous paper [5], Idris's *error reflection* allows programmatic rewriting of error messages. This can be used to provide domain-specific errors for embedded domain-specific languages, but most type errors are of the form "Can't unify t_1 with t_2", where t_1 and t_2 can be arbitrarily large terms. Additionally, dependent types often lead to a lot of redundant information being retained on terms in order to propagate type information - and much of this information is collected on implicit arguments that need to be inferred by the elaborator anyway. Without quasiquotation, pattern-matching on these terms inside of reflected error messages is extremely verbose and error-prone.

```
%error_handler
vectRewrite : Err -> Maybe (List ErrorReportPart)
vectRewrite (CantUnify x
                       `(Vect ~n ~a)
                       `(Vect ~m ~b)
                       _ _ _) =
  if n /= m
    then Just [TextPart "Mismatching lengths."]
    else Nothing
vectRewrite _ = Nothing
```

Figure 7. Error rewriter for vector lengths

As a simple example, Figure 7 demonstrates an error message rewriter provides a hint to users who attempt to use a vector whose length does not match the expected length. This code will cause unification errors between two vector types, when their lengths are not identical, to be replaced by the message "Mismatching lengths". For reasons of space, this is a very simple example. The equivalent of just one of the patterns, `` `(Vect ~n ~a) ``, is:

```
App (App (P (TCon _ 2)
            (NS (UN "Vect") ["Vect", "Prelude"])
            (Bind _
                  (Pi (P (TCon _ 0)
                         (NS (UN "Nat")
                             ["Nat", "Prelude"])
                         Erased))
                  (Bind _
                        (Pi (TType _))
                        (TType _))))
        n)
    a
```

where the universal patterns are a result of the special quoting rules that are applied in a pattern context.

This example was very simple. Many realistic error rewriting rules are much more complicated. Without quasiquotes, the error reflection feature would be worthless, because the effort required to manually elaborate terms would be far too great.

7. Conclusion and Future Work

This paper introduced a quasiquotation feature in the Idris language. These quotations can decrease the verbosity of reflection and allow the use of the implicit argument resolution mechanisms and type-driven overloading when constructing reflected terms. Idris's type-driven elaboration mechanism [3] needed only a small amount of new code in order to handle this unforeseen extension, providing evidence that the approach can scale to new features.

The present implementation of quasiquotation has one major limitation: the elaboration of some terms in the high-level Idris language results in auxiliary definitions, which are then referenced in the elaborated TT terms. This is because, in TT, all pattern matching must occur at the top level. As an example, `case` blocks and pattern-matching `let`s are elaborated into top-level functions. Presently, the quotations of these terms simply refer to names of definitions that do not exist. Potential solutions to this problem include rejecting terms with this kind of side effect or tracking the original syntax that results in auxiliary definitions, so that two quotations of the same high-level term will refer to the same auxiliary name. Neither potential solution is entirely satisfactory.

Presently, the elaboration of quasiquote patterns introduces a number of universal patterns in invisible parts of the term where the user would not be able to predict or control the contents, such as machine-generated unused names. However, the locations at which these patterns are inserted is currently not well founded in experience, and it may match too many terms. It would be useful to have a means of being more precise, or possibly even using dependent pattern matching to ensure a kind of restricted α-equivalence between the term being destructured and the quoted term in the pattern.

Acknowledgments

I would like to thank Edwin Brady for his assistance with the Idris implementation. Additionally, I would like to thank my Ph.D. advisor Peter Sestoft for his comments on drafts of this paper and Eugene Burmako and Denys Shabalin for correcting my misunderstandings of Scala's quasiquotes. This work was funded by the Danish National Advanced Technology Foundation (*Højteknologifonden*) grant 017-2010-3.

References

[1] L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM. .

[2] A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, 1999.

[3] E. Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[4] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13. ACM, 2013.

[5] D. R. Christiansen. Reflect on your mistakes! Lightweight domain-specific errors. Unpublished manuscript, 2014.

[6] D. de Rauglaudre. Camlp4 reference manual, 2003. URL `http://pauillac.inria.fr/camlp4/manual/`.

[7] D. Delahaye. A tactic language for the system coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science*, November 2000.

[8] M. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000.

[9] G. Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, pages 73–82. ACM, 2007.

[10] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[11] Microsoft. Expression trees (c# and visual basic), accessed August, 2014. URL `http://msdn.microsoft.com/en-us/library/bb397951.aspx`.

[12] W. v. O. Quine. *Mathematical Logic*. Harvard University Press, revised edition, 1981.

[13] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical Report 185242, École polytechnique fédérale de Lausanne, 2013.

[14] D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.

[15] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1):211–242, 2000.

[16] The Agda Team. The Agda Wiki, accessed 2014. URL `http://wiki.portal.chalmers.se/agda/`.

[17] The MetaOCaml Team. MetaOCaml, accessed 2014. URL `http://www.cs.rice.edu/~taha/MetaOCaml/`.

[18] P. van der Walt and W. Swierstra. Engineering proof by reflection in Agda. In R. Hinze, editor, *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 157–173. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41581-4. .

[19] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100. ACM, 2013.

FEDELE: A Mechanism for Exending the Syntax and Semantics for the Hybrid Functional-Object-Oriented Scripting Language FOBS

James Gil de Lamadrid
Bowie State University, 14000 Jericho Pk Rd, Bowie, MD. 20715
jgildelamadrid@bowiestate.edu

Abstract

A language FOBS-X (Extensible FOBS) is described. This language is an interpreted language, intended as a universal scripting language. An interesting feature of the language is its ability to be extended, allowing it to be adapted to new scripting environments. The interpretation process is structured as a core-language parser back-end, and a macro processor front-end. The macro processor allows the language syntax to be modified. A configurable library is used to help modify the semantics of the language, allowing the addition of the required capabilities for interacting in a new scripting environment. This paper focuses on the semantic extension of the language. A tool called FEDELE has been developed, allowing the user to add library modules to the FOBS-X library. In this way the semantics of the language can be enhanced, and the language can be adapted to new scripting environments.

Keywords functional, object-oriented, programming language

1. Introduction

The object-oriented programming paradigm and the functional paradigm both offer valuable tools to the programmer. Many problems lend themselves to elegant functional solutions. Others are better expressed in terms of communicating objects. FOBS-X is a single language with the expressive power of both paradigms allowing the user to tackle both types of problems, with fluency in only one language. FOBS-X is a modification to the FOBS language described in Gil de Lamadrid & Zimmerman [4]. The modification involves simplifications to the pointers used in the scoping rules.

FOBS-X has a distinctly functional flavor. In particular, it is characterized by the following features:

- A single, simple, elegant data type called a FOB, that functions both as a function and an object.
- Stateless programming. In the runtime environment, mutable objects are not allowed. Mutation is accomplished, as in functional languages, by the creation of new objects with the required changes.
- A simple form of inheritance. A sub-FOB is built from another super-FOB, inheriting all attributes from the super-FOB in the process.
- A form of scoping that supports attribute overriding in inheritance. This allows a sub-FOB to replace data or behaviors inherited from a super-FOB.
- A macro expansion capability, enabling the user to introduce new syntax.
- A tool for easily writing new library modules, allowing the semantics of FOBS-X to be modified to fit differing scripting requirements.

As with many scripting languages FOBS is weakly typed, a condition necessitated by the fact that it only has one data type. However, with interpreted languages the line between parsing and execution is more blurred than with compiled languages, and the necessity to perform extensive type checking before execution becomes less important.

Several researchers have built hybrid language systems, in an attempt to combine the functional and object-oriented paradigms, but have sacrificed referential transparency in the process. Yau et al. [11] present a language called PROOF. PROOF tries to fit objects into the functional paradigm with little modification to take into account the functional programming style. The language D by Alexandrescu [1] is a rework of the language C transforming it into a more natural scripting language similar to Ruby and Javascript.

Scala by Odersky et al. [12] is a language compiled to the Java Virtual Machine, which claims to implement a hybrid of functional and object-oriented paradigms, but tends toward the imperative language end of the spectrum. A class based language that is proposed as a tool to write web-servers, Scala is implemented as a small core language, and many of its capabilities are implemented in the library. FOBS has this same structure, allowing the capabilities of the language to be easily extended.

Two languages that seek to preserve functional features are FLC by Beaven et al. [2], and FOOPS by Goguen and Mesegner [6]. FOOPS is built around the addition of ADTs to functional features. We feel that the approach of FLC is conceptually simpler. In FLC, classes are represented as functions. This is the basis for FOBS also. In FOBS we have, however, removed the concept of the class. In a stateless environment, the job of the class as a "factory" of individual objects, each with their own state, is not applicable. In stateless systems a class of similar objects is better represented as a single prototype object that can be copied with slight modifications to produce variants.

IFL '14, October 1–3, 2014, Boston, MA, USA.

http://dx.doi.org/10.1145/

Another language that implements object-orientation while maintaining a mostly functional approach is OCAML[8]. Built around ML, OCAML has added elements enabling imperative and object orient programming. A record structure supports the creation of objects, and mutable objects support stateful programming. Later in the paper we discuss the importance of mutation in object-orientation. And, although important, we felt that mutation should be isolated and controlled, to help preserve the overriding computation model of FOBS, which prominently features referential transparency. In OCAML, mutable objects are tightly integrated into the computational model, giving it a distinct non-declarative nature.

Scripting languages have tended to shy away from the functional paradigm. Several object-oriented scripting languages such as Python [3] are available. Although mostly object-oriented, its support for functional programming is decent, and includes LISP characteristics such as anonymous functions and dynamic typing. However, Python lacks referential transparency. We consider this as one of the important advantages of FOBS. In the design of FOBS, we also felt that a simpler data structure could be used to implement objects and the inheritance concept, than was used in this popular language. FOBS combines object orientation and functional programming into one elegant hybrid, making both tools available to the user. Unlike languages like Python or FOOPS, this is not done by adding in features from both paradigms, but rather by searching for a single structure that embodies both paradigms, and unifies them.

2. Language Description

FOBS-X is built around a core language, core-FOBS-X. Core-FOBS-X has only one type of data: the FOB. A simple FOB is a quadruplet,

```
[m i -> e ^ ρ]
```

The FOB has two tasks. Its first task is to bind an identifier, i, to an expression, e. The e-expression is unevaluated until the identifier is accessed. Its second task is to supply a return value when invoked as a function. ρ (the ρ-expression) is an unevaluated expression that is evaluated and returned upon invocation.

The FOB also includes a modifier, m. This modifier indicates the visibility of the identifier. The possible values are: "'+", indicating public access, "'~", indicating protected access, and "'$", indicating argument access. Identifiers that are protected are visible only in the FOB, or any FOB inheriting from it. An argument identifier is one that will be used as a formal argument, when the FOB is invoked as a function. All argument identifiers are also accessible as public.

As an example, the FOB

```
['+x -> 3 ^ 6]
```

is a FOB that binds the variable x to the value 3. The variable x is considered to be public, and if the FOB is used as a function, it will return the value 6.

Primitive data is defined in the FOBS library. The types *Boolean*, *Char*, *Real*, and *String* have constants with forms close to their equivalent *C* types. The *Vector* type is a container type, with constants of a form close to that of the ML list. For example, the vector

```
["abc", 3, true]
```

represents an ordered list of a string, an integer, and a Boolean value. Semantically, a vector is more like the Java type of the same name. It can be accessed as a standard list, using the usual car, cdr, and cons operations, or as an array using indexes. It is implemented as a Perl list structure. Unlike the Java type, the FOBS-X type is immutable. The best approximation to the mutate operation is the creation of a brand new modified vector.

There are three operations that can be performed on any FOB. These are called *access*, *invoke*, and *combine*. An access operation accesses a variable inside a FOB, provided that the variable has been given a public or argument modifier. As an example, in the expression

```
['+x -> 3 ^ 6].x
```

the operator "." indicates an access, and is followed by the identifier being accessed. The expression would evaluate to the value of x, which is 3.

An invoke operation invokes a FOB as a function, and is indicated by writing two adjacent FOBs. In the following example

```
['$y -> _ ^ y.+[1]] [3]
```

a FOB is defined that binds the variable y to the empty FOB and returns the result of the expression $y + 1$, when used as a function. When the example is used as a function, since y is an argument variable, the binding of the variable y to the empty FOB is considered only a default binding. This binding is replaced by a binding to the actual argument, 3. To do the addition, y is accessed for the FOB bound to the identifier +, and this FOB is invoked with 1 as its actual argument. The result of the invocation is 4.

In an invocation, it is assumed that the second operand is a vector. This explains why the second operand in the above example is enclosed in square braces. Invocation involves binding the actual argument to the argument variable in the FOB, and then evaluating the ρ-expression, giving the return value.

A combine operation is indicated with the operator ";". It is used to implement inheritance. In the following example

```
['+x -> 3 ^ _] ; ['$y -> _ ^ x.+[y]]          (1)
```

two FOBs are combined. The super-FOB defines a public variable x. The sub-FOB defines an argument variable y, and a ρ-expression. Notice that the sub-FOB has unrestricted access to the super-FOB, and is allowed access to the variable x, whether modified as public, argument or protected.

The FOB resulting from Expression (1) can be accessed, invoked, or further combined. For example the code

```
(['+x -> 3 ^ _] ; ['$y -> _ ^ x.+[y]]).x
```

evaluates to 3, and the code

```
(['+x -> 3 ^ _] ; ['$y -> _ ^ x.+[y]]) [5]
```

evaluates to 8.

Multiple combine operations result in FOB stacks, which are compound FOBs. For example, the following code creates a FOB with an attribute x and a two argument function that multiplies its arguments together. The code then uses the FOB to multiply 9 by 2.

```
(['+x -> 5 ^ _] ; ['$a -> _ ^ _] ;
 ['$b -> _ ^ a.*[b]]) [9, 2]
```

In the invocation, the arguments are substituted in the order from top to bottom of the FOB stack, so that the formal argument a would be bound to the actual argument 2, and the formal argument b would be bound to 9.

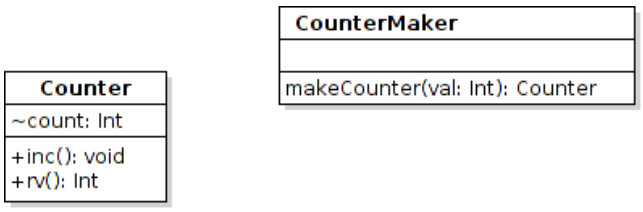


Figure 1. Class structure of Example (2)

In addition to the three FOBS operations, many operations on primitive data are defined in the FOBS library. These operations include the usual arithmetic, logic, and string manipulation operations. In addition, conversion functions provide conversion from one primitive type to another, when appropriate.

Example (2) presents a larger example to demonstrate how FOBS code might be used to solve more complex programming problems. In this example we define a FOB that implements a standard up-counter. The FOB structure is shown in Figure 1, using UML. The outermost FOB implements the UML class called *CounterMaker*, that copies a prototype to create new counters. The counters are known as the class *Counter* in Figure 1. *CounterMaker* creates a new *Counter* when its function *makeCounter* is called. The argument to *makeCounter*, *val*, becomes the initial value of the counter. The counter contains an instance variable, *count*, that contains the current count value, and a function *inc* that "increments" the counter. Since FOBS is stateless, what *inc* actually does is create a new *Counter* object with the incremented *count* variable.

```
## Implementation of a standard up-counter
([‘+makeCounter ->
     [‘$val -> 0 ^
          [‘~count -> val ^_];
          [‘+inc ->
               [‘~_ -> _^ makeCounter[
                    count.+[1]]]
           ^_];
          [‘~_ -> _ ^ count]
     ]
^_]
## test it
     .makeCounter[6].inc[].inc[])[]
#.
#!
```
(2)

Since UML is designed to model object-oriented systems, it is no surprise that using it to model a FOB requires extra notation to handle the ability to invoke a FOB as a function. In Figure 1. the notation *rv* is used to represent the operation of invoking the FOB as a function. The use of *rv* (return value) in the diagram indicates that, when the FOB Counter is invoked, it returns the current value of the variable *count*.

Larger examples, and a more complete definition of the FOBS language are given by Gil de Lamadrid and Zimmerman [4].

3. Core-FOBS Design Topics

Expression evaluation in FOBS-X is fairly straight forward. Three issues, however, need some clarification. These issues are: the semantics of the redefinition of a variable, the semantics of a FOB invocation, and the interaction between dynamic and static scoping.

3.1 Variable overriding

A FOB stack may contain several definitions of the same identifier, resulting in overriding. For example, in the following FOB

```
[‘$m -> ’a’ ^ m.toInt[]] ; [‘+m -> 3 ^ m]
```

the variable m has two definitions; in the super-FOB it is defined as an argument variable, and in the sub-FOB another definition is stacked on top with m defined as a public variable. The consequence of stacking on a new variable definition is that it completely overrides any definition of the same variable already in the FOB stack, including the modifier. In addition, the new return value becomes the return value at the top of the full FOB stack.

3.2 Argument substitution

As mentioned earlier, the invoke operator creates bindings between formal and actual arguments, and then evaluates the ρ-expression of the FOB being invoked. At this point we give a more detailed description of the process.

Consider the following FOB that adds together two arguments, and is being invoked with values 10 and 6.

```
([‘$r -> 5 ^ _] ; [‘$s -> 3 ^ r.+[s]]) [10, 6]
```

The result of this invocation is the creation of the following FOB stack

```
[‘$r -> 5 ^ _] ;
[‘$s -> 3 ^ r.+[s]] ;
[‘+r -> 6 ^ r.+[s]] ;
[‘+s -> 10 ^ r.+[s]]
```

In this new FOB the formal arguments are now public variables bound to the actual arguments, and the return value of the invoked FOB has been copied up to the top of the FOB stack. The return value of the original FOB can now be computed easily with this new FOB by doing a standard evaluation of its ρ-expression, yielding a value of 16.

3.3 Variable scope, and expression evaluation

Scoping rules in FOBS-X are, by nature, more complex than scoping used in most functional languages. Newer functional languages, such as Haskell and ML, typically use lexical scoping. Dynamic scoping is often associated with older dialects of LISP.

Pure lexical scoping does not cope well with variable overriding, as understood in the object-oriented sense, which typically involves dynamic message binding. To address this issue, FOBS-X uses a hybrid scoping system which combines lexical and dynamic scoping. Consider the following FOB expression.

```
[‘~y -> 1^_] ;
[‘~x ->
     [‘+n -> y.+[m] ^ n] ;
     [‘~m -> 2 ^_]
^_] ;
[‘~z -> 3 ^x.n]
```
(3)

This expression defines a FOB stack that is three deep, containing declarations for a protected variable y, with value 1, a protected variable x with a FOB stack as its value, and a protected variable z with the value 3 as its value. The stack that is the value of x consists of two FOBs, one defining a public variable n, and one defining a protected variable m.

We are currently mostly interested in the FOB stack structure of Expression (3), and can represent it graphically with the stack graph, given in Figure 2. In the stack graph each node represents a simple FOB, and is labeled with the variable defined in the FOB.

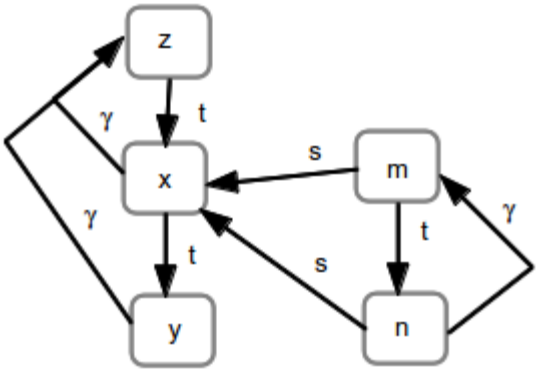


Figure 2. Stack graph of Example (3)

Three types of edges are used to connect nodes: the s-pointer, the t-pointer, and the γ-pointer. The s-pointer describes the lexical nested block structure of one FOB defined inside of another. The s-pointer for each node points to the FOB in which it is defined. For example m is defined inside of the FOB x.

The t-pointer for each node points to the super-FOB of a FOB. It describes the FOB stack structure of the graph. In Figure 2 there are basically two stacks: the top level stack consisting of nodes z, x, and y, and the nested stack consisting of nodes m, and n.

The γ-pointer is a back pointer, that points up the FOB stack to the top. This provides an easy efficient mechanism for finding the top of a stack from any of the nodes in the stack.

If the FOB z were invoked, it would access the FOB n for the value of n. This would cause the expression $y+m$ to be evaluated, a process that demonstrates the use of all three pointers. The process of resolving a reference in FOBS-X first examines the current FOB stack. The top of the current stack is reached by following the γ-pointer. Then the t-pointers are used to search the stack from top to bottom. If the reference is still unresolved, the s-pointer is used to find the FOB stack enclosing the current stack. This enclosing stack now becomes the current stack, and is now searched in the same fashion, from top to bottom, using the γ-pointer to find the top of the stack, and the t-pointers to descend to the bottom.

To summarize this procedure for the example, to locate the definition of the variable y, referenced in the FOB n, the γ-pointer for n is followed up to the FOB m, this FOB is examined, and then its t-pointer is followed down to the FOB n, which is also examined. Not having found a definition for the variable y, the s-point for FOB n is followed out to the FOB x, and then the γ-pointer is followed up to the FOB z. FOB z is examined, and its t-pointer is traversed to FOB x, which is also examined. Then the t-point for FOB x is finally followed down to the FOB y, which supplies the definition of y needed in the FOB n.

As mentioned above, the scoping for FOBS-X is a combination of lexical and dynamic scoping. S-pointers are lexical in nature, since the nesting of FOBs is a static property. T-pointers and γ-pointers are dynamic. These pointers must be created as new FOB stacks are created during execution.

4. The FOBS Library

As FOBS-X can be extended by adding new primitive FOBs to the library, we use the term *native primitive FOBs* to denote the primitive FOBs that are part of core-FOBS. The FOBS library contains

Libary FOB	Operation	Description
`Boolean`	`b.if[x, y]`	If Boolean value `b` is `true`, return x, otherwise return y
	`b.&[x]`	Return the boolean value of the expression $b \bigwedge x$
	`b.|[x]`	Return the boolean value of the expression $b \bigvee x$
	`b.![]`	Return the boolean value of the expression $\neg b$

Table 1. Operations for the *Boolean* FOB

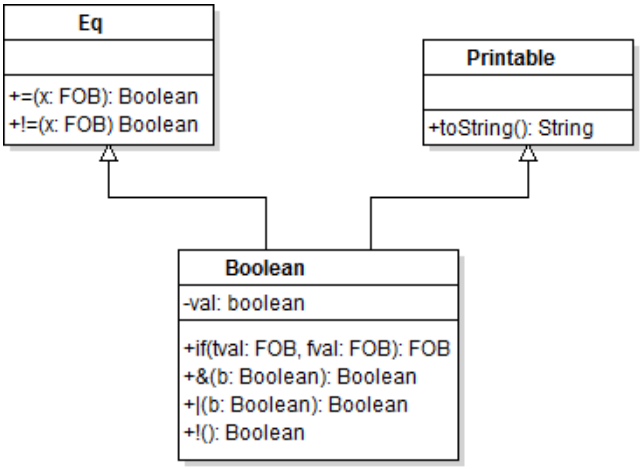


Figure 3. Interface for the *Boolean* FOB

definitions of all native primitive FOBs. The native primitive FOBs are *Int*, *Char*, *Real*, *Boolean*, *Vector*, *String*, and *FOBS*. In addition a set of "mix-in" FOBs are contained in the library, that serve the same purpose as mix-in classes described by Page-Jones [9], providing general capabilities to primitive FOBs. For example the *Boolean* FOB uses the mix-in FOBs *Eq*, and *Printable* to supply operations to compare Boolean values for equality, and the ability to be printed, respectively.

The native primitive FOBs mostly implement the native data types of the FOBS language. Each data type provides the wrapper for the data, along with a set of operations, used to manipulate the data. As an example, Table 1. shows the operations provided by the *Boolean* FOB. This operation structure is shown in the UML diagram of Figure 3. The operations for the *Boolean* FOB are implication, logical *and*, logical *or*, and logical *not*. The *Boolean* FOB inherits the operations of *equals*, and *not-equals* form the mix-in FOB *Eq*, and it inherits the *toString* function, that generates a print-string, from the FOB *Printable*.

The primitive FOB *FOBS* is the one primitive FOB that does not implement a data type. This FOB is, initially, largely empty. It, however, provides the mechanism for extending the FOBS-X language, allowing it to be adapted to differing scripting environments. The user of the FOBS-X language extends the language by adding modules to the *FOBS* FOB, one for each extension to the language.

5. Extensions

FOBS is a language that is designed to be extensible, both in terms of syntax, and semantics. To extend the language the user designs an *extension*. An extension is defined by an *extension module*, which is composed of two pieces: a macro file, and a collection of library modules.

5.1 Macro files

FOBS-X allows the syntax of the language to be changed in a limited fashion. The mechanism used to modify the syntax is macro expansion. Before a FOBS expression is parsed, a macro processor is used to expand macros used in the code. In this way, the user can alter the syntax of FOBS expression by writing and loading the appropriate macros to handle the changes.

Many programming languages have macro capabilities. These range from the fairly simple mechanisms in the programming language *C*, to the the relatively more sophisticated mechanisms of LISP. It was felt that these simple systems were inadequate for FOBS. In particular, to implement a fair degree of flexibility, we felt that the ability to modify syntax should be more extensive than these types of systems offer, including a limited ability to change delimiter symbols. The language MetaML [13] provides much more sophisticated macro capabilities. It is built for the manipulation of macro type code, and implements multi-stage meta-programming. The macro capability of FOBS is much lighter weight than that of MetaML, but ideas from MetaML have found their way into FOBS-X. In particular, we found the staging of macro expansion useful. The staging in our case is used to implement macro operator precedence.

Macro definitions are quadruples, which are described in detail by Gil de Lamadrd [5]. Example (4) gives a simple demonstration the form of macro definitions.

```
## the array mutate operation
#defleft
     #?x [ #*i ] <- #?y
#as
     ( #?x ) . -+ [ #*i , #*y ]
#level
     3
#end
```
(4)

The macro quadruple, $< S_1 \rightarrow S_2 : P, d >$, is composed of the following parts.

- S_1: the search string, which includes wild-card tokens
- S_2: the replacement string, which includes wild-card tokens.
- P: the priority of the macro, with priority 19 being highest priority, and priority 0 being the lowest.
- d: the direction of the scan, with *right* indicating right-to-left, and *left* indicating left-to-right.

In the FOBS notation of Example (4) the parts of the quadruple are specified using either the `#defleft`, or the `#defright` directive. Firstly, the directive specifies the direction d, depending on whether `#defleft` or `#defright` is used. Then the search string S_1, the replacement string S_2, and the priority P are specified, in order, separated by the two delimiters `#as`, and `#level`, and terminated by the `#end` directive.

The strings S_1, and S_2 are strings of FOBS lexicons, and wild-card tokens. Wild card tokens are all tokens that begin withe either the sequence "#?" or "#*", indicating a *single* wild card token, or a *mutiple* wild card token. A single wild card matches a single *atom*, and a multiple wild card matches a string of atoms. An atom is either a single FOBS token, or a balanced bracketed string, using one of the usual bracketing characters such as parentheses or braces.

Wild cards are named, so that the match in S_1 can be referred to in S_2. In Example (4), for example, the wild cards `#?x`, `#?y`, and `#*i` are matched in S_1, and their values are used in S_2.

The direction, d, and the priority of a macro, P, are used to control the associativity of the operator defined by the macro, and the precedence of the operator, respectively. To control associativity, macros defined with direction *left* are expanded left-to-right, resulting in the definition of a left-associative operator, and macros defined with a direction of *right* are expanded right-to-left, resulting in a right-associative operator. To control precedence, macros with higher priority are expanded before macros with lower priority, resulting in operators with different precedences.

5.2 The standard extension

The syntax in core-FOBS-X is a little cumbersome. It has been designed with minimalistic notation, allowing a concise formal description, given by Gil de Lamadrid & Zimmerman [4]. It is not necessarily attractive to the programmer. Standard extension (SE) FOBS-X attempts to rectify this situation. In particular, SE-FOBS-X includes constructs to enable the following.

- Allow infix notation for most operators.
- Eliminate the cumbersome syntax associated with declaring a FOB.
- Introduce English keywords to replace some of the more cryptic notation.
- Allow some parts of the syntax to be optionally omitted.

SE-FOBS-X is a language defined entirely using the macro processor. It demonstrates the flexibility of the FOBS-X macro capability to almost entirely rework the syntax of the language, without touching the back-end of the interpreter.

To help demonstrate the changes in syntax allowed by SE-FOBS, we rewrite the counter of Example (2) to in SE-FOBS.

```
#use #SE
## Implementation of a standard up-counter
(fob{
   public makeCounter
   val{
      fob{
         argument val
         ret{
            fob{
               count val{val} \
               public inc
               val{
                  fob{
                     ret{makeCounter
                        [count + 1]}
                  \}
               } \
               ret{count}
            \}
         }
      \}
   }
\}
## test it
   .makeCounter[6].inc[].inc[])[]
#.
#!
```
(5)

The `#use` directive loads the standard extension macro file. This file makes available the syntax used in the remainder of the code. The most salient syntax feature of the code is the *fob* structure, used to define FOB-stacks. Each FOB in the stack is listed in the *fob* construct, and terminated by the "\" delimiter.

A FOB declaration contains a modifier, the identifier, a *val* structure, and a *ret* structure. The *val* structure defines the e-expression for the FOB, and the *ret* structure gives the ρ-expression for the FOB. Any of the parts of the FOB declaration may be omitted, resulting in the use of appropriate default values. Modifiers in SE-FOBS are the keywords *public*, *private*, and *argument*, instead of the cryptic symbols "'+", "'~" , and "'$".

A final feature present in Example (5) is the use of the infix version of the addition operator. All common binary operators in SE-FOBS are available in their infix version, relieving the user from using the normal core-FOBS access and invoke notation.

5.3 Extension library modules

Macro files extend the syntax of the FOBS language. To extend the semantics, you must add modules to the FOBS library. The FOBS library is written in Perl, and so to add modules you simply write Perl modules, and add them into the appropriate library directory structure. This process initially may sound simple. On further reflection, it becomes obvious that to do this

- One needs to be fairly familiar with the structure of the FOBS library.
- One must be familiar with how to manipulate FOBs in Perl.

While it is reasonable to expect a user requiring complex semantic changes to learn the required material to develop library modules from scratch, it is an unreasonable burden to impose on the user that desires to make only minor changes to the semantics of FOBS. To make small changes it is more appropriate for the user to do so using a tool that simplifies the process. The tool that we have developed is the FOBS Extension Definition Language Extension (FEDELE).

When designing FEDELE, we first thought of a meta-language that was implemented as an external tool. However, since FOBS is a scripting language, and designed for just such work, we rapidly realized that it made sense to implement FEDELE as a FOBS extension. FEDELE is, therefor, a FOBS extension that helps the user create other FOBS extensions.

6. FEDELE Operating Environment

The standard extension is unusual in that it is an extension with only one component: the macro file. No semantic changes are made to FOBS; only syntactic changes. Most extensions contain both a macro file and library modules. FEDELE is a more usual extension. Library modules provide the capabilities of the package, and a macro file provides more convenient syntax for using it.

The FEDELE extension provides a simpler way of writing the library modules necessary for implementing an extension. The FEDELE language allows the user to specify the structure of the extension much in the same way that YACC (see Johnson [7]) allows a programming language designer to specify the structure of a new programming language. The specification is translated into a set of Perl modules implementing the extension. The modules are then placed in a directory, and the directory path is placed on the Perl include path *@INC*, extending the directories searched for library modules. This summarizes the process of extending the library, but to continue our discussion of FEDELE, we will need to examine the structure of the FOBS library in more detail.

6.1 The FOBS library implementation

The FOBS library is composed of a collection of primitive and utility FOBS. As explained previously primitive FOBs use utility FOBs to mix-in general capabilities. However, from the standpoint of structure, there is no difference between a primitive and a utility FOB. In this discussion we will therefor consider only the structure of a primitive FOB.

To illustrate the structure of a primitive FOB, we take as example the FOB *Boolean*. The Boolean FOB can be represented in UML as shown in Figure 3. It contains an instance variable *val* that contains the actual Boolean value, represented as a character string. It also contains the common Boolean operations of *and*, "&", *or*, "|", and *not*, "!". In addition it contains the operator *if* that implements the implication operator. The FOB *Boolean* inherits operations from the FOBs *Eq*, and *Printable*. From *Eq* it inherits the operations *equals*, "=", and *not-equals*, "!=". From *Printable* it inherits the operation *toString*, that converts a Boolean value into a printable string.

It should be noted that the term "inhertitance" for primitive FOBs is only loosely applied. In fact, the mechanism is more of a message-forwarding mechanism. That is to say that, for example, if a *Boolean* FOB receives an *equals* access request, the request is forwarded to its parent *Eq* FOB.

Implementing the *Boolean* FOB in Perl is done with two structures: a hash table, containing the data of the primitive FOB, and a Perl module, *Boolean*, that contains code for all of the operations in the primitive FOB.

6.2 Primitive FOB hash table structure

The hash-table representing the data in a primitive FOB stores information in attribute-value pairs. The attributes of interest are the following.

- *type* - This attribute gives the type of the FOB. A primitive FOB is of type "omega", using the notation described by Gil de Lamadrid & Zimmerman [4].
- *code* - This attribute stores the name of the primitive FOB. For the FOB *Boolean*, the code attribute would have the value "Boolean".
- Super-FOBs - This is a collection of attributes, one per parent FOB. Each of these attributes stores an instance of one of the parent FOBs. For the FOB *Boolean* there are two such attributes. *superEq* stores an instance of the primitive FOB *Eq*, and *superPrintable* stores an instance of the primitive FOB *Printable*.

In adition to the above standard attributes, the primitive FOB hash-table contains attributes that are specific to the particular primitive FOB. For the *Boolean* primitive FOB, there is only one more attribute: the attribute *val*, that holds the boolean value of the FOB, stored as a character string.

6.3 Primitive FOB module structure

The main library module for a primitive FOB has the same name as the primitive FOB. For example, for the primitive FOB *Boolean* there is a Perl module called *Boolean*. This module has four standard functions in it.

- *construct* - This function constructs the hash table representing an instance of the primitive FOB.
- *constructConstant* - This function is an extension of the function *construct*. It constructs the instance, using *construct*, and then initializes it by filling in any instance variables.

- *alpha* - This is the function α that is described by Gil de Lamadrid & Zimmerman [4]. It takes a single argument, a character string, and accesses the primitive FOB for the value of the identifier specified by the argument.
- *iota* - This is the function ι described by Gil de Lamadrid & Zimmerman [4]. It takes a single argument, a *Vector* FOB, and invokes the primitive FOB using the vector to supply its actual arguments.

6.4 Operation modules

The main module of a primitive FOB is not the only module needed to define the FOB. To understand why this is so, consider the following FOBS code, and the semantics of invocation.

```
false.&[true]                                    (6)
```

In this expression, the Boolean FOB *false* is being accessed for its *and* operation. The operation is then being invoked, with the argument *true*. However, the question arises, when we say that the operation is invoked, what is an operation? The simple answer is that if an operation is invoked, then it must be a FOB, because only FOBs are invoked. This observation becomes trivially clear when we look at an example that does not involve a primitive FOB.

```
['+ & -> ['~_ -> _ ^ false] ^ _] . & [true]      (7)
```

In this example, as in Example (6), a FOB is accessed for an "&" operation, and the operation is invoked with the Boolean FOB *true*. The difference is that in Example(7) the FOB being accessed is not a primitive FOB. What is produced by the access operation is a FOB, in this case, that always returns the value *false*. We observe that the same must be true of Example (6). That is to say that an access operation always produces a FOB, whether the FOB being accessed is a primitive FOB or not.

What the above discussion points out is that when we access an operator in a primitive FOB, what is produced is a FOB. That FOB, when invoked would perform the particular operation. Every operator in a primitive FOB must have defined a FOB that will perform the given operation. For a library FOB such as *Boolean* each of its operators is defined as a primitive FOB. For example the *and* operator for the FOB *Boolean* is defined as a primitive library FOB called *Boolean_and*. We refer to library modules for the operations of a primitive FOB, as *primitive operation modules*.

To summarize, a primitive FOB is represented as a set of library modules. These consist of the main library module, described above, and a set of operation modules, one per operation. An operation module contains the same functions as the main module. That is to say that the operation module will have a *construct* function, a *constructConstant* function, an *alpha* function, and an *iota* function, each with the same role as in he main module. Each of these functions would perform actions appropriate to the operator. That is to say that the *alpha* function would always return an empty FOB, and the *iota* function would perform the operation of the operation module.

6.5 Extension access

Once the user has defined an extension, the language FOBS must be able to allow the user to use the extension. This section describes the mechanism used to allow FOBS code to use an extension.

The modules of the extension can be placed at any location in the directory hierarchy of the operating system. The author of the extension then must inform FOBS where the extension is located. As discussed previously, this is done by ensuring that the extension directory is on the list of include directories for Perl, *@INC*. This is easily done by seting the environment variable PERL5LIB to the extension path.

Recall that the two components of an extension are the macro file, and the library modules. We discuss how the FOBS interpreter locates both of these components in this section. We begin with how the macro file is located. A macro file is loaded with the *#use* directive. An example might be

```
#use Count
```

This directive tells the FOBS interpreter to look for a file called *Count.fobs* containing the macros of the extension. What the FOBS interpreter does then is to search Perl include directories, listed in the array *@INC*. There are two exceptions to the procedure, as illustrated in the following *#use* invocations.

```
#use #SE
#use #FEDELE
```

The extensions *#SE*, the standard extension, and *#FEDELE* are considered part of the FOBS language, and as such are located in a separate default FOBS include directory.

We now turn to the location of library modules. The standard mechanism for accessing the library in FOBS is a reference to a constant. For example, if a FOBS expression contains a reference to the constant `true`, the FOBS interpreter observes that this is a *Boolean* constant. The interpreter then goes to the default library directory, locates the main *Boolean* module, and invokes its *constructConstant* constructor function to create the hash-table. *ConstructConstant* also links the main module to the hash-table, a Perl mechanism called *blessing*, effectively making the hash table an *object*, in the object-oriented sense, which is to say that the hash table can be sent messages corresponding to any of the functions defined in the main *Boolean* module.

When the user defines their own library module, the above procedure cannot be used, because there is no FOBS constant for the new primitive FOB that would trigger the FOB construction. Instead, the construction of a FOB is triggered using the FOB *FOBS*. This is illustrated in the following FOBS expression.

```
FOBS.Count.new[5]                                (8)
```

The FOB *FOBS* is a primitive FOB in the FOBS library used to present links to extensions to the user. In Example (8), the user is attempting to access the identifier *Count*, which is the name of an extension. This identifier is not explicitly defined in the FOB *FOBS*. However, the FOBS interpreter will consider it implicitly defined, and, when referenced, will attempt to load the main module for the extension from the list of Perl include directories.

If we assume that the *Count* FOB is defined along the lines of the UML diagram in Figure 1, the *Count* FOB has one operation, *inc*, explicitly defined. For every extension, generated by FEDELE or not, the primitive FOB must also contain a *new* operation. This operation, when called, generates a new instance of the FOB, and calls the *constructConstant* constructor for the FOB. In Example (8), the *new* operator is called to construct a primitive *Count* FOB, initialized to the value 5.

7. The FEDELE Extension

This section describes the components of the FEDELE extension. FEDELE has both a macro file, extending the syntax of FOBS to more easily specify extension components, and library modules, providing the operators required to specify the contents of the library modules of the new extension, and write the module out. We begin by describing the FEDELE operations.

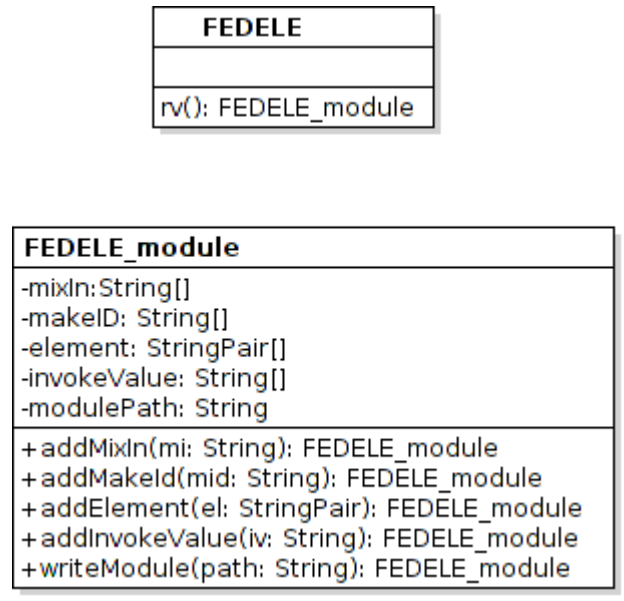


Figure 4. Interface for FEDELE

7.1 The FEDELE primitive FOB

The primitive FOB `FOBS.FEDELE` is a very uncomplicated FOB that has no accessible identifiers in it, and can only be invoked. The result of an invocation is a *FEDELE_module* FOB. The *FEDELE_module* is a data structure used to collect information on the new FOB being described. Figure 4 is the UML diagram showing the two FOBS: *FEDELE*, and *FEDELE_module*.

The *FEDELE_module* FOB contains variables for storing the following items

- *mixIn*: a list of primitive mix-in FOBs.
- *makeID*: a list of identifiers that will be included in the hash-table representing the FOB.
- *element*: a list of operations that will be included in the FOB. Each operation is represented as a pair consisting of the operation name, and a snippet of Perl code that will become the body of the *iota* function for the operation.
- *invokeValue*: a snippet of Perl code that will become the body of the *iota* function for the new FOB itself.
- *modulePath*: the directory on to which the files of the library module will be written.

In addition to the above variables, the *FEDELE_module* also contains operators for adding items to its data structures. Each operation adds an item and returns the modified *FEDELE_module*.

7.2 FEDELE macros

The second part of the FEDELE extension is a macro file that defines the FEDELE language, and allows easier specification of a primitive FOB. This FEDELE language is a structured language. The structures of the language are listed in Table 2.

A FEDELE specification, at the outer level is an *extension* structure. This structure would contain clauses; each clause being either a *mixIn*, a *make*, an *element*, or an *invoke* structure. This is illustrated more clearly in the next section. FEDELE translates the *extension* structure into FOBS code that creates a *FEDELE_module*. The clause structures are translated into *FEDELE_module* operations that add the appropriate elements to the module. For example, the *make* clause would translate into an invocation of the *addMakeId* operator shown in Figure 4.

Structure	Description
`extension "` *xtnd* `" { ` *clauses* ` } to` *path*	Defines an extension with name *xtnd*, to be written to the given directory path. It contains clauses giving the content of the extension *xtnd*.
`mixIn` *mixinFOB*	A clause indicating that the FOB *mixinFOB* from the library is a super-FOB for this FOB. This clause can be repeated to include several super-FOBs.
`make {` *idList* `}`	Describes the constant constructor for the FOB. The constructor will be available as the function `FOBS.`*xtnd*`.new`. *New* takes an argument for each identifier listed, and stores the argument as the value of the identifier. The *idList* is given as a list of strings, separated by commas.
`element "` *id* `" as "` *perlScript* `"`	Gives the name of an element, or operator, of the FOB available through the access operator. The included Perl script gives the value returned if the operator is invoked.
`invoke "` *perlScript* `"`	The Perl script gives the result of an invoke operation on the FOB itself.

Table 2. FEDELE macro operations

8. A FEDELE Example

We now present an example to illustrate how FEDELE is used. Suppose that the user wished to add a primitive FOB to the library that is similar to the counter FOB of Example (2). Remember that this example is illustrated in UML in Figure 1. The new library FOB, however, will be mutable. That is to say that a counter will have state, and each time the counter is incremented it will change its state, rather than produce a new counter with the modified state. This new counter will also support two new syntactic constructs: one to easily construct a counter, and one to increment the counter. The syntax of these operations is illustrated in the following example.

```
++(%C(5))
```

This example uses the "%C" operator to create a counter initialized to 5. The second operator illustrated, "++", is used to increment a counter.

Our new counter will also allow the user to increment it by any value, as opposed to just an increment of one. An increment of more than one will not be supported by the macros, but can still be accomplished by using the *inc* function itself, as in

```
c.inc[3]
```

that increases the value of the counter c by 3. Figure 5 shows the new FOB structure in UML.

8.1 The counter FEDELE specification

The extension specification for our new counter consists of a FEDELE specification describing the library modules, and a macro file defining the syntax of the constructor operator, and the increment operator. We begin by presenting the FEDELE code to generate the library modules.

```
## FEDELE specification to generate
##  the example counter
#use #FEDELE

extension "Count" {
     mixIn "Printable"
     make {"count"}
     element "inc" as "
          $args = lib::PrimitiveFobs->
             thunkDown($args->[0]);
          if($args eq $undef){
               return(lib::PrimitiveFobs::
                  getEmpty())
          };
          if($args->{type} eq \"omega\" &&
             $args->{code} eq \"Int\"){
               $it->{count}->{val} +=
                  $args->{val};
               return($it);
          }
          return(lib::PrimitiveFobs::
             getEmpty());
     "
     element "toString" as "
          my $v = $it->{count}->{val};
          return(lib::FEDELE::
             evalString(\"\\\"%C:\\\"
             .+[$v .toString[]]\"));
     "
     invoke "
          return($it->{count});
     "
} to "e:/fobs-x/code/Count"
#.
#!
```

(9)

Considering the overall structure of Example (9), it is , in fact, faithful to the UML description of Figure 5. It specifies a mix-in FOB *Printable*, an identifier *count*, two operations, *inc*, and *toString*, and a return value when invoked.

The code sections illustrate several issues concerned with the interface between FOBS and Perl. The first issue is how to enable a Perl segment to access the arguments of the function call. This is accomplished through the use of several special variables.

- `$it` - The FOB being operated on. That is to say that `$it` is the target of the invoke operation.
- `$args` - A *Vector* FOB containing the arguments of the invoke operation.

The variable `$it` contains all the identifiers declared in the FEDELE declaration as hash attributes. For instance, in the *Count* FOB, the sequence `$it->{count}` is the *count* identifier.

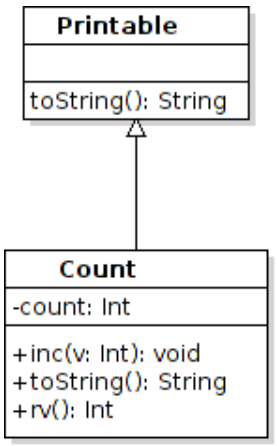


Figure 5. The mutable counter FOB

To access the arguments in the variable `$arg` a helper function is necessary. The arguments to FOBs are stored in thunks. To be used, the FOB inside the argument thunk must be unwrapped and evaluated. The function `lib::PrimitiveFobs->thunkdown` can be used for this purpose, as demonstrated in the definition of the operator *inc*.

There are a couple of other useful Perl functions used in Example (9). The function `lib::PrimitiveFobs::getEmpty` can be used to create an instance of the empty FOB, a FOB often used to signal an exception. Another function `lib::FEDELE::evalString` is used to evaluate FOBS expressions within the Perl code. This is a useful feature. Often it is easier to perform certain actions in FOBS, than in Perl. *EvalString* provides the ability to mix Perl with FOBS code, allowing the user to choose the more efficient implementation.

8.2 The counter macro file

The second component of the extension is the macro file that introduces more compact syntactic notation for the new counter operations. The contents of the file are shown in Example (10).

```
## macros for the Counter example FOB
#defleft
     % C ( #?op )
#as
     ( FOBS . Count . new [ #?op ] )
#level
     9
#end

#defleft
     ++ #?op
#as
     #?op . inc [ 1 ]
#level
     8
#end
#!
```

(10)

The macro file contains the definitions of two macros. The first one implements the constructor structure with the "%C" notation. It matches a string beginning with the character sequence "% C", and followed by a single atom enclosed in parentheses. This sequence is replaced by an invocation of `FOBS.Count.new` with the matched atom as an argument.

The second macro is for the increment operator. It matches the operator "++" followed by an atom, and replaces it with an invocation of the *inc* operation of the atom with argument 1.

8.3 Stateful and stateless programing

Example (10) demonstrates rather graphically one of the issues concerning the hybrid paradigm of FOBS. There is a dichotomy between functional programming and object-oriented programming. The object-oriented paradigm clearly involves the explicit maintenance of state. In fact we often refer to the bindings of instance variables as the state of the object. On the other hand, although state does exist in functional languages, and is usually maintained by the system stack, it is not manipulated explicitly, in the sense that the program does not change the state directly as is the case in imperative and object-oriented programs, but rather indirectly by invoking functions. But, this difference between the two paradigms often becomes significant, and produces awkward situations in FOBS.

One of the defining characteristics of FOBS is referential transparency. This puts FOBS squarely in the camp of stateless programming. This is seen when we observe, for example, that identifiers can be bound to a value only once. Mutable objects are not an option in this style of programming.

On the other hand, mutable objects are a staple of object-oriented programming. Also, state is often an integral part of scripting environments. For example, operating systems scripting often involves manipulating the state, represented as environment variables. To accommodate these situations in a language that advertises itself as an universal scripting language, it is not unreasonable for the user to wish to introduce state into FOBS. This is not difficult; the library can be extended to include mutable FOBS, as for example the *Count* FOB. However, it is still a stretch to use the language FOBS to manipulate these new mutable FOBS. In particular, what is needed to handle mutable FOBs is the ability to define operators whose return values are not used, but rather they are invoked for their side-effects on the state.

The problem of doing this type of stateful programming in a functional paradigm has been well researched, and has resulted in a body of literature on monadic programming (see Peyton Jones & Wadler [10], for example). Related to these results is a technique that has long been used in the object-oriented paradigm, called *method chaining*. This technique is used to pass multiple messages to the same object, as in the example

```
recipient.doX(xArg).doY(yArg)
```

in which the mutable object *recipient* is being first sent the message *doX* with the argument *xArg*, followed by the message *doY* with the argument *yArg*. Although the operations *doX*, and *doY*,naturally, might be thought of as returning no value, with the chaining technique they would instead return the object being operated on, *recipient*. In this way the next message in the chain is sent to the same recipient. One can think of the operators as passing the state along the chain from one operator to the next.

The technique of chaining is used in FOBS to handle mutable objects. Its effects can be observed in the code snippets of Example (9). The operator *inc* is defined to return the variable *$it*, which is the target FOB, and this allows FOBS expressions such as `++(++ %C(5))`, with a chain of increment operation being applied to the same FOB.

9. FOBS and Scripting

The intended use of FOBS is as a universal scripting language. Scripting languages are used to automate processes in a variety of environments. One of the most prevalent uses is in operating system interface. Scripting languages have also become very useful in creating dynamic web pages, and handling the collection of data using forms. They are also used in application programs, such as spreadsheets to automate calculations or procedures. In each of these applications the runtime system has two major components: an interpreter to execute scripts, and an interface that allows the script to interact with the environment. In FOBS-X, the library FOB *FOBS* is the interface to the environment. To adapt FOBS-X to a particular environment, an extension is created in the FOB *FOBS*. This extension contains all operations required for the interface, defined as FOBs.

As seen in the previous section, we have somewhat automated the process of creating these extensions. The user supplies a FEDELE description of an extension, and it is translated into a Perl definition. We have commenced the construction of a UNIX extension. We present an example of how this extension might be used in scripting.

A simple UNIX C-shell script follows that takes a command line argument, and prints out all files in the current directory containing that string.

```
#!/bin/csh
     if ( $#argv == 0 ) then
          echo Usage: $0 name
          exit 1
     else
          set user_input = $argv[1]
          ls | grep i $user_input
     endif
     exit 0
```

Assuming that an extension UNIX has been created, the above code could be translated into SE-FOBS-X as follows.

```
#use #SE
#use UNIX
     if {unix.args.length[] = 0} then {
          ## execute echo and exit in sequence,
          ## using the UNIX extension operation
          ## =>, (sequence)
          unix.echo["Usage: " + unix.args[0] +
             " name"] =>
          unix.exit[1]
     } else {
          fob {
               userInput
               val { unix.args[1] }
               ret {
                    ## use the UNIX package
                    ## operator || to perform
                    ## the UNIX pipe operation
                    ## on ls and grep, and use
                    ## the sequence operator to
                    ## follow this with an exit.
                    unix.ls[] || unix.grep["i",
                       userInput]
                       =>
                    unix.exit[0]
               } \
          }[]
     }
#.
#!
```

This script begins with two directives that inform the FOBS preprocessor that the standard and UNIX extensions are being used. The UNIX extension makes available the keyword `unix`, that is a con-

venience definition that allows the user to use this simple keyword, rather than the full specification, `FOBS.UNIX`.

Another notation defined in the UNIX extension is the operator "=>", which might be called the *sequence* operation. This operator is used to interact with UNIX, which is stateful, using the FOBS computational model, which is stateless. In UNIX, operations are performed in sequence, and although they return values, they are usually performed for their side effects. The sequence operator takes as operands two FOBs representing UNIX commands, performs them in sequence, alters the UNIX environment, and returns the return value of the last command as a FOB. The operator implements the chaining technique, discussed in Section 8.3.

A last notation used in the example is the operation "||". This is also part of the UNIX extension, and implements the UNIX pipe operation.

As a universal scripting language, FOBS will often be required to interact with stateful environments. The FOBS library gives FOBS that ability, although such interaction diminishes the referential transparency of the language. To ameliorate the situation, the library is structured to isolate all operations with side effects in the FOB *FOBS*.

10. Conclusion

We have briefly described a core FOBS-X language. This language is designed as the basis of a universal scripting language. It has a simple syntax and semantics.

FOBS-X is a hybrid language, which combines the tools and features of object oriented languages with the tools and features of functional languages. In fact, the defining data structure of FOBS is a combination of an object and a function. The language provides the advantages of referential transparency, as well as the ability to easily build structures that encapsulate data and behavior. This provides the user the choice of paradigms.

Core-FOBS-X is the core of an extended language, SE-FOBS-X, in which programs are translated into the core by a macro processor. This allows for a language with syntactic sugar, that still has the simple semantics of our core-FOBS-X language.

Because of the ability to be extended, which is utilized by SE-FOBS-X, the FOBS-X language gains the flexibility that enables it to be a universal scripting language. The language can be adapted syntactically, using the macro capability, to new scripting applications. The Extension FEDELE allows the semantics of the language to be adapted to new applications. FEDELE makes the process of extending the library easier, by automatically generating new library modules from a high-level specification language.

We are currently working on developing extensions for various scripting environments. Our next project is to produce a UNIX extension. Further in the future we plan to investigate using FOBS for web scripting applications.

References

[1] A. Alexandrescu *The D Programming Language*. Adison Wesley. 2010

[2] M. Beaven, R. Stansifer, D. Wetlow, : *A Functional Language with Classes*. In: Lecture Notices in Computer Science, Springer Verlag, 507. 1991

[3] D. Beazley, G. Van Rossum: *Python; Essential Reference*. New Riders Publishing, Thousand Oaks, CA. 1999

[4] J. Gil de Lamadrid, J. Zimmerman. *Core FOBS: A Hybrid Functional and Object-Oriented Language*. In: Computer Languages, Systems & Structures, 38. 2012

[5] J. Gil de Lamadrid. *Combining the Functional and Object-Oriented Paradigms in the FOBS-X Scripting Language*. In: International Journal of Programming Languages and Applications, AIRCC, Vol. 3, No. 2, Oct. 2013

[6] J. A. Goguen, J. Mesegner. *Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics*. In: Research Directions in Object-Oriented Programming, MIT Press, pp. 417-478. 1987

[7] S. C, Johnson. *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories. 2014

[8] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Remy, J. Vouillon. *The OCaml System Release 4.00: Documentation and Users Manual*. Institut National de Recherche en Informatique et en Automatique. 2012

[9] M. Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Addison Wesley, pp. 327-336. 2000

[10] S. L. Peyton Jones, P. Wadler. *Imperative Functional Programming*. POPL, Charleston, Jan, 1993

[11] S. S. Yau, X. Jia, D. H. Bae. *Proof: A Parallel Object-Oriented Functional Computation Model*. In: Journal of Parallel Distributed Computing, 12. 1991

[12] M. Odersky, L. Spoon, B. Venners. *Programming in Scala* , Artima, Inc. 2008

[13] T. Walid, T Sheard. *MetML and Multi-stage Programming with Explicit Annotations*, In: Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulation, pp. 203-217, Amsterdam, NL. 1997

Source-to-Source Compilation in Racket

You Want it in *Which* Language?

Tero Hasu
BLDL and University of Bergen
tero@ii.uib.no

Matthew Flatt
PLT and University of Utah
mflatt@cs.utah.edu

Abstract

Racket's macro system enables language extension and definition primarily for programs that are run on the Racket virtual machine, but macro facilities are also useful for implementing languages and compilers that target different platforms. Even when the core of a new language differs significantly from Racket's core, macros offer a maintainable approach to implementing a larger language by desugaring into the core. Users of the language gain the benefits of Racket's programming environment, its build management, and even its macro support (if macros are exposed to programmers of the new language), while Racket's syntax objects and submodules provide convenient mechanisms for recording and extracting program information for use by an external compiler. We illustrate this technique with Magnolisp, a programming language that runs within Racket for testing purposes, but that compiles to C++ (with no dependency on Racket) for deployment.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software; D.3.4 [*Programming Languages*]: Processors

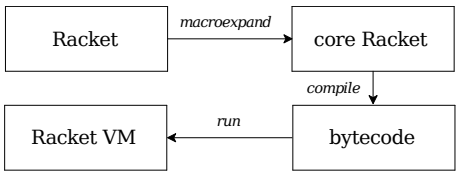
General Terms Design, Languages

Keywords Compiler frameworks, language embedding, macro systems, module systems, syntactic extensibility

1. Introduction

The Racket programming language (Flatt and PLT 2010) builds on the Lisp tradition of language extension through compile-time transformation functions, a.k.a. *macros*. Racket macros not only support language *extension*, where the existing base language is enriched with new syntactic forms, but also language *definition*, where a completely new language is implemented though macros while hiding or adapting the syntactic forms of the base language.

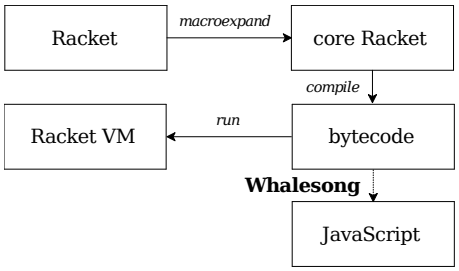
Racket-based languages normally target the Racket virtual machine (VM), where macros expand to a core Racket language, core Racket is compiled into bytecode form, and then the bytecode form is run:



The macro-expansion step of this chain is an example of a *source-to-source compiler* (or *transcompiler* for short), i.e., a translator that takes the source code in one language and outputs source code of another language. Transcompilers have potential benefits compared with compilation to machine code, such as more economical cross-platform application development by targeting widely supported languages, which enables the building of executables with various platform-vendor-provided toolchains.

A Racket-based language can also benefit by avoiding a runtime dependency on the Racket VM. Breaking the dependency can sometimes ease deployment, as the Racket VM is not well supported in every environment. Furthermore, for a mobile "app" to be distributed in an "app store," for example, it is desirable to keep startup times short and in-transit and in-memory footprints low; even in a stripped-down form, Racket can add significantly to the size of an otherwise small installation package. Factors relating to app-store terms and conditions and submission review process may also mean that avoiding linking in additional runtimes may be sensible or even necessary.

One example of an existing source-to-source compiler that avoids the Racket VM is Whalesong (Yoo and Krishnamurthi 2013), which compiles Racket to JavaScript via Racket bytecode:

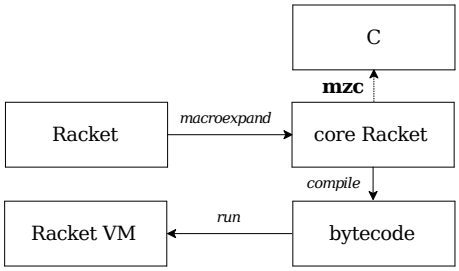


In this approach, a number of optimizations (such as inlining) are performed for bytecode by the normal Racket compiler, making it a sensible starting point for transcompilers that aim to implement variants of Racket efficiently.

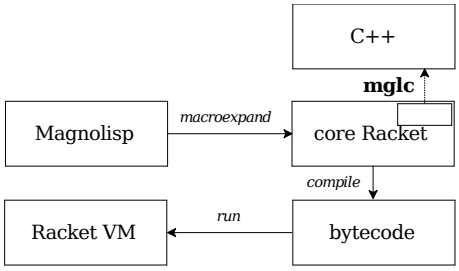
Compiling via Racket bytecode may be less appropriate when the language being compiled is not Racket or when optimizing for properties other than efficiency. Racket's bytecode does not retain all of the original (core) syntax, making it less suitable for implementing semantics-retaining compilation that happens primarily at the level of abstract syntax.

Thus, depending on the context, it may make more sense to compile from macro-expanded core language instead of bytecode.[1] Scheme-to-C compilers (e.g., CHICKEN and Gambit-C) typically work that way, as does the old `mzc` compiler for Racket:

[1] In Racket, one can acquire core language for a Racket source file by `read-syntax`ing the file contents and then invoking `expand` on the read (top-level) forms.



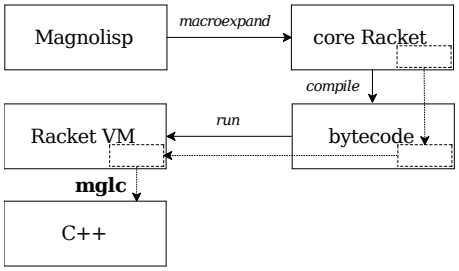
To try out a different Racket-exploiting transcompilation pipeline, we implemented a to-C++ compiler for a small programming language named *Magnolisp*. Conceptually, the Magnolisp implementation is like a Scheme-to-C compiler, but instead of handling all of Racket, it handles only a particular subset of Racket that corresponds to the expansion of the Magnolisp macros (although the "subset" includes additional macro-introduced annotations to guide compilation to C++). A traditional compilation pipeline for `mglc` (the Magnolisp compiler) would be the following, with the smaller box representing the part of the program to be transcompiled (additionally, there can be macro definitions, for example, which are not relevant when transcompiling):



However, directly manipulating core Racket S-expression syntax is not especially convenient from outside of the Racket pipeline. Racket's strength is in support for the macro-expansion phase, especially its support for multiple modules and separate compilation at the module level. It can be useful to be able to do back-end-specific work in macro expansion. In the Magnolisp case, such work includes recording type annotations and catching syntax errors early, for the benefit of `mglc`.

Magnolisp demonstrates a source-to-source compilation approach that takes advantage of the macro-expansion phase to prepare for "transcompile time."[2] More precisely, Magnolisp arranges for macro expansion to embed into the Racket program the information that the Magnolisp compiler needs. The compiler extracts the information by running the program in a mode in which transcompile-time code is evaluated. This results in the following, distinctly non-traditional compilation pipeline; here, the smaller boxes still correspond to the part of the program that is transcompiled, but they now denote code that encodes the relevant information about the program, and only gets run in the transcompile-time mode (as depicted by the longer arrow of the "run" step):

[2] Our use of the word "time" here refers to the idea behind Racket's submodules (Flatt 2013), which is to make it possible for programmers to define new execution phases beyond the macro-expansion and run-time phases that are built into the language model. In our case we want to introduce a transcompile-time phase, and designate some of the code generated during macro expansion as belonging to that phase. In practice, this is done by putting said code into a separately loadable module within a module, i.e. a submodule of a known name within the module whose transcompile-time code it is.



In essence, this strategy works because Racket is already able to preserve syntactic information in bytecode, so that Racket can implement separately compiled macros. Recent generalizations to the Racket syntax system—notably, the addition of submodules (Flatt 2013)—let us conveniently exploit that support for Magnolisp compilation.

The information that Magnolisp makes available (via a submodule) for compilation consists of abstract syntax trees (which incorporate any compilation-guiding annotations), along with some auxiliary data structures. As the particular abstract syntax is only for compilation, it need not conform to the Racket core language semantics; indeed, Magnolisp deviates from Racket semantics where deemed useful for efficient and straightforward translation into C++.

Even for a language that is primarily designed to support transcompilation, it can be useful to also have an evaluator. We envision direct evaluation being useful for simulating the effects of compiled programs, probably with "mock" implementations of primitives requiring functionality that is not available locally. The idea is to gain some confidence that programs (or parts thereof) work as intended before actually compiling them. Cross-compilation and testing on embedded devices can be particularly time consuming; compilation times generally pale in comparison to the time used to transfer, install, and launch a program.

The usual way of getting a Racket-hosted language to evaluate is to macro transform its constructs into the Racket core language, for execution in the Racket VM. Having macro-expansion generate separately loadable transcompile-time code does mean that it could not also generate evaluatable Racket run-time definitions. Magnolisp demonstrates this by supporting both transcompilation and Racket-VM-based evaluation.

1.1 Motivation for Racket-Hosted Transcompiled Languages

Hosting a language via macros offers the potential for extensibility in the hosted language. This means leveraging the host language both to provide a language extension mechanism and a language for programming any extensions. While a basic language extension mechanism (such as the C preprocessor or a traditional Lisp macro system) may be implementable with reasonable effort, safer and more expressive mechanisms require substantial effort to implement from scratch. Furthermore, supporting programmable (rather than merely substitution based) language extensions calls for a compile-time language evaluator, which may not be readily available for a transcompiled language.

Hosting in Racket offers safety and composability of language extensions through lexical scope and phase separation respecting macro expansion. Racket macros' hygiene and referential transparency help protect the programmer from inadvertent "capturing" of identifiers, making it more likely that constructs defined modularly (or even independently) compose successfully. *Phase separation* (Flatt 2002) means that compile time and run time have distinct bindings and state. The separation in the time dimension is particularly crucial for a transcompiler, as it must be possible to parse code without executing it. The separation of bindings, in turn, helps achieve language separation, in that one can have Racket bindings

in scope for compile-time code, and hosted-language bindings for run-time code.

Racket's handling of modules can also be leveraged to support modules in the hosted language, with Racket's `raco make` tool for rebuilding bytecode then automatically serving as a build tool for multi-module programs in the language. The main constraint is that Racket does not allow cycles among module dependencies.

Particularly for new languages it can be beneficial to reuse existing language infrastructure. With a Racket embedding one is in the position to reuse Racket infrastructure on the front-end side, and the target language's infrastructure (typically libraries) on the back-end side. Reusable front-end side language tools might include IDEs (Findler et al. 2002), documentation tools (Flatt et al. 2009), macro debuggers (Culpepper and Felleisen 2007), etc. Although some tools might not be fully usable with programs that cannot be executed as Racket, the run vs. compile time phase separation means that a tool whose functionality does not entail running a program should function fully.

Racket's language extension and definition machinery may be useful not only for users, but also for language implementors. Its macros have actually become a compiler front end API that is sufficient for implementing many general-purpose abstraction mechanisms in a way that is indistinguishable from built-in features (Culpepper and Felleisen 2006). In particular, a basic "sugary" construct is convenient to implement as a macro, as both surface syntax and semantics can be specified in one place.

1.2 Contributions

The main contributions of this paper are:

- we describe a generic approach to replacing the runtime language of Racket in such a manner that information about code in the language can be processed at macro-expansion time, and selectively made available for separate loading for purposes of source-to-source compilation to another high-level language;
- we show that the core language and hence the execution semantics of such a source-to-source compiled language can differ from Racket's;
- we suggest that it may be useful to also make a transcompiled language executable as Racket, and show that this is possible at least for our proof-of-concept language implementation; and
- we show that this approach to language implementation allows Racket's expressive macro and module systems to be reused to make the implemented language user extensible, and to make the scope of language extensions user controllable.

Some of the presented language embedding techniques have been previously used in the implementation of Dracula, to allow for compilation of Racket-hosted programs to the ACL2 programming language; they have remained largely undocumented, however.

The significance of the reported approach beyond the Racket ecosystem is that it supports transcompiler implementation for languages that have all three of the following properties:

- the language is extensible from within itself;
- the scope of each language extension can be controlled separately, also from within the language; and
- there are some guarantees of independently defined extensions composing safely.[3]

[3] In the case of Racket, macros that do not explicitly capture free variables are safe to compose in the limited sense that they preserve the meaning of variable bindings and references during macro expansion (Eastlund and Felleisen 2010).

2. Magnolisp

Magnolisp is a proof-of-concept implementation of a Racket-hosted transcompiled language, and the running example that we use to discuss the associated implementation techniques. As a language, Magnolisp is not exceptional in being suitable for hosting; the techniques described in this paper constitute a general method for hosting a transcompiled language in Racket.

Code and documentation for Magnolisp is available at:

`https://www.ii.uib.no/~tero/magnolisp-ifl-2014/`

2.1 The Magnolisp Language

To help understand the Magnolisp-based examples given later, we give some idea of the syntax and constructs of the language. We assume some familiarity with Racket macro syntax.

Magnolisp is significantly different from Racket in that its overriding design goal is to be amenable to static reasoning; Racket compatibility, for better reuse of Racket's facilities, is secondary. Magnolisp disallows many forms of runtime-resolved dispatch of control that would make reasoning about code harder. Unlike in Racket, all data types and function invocations appearing in programs are resolvable to specific implementations at compile time.

Requiring fully, statically typed Magnolisp programs facilitates compilation to C++, as the static types can be mapped directly to their C++ counterparts. To reduce syntactic clutter due to annotations, and hence to help retain Racket's untyped "look and feel," Magnolisp features whole-program type inference à la Hindley-Milner.

Magnolisp reuses Racket's module system for managing names internally within programs (or libraries), both for run-time names and macros. The exported C++ interface is defined separately through `export` annotations appearing in function definitions; only `export`ed functions are declared in the generated C++ header file.

The presented language hosting approach involves the definition of a Racket language for the hosted language. The Racket language for Magnolisp is named `magnolisp`. At the top-level of a module written in `magnolisp`, one can declare `function`s, for example. A function may be marked `foreign`, in which case it is assumed to be implemented in C++; such a function may also have a Racket implementation, given as the body expression, to also allow for running in the Racket VM. Types can only be defined in C++, and hence are always `foreign`, and `typedef` can be used to give the corresponding Magnolisp declarations. The `type` annotation is used to specify types for functions and variables, and the type expressions appearing within may refer to declared type names. The `#:annos` keyword is used to specify the set of annotations for a definition.

In this example, `get-last-known-location` is a Magnolisp function of type `(fn Loc)`, i.e., a function that returns a value of type `Loc`. The `(rkt.get-last-known-location)` expression in the function body might be a call to a Racket function from module `"positioning.rkt"`, simulating position information retrieval:

```
#lang magnolisp
(require (prefix-in rkt. "positioning.rkt"))

(typedef Loc (#:annos foreign))

(function (get-last-known-location)
  (#:annos foreign [type (fn Loc)])
  (rkt.get-last-known-location))
```

No C++ code is generated for the above definitions, as they are both declared as `foreign`. For an example that does have a C++ translation, consider the following code, which introduces Magno-

lisp's predefined `predicate` type for boolean values, `var`iable declarations, `if` expressions and statements, and `do` and `return` constructs. The latter two are an example of language that does not directly map into Racket core language; the `do` expression contains a sequence of statements, with any executed `return` statement determining the value of the overall expression. Magnolisp syntax is not particularly concise, but shorthands can readily be defined, as is here demonstrated by the `declare-List-op` macro for declaring primitives that accept a `List` argument:

```
#lang magnolisp
; list and element data types (defined in C++)
(typedef List (#:annos foreign))
(typedef Elem (#:annos foreign))

(define-syntax-rule (declare-List-op [n t] ...)
  (begin (function (n lst)
           (#:annos [type (fn List t)] foreign))
         ...))

; list and element primitives (implemented in C++)
(declare-List-op [empty? predicate]
                 [head Elem]
                 [tail List])
(function (zero)
  (#:annos [type (fn Elem)] foreign))
(function (add x y)
  (#:annos [type (fn Elem Elem Elem)] foreign))

; sum of first two list elements
; (or fewer for shorter lists)
(function (sum-2 lst) (#:annos export)
  (if (empty? lst)
      (zero)
      (do (var h (head lst))
          (var t (tail lst))
          (if (empty? t)
              (return h)
              (return (add h (head t)))))))
```

The transcompiler-generated C++ implementation for the `sum-2` function is the following (but hand-formatted for readability). The translation is verbose, and could be simplified with additional optimizations; its redeeming quality is that it closely reflects the structure of the original code, which was made possible by our use of GCC-specific C++ language extensions (e.g., "statement expressions"):

```
MGL_API_FUNC Elem sum_2(List const& lst)
{
  return (is_empty(lst)) ? (zero()) :
    (({ __label__ b;
        Elem r;
        {
          Elem h = head(lst);
          {
            List t = tail(lst);
            if (is_empty(t))
              { r = h; goto b; }
            else
              { r = add(h, head(t));
                goto b; }
          }
        }
        b: r;
      }));
}
```

2.2 Magnolisp Implementation

The collection of techniques for embedding a transcompiled language within Racket, as described in this paper, only concern the front end of a transcompiler. Wildly differing designs for the rest of the compilation pipeline are possible; we merely sketch the structure of our Magnolisp-to-C++ compiler as a concrete example.

Magnolisp is implemented in Racket, and in a way there are two implementations of the language: one targeting the Racket VM, and one targeting C++. The `magnolisp` Racket language has the dual role of defining execution semantics for the Racket VM, and also effectively being the front end for the transcompiler.

Figure 1 shows an overview of the transcompiler architecture, including both the `magnolisp`-defined front end, and the `mglc`-driven middle and back ends. One detail omitted from the figure is that the macro-expanded `"a.rkt"` module gets compiled before it or any of its submodules are evaluated; if this is not done ahead of time, with the result serialized into a file as bytecode, it will get done on demand by Racket when the for-transcompile-time submodule is accessed.

Figure 2 illustrates the forms of data running through the compilation pipeline. The `"a.rkt"` module's transcompile-time code gets run when its `magnolisp-s2s` submodule gets *instantiated*, which means that variables are created for module-level definitions. Transcompilation triggers instantiation by invoking `dynamic-require` to fetch values for said variables (e.g., `def-lst`); the values describe `"a.rkt"`, and are already in the compiler's internal data format. Any referenced dependencies of `"a.rkt"` (e.g., `"num-types.rkt"`) are processed in the same manner, and the relevant definitions are incorporated into the compilation result (i.e., `"a.cpp"` and `"a.hpp"`).

The middle and back ends may be accessed via the `mglc` command-line tool, or programmatically via the underlying API. The expected input for these is a set of modules for transcompiling to C++. The compiler loads any transcompile-time code in the modules and their dependencies. Dependencies are determined by inspecting binding information for appearing identifiers, as resolved by Racket during macro expansion. Any modules with a `magnolisp-s2s` submodule are assumed to be Magnolisp, but other Racket-based languages may also be used for macro programming or simulation. The Magnolisp compiler effectively ignores any code that is not run-time code in a Magnolisp module.

The program transformations performed by the compiler are generally expressed in terms of term rewriting strategies. These are implemented based on a custom strategy combinator library inspired by Stratego (Bravenboer et al. 2008). The syntax trees prepared for the transcompilation phase use data types supporting the primitive strategy combinators that the combinator library expects.

The compiler middle end implements whole-program optimization (by dropping unused definitions), type inference, and some simplifications (e.g., removal of condition checks where the condition is `(TRUE)` or `(FALSE)`). The back end implements translation from Magnolisp core to C++ syntax (including e.g. lambda lifting), C++-compatible identifier renaming, splitting of code into sections (e.g.: public declarations, private declarations, and private implementations), and pretty printing.

3. Hosting a Transcompiled Language in Racket

Building a language in Racket means defining a module or set of modules to implement the language. The language's modules define and export macros to compile the language's syntactic forms to core forms. In the case of a transcompiled language, the expansion of the language's syntactic forms might produce nested submodules to separate code than can be run directly in the Racket VM from information that is used to continue compilation to a different target.

In this section, we describe some of the details of that process for some transcompiled language L. Where the distinction matters, we use L_R to denote a language intended to also run in the Racket VM (possibly with mock implementations of some primitives), and

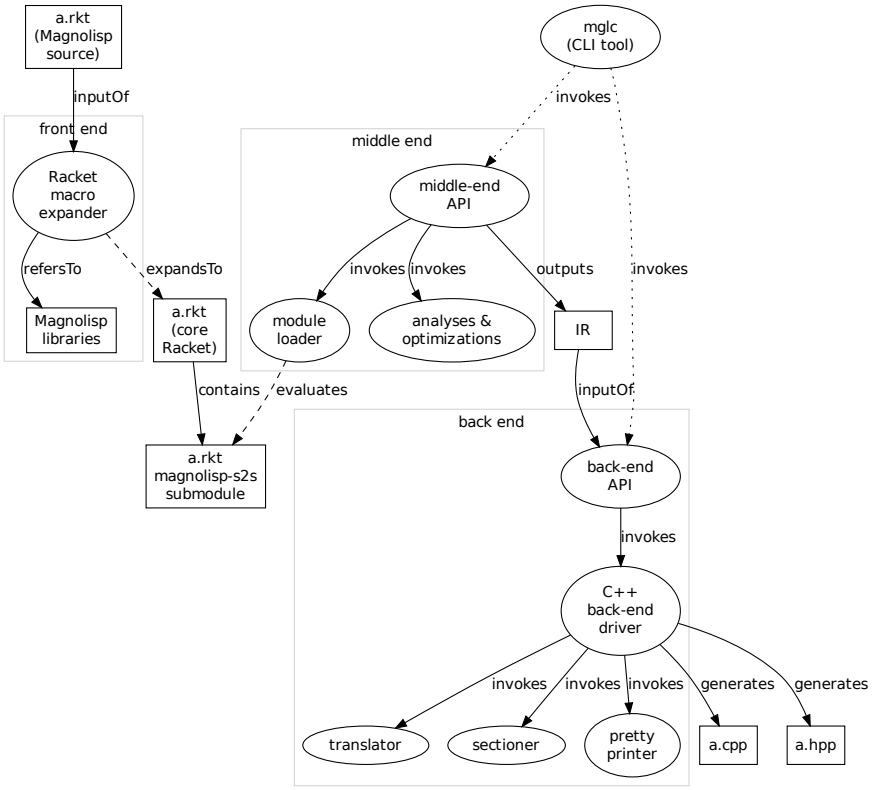


Figure 1: The overall architecture of the Magnolisp compiler, showing some of the components involved in compiling a Magnolisp source file `"a.rkt"` into a C++ implementation file `"a.cpp"` and a C++ header file `"a.hpp"`. The dotted arrows indicate that the use of the `mglc` command-line tool is optional; the middle and back end APIs may also be invoked by other programs. The dashed ‘‘evaluates’’ arrow indicates a conditional connection between the left and right hand sides of the diagram; the `magnolisp-s2s` submodule is *only* loaded when transcompiling. The ‘‘expandsTo’’ connection is likewise conditional, as `"a.rkt"` may have been compiled ahead of time, in which case the module is already available in a macro-expanded form.

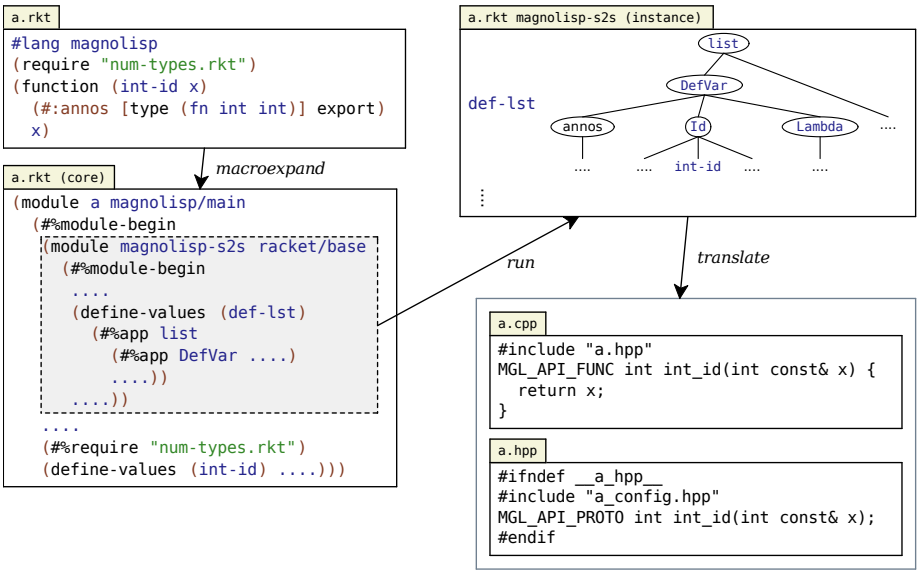


Figure 2: An illustration of the processing of a Magnolisp module as it passes through the compilation pipeline. Transcompile-time Racket code is shown in a dashed box.

L_C to denote a language that only runs through compilation into a different language.

3.1 Modules and `#lang`

All Racket code resides within some *module*, and each module starts with a declaration of its *language*. A module's language declaration has the form `#lang` L as the first line of the module. The remainder of the module can access only the syntactic forms and other bindings made available by the language L.

A language is itself implemented as a module.[4] A language module is connected to the name L—so that it will be used by `#lang` L—by putting the module in a particular place in the filesystem or by appropriately registering the module's parent directory.

In general, a language's module provides a *reader* that gets complete control over the module's text after the `#lang` line. A reader produces a *syntax object*, which is kind of S-expression (that combines lists, symbols, etc.) that is enriched with source locations and other lexical context. We restrict our attention here to a language that uses the default reader, which parses module content directly as S-expressions, adding source locations and an initially empty lexical context.

For example, to start the implementation of L such that it uses the default reader, we might create a `"main.rkt"` module in an `"L"` directory, and add a `reader` submodule that points back to `L/main` as implementing the rest of L:

```
#lang racket
(module reader syntax/module-reader L/main)
```

The S-expression produced by a language's reader serves as input to the macro-expansion phase. A language's module provides syntactic forms and other bindings for use in the expansion phase by exporting macros and variables. A language L can re-export all of the bindings of some other language, in which case L acts as an extension of that language, or it can export an arbitrarily restrictive set of bindings.

For example, if `"main.rkt"` re-exports all of `racket`, then `#lang L` is just the same as `#lang racket`:

```
#lang racket
(module reader syntax/module-reader L/main)
(provide (all-from-out racket))
```

A language must at least export a macro named `#%module-begin`, because said macro implicitly wraps the body of a module. Most languages simply use `#%module-begin` from `racket`, which treats the module body as a sequence of `require` importing forms, `provide` exporting forms, definitions, expressions, and nested modules, where a macro use in the module body can expand to any of the expected forms. A language might restrict the body of modules by either providing an alternative `#%module-begin` or by withholding other forms. A language might also provide a `#%module-begin` that explicitly expands all forms within the module body, and then applies constraints or collects information in terms of the core forms of the language.

For example, the following `"main.rkt"` re-exports all of `racket` except `require` (and the related core language name `#%require`), which means that modules in the language L cannot import other modules. It also supplies an alternate `#%module-begin` macro to pre-process the module body in some way:

```
#lang racket
(module reader syntax/module-reader L/main)
(provide
  (except-out (all-from-out racket)
              require #%require #%module-begin)
  (rename-out [L-module-begin #%module-begin]))
(define-syntax L-module-begin ....)
```

[4] Some language must be predefined, of course. For practical purposes, assume that the `racket` module is predefined.

The language definition facilities described so far are general, and useful regardless of whether the language is transcompiled or not. We now proceed to provide the specifics of this paper's transcompiled language implementation approach. In it, the `#%module-begin` macro in particular plays a key role, and overall, a Racket language L that is intended for transcompilation is defined as follows:

- Have the language's module export bindings that define the surface syntax of the language. The provided bindings should expand only to transcompiler-supported run-time forms. We describe this step further in section 3.2
- Where applicable, have macros record additional metadata that is required for transcompilation. We describe this step further in section 3.3
- Have the `#%module-begin` macro fully expand all the macros in the module body, so that the rest of the transcompiler pipeline need not implement macro expansion. We describe this step further in section 3.4
- After full macro expansion, have `#%module-begin` add externally loadable information about the expanded module into the module. We describe this step further in section 3.5
- Provide any run-time support for running programs alongside the macros that define the syntax of the language. We describe this step further in section 3.6

The export bindings of L may include variables, and the presence of transcompilation introduces some nuances into their meaning. When the meaning of a variable in L is defined in L, we say that it is a *non-primitive*. When its meaning is defined in the execution language, we say that it is a *primitive*. When the meaning of its appearances is defined by a compiler of L, we say that it is a *built-in*. As different execution targets may have different compilers, what is a built-in for one target may be a primitive for another. It is typically not useful to have built-ins for the Racket VM target, for which the `#%module-begin` macro may be considered to be the "compiler."

3.2 Defining Surface Syntax

To define the surface syntax of a language L, its implementation module's exports should ideally name a variable of L, a core language construct of L, or a macro that expands only to those constructs. Where the core language is a subset of Racket's, it should be ensured that only the transcompiler-supported set appears in an expansion. Where the core of L is a superset of Racket, the additional constructs should be encoded in terms of Racket's core forms.

Possible strategies for encoding foreign code forms include:

- **E1**. Use a variable binding to identify a core-language form. Use it in application position to allow other forms to appear within the application form. Subexpressions within the form can be delayed with suitable `lambda` wrappers, if necessary.
- **E2**. Attach information to a syntax object through its *syntax property* table; macros that manipulate syntax objects must propagate properties correctly.
- **E3**. Store information about a form in a compile-time table that is external to the module's syntax objects.
- **E4**. Use Racket core forms that are not in L (not under their original meaning), or combinations of forms involving such forms.

A caveat for **E2** and **E3** is that both syntax properties and compile-time tables are transient, and they generally become unavailable after a module is fully expanded, so any information to be preserved must be reflected as generated code in the expansion of the module; we describe this step further in section 3.5. Another caveat of such "out-of-band" storage locations is that where the stored data includes identifiers, one must beware of extracting identifiers out of syntax objects too early; if the identifier is in the scope of a binding form, then the binding form must be first expanded so that the identifier will include information about the binding.

In the case of L_R there are additional constraints to encoding foreign core forms, since the result of a macro-expansion should be compatible with both the transcompiler and the Racket evaluator. The necessary duality can be achieved if the surface syntax defining macros can adhere to these constraints: (**C1**) exclude Racket core form uses that are not supported by the compiler; (**C2**) add any compilation hints to Racket core forms in a way that does not affect evaluation (e.g., as custom syntax properties); and (**C3**) encode any transcompilation-specific syntax in terms of Racket core forms which only appear in places where they do not affect Racket execution semantics.

Where the constraints **C1**–**C3** are troublesome, the fallback option is to have `#%module-begin` rewrite either the run-time code, transcompile-time code, or both, to make the program conform to expected core language. Such rewriting may still be constrained by the presence of binding forms, however.

The principal constraint on encoding a language's form is that a binding form in L should be encoded as a binding form in Racket, because bindings are significant to the process of hygienic macro expansion. Operations on a fully expanded module's syntax objects, furthermore, can reflect the accumulated binding information, so that a transcompiler may possibly avoid having to implement its own management of bindings. For the cases where a language's forms do not map neatly to a Racket binding construct, Racket's macro API supports explicit *definition context*s (Flatt et al. 2012), which enable the implementation of custom binding forms that cooperate with macro expansion.

For an example of foreign core form encoding strategy **E1**, consider an L_C with a `parallel` construct that evaluates two forms in parallel. Said construct might be defined simply as a "dummy" constant, recognized by the transcompiler as a specific built-in by its identifier, translating any appearances of `(parallel e1 e2)` "function applications" appropriately:

```
(define parallel #f)
```

Alternatively, as an example of strategy **E2**, L_C's `(parallel e1 e2)` form might simply expand to `(list e1 e2)`, but with a `'parallel` syntax property on the `list` call to indicate that the argument expressions are intended to run in parallel:

```
(define-syntax (parallel stx)
  (syntax-case stx ()
    [(parallel e1 e2)
     (syntax-property #'(list e1 e2)
                      'parallel #t)]))
```

For L_R, `parallel` might instead be implemented as a simple pattern-based macro that wraps the two expressions in `lambda` and passes them to a `call-in-parallel` runtime function, again in accordance to strategy **E1**. The `call-in-parallel` variable could then be treated as a built-in by the transcompiler, and implemented as a primitive for running in the Racket VM:

```
(define-syntax-rule (parallel e1 e2)
  (call-in-parallel (lambda () e1) (lambda () e2)))
```

An example of adhering to constraint **C3** is the definition of Magnolisp's `typedef` form, for declaring an abstract type. A declared type `t` is bound as a variable to allow Racket to resolve type references; these bindings also exist for evaluation as Racket, but they are never referenced at run time. The `#%magnolisp` built-in is used to encode the meaning of the variable, but as `#%magnolisp` has no useful definition in Racket, the evaluation of any `(#%magnolisp ....)` expressions is prevented. The `CORE` macro is a convenience for wrapping `(#%magnolisp ....)` expressions in an `(if #f .... #f)` form to "short-circuit" the overall expression so as to make it obvious to the Racket bytecode optimizer that the enclosed expression is never evaluated as Racket. The `let/annotate` form is a macro that stores the annotations `a ...`, which might e.g. include the name of `t`'s C++ definition.

```
(define #%magnolisp #f)
(define-syntax-rule (CORE kind arg ...)
  (if #f (#%magnolisp kind arg ...) #f))

(define-syntax-rule (typedef t (#:annos a ...))
  (define t
    (let/annotate (a ...)
      (CORE 'foreign-type))))
```

Using a macro system for syntax definition offers several advantages compared to parsing in a more traditional way:[5]

- Where it is sufficient to define custom syntactic forms as macros, parsing is almost "for free." At the same time, the ability to customize a language's reader makes it possible for surface syntax not to be in Lisp's parenthesized prefix notation.
- Macros and the macro API provide a convenient implementation for "desugaring" and other rewriting-based program transformations. Such transformations can be written in a modular and composable way.
- For making L macro extensible, its implementation can simply expose a selection of relevant Racket constructs (directly or through macro adapters) to enable the inclusion of compile-time code within L modules.

3.3 Storing Metadata

We use the term *metadata* to mean data that describes a syntax object, but is not itself a core syntactic construct in the implemented language. Such data may encode information (e.g., optimization hints) that is meaningful to a transcompiler or other kinds of external tools. Some metadata may be collected automatically by the language infrastructure (e.g., source locations in Racket), some might be inferred by L's macros at expansion time, and some might be specified as explicit *annotations* in source code (e.g., the `export` annotation of Magnolisp functions, or the `weak` modifier of variables in the Vala language).

There is no major difference between encoding foreign syntax in terms of Racket core language, or encoding metadata; the strategies **E1**–**E4** apply for both. The main way in which metadata differs is that it does not tend to appear as a node of its own in a syntax tree. Any annotations in L do have surface syntax, but no core syntax, and hence they disappear during macro expansion; they do appear explicitly in unexpanded code, but such code cannot in general be directly analyzed, as unexpanded L code cannot be parsed. A more workable strategy is to have L's syntactic forms store any necessary metadata during macro expansion.

For metadata, storage in syntax properties is a typical choice. Typed Racket, for example, stores its type annotations in the a custom `'type-annotation` syntax property (Tobin-Hochstadt et al. 2011).

[5] A macro's process of validating and destructuring its input syntax can also be regarded as parsing, even though the input is syntax objects rather than raw program text or token streams (Culpepper 2012).

Compile-time tables are another likely option for metadata storage. For storing data for a named definition, one might use an *identifier table*, which is a dictionary data structure where each entry is keyed by an identifier. An *identifier*, in turn, is a syntax object for a symbol. Such a table is suitable for both local and top-level bindings, because the syntax object's lexical context can distinguish different bindings that have the same symbolic name. Magnolisp$_{it}$, a variant implementation of Magnolisp, uses an identifier table for metadata storage. Magnolisp$_{it}$ exports an `anno!` macro, which may be used to annotate an identifier, and is used internally e.g. by `function` and `typedef`. It is strictly a compile-time construct, and has no corresponding core syntax. Its advantage is that it may be used to post-facto annotate an already declared binding:

```
#lang magnolisp
(typedef int (#:annos foreign))
; MGL_API_FUNC int id(int const& x) { return x; }
(function (id x) x)
(anno! id [type (fn int int)] export)
```

It is also possible to encode annotations in the syntax tree proper, which has the advantage of fully subjecting annotations to macro expansion. Magnolisp adopts this approach for its annotation recording, using a special `'annotate`-property-flagged `let-values` form to contain annotations. Each contained annotation expression `a` is encoded as `(if #f (#%magnolisp 'anno ....) #f)` to prevent evaluation at Racket run time, and e.g. `[type ....]` expands to such a form, via the intermediate `CORE` form given in section 3.2:

```
(define-syntax-rule (type t) (CORE 'anno 'type t))

(define-syntax (let/annotate stx)
  (syntax-case stx ()
    [(_ (a ...) e)
     (syntax-property
      (syntax/loc stx
        (let-values ([() (begin a (values))] ...)
          e))
      'annotate #t)]))
```

The `let/annotate`-generated `let-values` forms introduce no bindings, and their right-hand-side expressions yield no values; only the expressions themselves matter. Where the annotated expression `e` is an initializer expression, the Magnolisp compiler decides which of the annotations are actually associated with the initialized variable.

3.4 Expanding Macros

One benefit of reusing the Racket macro system with *L* is to avoid having to implement an *L*-specific macro system. When the Racket macro expander takes care of macro expansion, the remaining transcompilation pipeline only needs to understand *L*'s core syntax (and any related metadata). Racket includes two features that make it possible to expand all the macros in a module body, and afterwards process the resulting syntax, all within the language.

The first of these features is the `#%module-begin` macro, which can transform the entire body of a module. The second is the `local-expand` (Flatt et al. 2012) function, which may be used to fully expand all the `#%module-begin` sub-forms. Using the two features together is demonstrated by the following macro skeleton, which might be exported as the `#%module-begin` of a language:

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(module-begin form ...)
     (let ([ast (local-expand
                 #'(#%module-begin form ...)
                 'module-begin null)])
       (do-some-processing-of ast))]))
```

The `local-expand` operation also supports *partial* sub-form expansion, as it takes a "stop list" of identifiers that prevent descending into sub-expressions with a listed name. At first glance one might imagine exploiting this feature to allow foreign core syntax to appear in a syntax tree, and simply prevent Racket from proceeding into such forms. The main problem with this strategy is that foreign binding forms would not be accounted for in Racket's binding resolution. That problem is compounded if foreign syntactic forms can include Racket syntax sub-forms; the sub-forms need to be expanded along with enclosing binding forms. To prevent these problems, a stop list is automatically expanded to include all Racket core forms if it includes any form so that partial expansion is constrained to the consistent case that stays outside of binding forms.

3.5 Exporting Information to External Tools

After the `#%module-begin` macro has fully expanded the content of a module, it can gather information about the expanded content to make it available for transcompilation. The gathered information can be turned into an expression that reconstructs the information, and that expression can be added to the overall module body that is produced by `#%module-begin`.

The expression to reconstruct the information should *not* be added to the module as a run-time expression, because extracting the information for transcompilation would then require running the program (in the Racket VM). Instead, the information is better added as compile-time code. The compile-time code is then available from the module while compiling other *L* modules, which might require extra compile-time information about a module that is imported into another *L* module. More generally, the information can be extracted by running only the compile-time portions of the module, instead of running the module normally.

As a further generalization of the compile-time versus run-time split, the information can be placed into a separate *submodule* within the module (Flatt 2013). A submodule can have a dynamic extent (i.e., run time) that is unrelated to the dynamic extent of its enclosing module, and its bytecode may even be loaded separately from the enclosing module's bytecode. As long as a compile-time connection is acceptable, a submodule can include syntax-quoted data that refers to bindings in the enclosing module, so that information can be easily correlated with bindings that are exported from the module.

For example, suppose that *L* implements definitions by producing a normal Racket definition for running within the Racket virtual machine, but also needs a syntax-quoted version of the expanded definition to compile to a different target. The `module+` form can be used to incrementally build up a `to-compile` submodule that houses definitions of the syntax-quoted expressions:

```
(define-syntax (L-define stx)
  (syntax-case stx ()
   [(L-define id rhs)
    (with-syntax ([rhs2 (local-expand #'rhs
                                      'expression null)])
      #'(begin
          (define id rhs2)
          (begin-for-syntax
            (module+ to-compile
              (define id #'rhs2)))))]))
```

Wrapping `(module+ to-compile ...)` with `begin-for-syntax` makes the `to-compile` submodule reside at compilation time relative to the enclosing module, which means that loading the submodule will not run the enclosing module. Within `to-compile`, the expanded right-hand side is quoted as syntax using `#'`.

Syntax-quoted code is often a good choice of representation for code to be compiled again to a different target language, be-

cause lexical-binding information is preserved in a syntax quote. Certain syntax-quoting forms—such as `quote-syntax/keep-srcloc`—additionally preserve source locations for syntax objects, so that a compiler can report errors or warnings in terms of a form's original source location.

Another natural representation choice is to use any custom intermediate representation (IR) of the compiler. Magnolisp, for example, processes each Racket syntax tree already within the module where macro expansion happens, turning them into its IR format, which also incorporates metadata. The IR uses Racket `struct` instances to represent abstract syntax tree (AST) nodes, while still retaining some of the original Racket syntax objects as metadata, for purposes of transcompile-time reporting of semantic errors. Magnolisp programs are parsed at least twice, first from text to Racket syntax objects by the reader, and then from syntax objects to the IR by `#%module-begin`; additionally, any macros effectively parse syntax objects to syntax objects. As parsing is completed already in `#%module-begin`, any Magnolisp syntax errors are discovered even when just evaluating programs as Racket.

The `#%module-begin` macro of `magnolisp` exports the IR via a submodule named `magnolisp-s2s`; it contains an expression that reconstructs the IR, albeit in a somewhat lossy way, excluding detail that is irrelevant for compilation. The IR is accompanied by a table of identifier binding information indexed by module-locally unique symbols, which the transcompiler uses for cross-module resolution of top-level bindings, to reconstruct the identifier binding relationships that would have been preserved by Racket if exported as syntax-quoted code. As `magnolisp-s2s` submodules do not refer to the bindings of the enclosing module, they are loadable independently.

3.6 Run-Time Support

The modules implementing a Racket language may also define run-time support for executing programs. For L, such support may be required for the compilation target environment; for L_R, any support would also be required for the Racket VM. Run-time support for L is required when the macro expansion of L can produce code referring to run-time variables, or when L exports bindings to run-time variables.

Any non-primitive run-time support variables are by definition defined in L itself, with each definition thus also compilable for the target. When L includes specific language for declaring primitives, then it may be convenient to define any variables corresponding to primitives in L, with any associated annotations; for L_R one would additionally specify any Racket VM implementation, either in Racket or another Racket-VM-hosted language. For variables representing built-ins of L_C, one might just use dummy initial value expressions, as the expressions are not evaluated, and the meaning of the variables is known to the compiler.

The primary constraint in implementing run-time support is that the Racket module system does not allow cyclic dependencies. Strictly speaking, then, any runtime library exported by a Racket module L *cannot* itself be implemented in L, but must use a smaller language. The `magnolisp` language, for example, exports the `magnolisp/prelude` module, which declares all the primitives of the language; the language of `magnolisp/prelude` is `magnolisp/base`, which does not include any runtime library.

The `magnolisp` language only exports four variables: the `#%magnolisp` built-in, and the `TRUE`, `FALSE`, and `predicate` primitives. The primitives are "semi-built-ins" in that the compiler knows that conditional expressions must always be of type `predicate`, and that the nullary operations `TRUE` and `FALSE` yield "true" and "false" values, respectively; this knowledge is useful during type checking and optimization:

```
#lang magnolisp/base

(require "surface.rkt")
(provide predicate TRUE FALSE)
(typedef predicate (#:annos [foreign
                             mgl_predicate]))
(function (TRUE) (#:annos [foreign mgl_TRUE]
                          [type (fn predicate)])
  #t)
(function (FALSE) (#:annos [foreign mgl_FALSE]
                           [type (fn predicate)])
  #f)
```

4. Evaluation

We believe that the presented Racket-hosted transcompilation approach is quite generic, in theory capable of accommodating a large class of languages. In practice, however, we would imagine it mostly being used to host *new* languages, with suitable design compromises made to achieve a high degree of reuse of the Racket infrastructure. Also, while macros are useful for language implementation alone, we would expect Racket's support for creating macro-extensible languages to be a significant motivation for choosing Racket as the implementation substrate.

Racket hosting should be particularly appropriate for research languages, as macros facilitate quick experimentation with language features, and design constraints should be acceptable if they do not compromise the researchers' ability to experiment with the concepts that are under investigation.

4.1 Language Design Constraints

The two design constraints for enabling effective Racket reuse that we have discovered are the following: (1) the hosted language's name resolution must be compatible with Racket's; and (2) S-expression-based syntax must be chosen to directly and effectively reuse Racket's default parsing machinery and existing macro programming APIs. The compilation requirement, in turn, may introduce constraints to the choice of core language, especially where one wants to output human-readable code.

Overloading as a language feature, for instance, appears a bad fit for Racket's name resolution. To alleviate the issue of naming clashes being more likely without overloading, Racket provides good support for renaming, including module system constructs such as `prefix-in` and `prefix-out` for mass renaming.

Defaulting to something S-expression based for surface syntax is advantageous, as then there is no custom reader to implement. Furthermore, as Racket syntax trees are also (enriched) S-expressions, and macros operate on them, one can then essentially use concrete syntax in patterns and templates for matching and generating code. This is comparable to the language-specific concrete syntax support in program transformation toolkits such as Rascal (Klint et al. 2009) and Spoofax (Kats and Visser 2010). Still, where important, other kinds of concrete syntaxes can be adopted for Racket languages, with or without support for expressing macro patterns in terms of concrete syntax; this has been demonstrated by implementations of Honu (Rafkind and Flatt 2012) and Python (Ramos and Leitão 2014), respectively.

In choice of core syntax, designing for natural and efficient mapping into the target language places fewer demands on the sophistication of the compiler's analyses and transformations. Magnolisp, for instance, is intended to map easily into most mainstream languages. It distinguishes between expressions and statements, for example, as do many mainstream languages (e.g., C++ and Java); making this distinction makes translation into said mainstream languages more direct.

4.2 Example Use Case: A Static Component System

As suggested above, macro-based extensibility might be an important motivation for implementing a Racket-based language, and

choosing a constrained core language might also be important for ease of transcompilation. One can reasonably wonder what the limits of macro-based expression then are, if constructs are defined in terms of their mapping into a limited run-time language. We address this question indirectly by considering a relatively advanced use case for macros as an example, namely that of component system implementation.

When organizing a collection of software building blocks, it can be useful to have a mechanism for "wiring up" and parameterizing said building blocks to form larger wholes (e.g., individual software products of a product line). Racket has a component system that includes such a mechanism; more specifically, the system supports *external linking*, i.e., parameterized reference to an arbitrary implementation of an interface (Owens and Flatt 2006). The system's *units* (Culpepper et al. 2005) are first-class, dynamically composed components.

Magnolisp lacks the run-time support for expressing units, and in this sense the language is severely constrained by its limited core language, and our lack of a comprehensive library of primitives for it. However, at compile time it has access to all of Racket, and hence enough power to implement a purely static component system. No such system is included, but to give an idea of how one might implement one, we provide a complete implementation of a rudimentary, yet potentially useful "component" system in figure 3.

Existing solutions suggest that it should also be possible to implement a more capable static component system in terms of Racket macros. Chez Scheme's `modules` support static, external linking, and have been shown to cater for a variety of use cases (Waddell and Dybvig 1999). Racket's built-in "packages" system (Flatt et al. 2012) resembles the Chez design, and is implemented in terms of macros, relying on features such as sub-form expansion, definition contexts, and compile-time binding. As packages are implemented statically, they require little from the run-time language.

```
#lang magnolisp
(define-syntax-rule (define<> x f e)
  (define-syntax f (cons #'x #'e)))

(define-syntax (use stx)
  (syntax-case stx (with as)
    [(_ f with new-x as fx)
     (let ([v (syntax-local-value #'f)])
       (with-syntax ([old-x (car v)] [e (cdr v)])
         #'(define fx
             (let-syntax ([old-x
               (make-rename-transformer #'new-x)])
             e))))]))

(typedef int (#:annos foreign))
(typedef long (#:annos foreign))

(function (->long x)
  (#:annos [type (fn int long)] foreign))

(define<> T id
  (let/annotate ([type (fn T T)])
    (lambda (x) x)))

; int int_id(int const& x) { return x; }
(use id with int as int-id)
; long long_id(long const& x) { return x; }
(use id with long as long-id)

; long run(int const& x)
;  { return long_id(to_long(int_id(x))); }
(function (run x) (#:annos export)
  (long-id (->long (int-id x))))
```

Figure 3: A primitive ''component'' system for Magnolisp. The macro `define<>` declares a named ''expression template'' `f`, and the macro `use` specializes such templates for a specific parameter `x`. Use of the two macros is demonstrated by a C++-inspired function template `id` with a type parameter `T`, also showing how macros can compensate for the lack of parametric polymorphism in Magnolisp. Corresponding `mglc`-generated C++ code is given in comments.

5. Related Work

While most languages previously implemented on Racket have been meant for execution only on the Racket virtual machine, a notable exception is Dracula (Eastlund 2012), which also compiles macro-expanded programs to ACL2. Dracula's compilation strategy follows the encoding strategy described in section 3.2 where syntactic forms expand to a subset of Racket's core forms, and applications of certain functions (such as `make-generic`) are recognized specially for compilation to ACL2. The part of a Dracula program that runs in Racket is expanded normally, while the part to be translated to ACL2 is recorded in a submodule through a combination of structures and syntax objects, where binding information in syntax objects helps guide the translation.

Sugar* (Erdweg and Rieger 2013) is a system for turning non-extensible languages into extensible ones. The resulting languages are extensible from within themselves, in a modular way, so that extensions are in scope following their respective module imports. While the aim of Sugar* is extended language into base language desugaring, one might also define a Sugar* "base language processor" that translates into another language before pretty printing. From among previously reported solutions, Sugar* perhaps comes closest to being a general solution to the implementation of transcompiled languages possessing the three characteristics listed in section 1.2. While Sugar* is liberal with respect to the definition of language grammars, one might arguably also gain guarantees of safe composition of language extensions through user-imposed discipline in defining them. The relative novelty of our solution is that it is itself based on a language-extension mechanism, whereas Sugar* is a special-purpose language implementation framework.

Silver (Wyk et al. 2010), like Racket, is a language capable of specifying extensible languages such that the extensions are modular and composable. Silver's specifications are based on attribute grammars (Knuth 1968), and the same formalism is used to specify both the base language and its extensions; therefore, even more so than with Racket, any extensions are indistinguishable from core language features. Silver supports safe composition of independently defined extensions by providing analyses to check whether extensions are suitably restricted to be guaranteed to compose; Racket provides some guarantees of safe composition, and even without analysis tools it tends to be obvious whether e.g. the "hygiene condition" (Kohlbecker et al. 1986) holds for the expansion of a given macro. While modular specification of syntax is supported by both Silver and Racket, only the former supports modular specification of semantic analyses. Such analyses—expressed as attribute grammar rules—may also be used to derive a translation to another language; Silver has been used to implement an extensible-C-to-C transcompiler, for example (Williams et al. 2014).

Lightweight Modular Staging (LMS) (Rompf and Odersky 2010) is similar to our technique in goals and overall strategy, but leveraging Scala's type system and overloading resolution instead of a macro system. With LMS, a programmer writes expressions that resemble Scala expressions, but the type expectations of surrounding code cause the expressions to be interpreted as AST constructions instead of expressions to evaluate. The constructed ASTs

can then be compiled to C++, CUDA, JavaScript, other targets, or to Scala after optimization. AST constructions with LMS benefit from the same type-checking infrastructure as normal expressions, so a language implemented with LMS gains the benefit of static typing in much the same way that a Racket-based language can gain macro extensibility. LMS has been used for languages with application to machine learning (Sujeeth et al. 2011), linear transformations (Ofenbeck et al. 2013), fast linear algebra and other data structure optimizations (Rompf et al. 2012), and more.

The Accelerate framework (Chakravarty et al. 2011; McDonell et al. 2013) is similar to LMS, but in Haskell with type classes and overloading. As with LMS, Accelerate programmers benefit from the use of higher-order features in Haskell to construct a program for a low-level target language with only first-order abstractions.

Copilot (Pike et al. 2013) is also a Haskell-embedded language whose expressions are interpreted as AST constructions. Like Racket, Copilot has a core language, into which programs are transformed prior to execution. The Copilot implementation includes two alternative back ends for generating C source code; there is also an interpreter, which the authors have employed for testing. Copilot's intended domain is the implementation of programs to monitor the behavior of executing systems in order to detect and report anomalies. The monitoring is based on periodic sampling of values from C-language symbols of the monitored, co-linked program. Since such symbols are not available to the interpreter, the language comes built-in with a feature that the programmer may use to specify representative "interpreter values" for any declared external values (Pike et al. 2012); this is similar to Magnolisp's support for "mocking" of `foreign` functions.

The Terra programming language (DeVito et al. 2013) appears to take an approach similar to ours, as it adopts an existing language (Lua) for compile-time manipulation of constructs in the run-time language (Terra). Like Racket, Terra allows compile-time code to refer to run-time names in a lexical scope respecting way. Ultimately, however, Terra is not designed to support transcompilation, and compiles to binaries via Terra as a fixed core language. Another difference is Terra's emphasis on supporting code generation at run time, while ours is on separation of compile and run times.

CGen (Selgrad et al. 2014) is a reformulation of C with an S-expression-based syntax, integrated into Common Lisp. An AST for source-to-source compilation is produced by evaluating the CGen core forms; this differs from our approach, where run-time Racket core forms are not evaluated. Common Lisp's `defmacro` construct is available to CGen programs for defining language extensions; Racket's lexical-scope-respecting macros compose in a more robust manner. Racket's macro expansion also tracks source locations, which would be a useful feature for a CGen-like tool. CGen uses the Common Lisp package system to implement support for locally and explicitly switching between CGen and Lisp binding contexts, so that ambiguous names are shadowed; Racket does not include a similar facility.

SC (Hiraishi et al. 2007) is another reformulation of C with an S-expression-based syntax. It supports language extensions defined by transformation rules written in a separate, Common Lisp based domain-specific language (DSL). The rules treat SC programs as data, and thus SC code is not subject to Lisp macro expansion (as in our solution) or Lisp evaluation (as in CGen). Fully transformed programs (in the base SC-0 language) are compiled to C source code. SC programs themselves have access to a C-preprocessor-style extension mechanism via which there is limited access to Common Lisp macro functionality.

6. Conclusion

Regardless of the implementation approach of a programming language, one might wish to extend it with additional features. Numerous motivating examples of language extensions are documented in literature (Hiraishi et al. 2007; Selgrad et al. 2014).

There are several technologies that specialize in language implementation (e.g., Rascal and Spoofax), and some of them (e.g., Silver) even focus on supporting the implementation and composition of independently defined language extensions. However, existing solutions generally lack specific support for the implementation of languages that are extensible from within themselves, and still aim to support convenient definition of extensions that compose in a safe manner. One exception is Racket, which supports the implementation of languages as libraries (Tobin-Hochstadt et al. 2011), and aims for safe composition of not only functions, but also syntactic forms. However, Racket-based languages have traditionally been run on the Racket VM, making Racket an unlikely choice for hosting transcompiled languages.

We have described a generic approach for having Racket host the front end of a source-to-source compiler. It involves a "proper" embedding of the hosted language into Racket, such that Racket's usual language definition facilities are not bypassed. Notably, the macro and module systems are still available, and may be exposed to the hosted language, to provide a way to implement and manage language extensions within the language. Furthermore, tools such as the DrRacket IDE still recognize the hosted language as a Racket one, are aware of the binding structure of programs written in it, and can usually trace the origins of macro-transformed code, for example.

Racket's macro system is expressive, allowing the syntax and semantics of a variety of language extensions to be specified in a robust way; general compile-time bindings for sharing of information between macros, for example, are supported. Scoping of language constructs can be controlled in a fine-grained manner using Racket's module system, and it is also possible to define or import macros for a local scope. With typical macros composing safely, and scoping control reducing the likelihood of macro naming clashes and allowing macros to be defined privately, pervasive use of syntactic abstraction becomes a real alternative to manual or tools-assisted writing of repetitive code.

The benefits of syntactic abstraction can furthermore be extended to any program-describing metadata, whether present to support transcompilation, or for other reasons; this can be done simply by having "data-as-code," thus making it subject to macro expansion.

Racket-hosted base language implementations can likewise leverage Racket's syntax manipulation facilities to perform macro-expansion-based transformations that produce non-Racket code. The approach indeed *requires* some macro-expansion time work to prepare separately loadable information for "transcompile time;" this does not preclude additional work performed in preparation for any optional Racket-VM-based run time.

Racket, with its general-purpose features and libraries, and ability to host program transformation domain specific sub-languages, may also be an attractive substrate for implementing the rest of a transcompilation pipeline.

Acknowledgements Carl Eastlund provided information about the implementation of Dracula. Magne Haveraaen, Anya Helene Bagge, and the SLE 2014 anonymous referees provided useful comments on drafts of this paper. This research has in part been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

Bibliography

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming* 72(1-2), pp. 52–70, 2008.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proc. Wksp. Declarative Aspects of Multicore Programming*, 2011.

Ryan Culpepper. Fortifying Macros. *J. Functional Programming* 22, pp. 439–476, 2012.

Ryan Culpepper and Matthias Felleisen. A Stepper for Scheme Macros. In *Proc. Wksp. Scheme and Functional Programming*, 2006.

Ryan Culpepper and Matthias Felleisen. Debugging Macros. In *Proc. Generative Programming and Component Engineering*, pp. 135–144, 2007.

Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic Abstraction in Component Interfaces. In *Proc. Generative Programming and Component Engineering*, pp. 373–388, 2005.

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A Multi-Stage Language for High-Performance Computing. *ACM SIGPLAN Notices* 48(6), pp. 105–116, 2013.

Carl Eastlund. Modular Proof Development in ACL2. PhD dissertation, Northeastern University, 2012.

Carl Eastlund and Matthias Felleisen. Hygienic Macros for ACL2. In *Proc. Sym. Trends in Functional Programming*, 2010.

Sebastian Erdweg and Felix Rieger. A Framework for Extensible Languages. In *Proc. Generative Programming and Component Engineering*, pp. 3–12, 2013.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *J. Functional Programming* 12(2), pp. 159–182, 2002.

Matthew Flatt. Composable and Compilable Macros: You Want it *When?* In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.

Matthew Flatt. Submodules in Racket: You Want it *When*, Again? In *Proc. Generative Programming and Component Engineering*, 2013.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 109–120, 2009.

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *J. Functional Programming* 22(2), pp. 181–216, 2012.

Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. Experience with SC: Transformation-based Implementation of Various Language Extensions to C. In *Proc. Intl. Lisp Conference*, pp. 103–113, 2007.

Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 444–463, 2010.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. IEEE Intl. Working Conf. Source Code Analysis and Manipulation*, pp. 168–177, 2009.

Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical System Theory* 2(2), pp. 127–145, 1968.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. Lisp and Functional Programming*, pp. 151–181, 1986.

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *Proc. ACM Intl. Conf. Functional Programming*, 2013.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proc. Generative Programming and Component Engineering*, 2013.

Scott Owens and Matthew Flatt. From Structures and Functors to Modules and Units. In *Proc. ACM Intl. Conf. Functional Programming*, 2006.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience Report: A Do-It-Yourself High-Assurance Compiler. In *Proc. ACM Intl. Conf. Functional Programming*, 2012.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring Embedded Systems. *Innovations in Systems and Software Engineering* 9(4), pp. 235–255, 2013.

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. Generative Programming and Component Engineering*, pp. 122–131, 2012.

Pedro Ramos and António Menezes Leitão. An Implementation of Python for Racket. In *Proc. European Lisp Symposium*, 2014.

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. Generative Programming and Component Engineering*, pp. 127–136, 2010.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging. In *Proc. ACM Sym. Principles of Programming Languages*, 2012.

Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. Defmacro for C: Lightweight, Ad Hoc Code Generation. In *Proc. European Lisp Symposium*, 2014.

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proc. Intl. Conf. Machine Learning*, 2011.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. *SIGPLAN Not.* 47(6), pp. 132–141, 2011.

Oscar Waddell and R. Kent Dybvig. Extending the Scope of Syntactic Abstraction. In *Proc. ACM Sym. Principles of Programming Languages*, 1999.

Kevin Williams, Matt Le, Ted Kaminski, and Eric Van Wyk. A Compiler Extension for Parallel Matrix Programming. In *Proc. Intl. Conf. Parallel Processing (to appear)*, 2014.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An Extensible Attribute Grammar System. *Science of Computer Programming* 75(1-2), pp. 39–54, 2010.

Danny Yoo and Shriram Krishnamurthi. Whalesong: Running Racket in the Browser. In *Proc. Dynamic Languages Symposium*, 2013.

Combining Shared State with Speculative Parallelism in a Functional Language

Matthew Le
Rochester Institute of Technology
ml9951@cs.rit.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract

Purely functional programming languages have proven to be an attractive option for implementing parallel applications. The lack of mutable state eliminates the possibility for race conditions, which relieves programmers of reasoning about the exponential interleavings of threads and nondeterministic behavior. Unfortunately, there are applications that by making use of shared state can achieve significant constant factor speedups compared to their purely functional counterparts.

IVars have been proposed as a possible solution, allowing threads to share information via write-once references, while preserving a deterministic semantics. However, in the presence of speculative parallelism (cancelation), this determinism guarantee is lost. In this work we show how to go about combining these two concepts by proposing a dynamic rollback mechanism for enforcing determinism. We have formalized the semantics of a parallel functional language extended with IVars, speculative parallelism, and our proposed rollback mechanism using the Coq proof assistant, and have proven that it preserves determinism. Additionally, we describe a preliminary implementation in the context of the Manticore project, and give some initial performance results.

1. Introduction

Writing parallel applications is a notoriously difficult task. Programmers are forced to reason about the nondeterministic behavior arising from an exponential interleaving of threads. One way of avoiding this difficulty is to use a functional language when developing parallel applications, since functional languages prohibit the alteration of shared state. Race conditions and nondeterminism arise when multiple threads attempt to read from and write to the same location in memory. Since functional languages do not allow writes, we avoid race conditions altogether, making determinism an easy property to enforce. Unfortunately, there are applications that can be more efficiently or more naturally implemented when shared state is used. One attempt at addressing this problem is the use of IVars [ANP89], which are shared references that may only be written to once. IVars have been proven to preserve determinism in an otherwise purely functional parallel language [MNP11, BBC$^+$10], while allowing threads to communicate intermediate results to one another via shared memory.

In this work we show that the determinism guarantee for IVars does not hold in the presence of speculation – a method for parallelizing programs, where unneeded tasks may be canceled. Additionally this paper makes the following contributions:

- We propose a rollback mechanism that can be used to restore deterministic execution in the presence of speculation and IVars.
- We provide a formal semantics of a parallel language with IVars, speculative parallelism, and the proposed rollback mechanism.
- We give a mechanized proof, using the Coq Proof Assistant, that the rollback mechanism preserves determinism of the language that combines IVars and speculative parallelism.
- We describe an implementation that is under development in the Manticore project and give some preliminary results.

Source code for the Coq formalization can be found at: `http://people.rit.edu/ml9951/research.html`.

2. Background

2.1 IVars

IVars are shared memory references that may only be written to once, originally proposed as part of the parallel functional language Id [ANP89]. The interesting property about IVars is that they do not compromise the determinism guarantees that one can make about an otherwise purely functional parallel language. Meanwhile, they strictly increase the expressiveness of the language. As an example, consider an application implementing producer-consumer parallelism where two threads are running in parallel, one of which writes data into a shared buffer and the other processes this data as soon as it becomes available. This sort of pattern cannot be efficiently implemented in a purely functional language. In such a language, the producer would be required to produce all of its elements before the consumer could start processing them. On the other hand, if we are able to make use of IVars, we could implement the shared buffer as a linked list giving us the desired behavior.

Informally, the semantics of IVars are as follows. When an IVar is created it is empty. When it is written to, it becomes full and if a thread attempts to write to it again, a runtime error is produced and the program terminates. If a thread tries to read from an IVar that is empty, it blocks until the contents are filled, after which it can read from the IVar an arbitrary number of times without synchronization.

2.2 Speculative Parallelism

Speculation is a method for parallelizing applications, where some number of parallel tasks are created, and if it turns out that any of these tasks are unneeded, they are canceled. This sort of pattern arises frequently in search problems where we want to search multiple paths in parallel, and then when a solution is found, we would like to cancel the rest of the search threads so as to free up resources for future computations. The research literature has given rise to many examples of speculatively parallel algorithms [PRV10, Bur85, JLM$^+$09].

```
1  exception E
2  val i = IVar.new()
3  val _ = (|raise E, IVar.put(i, 10)|)
4          handle E => ((), ())
5  val x = IVar.get i
```

Figure 1. Nondeterministic Example

2.3 Manticore

Manticore is a compiler for a purely functional subset of Standard ML which has been extended with parallel features. These parallel features are given a sequential semantics, allowing programmers to reason about parallel computations in the same way they would their sequential counterparts. The most basic parallel construct in Manticore is the parallel tuple, denoted

$$(|e_1, \ldots, e_n|)$$

Parallel tuples express fork-join parallelism, where each expression e_i is evaluated in parallel. The result of the entire expression is a data structure containing the results of each e_i.

Additionally, we provide a construct for asynchronously spawning threads via **fork**. The **fork** construct takes an expression that gets evaluated in a separate thread allowing the main thread to continue with the execution of the remainder of the program.

In addition to parallelism, Manticore also supports exception handling. The semantics for a regular sequential tuple is to evaluate each e_i in left to right order, so if an exception gets raised, it will always be the leftmost exception in the tuple that gets propagated. Enforcing the sequential semantics for parallel tuples in the presence of exceptions works as follows. If expression e_i raises an exception, then the threads evaluating expressions e_{i+1} through e_n are canceled and we wait for the previous $i-1$ elements to terminate to check if they raise an exception.

2.4 Determinism

In Manticore we can encode a notion of speculative parallelism using the parallel constructs and exception handling features while maintaining the sequential semantics [FRRS08]. Unfortunately, if we were to incorporate IVars into the language, we would lose this guarantee due to the cancelation associated with raised exceptions. As an example, consider the code in Figure 1. In line 2 we create an empty IVar, and then in parallel raise an exception and write to this IVar. There are two ways in which this can play out. The cancelation can go through before the write, leaving the IVar empty, or the write can go through before the cancelation leaving the IVar full with the value 10. These two scenarios lead to two different observable behaviors of our program, either it could block indefinitely due to a read from an empty IVar, or it could terminate with x bound to the value 10. In order to enjoy the benefits of a deterministic parallel language, we must extend the Manticore runtime system to avoid these race conditions.

3. Preserving Determinism

In order to preserve a deterministic semantics for parallel tuples in the presence of exception handling we would like to make it seem to the programmer as if canceled threads "never happened" in the first place. Implementing this for a purely functional language is not too difficult, however, in the presence of shared references such as IVars, it becomes substantially more complex.

The first step is to "undo" the effects of a thread when it is being canceled due to a raised exception. With IVars this simply amounts to resetting the contents of full IVars to empty. However, it is possible that before the cancelation occurred, other threads concurrently running were able to read the contents of this IVar.

```
1  exception E
2  val i = IVar.new()
3  val j = IVar.new()
4  val _ = fork((|raise E, IVar.put(i, 10)|)
5              handle E => ((), ()))
6  val (_, x) = (|IVar.put(j, IVar.get i), IVar.get j|)
```

Figure 2. Transitive Rollback

	x	$\in$	Var
Values	V	::=	$x \mid i \mid \backslash x.M \mid \textbf{return}\ M \mid M >>= N$
		$\mid$	$\textbf{runPar}\ M \mid \textbf{fork}\ M \mid \textbf{new} \mid \textbf{put}\ i\ M$
		$\mid$	$\textbf{get}\ i \mid \textbf{done}\ M \mid \textbf{spec}\ M\ N$
		$\mid$	$\textbf{specRun}(M, N) \mid \textbf{specJoin}(M, N)$
		$\mid$	$\textbf{raise}\ M \mid \textbf{handle}\ M\ N$
Terms	M, N	::=	$V \mid M\ N \mid \cdots$
Heap	H	::=	$H, x \mapsto iv \mid \cdot$
Speculative State	s	::=	$\mathbf{S} \mid \mathbf{C}$
IVar State	iv	::=	$\langle s \rangle \mid \langle s_1, ds, s_2, \Theta, M \rangle$
Thread ID	Θ	::=	$\cdot \mid \Theta : n \quad n \in \mathbb{N}$
Thread Pool	T	::=	$\cdot \mid (T_1 \mid T_2) \mid \Theta[S_1, S_2, M]$
Action	A	::=	$(R, x, M) \mid (W, x, M) \mid (S, M)$
		$\mid$	$(A, x, M) \mid (F, \Theta, M) \mid CSpec$
Action Queue	S	::=	$\cdot \mid S : A$
Evaluation Context	E	::=	$[\cdot] \mid E >>= M \mid \textbf{specRun}(E, M)$
		$\mid$	$\textbf{handle}\ E\ N \mid \textbf{specJoin}(N, E)$
Configuration	σ	::=	$H; T \mid Error$

Figure 3. Speculative Par Monad Syntax

In this case, the runtime system must also rollback these threads to the point in which they read from the IVar. At this point, we also need to "undo" any effects that these threads may have done and rollback any threads that might have read from these IVars. This rollback continues until we reset all IVars and dependent readers that are transitively reachable from the effects done by the original thread that was canceled.

As an example of this transitive closure property, consider the code in Figure 2. In line 4 we fork a new thread to evaluate a parallel tuple that raises an exception and writes the value 10 to IVar i. The thread writing to the IVar is then canceled due to the raised exception, requiring the contents of i to be reset to empty. In line 6 we read the value written to i and write it into j. If this read occurs before the cancelation, then we must reset this thread back to before it performed the read. If we are going to reset this thread to before it did the read, then we must also "undo" the write to IVar j. Furthermore, if the second element of the parallel tuple is able to read from j, then this must also get rolled back to the point before it performed the read.

4. Formal Semantics

In this section we provide a formal semantics describing a method for performing the rollback that was alluded to in the previous section. The semantics presented in this paper are an extension of the Par Monad semantics as presented in [MNP11], which helps facilitate our proof of determinism described in the next section. Figure 3 gives the syntax of the language. Relative to [MNP11], We have added syntax for speculative computations, where **specRun** and **specJoin** are intermediate forms that arise throughout the execution of a program, and are not terms available in the surface language, as is **done**.

A heap is a finite map from IVar names to IVar states where an IVar state can be empty or full. If it is empty, we also indicate

$$\text{RunPar}\ \frac{\cdot; 1[\cdot, \cdot, M >>= \backslash x.\textbf{done}\ x] \rightarrow_s^* H'; T \mid \cdot : 1[\cdot, S_2, \textbf{done}\ N], \quad N \Downarrow_s V, \quad \text{Finished}(T)}{\textbf{runPar}\ M \Downarrow_s V}$$

$$\text{RunParError}\ \frac{\cdot; 1[\cdot, \cdot, M >>= \backslash x.\textbf{done}\ x] \rightarrow_s^* \textbf{Error}}{\textbf{runPar}\ M \Downarrow_s \textbf{Error}} \qquad \text{RunParDiverge}\ \frac{\cdot; 1[\cdot, \cdot, M >>= \backslash x.\textbf{done}\ x] \rightarrow_s^\infty}{\textbf{runPar}\ M \Downarrow_s \infty}$$

$\boxed{H; T \rightarrow_s \sigma}$

$$\text{Eval}\ \frac{M \neq V \quad M \Downarrow_s V}{H; \Theta[S_1, S_2, E[M]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[V]] \mid T} \qquad \text{Bind}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{return}\ N >>= M]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[M\ N]] \mid T}$$

$$\text{BindRaise}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{raise}\ M >>= N]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[\textbf{raise}\ M]] \mid T}$$

$$\text{Handle}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{handle}(\textbf{raise}\ M)N]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[N\ M]] \mid T}$$

$$\text{HandleReturn}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{handle}(\textbf{return}\ M)N]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[\textbf{return}\ M]] \mid T}$$

$$\text{Fork}\ \frac{n = \text{numSpawns}(s1, s2)}{H; \Theta[S_1, S_2, E[\textbf{fork}\ M]] \mid T \rightarrow_s H; \Theta[(F, \Theta : n, E[\textbf{fork}\ M]) : S_1, S_2, E[\textbf{return}()]] \mid \Theta : n[CSpec, \cdot, M] \mid T}$$

$$\text{New}\ \frac{H' = H[x \mapsto \langle \mathbf{S} \rangle] \quad x \notin Domain(H)}{H; \Theta[S_1, S_2, E[\textbf{new}]] \mid T \rightarrow_s H'; \Theta[(A, x, E[\textbf{new}]) : S_1, S_2, E[\textbf{return}\ x]] \mid T}$$

$$\text{Get}\ \frac{H(x) = \langle s_1, ds, s_2, \Theta', M \rangle, \quad H' = H[x \mapsto \langle s_1, \Theta \uplus ds, s_2, \Theta', M \rangle]}{H; \Theta[S_1, S_2, E[\textbf{get}\ x]] \mid T \rightarrow_s H'; \Theta[(R, x, E[\textbf{get}\ x]) : S_1, S_2, E[\textbf{return}\ M]] \mid T}$$

$$\text{Put}\ \frac{H(x) = \langle s \rangle, \quad H' = H[x \mapsto \langle s, \emptyset, \mathbf{S}, \Theta, M \rangle]}{H; \Theta[S_1, S_2, E[\textbf{put}\ x\ M]] \mid T \rightarrow_s H'; \Theta[(W, x, E[\textbf{put}\ x\ M]) : S_1, S_2, E[\textbf{return}()]] \mid T}$$

$$\text{Overwrite}\ \frac{\begin{array}{c} H(x) = \langle s_1, ds, \mathbf{S}, \Theta', N \rangle, \quad \Theta'[S_1 : (W, x, N) : S_1', S_2', N'] \in T \\ \text{rollback}(\Theta', S_1', H, T) \rightsquigarrow (H', T'), \quad H'' = H'[x \mapsto \langle \emptyset, \cdot, \Theta, M \rangle] \end{array}}{H; \Theta[\cdot, S_2, E[\textbf{put}\ x\ M]] \mid T \rightarrow_s H''; \Theta[\cdot, S_2, E[\textbf{return}()]] \mid T'} \qquad \text{ErrorWrite}\ \frac{H(x) = \langle \mathbf{C}, ds, \mathbf{C}, \Theta', N \rangle}{H; \Theta[\cdot, S_2, E[\textbf{put}\ x\ M]] \mid T \rightarrow_s Error}$$

$$\text{Spec}\ \frac{n = \text{numSpawns}(s1, s2)}{H; \Theta[S_1, S_2, E[\textbf{spec}\ M\ N]] \mid T \rightarrow_s H; \Theta[(F, \Theta : n, E[\textbf{spec}\ M\ N]) : S_1, S_2, E[\textbf{specRun}(M, N]] \mid \Theta : n[(S, N) : CSpec, \cdot, N] \mid T'}$$

$$\text{SpecRB}\ \frac{\text{rollback}(\Theta : n, \cdot, H, \Theta : n[S_1' : (S, N_0), S_2', N] \mid T) \rightsquigarrow (H', \Theta : n[\cdot, S_2', N'] \mid T')}{H; \Theta[\cdot, S_2, E[\textbf{specRun}(\textbf{raise}\ M, N_0]] \mid \Theta : n[S_1' : (S, N_0), S_2', N] \mid T \rightarrow_s H'; \Theta[\cdot, S_2, E[\textbf{raise}\ M]] \mid T'}$$

$$\text{SpecJoin}\ \frac{\Theta : n[S_1' : (S, N_0), S_2', N] \in T}{H; \Theta[\cdot, S_2, E[\textbf{specRun}(\textbf{return}\ M_1, N_0)]] \mid T \rightarrow_s H; \Theta : n[adopt(S_1', E, \textbf{return}\ M_1), S_2', E[\textbf{specJoin}(\textbf{return}\ N_1, N)]] \mid T}$$

$$\text{SpecDone}\ \frac{}{H; \Theta[\cdot, S_2, E[\textbf{specJoin}(\textbf{return}\ N_1, \textbf{return}\ N_2)]] \mid T \rightarrow_s H; T \mid \Theta[\cdot, S_2, E[\textbf{return}(N_1, N_2)]]}$$

$$\text{SpecRaise}\ \frac{}{H; \Theta[\cdot, S_2, E[\textbf{specJoin}(\textbf{return}\ N_1, \textbf{raise}\ E)]] \mid T \rightarrow_s H; \Theta[\cdot, S_2, E[\textbf{raise}\ M]] \mid T}$$

$$\text{PopRead}\ \frac{H(x) = \langle \Theta, \mathbf{C}, \uplus ds, \mathbf{C}, \Theta', M \rangle}{H; \Theta[S_1 : (R, x, N'), S_2, N] \mid T \rightarrow_s H; \Theta[S_1, (R, x, N') : S_2, N] \mid T}$$

$$\text{PopWrite}\ \frac{H(x) = \langle \mathbf{C}, ds, \mathbf{S}, \Theta, M \rangle, \quad H' = H[x \mapsto \langle \mathbf{C}, ds, \mathbf{C}, \Theta, M \rangle]}{H; \Theta[S_1 : (W, x, N'), S_2, N] \mid T \rightarrow_s H'; \Theta[S_1, (W, x, N') : S_2, N] \mid T)}$$

$$\text{PopNewFull}\ \frac{H(x) = \langle \mathbf{S}, ds, \mathbf{S}, \Theta', M \rangle, \quad H' = H[x \mapsto \langle \mathbf{C}, ds, \mathbf{S}, \Theta', M \rangle]}{H; \Theta[S_1 : (A, x, M''), S_2, M'] \mid T \rightarrow_s H'; \Theta[S_1, (A, x, M'') : S_2, M'] \mid T}$$

$$\text{PopNewEmpty}\ \frac{H(x) = \langle \mathbf{S} \rangle, \quad H' = H[x \mapsto \langle \mathbf{C} \rangle]}{H; \Theta[S_1 : (A, x, M'), S_2, M] \mid T \rightarrow_s H'; \Theta[S_1, (A, x, M') : S_2, M] \mid T}$$

$$\text{PopFork}\ \frac{}{H; \Theta[S_1 : (F, \Theta', M'), S_2, M] \mid \Theta'[S_1' : CSpec, S_2', N] \mid T \rightarrow_s H; \Theta[S_1, (F, \Theta', M') : S_2, M] \mid \Theta : 1[S_1', CSpec : S_2', N] \mid T}$$

Figure 4. operational Semantics

whether or not is was allocated speculatively. If an IVar is full we record if it was allocated speculatively, the thread IDs of those who have read the IVar, whether or not it was written speculatively, the ID of the writer, and the term written to the IVar. A thread pool is a multiset of threads, where each thread has a thread ID, a list (queue) of speculative actions it has performed, a list of actions it has committed, and a term that it is evaluating. Action queues are a list of actions, where an action can be a read, write, spec, allocation, fork, or an action indicating it was created speculatively. Lastly, a configuration is a heap paired with a thread pool, or the error state.

The overall semantics of the language is described by a big step relation, which is used to represent the "usual" Haskell semantics. In this presentation and in [MNP11], we only give the big-step rule for **runPar** as the rest is entirely conventional. The RunPar rule then depends on a small step relation for the Speculative Par Monad presented in Figure 4. The small step semantics relates a heap H and a thread pool T to either a new Heap and new thread pool, or the error state if multiple writes occurred to a single IVar. Rules Bind, BindRaise, Handle, and HandleReturn are standard monadic bind and exception handling rules. The Eval rule dispatches back to the big step semantics for reducing non-monadic terms (such as beta reduction, creating tuples, projecting tuples, etc...).

The Fork rule spawns a new thread, and records a fork action on the thread performing the fork along with the thread ID of the forked thread. we uniquely name threads by adding a number onto the forking thread's ID that is equal to the number of threads that have already been created by this thread. The forked thread is then created with an action on its stack indicating it was created speculatively, and not allowing it to commit any actions. When the forking thread has a fork action at the head of its action queue, it can commit this action, moving the fork action over to its commit list, and moving the $CSpec$ action over to the forked thread's commit list. The New rule allocates a new IVar, marking it as having been allocated speculatively. When the allocation action makes its way to the head of the action queue, it can then change the state of the IVar from speculative to commit using the PopNewFull or PopNewEmpty rule. The Get rule is used to read from an IVar, if the IVar is full, then we add a read action to the threads speculative action queue, and record the thread's ID in the IVar indicating that if this IVar is rolled back, this thread is a dependent reader. The PopRead rule can then be used to commit this read action assuming the IVar is now in commit mode. The Put rule is used to write to an IVar, assuming it is empty, we fill the contents of the IVar and add a write action to the thread's action list. This action can then be committed using the PopWrite rule, which sets the status of the IVar to commit written.

The Overwrite rule applies when a thread has no speculative actions (i.e. it is in commit mode) and is attempting to write to an IVar that is speculatively full. When looking up the IVar in the heap we see that it previously was written by thread Θ', which we then lookup in the pool and split its speculative action queue into those actions that happened after the write, and those that happened before the write to this IVar. We then perform a rollback with respect to thread Θ', which is described later. For now it suffices to know that it undoes all actions performed by Θ', up to S_1', which correspond to the actions performed before the write to x. We then update IVar x to contain the value being written by thread Θ. The ErrorWrite rule is similar to Overwrite, except the IVar being written is commit full, which corresponds to an error.

The Spec rule begins a speculative computation, which behaves similarly to the Fork rule with a few differences. First, notice that we add two actions to the created thread's speculative action list, The first is an action indicating it was created speculatively as is done in the Fork rule, but we also include the (S, N) action indicating that it is the right branch of a speculative computation

$$\boxed{\text{rollback}(\Theta, S, H, T) \rightsquigarrow (H', T')}$$

$$\text{RBDone}\ \frac{}{\text{rollback}(\Theta, S, H, \Theta[S, S_2, M] \mid T) \rightsquigarrow (H, \Theta[S, S_2, M] \mid T)}$$

$$\text{RBRead}\ \frac{\begin{array}{c} H(x) = \langle s_1, \Theta' \uplus ds, \mathbf{S}, t, M\rangle, \\ H' = H[x \mapsto \langle s_1, ds, \mathbf{S}, t, M\rangle] \\ \text{rollback}(\Theta, S, H', \Theta'[S_1, S_2, M'] \mid T) \rightsquigarrow (H'', T') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(R, x, M') : S_1, S_2, M] \mid T) \rightsquigarrow (H'', T')}$$

$$\text{RBFork}\ \frac{\begin{array}{c} T = \Theta''[CSpec, S_2', M''] \mid T' \\ \text{rollback}(\Theta, S, H, \Theta'[S_1, S_2, M'] \mid T') \rightsquigarrow (H', T'') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(F, \Theta'', M') : S_1, S_2, M] \mid T) \rightsquigarrow (H', T'')}$$

$$\text{RBWrite}\ \frac{\begin{array}{c} H(x) = \langle s, \emptyset, \mathbf{S}, \Theta', M\rangle, \quad H' = H[x \mapsto \langle s\rangle] \\ \text{rollback}(\Theta, S, H', \Theta'[S_1, S_2, M'] \mid T) \rightsquigarrow (H'', T') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(W, x, M') : S_1, S_2, M] \mid T) \rightsquigarrow (H'', T')}$$

$$\text{RBNew}\ \frac{\begin{array}{c} H(x) = \langle \mathbf{S}\rangle, \quad H' = H \backslash x \\ \text{rollback}(\Theta, S, H', \Theta'[S_1, S_2, M'] \mid T) \rightsquigarrow (H'', T') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(A, x, M') : S_1, S_2, M] \mid T) \rightsquigarrow (H'', T')}$$

Figure 5. Rollback

with initial term N. When the fork action makes its way to the front of Θ's action list, we remove the $CSpec$ action, but the (S, N) action remains on the speculative list, disallowing this thread from committing anything until it joins with its corresponding commit thread in the SpecJoin rule.

The SpecRB rule corresponds to canceling a speculative thread, where we rollback the canceled thread's actions similarly to what is done in the Overwrite rule. The SpecJoin rule is used for joining a speculative computation. When the thread executing the left branch of a speculative computation is finished, we adopt the term being evaluated by the speculative thread, and all of its speculative actions and transition to the **specJoin** intermediate form. The SpecDone and SpecRaise rules are used to finish a speculative computation when the right branch evaluates to a returned value or raised exception respectively.

Figure 5 provides the semantics for performing a rollback. The rollback function takes a thread ID, Θ, to rollback with respect to, a list of actions, S, such that the rollback stops when thread Θ has this list of actions S, a heap, and a thread pool. The result of a rollback is then a new heap and a new thread pool.

The RBDone rule indicates that the rollback is complete when thread Θ has as its action list S. The RBRead rule is used to undo a read action. It must be the case that the thread's ID is present in the set of dependent readers on the IVar when looked up in the heap, so we remove the ID from the set, and continue with the rollback, resetting the thread to the term associated with the read action. RBFork is applicable when the thread associated with a fork action has nothing but the created speculative action in its speculative list, we then proceed with the rollback by throwing away the forked thread, and reset the forking thread to the term associated with the action. RBWrite undoes a write action when the IVar written to has no recorded dependent readers, we then proceed by resetting the IVar to empty and resetting the writing thread to the term associative with the write action. RBNew undoes an allocation action when the IVar being rolled back was speculatively created, we remove it from the heap and continue after resetting the thread back to the term associated with the allocation action.

$$
\begin{array}{rcl}
\mathcal{E}[\![H;T]\!] & = & \mathcal{E}[\![H]\!];\mathcal{E}[\![T]\!] \\
\mathcal{E}[\![T_1 \mid T_2]\!] & = & \mathcal{E}[\![T_1]\!] \mid \mathcal{E}[\![T_2]\!] \\
\mathcal{E}[\![\Theta[S_1 : (R, x, M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (W, x, M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (A, x, M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (F, \Theta', M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (S, M'), S_2, M]]\!] & = & \cdot \\
\mathcal{E}[\![\Theta[S_1 : CSpec, S_2, M]]\!] & = & \cdot \\
\mathcal{E}[\![\Theta[\cdot, S_2, M]]\!] & = & M \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{S} \rangle]\!] & = & \mathcal{E}[\![H]\!] \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{C} \rangle]\!] & = & \mathcal{E}[\![H]\!], x \mapsto \langle\rangle \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{S}, ds, s, \Theta, M \rangle]\!] & = & \mathcal{E}[\![H]\!] \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{S}, \Theta, M \rangle]\!] & = & \mathcal{E}[\![H]\!], x \mapsto \langle\rangle \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{C}, \Theta, M \rangle]\!] & = & \mathcal{E}[\![H]\!], x \mapsto \langle M \rangle
\end{array}
$$

Figure 6. Erasure

$$
\begin{array}{rcl}
\mathcal{US}[\![H;T]\!] & = & \mathcal{US}[\![H]\!];\mathcal{US}[\![T]\!] \\
\mathcal{US}[\![T_1 \mid T_2]\!] & = & \mathcal{US}[\![T_1]\!] \mid \mathcal{US}[\![T_2]\!] \\
\mathcal{US}[\![\Theta[S_1 : (R, x, M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (W, x, M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (A, x, M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (F, \Theta', M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (S, M'), S_2, M]]\!] & = & \Theta[\cdot : (S, M'), S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : CSpec, S_2, M]]\!] & = & \cdot \\
\mathcal{US}[\![\Theta[\cdot, S_2, M]]\!] & = & \Theta[\cdot, S_2, M] \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{S} \rangle]\!] & = & \mathcal{US}[\![H]\!] \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{C} \rangle]\!] & = & \mathcal{US}[\![H]\!], x \mapsto \langle \mathbf{C} \rangle \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{S}, ds, s, \Theta, M \rangle]\!] & = & \mathcal{US}[\![H]\!] \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{S}, \Theta, M \rangle]\!] & = & \mathcal{US}[\![H]\!], x \mapsto \langle \mathbf{C} \rangle \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{C}, \Theta, M \rangle]\!] & = & \mathcal{US}[\![H]\!], x \mapsto \langle \mathbf{C}, \emptyset, \mathbf{C}, \Theta, M \rangle
\end{array}
$$

Figure 7. Unspeculate

5. Proof of Determinism

The overall proof strategy is to first prove an equivalence to the original Par Monad, which is known to be deterministic [MNP11, BBC$^+$10], and then deducing determinism for our speculative extension from this equivalence. For the reader's convenience, we have restated the semantics of the original Par Monad in the Appendix. Those familiar with [MNP11] will notice some slight differences between the two presentations. First, we have used an explicit heap for IVars, where as the original semantics mixes threads with IVars in the style of the π-calculus. Second, we have added syntax for speculative computations in Par, however, it is evaluated sequentially and essentially equivalent to a special case of the bind construct. More concretely, **spec** M N can be de-sugared to $M >>= \backslash i.(N >>= \backslash j.\mathbf{return}(i, j))$ where i does not occur free in N. Lastly, in the original semantics, threads were allowed to terminate in the middle of a computation when they complete, where as in our presentation, we keep them around to the end of a **runPar**.

Before stating our equivalence theorem, we first introduce an erasure in Figure 6 that relates speculative program states to non speculative (Par Monad) program states. Intuitively, the erasure recursively goes through the program state, and "throws away" speculative work. If a thread has speculative actions, we reset them to the term associated with the oldest action in their list for read, write, allocation, and fork actions. If the oldest action indicates that it was created speculatively, or it is a thread executing the right branch of a **spec**, then we simply throw away the thread as these threads would not yet have been created in the non speculative semantics. When erasing the heap, we throw out any IVars that were speculatively created. If an IVar was commit created, but was speculatively written, then the erasure simply resets it to empty.

We can now relate the behaviors in one language to the behaviors in the other, where behaviors are defined as:

$$
\begin{array}{rcl}
\beta_s[M] & = & \{V \mid \mathbf{runPar}\ M \Downarrow_s b\} \\
\beta_p[M] & = & \{V \mid \mathbf{runPar}\ M \Downarrow_p b\}
\end{array}
$$

Here the s subscript is used to denote a large step in the speculative semantics and a p subscript is used to denote a large step in the non speculative (Par) semantics. Also, in this case b represents all possible outcomes of **runPar** (i.e. b could be some term M, **Error**, or ∞).

There is an interesting point to be made about proving an equivalence between diverging programs. In the speculative language it is possible to have divergent programs that can converge in the nonspecualtive language if care is not taken. As an example consider the program:

$$\mathbf{runPar}\ (\mathbf{spec}\ (\mathbf{raise}\ M)\ N)$$

Where N is a divergent computation. In the speculative language, there is nothing that forces us to make progress on the commit portion of a speculative computation, therefore this program could infinitely take steps on N, despite the fact that if the left branch of the **spec** ever got a chance to run it would cancel the divergent computation. In the non speculative language this is not an issue as progress cannot be made on the right branch of a **spec** until the left branch has been evaluated to a raised exception or returned value. Typically one would define divergence as:

$$\frac{H;T \rightarrow_s H';T' \quad H';T' \rightarrow_s^\infty}{H;T \rightarrow_s^\infty}$$

However for our purposes we must state a more restrictive version of divergence:

$$\frac{H;T \rightarrow_{\text{spec}}^* H';T' \quad H';T' \rightarrow_{\text{commit}} H'';T'' \quad H'';T'' \rightarrow_s^\infty}{H;T \rightarrow_s^\infty}$$

Where the $\rightarrow_{\text{commit}}$ relation is the same as the step relation presented in Figure 4 except that we restrict that the thread taking the step does not have any uncommitted actions and the $\rightarrow_{\text{spec}}$ relation is the complement of $\rightarrow_{\text{commit}}$. Essentially we are enforcing a fairness policy requiring that progress must be made on a commit thread in order for a program state to be divergent. Note that this leaves the class of speculatively divergent programs undefined in our formalism, however, we do not believe this is an issue as those programs will have a defined behavior in an actual implementation assuming a fair scheduling policy.

At this point we are able to state our equivalence theorem

Theorem 1 (Equivalence)**.** $\forall M, \beta_s[M] = \beta_p[M]$

Proof Sketch. We show $\forall b \in \beta_s[M] \Rightarrow b \in \beta_p[M]$ and $\forall b \in \beta_p[M] \Rightarrow b \in \beta_s[M]$. The most interesting case is showing $V \in \beta_s[M] \Rightarrow V \in \beta_p[M]$ where V is the result of a successfully converging program in the speculative language (i.e. not an error or divergent program), which follows from Lemma 1 □

Lemma 1 (Speculative Implies Nonspeculative)
If $\cdot; 1[\cdot, \cdot, M >>= \backslash x.\mathbf{done}\ x] \rightarrow_s^* H_s; T_s \mid 1[\cdot, S_2, \mathbf{done}\ N]$ *and Finished*(T_s) *then*
$\exists H_p\ T_p, \cdot; M >>= \backslash x.\mathbf{done}\ x \rightarrow_p^* H; T_p \mid \mathbf{done}\ N$ *and* $\mathcal{E}[\![H_s; T_s]\!] = H_p; T_p$ *and Finished*(T_p)

This is proven with a good amount of infrastructure behind it. First we define a metafunction in Figure 7 similar to erasure that relates a program state to its "commit frontier" which essential abandons any speculative work that has been done. This unspeculate function is then used to state a well-formedness property on speculative program states:

$$\frac{\mathcal{US}[\![H;T]\!] \rightarrow_s^* H;T}{\mathrm{WF}(H;T)}$$

Intuitively, this says that a program state is well formed if we can abandon all speculative work that has been done and get back to the exact point we were at before unspeculating. Lemma 1 then follows from a more general restatement.

Lemma 2 (Speculative Implies Nonspeculative WF)
If $WF(H_s;T_s)$ *and* $H_s;T_s \rightarrow_s^* H_s';T_s'$ *then*
$\exists H_p'\ T_p', \mathcal{E}[\![H_s;T_s]\!] \rightarrow_p^* H_p';T_p'$ *and* $\mathcal{E}[\![H_s';T_s']\!] = H_p';T_p'$

Proof Sketch. By induction on the derivation of $H_s;T_s \rightarrow_s^* H_s';T_s'$ and case analysis on the first step taken in the derivation. If the first step is a speculative step (i.e. the thread taking the step has uncommitted actions), then take zero steps in the non speculative semantics as $\mathcal{E}[\![H_s;T_s]\!] = \mathcal{E}[\![H_s';T_s']\!]$. If the first step corresponds to Eval, Bind, BindRaise, Handle, HandleReturn, Fork, New. Get, Put, Overwrite, ErrorWrite, Spec, SpecRB, SpecJoin, SpecDone, or SpecRaise, and the thread taking the step does not have any uncommitted actions, then we take the one corresponding step in the non speculative semantics. If the first step corresponds to PopRead, PopWrite, PopNewFull, PopNewEmpty, or PopFork, then the speculative program must "catch up" by performing the action being committed and all of the pure steps between the action being committed and the next uncommitted action if any. Fortunately, the sequence of steps necessary to catch up is given to us by the well-formedness derivation. □

Once we have established the equivalence, we can deduce determinism easily assuming that the non speculative language is deterministic

Theorem 1 (Par Monad Deterministic)
If **runPar** $M \Downarrow_p V_1$ *and* **runPar** $M \Downarrow_p V_2$*, then* $V_1 = V_2$

Proof Sketch. This is assumed based on previous work □

Theorem 2 (Speculative Par Monad Deterministic)
If **runPar** $M \Downarrow_s V_1$ *and* **runPar** $M \Downarrow_s V_2$*, then* $V_1 = V_2$

Proof Sketch. By case analysis on both **runPar** $M \Downarrow_s V_1$ and **runPar** $M \Downarrow_s V_2$. If V_1 and V_2 are the results of successfully converging programs, then by Lemma 1 we have **runPar** $M \Downarrow_p V_1$ and **runPar** $M \Downarrow_p V_2$. From Theorem 1 we have $V_1 = V_2$. The other cases are proven similarly.

□

Note that many cases and supporting lemmas are left out for brevity and that the proof sketches provided are only meant to give the reader a high level intuition as to how the details of the proof fit together. Full details about the proof can be found in the Coq formalization at `http://people.rit.edu/ml9951/research.html`

6. Implementation

In addition to the formal semantics and determinism proof we have also begun a preliminary implementation as a part of the Manticore project. We have implemented the rollback mechanism and an IVar library using the BOM intermediate language that is used for much of the rest of the runtime system and thread scheduling infrastructure [FRR08]. One key feature that the BOM intermediate language has is first class continuations, which allow us to "reset" threads to previous points in their evaluation.

6.1 Threads in Manticore

In Manticore, threads are simply represented as a unit continuation and a pointer to thread local storage. We store the action list described in the formal semantics inside of thread local storage. When a thread is created, we can provide a cancelable object such that each time the thread is scheduled, it first checks to see if a flag in the cancelable object has been set and if so, it terminates. More details about about cancelation and thread scheduling can be found in [FRRS11].

6.2 IVars

An IVar is represented as a record almost identically as it is in the formal semantics. The main difference is in the list of dependent readers of an IVar. In the formal semantics, this is simply a multi set of thread IDs, however, in our implementation it is actually a tuple containing the cancelable object associated with the reader, a continuation corresponding to the current continuation of the reader at the point in which it read the IVar, and a pointer to the list of actions it has performed. When a thread reads from an IVar, it captures its current continuation, and stores it in the IVar along with its cancelable object and action pointer. In the event that a rollback is invoked, we recursively go through the list of actions to be rolled back doing the following for each action:

- If the action is a fork action, cancel the forked thread (cancelable object is stored in the action object) and append all of the forked thread's actions to the list of actions to be rolled back
- If the action is a read action, we simply continue with the rollback
- If the action is a write action, reset the IVar to empty, and process each of the dependent readers associated with this IVar.
  - When processing dependent readers, we recurse down the list of actions they have been performed and look for the oldest read action to the IVar being rolled back. Note that it must be the oldest action because if the thread read from the IVar multiple times, we need to reset it back to the point at which it read from the IVar for the first time.
  - We then reset this thread to the continuation associated with this read action and append the actions occurring after the read to the list of actions to be rolled back.

Note that we do not record an action for allocating an IVar. This is done in the formal semantics for the purposes of maintaining the well-formedness property and is not necessary to rollback the creation of IVars as they will simply be garbage collected. The reason this is important for preserving the well-formedness property is that after unspeculating a program state, it must be able to run forward to exactly the state it was in prior to unspeculating. This means that the names chosen for IVars in the heap must be the same as they were previously, which would not be possible if speculatively allocated IVars where not removed from the heap.

As a final technical detail, when "resetting" threads to previous points in their evaluation we actually simply cancel the thread to be reset. We then create a new thread with the same identity, except that it begins its evaluation at the continuation corresponding to the point in which it is to be "reset".

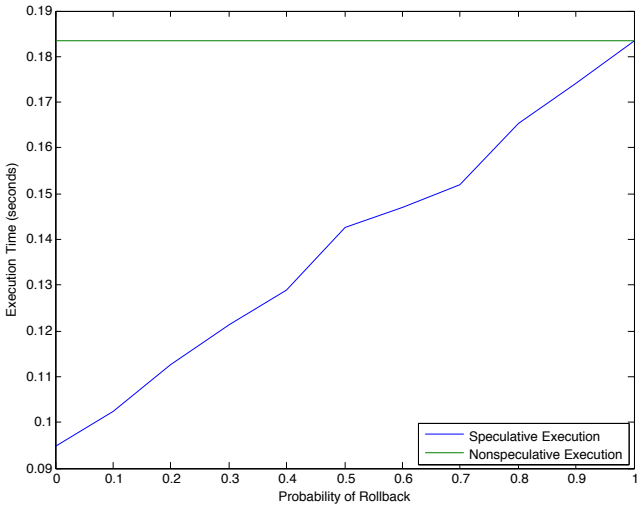


Figure 8. Rollback Overhead

7. Preliminary Results

Our implementation is still in its early stages, however, we have been able to perform some preliminary evaluation in order to give the reader a sense as to what sort of overhead is introduced by our logging and rollback mechanism in Manticore.

7.1 Producer Consumer

The first benchmark is a simple program that spawns two threads, one that repeatedly writes some arbitrary data to a linked list of IVars and another that reads each element of the linked list as it becomes available. This gives us some idea as to what sort of price we pay even if we have no interest in doing any sort of speculative computation. For a program that writes 5,000 IVars we see only a 5% slowdown relative to an implementation that performs no logging.

7.2 Measuring Rollback

In an effort to measure the overhead introduced by the rollback mechanism we have constructed a synthetic benchmark that forks a thread that speculatively writes to an IVar, and with a given probability raises an exception to rollback the write after a predetermined amount of time. After forking this thread, the main thread then reads from the speculatively written IVar in order to record a dependent reader and then enters a spin loop for the same predetermined amount of time as the forked thread. When a rollback occurs, the runtime system will then reset the written IVar to empty and reset the main thread to before the point that it read from the IVar. If a rollback does not occur, then the two spin loops are executed in parallel and should, in theory, achieve 2X speedup. Figure 8 shows the results of the experiment varying the probability of performing a rollback from 0 to 1 in 0.1 increments. The execution times for each probability interval are the average of 500 iterations. For the non speculative results, we simply run the two spin loops sequentially in order to get a baseline execution that does not involve the runtime system. The results indicate that for this particular scenario, the overhead of a rollback is essentially free. The average runtime of the non speculative case is 0.1833 seconds vs. 0.1834 for the average speculative runtimes with a 100% chance of a rollback occurring.

Certainly these results will vary based on the "size" of the rollback, meaning if we had more threads dependently reading from the IVar, or we were speculatively writing to more IVars, the execution time would definitely be different as the runtime system would need to do more work. Future work will include a more in-depth analysis of these parameters.

8. Related Work

This work builds on two broad categories of related projects, those that deal with deterministic parallelism in the presence of shared state, and those that deal with speculative parallelism.

8.1 Shared State

IVars were first proposed in the language Id [ANP89], which is also a parallel functional language, however, they sacrifice determinism by also adding MVars, which are shared references that can be written an arbitrary number of times with implicit synchronization. More recently, IVars have been adopted by parallel languages such as the Par Monad of Haskell [MNP11] and some of the Concurrent Collections work[BBC+10], however, neither of these works support speculative parallelism.

LVars are a new abstraction that were recently proposed by Kuper et al. [KN13, KTKN14] that generalize IVars to allow multiple writes, but restrict that they must be monotonically increasing in some fashion. LVars suffer from the same problem as IVars in that they also lose their determinism guarantee in the presence of cancelation. More recently, they have proposed an elegant solution for combining LVars with speculative parallelism [KTTN14], where threads can perform speculative work (i.e. can potentially be canceled) if they are read only. They do however, allow speculative threads to write to memoization tables such that they can "help out" other threads, however, one shortcoming to this solution is that performance becomes difficult to reason about as a programmer. Note that parallel speedup is only achieved if the speculative thread is able to write to the memoization table before another thread needs this result. If it does not make it there in time, then not only is there no benefit, but the speculative thread corresponds to wasted work. On the other hand in this work, if the commit portion of a parallel tuple finishes before the speculative threads, it simply waits for them to complete and then joins with them.

Welc et al. proposed a solution for enforcing a sequential semantics for Java futures [WJH05, NZJ08], a concurrency abstraction taken from Multilisp [Hal85]. They too extend their runtime system to enforce deterministic execution, but in a very different way relative to our approach. First, for each thread that is spawned, they create a new copy for each object that it writes to. This does not allow for the type of fine grained sharing that we are able to support in our producer-consumer benchmark. Additionally, if their runtime system detects that a thread has violated the sequential semantics, they restart the thread from the beginning, where as our approach is able to simply rollback a thread to the exact point in which the violation occurred, avoiding redundant work.

Bocchino et al. give a region based type and effect system for guaranteeing determinism at compile time for parallel Java programs [BAD+09]. Their approach requires annotations on Java programs specifying what "regions" objects are allocated in. They then extend their Java compiler to statically verify that concurrently executing threads do not manipulate objects that are allocated in the same regions.

8.2 Speculative Parallelism

There is a large body of work that has been done on transparent speculative parallelism, where the compiler and runtime system automatically perform value prediction and control the amount of parallelism in the program, however, more relevant to this work is the notion of programmable speculative parallelism. Programmable

speculative parallelism was first introduced in [Bur85] in the context of the Mirranda language. Their approach uses a purely functional language, so they do not deal with any of the rollback issues that we present in this work.

More recently, Prabhu et al. propose language constructs for specifying speculatively parallel algorithms and formalize their semantics using the lambda calculus extended with shared references [PRV10]. Rather than providing a runtime system that performs rollbacks in the event of a miss-speculated value, they describe an analysis that is performed at compile-time that guarantees that they will never need to perform any rollbacks. Their analysis is necessarily conservative, making certain types of sharing patterns not expressible in their language.

Software Transactional Memory (STM) can be seen as a form of speculative parallelism. Transactional memory allows programmers to wrap code in "atomic" blocks that the runtime system guarantees to be executed in isolation [ST95]. STM uses a form of "optimistic" concurrency where threads execute code inside of transactions and upon completion check to see if any of the memory locations they read or wrote were compromised by other concurrently running threads. If so, they abort the transaction and restart from the beginning. Transactional memory is different from our work in the sense that they provide no guarantees about deterministic execution, and is concerned only with atomicity.

9. Conclusions and Future Work

Giving parallel constructs a deterministic semantics makes reasoning about parallel programs substantially easier. In this work we have shown how we can extend the expressiveness of Manticore by adding IVars and still be able to guarantee deterministic execution. We have formalized the semantics of this extended language and provided a proof of its correctness using the Coq proof assistant.

For our preliminary implementation we have tried to remain faithful to the formal semantics as much as possible to ensure correctness without worrying too much about performance. In the immediate future we plan on fine tuning our implementation of the runtime system in Manticore to improve efficiency and perform a more thorough evaluation. This idea of combining speculative parallelism with IVars is a new programming model that has not been explored elsewhere so coming up with interesting benchmark programs is also a bit of a challenge and something we look to explore further in the future. Lastly, we believe it would be interesting in generalizing our approach to the LVars programming model. As mentioned in the previous section, this is an extension of the IVars model that permits multiple writes to shared references, so extending both our implementation and our formal semantics presents some interesting challenges.

References

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, **11**(4), October 1989, pp. 598–632.

[BAD+09] Bocchino, Jr., R. L., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA'09*, Orlando, Florida, USA, 2009. ACM, pp. 97–116.

[BBC+10] Budimlić, Z., M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tacsirlar. Concurrent collections. *Sci. Program.*, **18**(3-4), August 2010, pp. 203–217.

[Bur85] Burton, F. W. Speculative computation, parallelism, and functional programming. *IEEE Trans. Computers*, **34**(12), 1985, pp. 1190–1193.

[FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, Victoria, BC, Candada, September 2008. ACM, pp. 241–252.

[FRRS08] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, Victoria, BC, Candada, September 2008. ACM, pp. 119–130.

[FRRS11] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, **20**(5–6), 2011, pp. 537–576.

[Hal85] Halstead Jr., R. H. Multilisp: A language for concurrent and symbolic computation. *ACM TOPLAS*, **7**, 1985, pp. 501–538.

[JLM+09] Jones, C. G., R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, Berkeley, California, 2009.

[KN13] Kuper, L. and R. R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *FHPC'13*, Boston, Massachusetts, USA, 2013. ACM, pp. 71–84.

[KTKN14] Kuper, L., A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. In *POPL'14*, San Diego, California, USA, 2014. ACM, pp. 257–270.

[KTTN14] Kuper, L., A. Todd, S. Tobin-Hochstadt, and R. Newton. Taming the parallel effect zoo. In *PLDI'14*, Edinburgh, UK, 2014. ACM.

[MNP11] Marlow, S., R. Newton, and S. Peton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*. ACM, 2011, pp. 71–82.

[NZJ08] Navabi, A., X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPOPP'08*, Salt Lake City, UT, USA, 2008. ACM, pp. 23–32.

[PRV10] Prabhu, P., G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI'10*, Toronto, Ontario, Canada, 2010. ACM, pp. 50–61.

[ST95] Shavit, N. and D. Touitou. Software transactional memory. In *PODC '95*, Ottowa, Ontario, Canada, 1995. ACM, pp. 204–213.

[WJH05] Welc, A., S. Jagannathan, and A. Hosking. Safe futures for java. In *OOPSLA'05*, San Diego, CA, USA, 2005. ACM, pp. 439–453.

Acknowledgments

Thanks to Vitor Rodriquez and Zack Fitzimmons for their many useful comments that helped to improve this work. This research is supported by the National Science Foundation under Grants CCF-0811389 and CCF-101056

10. Appendix

$$\boxed{\text{Finished}(H; T)}$$

$$\frac{\text{Finished}(H; T_1) \mid \text{Finished}(H; T_2)}{\text{Finished}(H; T_1 \mid T_2)} \qquad \frac{H(x) = \langle \mathbf{C} \rangle}{\text{Finished}(H; \Theta[S_1, S_2, E[\textbf{get } x]])}$$

$$\frac{}{\text{Finished}(H; \Theta[\cdot, S_2, \textbf{return } M])} \qquad \frac{}{\text{Finished}(H; \Theta[S : A, S_2, M])}$$

Figure 9. Finished Thread Pool

$$
\begin{array}{rcl}
adopt(S : (R, x, M), E, M') & = & adopt(S, E, N) : (R, x, E[\textbf{specJoin}(N, M)]) \\
adopt(S : (W, x, M), E, M') & = & adopt(S, E, N) : (W, x, E[\textbf{specJoin}(N, M)]) \\
adopt(S : (A, x, M), E, M') & = & adopt(S, E, N) : (A, x, E[\textbf{specJoin}(N, M)]) \\
adopt(S : (F, \Theta, M), E, M') & = & adopt(S, E, N) : (F, \Theta, E[\textbf{specJoin}(N, M)]) \\
adopt(S : (S, M), E, M') & = & adopt(S, E, N) : (S, E[\textbf{specJoin}(N, M)]) \\
adopt(S : CSpec, E, M') & = & adopt(S, E, N) : CSpec
\end{array}
$$

Figure 10. Action Adoption

$$
\begin{array}{rrcl}
 & x & \in & Var \\
\text{Values} & V & ::= & x \mid i \mid \backslash x.M \mid \textbf{return}\ M \mid M >>= N \mid \textbf{runPar}\ M \mid \textbf{fork}\ M \mid \textbf{new} \mid \textbf{put}\ i\ M \\
 & & \mid & \textbf{get}\ i \mid \textbf{done}\ M \mid \textbf{spec}\ M\ N \mid \textbf{specRun}(M, N) \mid \textbf{specJoin}(M, N) \mid \textbf{raise}\ M \\
 & & \mid & \textbf{handle}\ M\ N \\
\text{Terms} & M, N & ::= & V \mid M\ N \mid \cdots \\
\text{Heap} & H & ::= & H, x \mapsto iv \mid \cdot \\
\text{IVar State} & iv & ::= & \langle\rangle \mid \langle M \rangle \\
\text{Evaluation Context} & E & ::= & [\cdot] \mid E >>= M \mid \textbf{specRun}(E, M) \mid \textbf{handle}\ E\ N \mid \textbf{specJoin}(N, E) \\
\text{Thread Pool} & T & ::= & \cdot \mid (T_1 \mid T_2) \mid M \\
\text{Configuration} & \sigma & ::= & H; T \mid Error
\end{array}
$$

Figure 11. Original Par Monad Syntax

$$
\text{FPar}\ \frac{\text{Finished}_\text{p}(H; T_1) \quad \text{Finished}_\text{p}(H; T_2)}{\text{Finished}_\text{p}(H; T_1 \mid T_2)} \qquad \text{FBlocked}\ \frac{H(x) = \langle\rangle}{\text{Finished}_\text{p}(H; E[\textbf{get}\ x])} \qquad \text{FDone}\ \frac{}{\text{Finished}_\text{p}(H; \textbf{return}\ M)}
$$

Figure 12. Original Par Monad Finished

$$
\text{RunPar}\ \frac{(M >>= \backslash x.\textbf{done}\ x) \rightarrow_p^* \textbf{done}\ N \mid T \quad N \Downarrow V \quad \text{Finished}_\text{p}(T)}{\textbf{runPar}\ M \Downarrow V}
$$

$$
\text{Eval}\ \frac{M \Downarrow V}{H; T \mid E[M] \rightarrow_p H; T \mid E[V]} \qquad \text{Bind}\ \frac{}{H; T \mid E[\textbf{return}\ N >>= M] \rightarrow_p H; T \mid E[M\ N]}
$$

$$
\text{BindRaise}\ \frac{}{H; T \mid E[\textbf{raise}\ N >>= M] \rightarrow_p H; T \mid E[\textbf{raise} N]} \qquad \text{Handle}\ \frac{}{H; T \mid E[\textbf{handle}(\textbf{raise}\ M) N] \rightarrow_p H; T \mid E[M\ N]}
$$

$$
\text{HandleRet}\ \frac{}{H; T \mid E[\textbf{handle}(\textbf{return}\ M) N] \rightarrow_p H; T \mid E[\textbf{return}\ M]} \qquad \text{Fork}\ \frac{}{H; T \mid E[\textbf{fork}\ M] \rightarrow_p H; E[\textbf{return}()] \mid M \mid T}
$$

$$
\text{New}\ \frac{x \notin Domain(H) \quad H' = H[x \mapsto \langle\rangle]}{H; E[\textbf{new}] \mid T \rightarrow_p H'; T \mid E[\textbf{return}\ x]} \qquad \text{Get}\ \frac{H(x) = \langle M \rangle}{H; E[\textbf{get}\ x] \mid T \rightarrow_p H; E[\textbf{return}\ M] \mid T}
$$

$$
\text{Put}\ \frac{H(x) = \langle\rangle \quad H' = H[x \mapsto \langle M \rangle]}{H; E[\textbf{put}\ x\ M] \mid T \rightarrow_p H'; E[\textbf{return}()] \mid T} \qquad \text{Spec}\ \frac{}{H; E[\textbf{spec}\ M\ N] \mid T \rightarrow_p H; E[\textbf{specRun}(M, N)] \mid T}
$$

$$
\text{SpecRun}\ \frac{}{H; E[\textbf{specRun}(\textbf{return}\ M, N)] \mid T \rightarrow_p H; E[\textbf{specJoin}(\textbf{return}\ M, N)] \mid T}
$$

$$
\text{SpecRaise}\ \frac{}{H; E[\textbf{specRun}(\textbf{raise}\ M, N)] \mid T \rightarrow_p H; E[\textbf{raise}\ M] \mid T}
$$

$$
\text{specJoin}\ \frac{}{H; E[\textbf{specJoin}(\textbf{return}\ M, \textbf{return}\ N)] \mid T \rightarrow_p H; E[\textbf{return}(M, N)] \mid T}
$$

$$
\text{specJoinRaise}\ \frac{}{H; E[\textbf{specJoin}(\textbf{return}\ M, \textbf{raise}\ N)] \mid T \rightarrow_p H; E[\textbf{raise}\ N] \mid T}
$$

Figure 13. Original Par Monad Operational Semantics

Towards Execution of the Synchronous Functional Data-Flow Language SIG

[Draft Paper]

Baltasar Trancón y Widemann
Ilmenau University of Technology
baltasar.trancon@tu-ilmenau.de

Markus Lepper
semantics GmbH

Abstract

SIG is the prototype of a purely declarative programming language and system for the processing of discrete, clocked synchronous, potentially real-time data streams. It aspires to combine good static safety, scalability and platform independence, with semantics that are precise, concise and suitable for domain experts. Its semantical and operational core has been formalized. Here we discuss the general strategy for making SIG programs executable, and describe the current state of a prototype compiler. The compiler is implemented in Java and targets the JVM. By careful cooperation with the JVM JIT compiler, it provides immediate executability in a simple and quickly extensible runtime environment, with code performance suitable for moderate real-time applications such as interactive audio synthesis.

1. Introduction

SIG is the prototype of a purely declarative programming language and system for the processing of discrete, clocked synchronous, potentially real-time data streams. It is designed to support both visual (data-flow diagram) and textual (functional) programming styles, to be scalable to complex tasks, and to be interoperable with a wide variety of execution platforms and legacy code bases.

The potential application fields for SIG are in science, such as modelling and simulation of system dynamics, in engineering, such as sensor data processing and control in embedded systems, as well as in art, such as audio synthesis and computational music.

The strategic vision of the SIG project is to leverage the safety and productivity of modern language technology in a system that can be used effectively, and its actual semantics understood, by domain experts. We believe that this could constitute a significant improvement over the state of the art, which is plagued by twin evils: Application development that uses the established domain-specific tools must deal with their outdated technology and ill-defined semantics; while development that avoids them exposes domain experts as programming laymen to low-level general-purpose programming languages with inadequate expressivity.

The full realization of this vision is of course a long-term goal, and would require substantial effort in order to implement an infrastructure consisting of development tools, runtime environments, algorithmic libraries, bindings for indispensable legacy code, etc. A first major step has been reported on in [7], where the computational framework of SIG (i.e. denotational semantics, core operations, intermediate code representation, and their precise relationships) are discussed in due technical detail.

In the present paper, we report on the next step: a prototype SIG runtime environment that emphasizes integrated tool chains, and immediate and transparent execution of code in various phases of the interpreted–compiled spectrum. This allows us to demonstrate SIG in application areas with interactive systems and moderate real-time requirements, simultaneously showcasing the expressivity and practical feasibility of the language. A running demo package for audio synthesis has recently been published [8].

2. SIG at Work

2.1 Design Considerations

With regard to notation, the data-stream programming world is divided into a visual and a textual camp.

The visual approach, employing data-flow diagrams as the main notation for algorithms, is traditionally favoured by domain experts. Typical programming systems include Simulink for engineering applications, Max/MSP for audio and artistic performance, or the "system dynamics" school of computational modelling of complex systems. Programs are graphs built from *boxes* that specify computations, and *wires* that carry data flow. In spite of the appealing ability to visualize the routing of data flow very intuitively, the diagram approach is known to suffer from poor scalability, frequent confusion of layout and semantics, and lack of support for other essential aspects of algorithms: data types, case distinctions, abstraction and reuse, state and initialization.

These weaknesses are conspicuously absent in functional programming, which features well-understood remedies such as type inference, algebraic data types and pattern matching, anonymous and higher-order functions, and purely declarative semantics. It is therefore no surprise that *functional reactive programming* (FRP) is hailed as an elegant foundation for data-stream programming by the more semantically-minded. Diagrams can be expressed in this framework in terms of *arrows* [2].

The SIG approach aims at neutrality between visual and textual frontend representations, and consequently has been designed around a functional core representation that can represent both naturally; see [7]. In comparison with FRP, SIG takes a characteristically different route: On the one hand, the model of time as discretized by clock ticks at one or several constant rates, is much

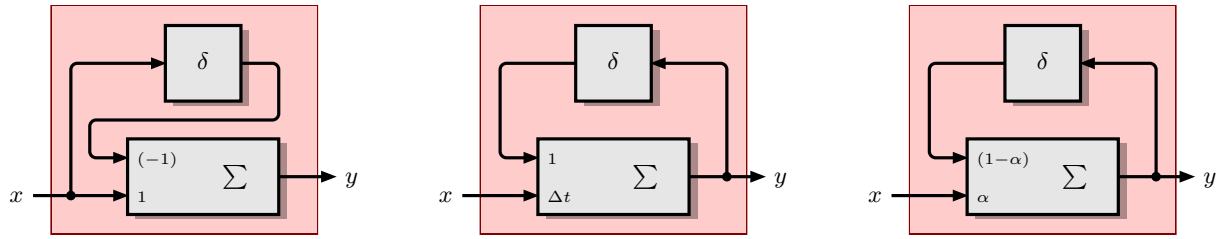


Figure 1. Linear stream programming with delay: *left to right* – backward difference; discretized integral; first-order low-pass filter.

less generic and abstract than in general FRP, which supports also spontaneous events and continuous signals. On the other hand, this restrictiveness is exploited in a computational model that brings denotational semantics and low-level implementation techniques to a very close congruence, and is more orthogonal to other features of functional programming, most notably pattern matching, than current arrow-based FRP frameworks.

2.2 Frontend

True to the tradition of synchronous data-flow programming, SIG programs are represented in a style that abolishes all kinds of explicit sequential control flow, such as blocks, loops or recursion. All computations are specified as if operating on instantaneous data at a single clock tick, and are understood to be implicitly lifted to whole streams by iteration at their respective clock rate, without spontaneous events or termination. All data flow is conceptually instantaneous, unless explicitly delayed. Nontrivial behavior in general (anything other than a function mapped over a stream) arises from delayed interference, and state in particular arises from delayed feedback. Instantaneous feedback (i.e. circular data flow *not* passing through a delay operator) is forbidden. Hence no causal singularities arise; scheduling can be decided modularly and statically, and no fixpoints need be considered. For simplicity, we consider only one primitive delay operator δ, which delays an arbitrary stream for exactly one clock tick (i.e. prepends some specified or default initial value).

A great variety of important building blocks for stream processing algorithms can already be specified in the simplest form of this style; see Figure 1 for a gallery of ubiquitous components built from elementary arithmetics and delay.

Evidently, the diagram approach shines where data has *product* structure and routing is static: a tuple of values is nicely visualized as a bus of wires. By contrast, data with *coproduct* structure, where routing depends on dynamic case distinctions, is handled rather awkwardly. It is hence no surprise that automata (a principal algorithmic manifestation of coproduct-oriented computation) are supported by a *different* diagram language in visual approaches (e.g. Stateflow for Simulink), if at all.

As an archetypal running example, consider the *sample and hold* (S&H) operator, which either forwards its current input x or retains its previous output y, depending on an external trigger t taking the values $\{\mathsf{S}, \mathsf{H}\}$. This functionality can be specified conveniently in a diagram as depicted in Figure 2, using an ad-hoc multiplexer component. Note that we refrain from the "engineering practice" of encoding the range of t numerically, for obvious reasons of clarity and safety. An equivalent specification can be given textually as depicted in Figure 3, using an enumerated type and the SIG box notation. Note that this notation differs from lambda terms by naming both inputs and outputs explicitly and symmetrically.

The multiplexer approach to control flow, while handy for simple situations, has rather poor expressivity and scalability. For instance, consider the evident refactoring of the S&H component

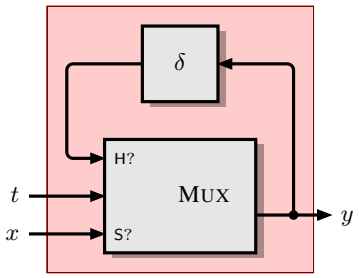


Figure 2. Triggered S&H; diagram with multiplexer

```
type trigger = { S, H }

[
  in x : real , t : trigger
  out y : real
where
  y := case t of {
         S → x
         H → delay(y)
       }
]
```

Figure 3. Triggered S&H; SIG notation (box)

from a functional programmer's viewpoint: Since the input x is irrelevant in the *hold* case, a more economic interface would fuse the two inputs, using a well-known algebraic datatype as depicted in Figure 4. Note that Scala vernacular is used, Haskell enthusiasts may substitute *Maybe*. Whereas this encoding is easily processed with pattern matching clauses, there is no obvious viable generalization of multiplexing to do the job. Apparently the challenging feature is the combination of case distinction and data unpacking, as effected by pattern constructors, as a single atomic operation.

To bring the S&H example even closer to traditional functional programming style, a lambda-style asymmetric function abstraction and named access pattern may be used, as depicted in Figure 5. Note that delayed feedback from the output, a ubiquitous pattern in stream programming, practically prevents the function body expression from being anything than a locally bound variable, hence the gain in conciseness over the box notation is not quite as great as in noncircular cases.

However, the **where** clause gives an impression of the unification of the diagram and expression paradigms that SIG aspires to. Ideally, the programmer should be free to combine the notations orthogonally, each where it shines: Expressions for tree-shaped flow with irrelevant intermediates and coproduct-structured data; diagrams for irregular and circular flow and product-structured data.

```
type option(t) = { some(t), none }

[
  in x : option( real )
  out y : real
where
  y := case x of {
         some(v) → v
         none    → delay(y)
       }
]
```

Figure 4. Triggerless S&H; SIG notation (box)

```
{
  x : option( real ) → y
  where y := getOrElse(x, delay(y))
}
```

Figure 5. Triggerless S&H; SIG notation (lambda)

The SIG language addresses these issues by program reduction to a core layer with primitive operations that can implement multiplexers and pattern matching equally naturally, and deal with delay in a semantically clean and operationally useful way.

2.3 Core

The key insight behind the semantic framework of SIG is that three essential description formats can be made to coincide [7]:

1. adjacency-based algebraic hypergraph representation of data-flow diagrams (with wires as nodes and boxes as hyperedges, respectively);
2. administrative normal form of functional program expressions, or rather the equivalent static single-assignment (SSA) form;
3. intensional definition of local, elementwise semantics given as a Mealy-style combined I/O-and-transition relation (giving rise to global, stream function semantics by coinduction).

The full details of the theoretical foundation of (3.) and the algorithmic derivation of (2.) from a functional frontend notation can be found in [7]. In the present section, we summarize the key points. The following sections give the main technical contribution of the present paper, by discussing the further use of (2.) in a compiler pipeline.

2.3.1 Delay Elimination

The notation of stream computations in terms of per-element and delay operators, while intuitively convenient, is awkward to reason with directly in a declarative language processing framework. Stream-level behavior is not specified fully by element-level input/output relations, as delay operators break referential transparency.

Hence SIG eliminates delay operators en route to the core layer, by introducing a matching pair of pre- and post-state variables for each occurrence of δ, which then becomes a pair of independent simple equations, forwarding input to post-state and pre-state to output, respectively. Apparently circular data flow is admissible if and only if the circles are eliminated by the splitting of all delay operators.

It is implied that the post-state values of each clock cycle flow to the corresponding pre-state variables of the next cycle. That is, the quaternary relation of input, output, pre- and post-state specifies a stream-transducing Mealy machine.

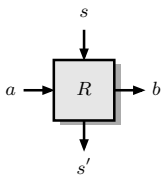


Figure 6. Stateful single-step computation model

```
[
  in x : real , t : { S, H }
  out y : real
  state z : real                  // implies z' : real
where
  y := case t of {
         S → x
         H → z
       }
  z' := y
]
```

Figure 7. Triggered S&H; SIG notation (no delay)

The approach can be visualized as depicted in Figure 6. Explicit data flow in the sense of the functional composition of computations, proceeds left to right. Temporal data flow proceeds top to bottom. The stream-level global semantics of a program is given by replicating its element-level relation ω times along the vertical axis, up to initial values for the top end. If the element-level relation is a total function, as dictated for complete SIG component definitions, then the corresponding denotational semantics is captured neatly by coinduction, as elaborated in [7].

The reduction of delay to state can also be notated textually. Figure 7 depicts the result of delay elimination from the program in Figure 3. Note that reduction to the core layer also implies the naming of all intermediate values, as customary for administrative normal or SSA form, although the S&H example does not exhibit this feature.

2.3.2 Control Elimination

Control flow is an awkward feature from a data flow-centric perspective. The SIG approach reduces control flow to data flow for the purpose of maximally parallel core-layer semantics. Backends are free to implement these directly, as in hardware, or to emulate them by reconstructed control flow, as on sequential machines.

The rationale here is that the automatic sequentialization of parallel programs is conceptually much simpler, and effectively achieved with standard compiler technology, than the reverse problem, which remains the elusive holy grail of traditional high-performance computing.

The elimination of control is achieved by creatively abusing the φ operator introduced by SSA, and complementing it with a novel, dual γ operator, to be introduced below. In its original sense, φ multiplexes a number of inputs, understood as alternative values of the same variable produced by different control predecessors.

Of course, there is to be no such thing in SIG; the very purpose of the core layer is to gather all computations in a single basic block. Instead, the SIG-style φ operator multiplexes values from (the right hand sides of) different clauses of a case distinction, depending on the success of pattern matching (of their respective left hand sides).

To this end, all *internal* variables of a component are tacitly augmented to admit an additional value $\perp$. Note that $\perp$ merely signifies that no value is currently available. This is a decidable situation, since program divergence, the usual meaning of $\perp$ in the

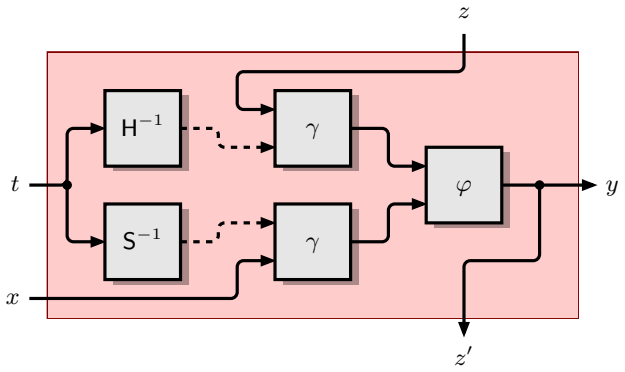


Figure 8. Triggered S&H; diagram (core)

```
[
  in x : real, t : { S, H }
  out y : real
  state z : real                    // implies z' : real
where
  local c, d : control
  local v, w : real
  c := S⁻¹(t)
  v := guard(x, c)
  d := H⁻¹(t)
  w := guard(z, d)
  y := choose(v, w)
  z' := y
]
```

Figure 9. Triggered S&H; SIG notation (core SSA)

semantics of recursive functions, is excluded. Ordinary elementary operations are lifted strictly; if any input is $\bot$, then so are all outputs.

Each partial computation, such as a single clause of a case distinction, can be represented uniformly as a "left-top-total" relation in the sense of Figure 6, where missing cases are mapped to $\bot$. A φ node then simply chooses nondeterministically among its non-$\bot$ inputs, or yields $\bot$ if there is none.

The success of pattern matching is communicated by adding to each pattern constructor an additional "control" output indicating success. The type of these is nominally a singleton $\{\top\}$, augmented to a Boolean control type $\{\top, \bot\}$. If the pattern succeeds, then regular outputs unpack the data constructor argument values, which are non-$\bot$ by strictness, and the control output is $\top$. If the pattern fails, then all outputs are $\bot$. Note that this encoding may appear redundant for data constructors with arguments, but it is *not* for the common case of nullary constructors, where the corresponding pattern is a Boolean test.

The selection of computations is expressed by a guard operator γ. It takes a single data input and arbitrarily many control inputs. The data is forwarded if no control is $\bot$. Otherwise, the output is $\bot$ as well. A clause from a case distinction is then selected by guarding each result of its right hand side with all control values issued by its left hand side.

The elimination of control can be specified formally by tedious but straightforward syntax-directed rewrite rules, see [7]. The application to the S&H example is depicted as a diagram in Figure 8. Data and control wires are indicated by solid and dashed lines, respectively.

A textual representation is depicted in Figure 9. As stated in the beginning of section 2.3, the set of assignments can be read consistently in several ways: as the adjacency list of the hypergraph corresponding the diagram in Figure 8; as a normalized functional program in SSA form consisting of a monolithic basic block; as the intensional definition of an element-level semantic relation by set comprehension in the style of the Z notation.

With respect to the former two, note that the textual single-assignment constraint coincides with the usual diagram constraint that distinct outputs must not collide on a shared wire.

Note that γ and φ nodes act as data-carrying conjunction and disjunction operators, respectively. They can be reduced further to logical expressions in conjunctive normal form. Hence interesting static properties such as definite single assignment of outputs can be checked using off-the-shelf SAT solver technology.

2.4 Backend

While SIG is designed with maximal platform independence in mind, there are a number of general assumptions that constitute a loose execution model.

2.4.1 Composition of Components

A key feature of SIG for scalability and efficient use is full compositionality. The computational box abstraction unifies primitive computations and user-defined subprograms. Thus complex stream-processing systems are constructed and scoped hierarchically. A reference to a defined component can be inlined (i.e. the box replaced by its innards) without affecting program semantics.

This seems like an obvious, almost trivial, property of a functional language, but has decidedly nontrivial consequences in a time-aware setting. Most importantly, the wire abstraction of data flow must not have intrinsic delay, as this would break the scale-free semantics and disallow the optimizations that routinely go with inlining, such as copy propagation. All data flow, except for explicit delay, must be undistinguishable from instantaneous transport.[1] This places strict bounds on the depth of computational networks that can be implemented with given real-time constraints.

2.4.2 Global Control

The realization of a component performs a single step that processes one element of each connected data stream. This involves the updating of pre-state from the preceding post-state, the consumption of inputs, and the production of intermediate values, post-state and outputs, with no particular order of the subtasks specified. During the execution of a step, each component is responsible for having its subcomponents executing a step of their own, respecting data flow constraints.

On a sequential platform, this means that the producer of each stream must execute before its respective consumers. SIG is designed such that a schedule can be devised compositionally and ahead of time. Note that the order of assignment statements depicted in Figure 9, while semantically irrelevant, is a valid sequential schedule of the component; each variable is written before it is read. A sequential implementation is free to choose this or any other valid order, as long as the choice remains transparent to the external observer.

In many cases, ahead of time means at compile time, but various advanced but typical applications require the reconfiguration of computations by parts of the program running at a slower rate. Support for such hierarchically dynamic systems in SIG is a matter for future research.

[1] To use a physical metaphor, the SIG model of spacetime is the Newtonian $c \to \infty$ limit of relativity.

Parts of a SIG program may operate at the same or at different clock rates. The complete program is sliced into its synchronous parts (i.e. each operating at a single rate) and re-sampling connectors. The details are a matter of future research. The runtime environment triggers the execution steps of each root component centrally, with the prescribed rate, in a conceptually infinite loop. Components may not choose to terminate this loop spontaneously.

2.4.3 Inter-Component Communication

Communication between components (i.e. how wires work conceptually) in SIG is characterized by pull-based shared memory. Actual implementations may use arbitrary mechanisms to achieve the specified behavior.

Each component has the exclusive ownership of a distinct writable storage location for each of the output streams it produces. Consumers can access the current value of a stream by reading from this location. All activity is driven by the external clock; neither production nor consumption constitutes an observable event.

On the one hand, the current element of each stream is defined by the value of its location at the clock tick. The producer must be given the opportunity to write an up-to-date value in time. Otherwise, the previous value is tacitly retained. By writing to a location, the previous value is generally made inaccessible. If needed, it must be retained elsewhere, typically using delay.

All outputs of a component change apparently simultaneously. Inconsistent states, such as temporarily arising from implementation by a sequence of write operations, must not be observed. Spontaneous events of the execution environment must be quantized at some clock rate, and reacted on by polling.

On the other hand, each component is oblivious to the consumers of the outputs it produces. Reading the current element of a stream from a location does not notify the producer. Demand for a value does not trigger its computation, nor does absence of demand prevent it. Weird effects such as the infamous *time leaks* of lazy FRP do not arise.

2.4.4 Total Computations

The shared-memory communication model implies that, without additional out-of-band information, is is conceptually impossible *not* to yield a result. In embedded systems, this is often a very practically the case.[2] SIG components are generally implementations of total functions; they must not fail to define their outputs for any valid combination of input and pre-state.

By contrast, arbitrary networks of components have more freedom. They can produce $\perp$ values, and even be nondeterministic. In the disciplined textual frontend language of SIG, the former arises from partial computations such as incomplete case distinctions, and the latter arises from overlapping cases, since SIG has no implicit first- or best-fit disambiguation rules. By liberal use of the core operations γ and φ, a wider variety of similar situations can be created.

Only when a network of components is explicitly designated as the definition of a component by the programmer, a proof obligation for totality and determinism is entailed. Since the question is evidently undecidable in general for all nontrivial collections of primitive operations, a statically checkable approximation is needed. For the disciplined approach (where unsafe subcomputations arise from pattern matching), the requirement that case distinctions be complete and non-overlapping is a natural candidate, and can be checked effectively using standard compiler technology. Possible relaxations, as well as the general case of arbitrarily mixed core operations, are left for future research.

[2] As has been demonstrated drastically by the botched first launch of the Ariane 5 rocket.

3. Compiler

3.1 Architecture and Environment

The current SIG compiler and execution environment is written in Java. The parser is generated by a variant of the ANTLR[3] tool. Syntax trees are mapped to an intermediate representation (IR) as specified in [7], and the various subsequent program transformations towards the SSA form are implemented using a visitor style pattern approach. The IR data model and the visitor machinery are generated from a very concise ($\sim$200 lines) specification by the UMOD tool [3].

Programs in IR can be executed on the fly by an interpreter, or translated to JVM bytecode for better performance. A joint communication API makes the choice of the execution strategy transparent, on a per-component basis. Bytecode is produced in a closed loop and fed directly to the JVM class loader, without the need to call external tools. Alternatively, the bytecode can be stored and compiled to machine code by an external static code generator.

Theoretically, SIG programs can be modularized and compiled separately, although the frontend notation has no module system yet. However, for real-time applications, we expect that satisfactory results require whole-program compilation, in particular since many important analyses (e.g. worst case execution time) work best globally. The non-recursive nature of SIG data flow networks ensures that conceptual boundaries which exist in well-structured source code can be eliminated during compilation by aggressive inlining. Performance-critical application tend to be small enough for whole-program compilation to be feasible.

3.2 Runtime Interface

3.2.1 Type Specialization

Several basic data types of the SIG frontend are mapped directly to their Java/JVM counterparts, such that primitive operations can be used and the dynamic allocation of boxing objects can be avoided. Computations that declare only variables of such types are guaranteed to run without the use of the JVM allocator, and hence without triggering the garbage collector, which greatly enhances real-time responsiveness.

In particular, the Java/JVM types `int` and `double` are supported. The Java frontend type `boolean` is supported as well, which is encoded as the subset $\{0, 1\}$ of `int` on the JVM. Following this example, arbitrary user-defined enumerated types (i.e. algebraic data types with nullary constructors only) are encoded as subsets $\{0, \ldots, n-1\}$ of `int`. The extra value $\perp$ is encoded by pairing each variable of primitive type with a `boolean` control variable. Types that have no primitive mapping are encoded as objects.

3.2.2 Data Interfaces

For the sake of abstraction, the interfaces of components admit two different perspectives. The internal perspective is symmetrical with respect to input and output. It identifies variables of both kinds formally by locally scoped names, and operationally by self-owned storage locations. This view has been demonstrated in the examples of the SIG box notation.

By contrast, the external view treats input and output asymmetrically. Each component publishes its outputs passively by implementing an API Source for querying their current values. Conversely, inputs are supplied by reference to another instance of the API Source, which the component may query actively. Variables of either kind are identified by their position in the list of respective parameter declarations, regardless of their internal names. Thus the principle of alpha equivalence carries over from conventional functional programming.

[3] http://www.antlr.org

```
interface Source {
  int     getInt     (int index);
  double  getDouble  (int index);
  boolean getBoolean (int index);
  Object  getValue   (int index);
}
```

Figure 10. Data API

The API needs to strike a pragmatic balance. On the one hand, static safety and efficiency of data flow demand a high degree of specialization. On the other hand, ease of use and efficiency of caller logic demand a uniform access pattern. In the current implementation, we have chosen a middle road. The uniform interface is depicted in Figure 10. It specializes access according to implementation data types, but not according to parameter position. A critical evaluation of the actual performance and comparison with alternative approaches is a matter for future research.

Note that the API is mainly employed at system boundaries. Globally, instances are supplied by the runtime environment and the compiled program for system inputs and outputs, respectively. Locally, API encapsulation arises at metaprogramming stage boundaries, where one part of the running system configures another, to be run at a faster rate. Within relatively static component networks, the SIG compiler is expected to perform whole program optimization, resulting in the elimination of intermediate interfaces by inlining.

3.2.3 Component Instantiation

Metaprogramming capabilities are essential to the SIG approach, because the greatly amplify the scalability of the program development process. We follow the *staged* metaprogramming paradigm à la MetaML[6], where code fragments can be *quoted* and *spliced*, the meta equivalent of function abstraction and application, respectively, in ordinary higher-order functional programming. The current implementation has prototypic support. The details of notation and semantic constraints of SIG metaprogramming are a matter for future research.

Care must be taken when lifting a first-class notion of computation as data to the scenario of elementwise stream processing. There is no evident canonical explication of, say, a stream of stream functions. The dilemma is rather subtle: On the one hand, application to single elements is not functional application, due to hidden state transitions. On the other hand, application to whole streams, which is perfectly functional, is never expressed directly in the language.

How staged metaprogramming can help to clarify these issues is a matter of ongoing research. The full details are to be discussed in a forthcoming companion paper. For the present, it suffices to state informally that stages break synchronization. From the perspective of earlier stages, later stages are code objects to be configured to run independently, at a faster rate. Conversely, from the perspective of later stages, earlier stages are represented as creation-time snapshots only. This asymmetricity allows to keep violations of referential transparency, incurred by the use of delay/state, under the hood of the implementation.

The technical realization of this scheme in the Java-based runtime environment uses a three-tiered factory model, with one layer of abstraction each above and below the representation of components.

The highest level of abstraction, and the unit of *implementation*, is the Template. It corresponds to the defining expression of a component object (i.e. a quotation in the frontend language), out of context. In higher-order functional terminology, templates can be thought of as *lambda-lifted* local functions. A template can be instantiated with an environment snapshot of the current values of its free (cross-stage) variables to produce a Component, see Figure 11. This is done implicitly by the quotation operator. Different implementation strategies can coexist transparently through different subclasses of Template.

```
interface Template {
  Component newInstance (Source environment);
}
```

Figure 11. Runtime factory; upper level

```
interface Component {
  Session newSession ();
}
```

Figure 12. Runtime factory; middle level

The middle level of abstraction, and the unit of *configuration*, is the Component. Components represent referentially transparent stream functions. In higher-order functional terminology, templates can be thought of as *closure-converted* local functions. In order to make components reentrant in spite of local state, they must be instantiated for each stream-level application to produce a Session, see Figure 12. This is done implicitly at initialization time of the containing computation.

The lowest level of abstraction, and the unit of elementwise *computation*, is the Session. Sessions represent intermediate states of stream computations, and are thus not referentially transparent. They are never exposed to the user, but handled only internally. A session need to be initialized (init), and subsequently invoked (step) once per clock tick to execute a step. This is done implicitly at initialization time of the containing computation, and by the splicing operator, respectively. See Figure 13.

Sessions communicate by the Source API. Each session must be connected to a source from which it can pull the current elements of its inputs streams at each step. Conversely, each session implements the Source interface to provide public access to the current elements of its output streams. The wiring is performed implicitly at initialization time of the containing computation; the actual pulling of outputs is done by the splicing operator.

In principle, sessions can be reused sequentially by reconnection to new inputs and reinitialization, although concurrent reuse is obviously unsafe and must be avoided.

Each step of a session consists of three subtasks that update pre-state (tick), inputs (input), and post-state and outputs (action), respectively. Subclasses of Session must override all abstract methods to implement the computation of the represented component, as well as allocate exclusive storage for all local variables. The current implementation mandates that a copy of the pulled values be stored during the input phase. Thus, all variables in the scope of the component body can have the same storage and access pattern, and there is no need for distinct "operand modes" of primitive operations.

The API has been designed consciously such that no advanced features of Java are used, hence it could be mapped with little effort and no significant overhead to more low-level languages such as C. Thus, by the implementation of a C code generator, SIG components could be made usable as libraries in a very wide variety of systems.

```
abstract class Session implements Source {
  private Source inSource;
  public void setInSource (Source inSource) {
    this.inSource = inSource;
  }

  public abstract void init ();

  public void step () {
    tick ();
    input(inSource);
    action () ;
  }

  protected abstract void tick ();
  protected abstract void input (Source source);
  protected abstract void action ();
}
```

Figure 13. Runtime factory; lower level

3.3 Code Generation

3.3.1 State Transition

From the perspective of the SIG core layer, each delay operator gives rise to a pair of distinct variables x and x' for pre- and post-state, respectively. The code for a single step of a component relies on the calling environment to update its pre-state, namely with initial values on the first call, and with the previous value of the corresponding post-state on each subsequent call. How this state transition is actually effected is up to the particular implementation. There are several reasonable tactics with different usage profiles:

Transport The pair of conceptual variables can be taken literally, and an actual move operation can be used to copy values from each post-state variable to its pre-state counterpart. This is a semantically safe fallback tactic that works in all cases, but not particularly efficient. It is used by the current compiler implementation by default.

Double Buffering The behavior of the step code can be made to alternate between two variants, either by a global Boolean indirection switch, or by flipping between two clones of the code where the respective roles of pre- and post-state are mirrored. This tactic is likely more efficient than literal transport if there are many state variables. It is supported by the current compiler implementation as a configurable alternative to the default.

Overlay During code generation for a sequential machine, the SSA variables of the core representation are likely allocated to pseudo-registers anyway, such that in general values with non-overlapping life times can share a storage location. Additional constraints can be placed on the instruction schedule, such that all operations reading a pre-state variable must occur before the operation writing the corresponding post-state variable. Then pre- and post-state are non-overlapping, and may share a storage location. This tactic can save space as well as time, but does not work in all cases; see Figure 14 for a counterexample. It is used by the current compiler implementation heuristically on an all-or-nothing basis; selective use is planned for a future revision.

Indirect Buffering Multi-step delay of data must be expressed as a chain of single-step delay operations. No matter which of the preceding tactics is used, this yields a naive FIFO buffer implementation in terms of state variables, where values are actually transported from the input to the output end, see Figure 15. Except for near-trivial cases, an indirectly addressed (ring buffer) implementation is preferrable, where the current position of the input and

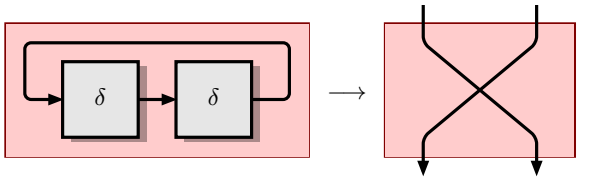


Figure 14. Delay reducing to non-overlayable state variables

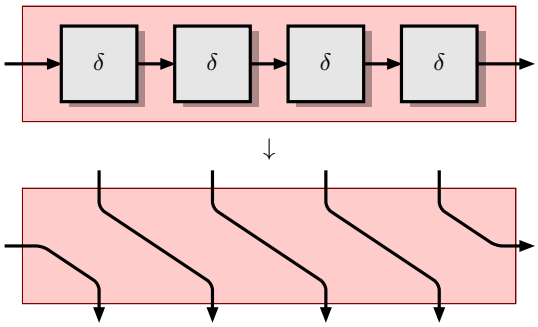


Figure 15. Delay chain reducing to FIFO

output ends move, rather than the stored data. This tactic needs to be applied selectively for suitably long delay chains in order to pay off. Support is planned for a future revision of the compiler.

3.3.2 Parallel and Sequential Evaluation

The semantics and core operations of SIG have designed carefully to allow for maximal parallelism, constrained only by explicit data flow. The encoding of control flow into data flow that embodies this principle, and is achieved by means of γ and φ operations as described above, seems unnatural from the perspective of execution on a conventional sequential machine: rather than choosing proactively between alternative branches, all branches are evaluated independently, and unneeded results are only discarded after the fact. Compare this behavior to the eager operators `&` and `|` in the C language family, as opposed to the short-circuiting operators `&&` and `||`, respectively. Several arguments need to be considered in favor of either operational approach:

In a side-effect free language, the two variants are behaviorally indistinguishable. Implementations may choose either on the grounds of convenience and efficiency. On a simple sequential execution platform, avoiding unneeded computations by conditional *branches* is virtually always a win. On modern CPUs with deep pipelines, branchless solutions that overlap alternatives and select results by conditional *moves* may be preferrable, as long as alternatives are few in number and not disproportionately expensive. Opportunistic choices need to be made, based on accurate cost models for the specific processor architecture, for good performance. By contrast, on non-sequential platforms such as field-programmable gate arrays (FPGAs), a literally parallel layout of alternatives followed by multiplexers is the canonical solution.

The current implementation of the SIG compiler takes the parallel semantics of control at face value, and translates γ and φ operators to code as they appear. Clearly, this solution scales badly on its target platform, the strongly sequential JVM. Fortunately, the sequentialization of parallel programs is turning out to be a much more tractable problem than its converse. A compiler pass that identifies conditionally needed code in the SSA form and substitutes conditional branches for γ and φ nodes is being developed.[4]

[4] Andreas Loth. Master's thesis.

```
abstract class Action {
  public abstract void run (State state);
  // ...
}
```

Figure 16. Threaded code substep

```
class State {
  public Action pc ;

  public final Object[]  registers ;
  public final double[]  registers_double ;
  public final int[]     registers_int ;
  public final boolean[] registers_bool ;

  public final boolean[] registers_control ;
}
```

Figure 17. Interpreter state

3.3.3 Interpreter

The interpreter variant of the current SIG execution environment operates almost directly on the SSA core form. Operations are scheduled statically in some valid sequential order, variables are allocated to numbered reusable "registers", and frequently occuring generic operations are specialized for their operand cound (if variadic) and/or type (if polymorphic), respectively. Otherwise, there is a one-to-one relationship between SSA statements and substeps of the actual execution.

The substeps are reified as individual Action objects, see Figure 16, organized in an object-oriented form of the traditional *threaded code* approach. The allocated virtual registers are realized as a family of equally-shaped arrays of the various supported primitive types, bundled together with a program counter (i.e. reference to the next substep) in a State object; see Figure 17. The interpreter invokes each substep in turn (run), allowing it to modify the current state. How the program counter is updated has been omitted for simplicity. In the current implementation, a linear list of substeps according to the predetermined schedule is traversed. However, the mechanism generalizes to nontrivial control flow with branching instructions, which shall be supported in a future revision.

The threaded code implementation has been designed to maximize the use of primitive data and array features of the JVM, as opposed to "clean" high-level object-oriented APIs. Consequently, actual operations coded as subclasses of Action invoke few JVM instructions with little execution overhead each, thus encouraging the JVM JIT compiler to compensate the interpretative overhead by aggressive inlining and specialization.

The interpreter, instantiated with the preprocessed code and register layout of a component, is encapsulated behind the Template interface. It can be mixed transparently with other means of implementation, as long as they use the Source API for communication.

The threaded code approach fulfills the requirement for extensible instruction sets nicely. All that is needed to add a new instruction is a new subclass of Action that mutates a State object accordingly, and a corresponding rule in the instruction selection procedure of the interpreter. The Action abstraction also allows for easy unit testing, tracing and profiling of instruction set extension candidates.

3.3.4 Compiler

The threaded code interpreter, while reasonably fast and very flexible, contains two indirections that cause runtime overhead on *every* instruction: dynamic array-based access to local variables, and virtual method invocation of Action.run.

```
public class ... extends Session {
  double in0;                    // x
  int in1;                       // t
  double out0;                   // y
  double pre0;                   // z
  double post0;                  // z'

  // Session method implementations
}
```

Figure 18. Triggered S&H; compiled class

```
abstract class Action {
  // ...
  public void compile(CompilationContext ctx);
}
```

Figure 19. Threaded code substep

We have added an "afterburner" code generation phase that compiles threaded code objects to JVM bytecode. Dedicated subclasses of Session, and their factory progenitors Template and Component, are created for each compiled SIG component. Local variables are mapped to individual member fields of the appropriate primitive type; see Figure 18 for the S&H example. Instructions are compiled to JVM bytecode fragments, which are then glued together to implement Session.action; see Figure 20 in comparison to Figure 9. The resulting code can be loaded directly into the host JVM by a ClassLoader, or stored as class files for external use.

Compilation is implemented in a distributed fashion by Action subclasses. Namely, a method compile receives a CompilationContext object that can resolve variables to JVM constant pool entries, and act as a sink for bytecode instructions. This design retains as much extensibility and traceability of the instruction set as possible, even if fragmented bytecode generation is somewhat harder to test and debug than threaded code. The downside is that, because instruction selection is performed in isolation, the resulting bytecode contains a number of redundancies, easily seen in Figure 20.

Theoretically, an extra optimization pass on the JVM bytecode format could be used for cleanup. But we have found that JVM JIT compilers do that job well already. For the S&H example, the machine code produced by Oracle's Hotspot JVM 1.8.0_20, on a test machine specified in the following subsection, is depicted in Figure 21.

The redundancy that remains in the depicted code, namely that patterns are matched twice, stems from the incongruency of control flow which is parallel in SIG and sequential on the JVM. A transformation-based systematic solution notwithstanding, we have found that existing compilers are quite capable of eliminating the redundancy in simple cases. In particular, the machine code produced by GCJ 4.8.2 with the `-O3` option, invoked with the same bytecode on the same target machine, is depicted in Figure 22. Comparison of Figures 21 and 22 illustrates the typical tradeoff between just-in-time and ahead-of-time compilation: more aggressive use of processor-specific capabilities (here, SSE2 extensions) for the former, and more thorough (here, optimal) application of expensive optimizations (here, sparse conditional constant propagation) for the latter.

3.4 Experimental Evaluation

We have tested the performance of both interpreted and compiled code with a simple but nontrivial sound synthesis application. It

```
protected void action();
  Code:
     0: aload_0
     1: getfield     #37                 // t
     4: iconst_1
     5: isub                             // S?
     6: ifne         14
     9: iconst_1
    10: istore_1
    11: goto         16
    14: iconst_0
    15: istore_1

    16: iload_1
    17: ifne         28
    20: dconst_0
    21: dstore_2
    22: iconst_0
    23: istore       4
    25: goto         36
    28: aload_0
    29: getfield     #31                 // x
    32: dstore_2
    33: iconst_1
    34: istore       4

    36: aload_0
    37: getfield     #37                 // t
    40: iconst_0
    41: isub                             // H?
    42: ifne         50
    45: iconst_1
    46: istore_1
    47: goto         52
    50: iconst_0
    51: istore_1

    52: iload_1
    53: ifne         65
    56: dconst_0
    57: dstore       5
    59: iconst_0
    60: istore       7
    62: goto         74
    65: aload_0
    66: getfield     #47                 // z
    69: dstore       5
    71: iconst_1
    72: istore       7

    74: aload_0
    75: iload        4
    77: ifeq         84
    80: dload_2
    81: goto         102
    84: iload        7
    86: ifeq         94
    89: dload        5
    91: goto         102
    94: // abort (t out of range)
   102: putfield     #39                 // y

   105: aload_0
   106: aload_0
   107: getfield     #39                 // y
   110: putfield     #44                 // z'
   113: return
```

Figure 20. Triggered S&H; bytecode

```
action:
        mov     0x38(%rsi), %r11d # t
        mov     %r11d, %r10d
        dec     %r10d
        xorpd   %xmm0, %xmm0, %xmm0
        test    %r10d, %r10d      # S?
        je      .Le
        xorpd   %xmm1, %xmm1, %xmm1
.La:
        test    %r11d, %r11d      # H?
        jne     .Lb
        movsd   0x28(%rsi), %xmm0 # z
.Lb:
        test    %r10d, %r10d      # S?
        je      .Ld
        test    %r11d, %r11d      # H?
        jne     .Lf
.Lc:
        movsd   %xmm0, 0x20(%rsi) # y
        movsd   %xmm0, 0x30(%rsi) # z'
        ret
.Ld:
        movapd  %xmm1, %xmm0
        jmp     .Lc
.Le:
        movsd   0x18(%rsi), %xmm1 # x
        jmp     .La
.Lf:
        # abort (t out of range)
```

Figure 21. Triggered S&H; machine code (JRE)

```
action:
        movl    48(%rdi), %eax    # t
        testl   %eax, %eax        # H?
        je      .L17
        cmpl    $1, %eax          # S?
        jne     .L26
        movsd   40(%rdi), %xmm0   # x
        movsd   %xmm0, 56(%rdi)   # y
        movsd   %xmm0, 72(%rdi)   # z'
        ret
.L17:
        movsd   64(%rdi), %xmm0   # z
        movsd   %xmm0, 56(%rdi)   # y
        movsd   %xmm0, 72(%rdi)   # z'
        ret
.L26:
        # abort (t out of range)
```

Figure 22. Triggered S&H; machine code (GCJ)

implements a digital organ with a range of four chromatic octaves. Each of the 49 notes consists of two SIG components, namely a sine wave generator and an ADSR envelope generator, running at the audio rate of 44.1 kHz and the 64 times slower control rate, respectively. The precise algorithms are specified in [8]. They translate to 4 and 54 core operations, respectively.

A hand-coded environment runs all 49 notes in quasi-parallel for full polyphony, and mixes them together according to input from a MIDI keyboard, for interactive real-time CD quality output. The resulting audio stream is fed to the push-based Java audio system. Hence the audio and control rate clocks operate in pseudo-real time: the control loop runs at full speed when there is sufficient space in the audio output buffer, and blocks when the buffer is full. By limiting the buffer size, latency is bounded to 10–100 ms.

The actual time spent in computation (i.e. component execution and mixing) is recorded with the precision and accuracy of Java `System.nanoTime()`. Optimizations that turn off silent voices have been deactivated for the sake of regular load and stable measurements. On our test system, with a Core i5-3317U CPU at 1.7 GHz, Ubuntu 14.04 OS, and Oracle JDE 1.8.0_20, we have observed an effective rate (number of samples produced divided by time spent computing) of 229±3 kHz for interpreted code, and 2740±60 kHz for compiled code, respectively.[5] These figures translate to an average effort of about 152 and 13 CPU cycles per voice-sample, or to a load of 19.6 % and 1.6 %, respectively. The speedup by compilation is a factor of 12. All experiments use only a single CPU core for SIG computations, although JVM system threads may run concurrently on other cores.

In summary, the interpreted version, on stock hardware and without JVM tweaking, performs fast enough for a real-time demonstration by a comfortable margin. The compiled version has enough computational reserves that it can be expected to scale up to audio synthesis tools of artistically acceptable quality.

4. Conclusion

The SIG language is highly domain-specific, and hence poses specific problems for effective and efficient execution. On the one hand, the purely and totally functional approach, and the rigid control flow enable or simplify a great number of analyses and optimizations. On the other hand, the prototype nature of the current implementation and applications, and the fact that type system and instruction sets are far from fixed, calls for a compiler design that is more a laboratory environment than a closed tool.

As a notable practical lesson from the construction of the SIG compiler, we have corroborated the hypothesis that bytecode platforms are suitable backends for rapid prototyping. Many errors in the code generator have been detected statically by standard JVM bytecode verification tools. In other cases that fail at runtime, debugging is fairly convenient, even without a working generator for symbol tables or source location metadata.

The Java platform has extensive support for real-world interaction, such as GUIs, sampled audio output and MIDI audio input, in terms of on-board libraries that work out-of-the box and with decent efficiency/safety tradeoffs. Using these, tangible demonstrations of SIG programs in a live loop, as in [8], can be constructed with very moderate effort.

The JVM JIT compiler allows to explore the interpreter–compiler continuum in search for a sweet spot for the prototype implementation of a novel language rather freely, by keeping the performance penalties for higher levels of backend abstraction within reasonable limits.

4.1 Related Work

- FRP [2, 4, 9]
- Hume [1]
- Faust [5]
- Trace-based compilation techniques

4.2 Future Work

- Branch-based implementation of φ nodes
- C backend
- FPGA backend
- Worst-case execution time analysis

[5] Reported errors are mean absolute deviations.

References

[1] Kevin Hammond and Greg Michaelson. The design of Hume: A high-level language for the real-time embedded systems domain. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2003.

[2] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming (AFP 2002)*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[3] Markus Lepper and Baltasar Trancn y Widemann. Optimization of visitor performance by reflection-based analysis. In J. Cabot and E. Visser, editors, *Proceedings 4th International Conference on Theory and Practice of Model Transformations (ICMT 2011)*, volume 6707 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 2011.

[4] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, pages 51–64. ACM, 2002.

[5] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632, 2004.

[6] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[7] Baltasar Trancón y Widemann and Markus Lepper. Foundations of total functional data-flow programming. In *Mathematically Structured Functional Programming (MSFP 2014)*, volume 154 of *Electronic Proceedings in Theoretical Computer Science*, pages 143–167, 2014.

[8] Baltasar Trancón y Widemann and Markus Lepper. Sound and soundness – practical total functional data-flow programming. In *2nd International Workshop on Functional Art, Music, Modeling and Design (FARM 2014)*. ACM Digital Library, 2014.

[9] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, 2000.

Declaration-level Change and Dependency Analysis of Hackage Packages

Extended Abstract

Philipp Schuster and Ralf Lämmel

Software Languages Team, Department of Computer Science, University of Koblenz-Landau, Germany

Abstract

Version numbers for Haskell packages on Hackage communicate when an update is possibly breaking and prevent installation of such updates. However, version numbers say nothing about which declarations actually changed. Similarly, version bounds on dependencies do not take into account which declarations a package actually uses. This leads to cases where installation of a package is unnecessarily prohibited. We describe a methodology and an supporting infrastructure, HackPackUp, for determining if and how an update affects a package. In a sample of 1,578 packages, we find 5,404 scenarios where an update is prohibited even though a package uses none of the changed declarations.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features; D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement; F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages

Keywords Haskell. Hackage. Package Update. Version Bound. Program Analysis. Mining Software Repositories. HackPackUp.

1. Motivation

Imagine a hypothetical package *favorites* of version 0.2.0. Haskell or Hackage's *Package Versioning Policy* (PVP)[1] defines the major part of a version number to be the first two digits (in our example: 0.2) and the minor part to be the rest (in our example: 0). Version numbers are ordered lexicographically. It also specifies that all dependencies on *favorites* should be constrained with a lower and an upper bound. The lower bound excludes versions the package was not tested with. The upper bound includes all future minor versions but excludes all future major versions. So if for example some package was tested with *favorites* version 0.1.1 and 0.2.0 the version bounds should be $favorites >= 0.1.1 \&\& < 0.3$).

Further imagine, there is a function declaration *color* in a module in *favorites*. If a new version of *favorites* with an improved but backwards compatible implementation for *color* is released, its version number would only be increased in the minor part (in our example: 0.2.1). In this way, future installations of packages depending on *favorites* could use the improved version. If however, in a new version of *favorites*, the declaration for *color* would be removed, then the change would be backwards incompatible and the version number would have to be increased in the major part (in our example: 0.3.0). This means no existing package depending on *favorites* can be installed with the new version. They all have to be checked manually for compatibility and updated accordingly. If a package did not even use *color*, then the update would not affect the package, thereby prohibiting installation unnecessarily. We want to know if such cases exist in practice and how significant of a problem this is. To this end, we need information about the changes and the dependencies at the level of individual declarations for packages on Hackage.

The research reported in this extended abstract is an instance of 'mining software repositories'; see [4] for a survey. Such mining has also been researched in a Haskell/Hackage context with an objective different from ours; see the analysis of generic programming on Hackage [1]. The analysis of declaration-level changes and dependencies, as it is central to our research, also relates to change impact analysis, which is an established subject specifically for imperative languages; see [5] for a survey. For instance, a fine grained impact analysis which includes mining for an evolving software repository is reported in [3]. On the Haskell front, there exist tools to check what symbols a Hackage package update changes, e.g., *hackage-diff*,[2] but the analysis of changes at a declaration-level and their impact in terms of dependencies has not been the subject of research.

2. Research question

Our initial research question is this: *Do unnecessarily prohibited update scenarios exist on Hackage?* An update scenario is characterized by a package, a dependency of that package that satisfies the dependency constraints and any later version of that dependency. A prohibited update scenario is one where the later version of the dependency does not satisfy the constraints. An unnecessarily prohibited update scenario is one that could be permitted based on the criterion that the package is not 'affected' by revised or deleted declarations in the newer version of the dependency. Just like the PVP, we do not consider additions to be breaking because while added

[1] PVP: `http://www.haskell.org/haskellwiki/Package_versioning_policy` Explanation of the idea behind the PVP: `http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue2/EternalCompatibilityInTheory`

[2] `http://hackage.haskell.org/package/hackage-diff`

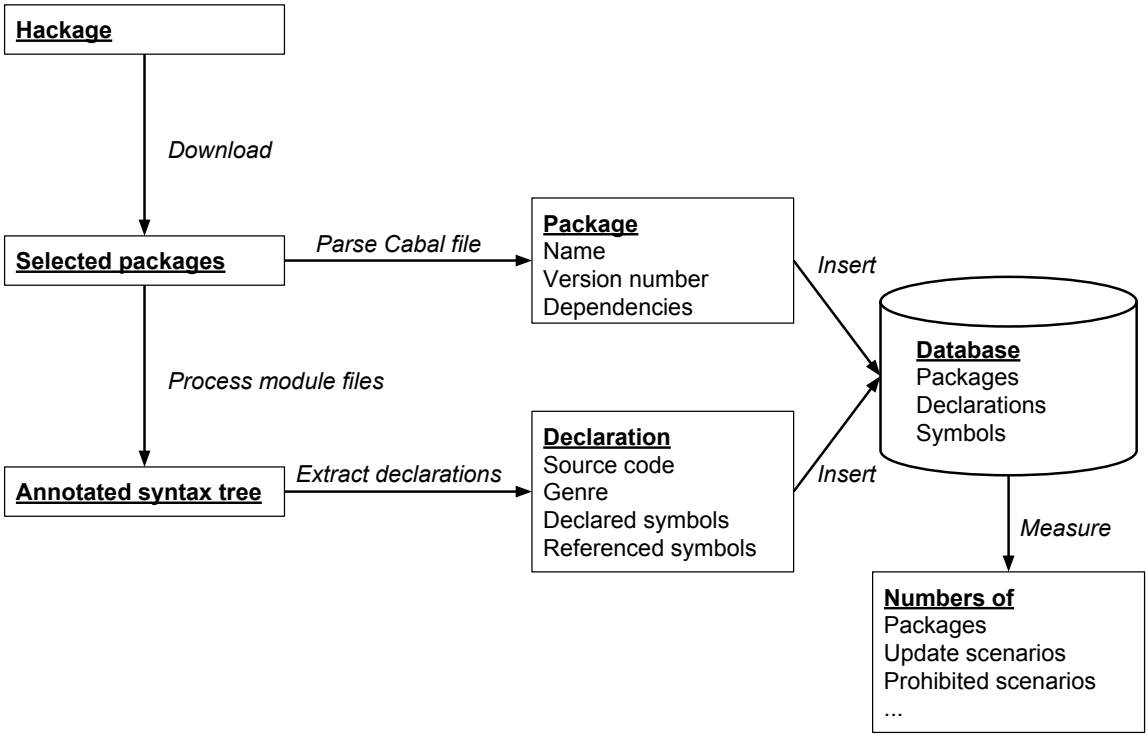


Figure 1. Overview of the fact extraction and measurement process.

declarations could cause name clashes these can be prevented with explicit import lists.

In fact, we would like to find out whether the problem of unnecessarily prohibited update scenarios is a significant one. Thus, we complement our research question as follows: *How many unnecessarily prohibited update scenarios exist on Hackage?* Accordingly, we perform (roughly) the following measurements by analyzing (a sample of) Hackage packages:

- How many update scenarios are there?
- How many of those are prohibited?
- How many of those prohibited are 'unnecessary'?

3. Fact extraction model

Fact extraction is applied to a set of packages (i.e., all of Hackage or a sample thereof). A *package* is uniquely identified by its *name* and its *version number*. A package p *depends* on package p' if the name of p' is listed as a dependency in the package description of p and the version number of p' satisfies the constraints in the package description. An *update* is a pair of packages with the same name and where the version number of the first package is smaller than the version number of the second package. The version numbers do not have to be consecutive, an *update* may skip several versions. A *major update* is where the major parts of the version numbers are different; otherwise we speak of a *minor update*.

A package declares a set of declarations, possibly subdivided into several modules. Every declaration has these properties: a *genre* (i.e., *function*, *type*, *class*, *instance*), sets of *declared* and *referenced symbols*, and the *source code* or *AST* underlying the declaration. Symbols are to be qualified by module names and genres (because of separate namespaces).

Using these basic facts, we can compute what symbols of a dependency a package requires and in what way an update changes a symbol. This allows us to decide if an update affects a package. We are inspired by other work on classifying software changes [2]. By collecting all such information and general package descriptions for Hackage packages (all of Hackage or a sample thereof), we can compute the measurements of §2.

4. Methodology

We address the research question through the following methodology. Figure 1 gives an overview.

- Download Hackage packages. We may need to trade scalability for completeness; see the actual selection of packages in the case study (§5).
- For each package's Cabal file, save these properties in the database:
  - The package's name and version number.
  - Which of all the packages under investigation it is allowed to depend on according to the dependency constraints. Usually, multiple different minor and major versions are allowed.

Packages	1,578
Declarations	332,663
Symbols	36,603
Update scenarios	212,176
Minor	113,030
Allowed	113,000
Prohibited	30
Major	99,146
Allowed	77,418
Affected	40,479
Unaffected	36,939
Prohibited	21,728
Affected	16,354
Unaffected	5,374

Figure 2. Measurement results of the case study.

  - An immediate next version, if it exists, together with the status whether it differs in the major version part. From the immediate next versions we can generate all updates. We are primarily interested in major updates, but the minor updates are interesting for validating the methodology.
- Attempt to install every package.
- Run the HackPackUp processor instead of the regular compiler:
  - Preprocess and parse to get the abstract syntax tree (AST).
  - Resolve names to annotate every symbol occurrence in the AST with its origin.
  - Save properties of each declaration in a database:
    - Source code
    - Genre
    - Declared symbols
    - Referenced symbols
- Run measurements (§2) against that database.

5. Case study

While the initial goal was to analyze all of Hackage, for scalability reasons, we had to choose some segment of Hackage. Without such selection, we would have to exercise too many dependencies and validate too many results. We looked for a list of packages that share many dependencies to minimize the number of packages we have to process. Stackage is a Haskell package repository with fewer packages than Hackage has. It is also complete in the sense that all dependencies of all its packages are included in it. We got the list of packages from Stackage and took all versions of all those packages from Hackage.

Table 2 lists number of entities resulting from fact extraction and storage in the database (§3 and §4).

We computed 212,176 update scenarios from our data. An update scenario consists of three parts: A package, a dependency of that package that satisfies the constraints and any later version of that dependency. In 113,030 of these update scenarios the update does not involve a major version change which means they should not be prohibited and indeed only 30 are. The other 99,146 update scenarios involve a major update. In 21,728 of these the version of the later dependency does not satisfy the constraints anymore. This means they are prohibited by an upper version bound.

We find that in 5,404 of the prohibited update scenarios the update does not affect the package and are therefore unnecessarily prohibited. We conclude this because none of the symbols the package requires from the first version of its dependency are absent or different in the second. If on the other hand we look at the 77,418 major update scenarios that are allowed we find that in 40,479 of those the update does indeed remove or alter at least one of the symbols required by the package. This could mean that the upper version bounds are wrong or missing. This could also mean that there are backwards compatible changes to parts of a package and backwards incompatible changes to another part requiring the update to be classified as *major*. We plan to investigate this problem in future work.

Besides the fact that we have only taken a sample of Hackage there are other threats to validity. Not all packages of the sample can be processed with HackPackUp. We use *haskell-src-exts*[3] for parsing and *haskell-names*[4] for name resolution. Most packages are only tested to work with GHC and make implicit assumptions about preprocessor flags, language extensions and builtin modules and therefore are not immediately parseable or some symbols fail to be resolved. We do not consider the *base* package as a dependency because it is a dependency of almost every package and for simplicity we only have one version and no information about the declarations of *base* available. We currently ignore type class instances because their resolution is out of scope of our tool.

At the time of writing, we are working on validating our classification of the update scenarios as unnecessarily prohibited. To this end, we need to install the packages with the varying dependencies. This proves difficult because of the number of installs and the time they take to build, especially in the view of a clean state needed for each test install.

In summary we can say that we indeed have found a significant number of cases where an update scenario is prohibited while the update would not affect the package. We have found, to our surprise, that in many update scenarios a major update does affect the package while it is still allowed. This observation calls for future work.

References

[1] N. Bezirgiannis, J. Jeuring, and S. Leather. Usage of generic programming on hackage: Experience report. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 47–52. ACM, 2013.

[2] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, Sept. 2005.

[3] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 105–111. ACM, 2006.

[4] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, Mar. 2007.

[5] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Softw. Test., Verif. Reliab.*, 23(8):613–646, 2013.

[3] `http://hackage.haskell.org/package/haskell-src-exts`

[4] `http://hackage.haskell.org/package/haskell-names`

An Efficient Type- and Control-Flow Analysis for System F

Connor Adsit
Rochester Institute of Technology
cda8519@rit.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract

At IFL'12, we presented a novel monovariant flow analysis for System F (with recursion) that yields both *type-flow* and *control-flow* information. [4] The type-flow information approximates the type expressions that may instantiate type variables and the control-flow information approximates the λ- and Λ-expressions that may be bound to variables. Furthermore, the two flows are mutually beneficial: control flow determines which Λ-expressions may be applied to which type expressions (and, hence, which type expressions may instantiate which type variables), while type flow filters the λ- and Λ-expressions that may be bound to variables (by rejecting expressions with static types that are incompatible with the static type of the variable under the type flow).

Using a specification-based formulation of the type- and control-flow analysis, we proved the analysis to be sound, decidable, and computable. Unfortunately, naïvely implementing the analysis using a standard least fixed-point iteration yields an $O(n^{13})$ algorithm.

In this work, we give an alternative flow-graph-based formulation of the type- and control-flow analysis. We prove that the flow-graph-based formulation induces solutions satisfying the specification-based formulation and, hence, that the flow-graph-based formulation of the analysis is sound. We give a direct algorithm implementing the flow-graph-based formulation of the analysis and demonstrate that it is $O(n^4)$. By distinguishing the size l of expressions in the program from the size m of types in the program and performing an amortized complexity analysis, we further demonstrate that the algorithm is $O(l^3 + m^4)$.

1. Introduction

2. Language and Semantics

2.1 Syntax

In IFL'12, we introduced an ANF Intermediary Representation for System F. We give a specification for a modified ANF System F language in Figure 1.

We maintain the same types as before, but we restrict the form of syntactic types in order to remove the recursion found in the previous definition of types. Instead of having type constructors be composed of smaller types (ie. for function and forall types), they must be constructed with type variables. We also introduce DeBruijn indices into the type system to describe any type parameterized by a Λ-abstraction.

In addition to adding a distinction between simple binds (values) and complex binds (applications), we expand upon the definition of expressions to accommodate the changes made to the type system. The syntax is extended to include `let`-bindings for type variables. Additionally, we mandate that all binding occurances of expression variables be annotated with a type variable instead of a type. Similarly, the recursive function variables and λ-parameter variables appearing in simple binds must also have type variable annotations. We changed the specification of type applications so that only type variables may be passed as arguments.

Our motivation for using type variables is to promote type reuse, limiting the pool of possible types to as little a number as possible. As we will see later, having as few types as possible will reduce the overall complexity of the algorithm.

2.2 Scanning Input Program

Figure 2 introduces relations that finds all nested expressions, expression binds, type binds, type variables, and expression variables in a program.

We begin our recursive descent into a program by saying the entire program can be found inside itself. An expression can be found in the program if it is the body of a `let`-binding or if it belongs to a λ- or Λ-abstraction. An expression bind belongs to a program if the program contains a `let` x with that particular expression bind on the right-hand side. Similarly, a type bind belongs to a program if it participates in a `let` α expression that also belongs to the program. All type variables found in any nested `let`α or as parameters to Λ-abstractions belong to the program. The relation behaves similarly with expression variables and `let` x bindings and λ-abstractions. We must also consider that the recursive function variables, f, belong to a program if the encapsulating λ- or Λ-abstraction also belongs to the program.

2.3 Semantics

We assume the usual operational semantics for the ANF System using a CESK machine. More details can be found in [4].

The static semantics will need to be extended to include support for DeBruijn indices. In particular, when a value is used, we need to ensure that all indices present in the type are encapsulated by the proper amount of foralls. We also must mandate that type variables are bound before use, which has been implemented successfully in [12]. Whenever an expression variable is bound to a value, the annotated type variable must map to a type in the current context that is compatible with the type of the value.

It is worth noting that all forms of the analysis and algorithm will hold for a program even if it is untyped. It is only when the program is well typed that the flow-graph based analysis and the algorithm will yield a sound result.

$$
\begin{array}{llcll}
\textit{Type variables} & \mathit{TyVar} & \ni & \alpha, \beta, \ldots & \\
\textit{Type indices} & \mathit{TyIdx} & \ni & n & ::= \quad \mathtt{0} \mid \mathtt{1} \mid \cdots \\
\textit{Type binds} & \mathit{TyBnd} & \ni & \tau & ::= \quad \alpha_a \to \alpha_b \mid \forall.\ \alpha_b \mid \#n \\
\\
\textit{Expression variables} & \mathit{ExpVar} & \ni & x, y, z, f, g, \ldots & \\
\textit{Expression binds (simple)} & \mathit{ExpBnd_s} & \ni & b_\mathsf{s} & ::= \quad \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \mid \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \\
\textit{Expression binds (complex)} & \mathit{ExpBnd_c} & \ni & b_\mathsf{c} & ::= \quad x_f\ x_a \mid x_f\ [\alpha_a] \\
\textit{Expression binds} & \mathit{ExpBnd} & \ni & b & ::= \quad b_\mathsf{s} \mid b_\mathsf{c} \\
\\
\textit{Expressions} & \mathit{Exp} & \ni & e & ::= \quad \mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \mid \mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \mid x \\
\textit{Programs} & \mathit{Prog} & \ni & P & ::= \quad e
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{ResOf}(\cdot) & :: & \mathit{Exp} \to \mathit{ExpVar} \\
\mathsf{ResOf}(\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e) & = & \mathsf{ResOf}(e) \\
\mathsf{ResOf}(\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e) & = & \mathsf{ResOf}(e) \\
\mathsf{ResOf}(x) & = & x
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{TyOf}(\cdot) & :: & \mathit{ExpBnd_s} \to \mathit{TyVar} \\
\mathsf{TyOf}(\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b) & = & \alpha_f \\
\mathsf{TyOf}(\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b) & = & \alpha_f
\end{array}
$$

Figure 1. Syntax of ANF System F

$\boxed{e \preceq_{\mathit{Exp}} P}$

$$\frac{}{P \preceq_{\mathit{Exp}} P} \qquad \frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{e \preceq_{\mathit{Exp}} P} \qquad \frac{\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{e \preceq_{\mathit{Exp}} P}$$

$$\frac{\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{e_b \preceq_{\mathit{Exp}} P} \qquad \frac{\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{e_b \preceq_{\mathit{Exp}} P}$$

$\boxed{b \preceq_{\mathit{ExpBnd}} P}$

$$\frac{\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{b \preceq_{\mathit{ExpBnd}} P}$$

$\boxed{\tau \preceq_{\mathit{TyBnd}} P}$

$$\frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{\tau \preceq_{\mathit{TyBnd}} P}$$

$\boxed{\alpha \preceq_{\mathit{TyVar}} P}$

$$\frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{\alpha \preceq_{\mathit{TyVar}} P} \qquad \frac{\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{\beta \preceq_{\mathit{TyVar}} P}$$

$\boxed{x \preceq_{\mathit{ExpVar}} P}$

$$\frac{\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{x \preceq_{\mathit{ExpVar}} P} \qquad \frac{\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{f \preceq_{\mathit{ExpVar}} P} \qquad \frac{\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{z \preceq_{\mathit{ExpVar}} P} \qquad \frac{\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{f \preceq_{\mathit{ExpVar}} P}$$

Figure 2. Sub-term Relations

3. Specification-Based Formulation of TCFA

The specification formulation is safe under an abstract type environment, which keeps track of type variables and the possibly many types that may be bound to the variables, and also an abstract value environment, which behaves the same way with expression variables and binders. In order to check for type compatibility, we make use of an abstract type environment to recursively replace all type variables with any type found in the environment's entry for the variable. We say that a type is closed when there are no longer any type variables in the expanded type. Two type variables, α_1 and α_2, are compatible under an abstract type environment if it is possible to derive the same closed type from both α_1 and α_2.

An abstract type environment, $\hat{\phi}$, and an abstract value environment, $\hat{\rho}$, safely approximate an expression by an inductive analysis of the expression.

If the expression is a `let`-α expression, we must make sure that the τ being bound in the syntax appears in the possible bindings for α as described by $\hat{\phi}$ and that the remaining expressions are also safe under the same $\hat{\phi}$ and $\hat{\rho}$.

If the expression is a `let`-x expression and the binder is a simple bind, we read the type off of x and the binder. If the two are compatible we need to make sure that the binder appears in the possible bindings of x in $\hat{\rho}$. Additionally, we need to push the analysis through the body of the binder and the remaining expressions.

Otherwise, if a complex bind appears on the right side of a `let`-x expression, we need to iterate through the possible λ-abstractions recorded in $\hat{\rho}$ if the binder is a value application. For every possible argument in the entry for x_a, if the type of the argument is compatible with the type of the formal parameter under $\hat{\phi}$, we need to assert that the argument also appears in the entry for the parameter. Likewise, for all values returned by the abstraction, if the type of the value is compatible with the type of the bound variable, it must appear in the entry for the bound variable in $\hat{\phi}$.

Similarly, if a type application is being performed, we iterate through all possible Λ-abstractions that could be bound to the function variable. We assert that all types potentially bound to the type argument must appear in the description of the type parameter in $\hat{\phi}$. As with value application, any type-compatible results returned from the abstraction must be present in $\hat{\phi}$'s entry for the bound variable.

Finally, in the case that the expression is a simple variable, we assume that the analysis is sound.

3.1 Soundness, Decidability, and Computability

Our previous work [4, 5] showed that the specification-based formulation of the type- and control-flow analysis is sound with respect to the operational semantics, that the acceptability of given (finite) abstract type and value environments with respect to a program is decidable, and that the minimum acceptable abstract type and value environments for a program are computable in polynomial time. We briefly recall the essence of these arguments.

Soundness of the specification-based formulation of the type- and control-flow analysis asserts that any acceptable pair of abstract environments for a well-typed program approximates the run-time behavior of the program. In particular, the abstract type and value environments approximate every concrete type and value environment that arises during execution of the program. Flow soundness relies crucially on the well-typedness of the program. Soundness of the type system guarantees that, at run time, an expression variable will only be bound to a well-typed closed value of a closed type and that the expression variable's type annotation must be interpreted as that closed type. Hence, if there is no closed type at which both the static type of the expression variable and the static type of the value might be instantiated, then that variable will never be bound to that value at run time. The critical component of the proof is that the type compatibility judgment $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \alpha_1 \approx\!\!\!\approx \alpha_2$ is derivable whenever there is a common closed type at which both α_1 and α_2 are instantiated.

Although there are an infinite number of pairs of abstract type and value environments that are acceptable for a given program, we are primarily interested more precise pairs over less precise pairs. For a given program, we can limit our attention to the "finite" abstract type and value environments that map the type variables that occur in the program to sets of type binds that appear in the program (and map all type variables that do not occur in the program to the empty set) and map the expression variables that occur in the program to sets of simple expression binds that appear in the program (and map all expression variables that do not occur in the program to the empty set).

The decidability of the acceptability judgment $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e$ relies upon the decidability of the type compatibility judgment $\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx\!\!\!\approx \alpha_2$. Due to "recursion" in the abstract type environment, whereby a type variable may be mapped (directly or indirectly) to a set of type binds in which the type variable itself occurs free, we cannot simply enumerate the (potentially infinite sets of) closed types θ_1 and θ_2 such that $\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \twoheadrightarrow \theta_1$ and $\hat{\phi} \vDash_{\mathsf{S}} \alpha_2 \twoheadrightarrow \theta_2$ in order to decide whether or not the judgment $\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx\!\!\!\approx \alpha_2$ is derivable. To address this issue, we take inspiration from the theory and implementation of regular-tree grammars [1, 3, 6]. By interpreting an abstract type environment as (the productions for) a regular-tree grammar, a derivation of the judgment $\hat{\phi} \vDash_{\mathsf{S}} \alpha \twoheadrightarrow \theta$ is exactly a parse tree witnessing the derivation of the ground tree θ from the starting non-terminal α in the regular-tree grammar $\hat{\phi}$ and the judgment $\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx\!\!\!\approx \alpha_2$ is derivable iff languages generated by taking α_1 and α_2, respectively, as the starting non-terminal in the regular-tree grammar $\hat{\phi}$ have a non-empty intersection. Since regular-tree grammars are closed under intersection and the emptiness of a regular-tree grammar is decidable [6, 9], the type compatibility judgment $\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx\!\!\!\approx \alpha_2$ is decidable.

Finally, the minimum acceptable pair of abstract type and value environments for a given program is computable via a standard least-fixed point iteration. We interpret the acceptability judgment $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e$ as defining a monotone function from pairs of abstract environments to pairs of abstract environments; the "output" abstract environments are formed from the "input" abstract environments joined with all discovered violations.

We conclude with a crude upper-bound on computing the minimum acceptable pair of abstract type and value environments for a given program, of size n, via a standard least-fixed point computation. We represent $\hat{\phi}$ and $\hat{\rho}$ as two-dimensional arrays (indexed by α/τ and x/b_{s}, respectively), requiring $O(n^2)$ space.[1] Thus, the two abstract environments are lattices of height $O(n^2)$. Each (naïve) iteration of the monotone function is syntax directed ($O(n)$) and dominated by the function-application bind, which loops over all of the elements of $\hat{\rho}(x_f)$ and $\hat{\rho}(\mathsf{ResOf}(e_b))$ ($O(n)$), loops over all of the elements of $\hat{\rho}(x_a)$ ($O(n)$), and computes type compatibility via a regular-tree grammar intersection ($O((n^2)^2)$, because the output regular-tree grammar is, worst-case, quadratic space with respect to the size of the input regular-tree grammar) and emptiness test ($O(((n^2)^2)^2)$, because the emptiness query is quadratic time with respect to the input regular-tree grammar). Hence, our analysis is computable in polynomial time: $O(n^{13}) = (O(n^2) + O(n^2)) \times (O(n) \times O(n) \times O(n) \times (O(n^4) + O(n^8)))$.

[1] See Sections 5.2.1 and 5.2.2 for more discussion of assumptions about the representation of the input program and data structures and operations.

$$
\begin{array}{lrcll}
\textit{Types (closed)} & \mathit{TyClsd} & \ni & \theta & ::= \quad \theta_a \to \theta_b \mid \forall.\ \theta_b \mid \#n \\
\textit{Abstract type environments} & \mathit{ATyEnv} = \mathit{TyVar} \to \mathcal{P}(\mathit{TyBnd}) & \ni & \hat{\phi} & ::= \quad \{\alpha \mapsto \{\tau, \ldots\}, \ldots\} \\
\textit{Abstract value environments} & \mathit{AValEnv} = \mathit{ExpVar} \to \mathcal{P}(\mathit{ExpBnd_s}) & \ni & \hat{\rho} & ::= \quad \{x \mapsto \{b_\mathsf{s}, \ldots\}, \ldots\}
\end{array}
$$

$\boxed{\hat{\phi} \vDash_\mathsf{S} \tau \twoheadrightarrow \theta}$

$$\frac{\hat{\phi} \vDash_\mathsf{S} \alpha_a \twoheadrightarrow \theta_a \qquad \hat{\phi} \vDash_\mathsf{S} \alpha_b \twoheadrightarrow \theta_b}{\hat{\phi} \vDash_\mathsf{S} \alpha_a \to \alpha_b \twoheadrightarrow \theta_a \to \theta_b} \qquad \frac{\hat{\phi} \vDash_\mathsf{S} \alpha_b \twoheadrightarrow \theta_b}{\hat{\phi} \vDash_\mathsf{S} \forall.\ \alpha_b \twoheadrightarrow \forall.\ \theta_b} \qquad \frac{}{\hat{\phi} \vDash_\mathsf{S} \#n \twoheadrightarrow \#n}$$

$\boxed{\hat{\phi} \vDash_\mathsf{S} \alpha \twoheadrightarrow \theta}$

$$\frac{\tau \in \hat{\phi}(\alpha) \qquad \hat{\phi} \vDash_\mathsf{S} \tau \twoheadrightarrow \theta}{\hat{\phi} \vDash_\mathsf{S} \alpha \twoheadrightarrow \theta}$$

$\boxed{\hat{\phi} \vDash_\mathsf{S} \alpha_1 \approx\!\!\!\approx \alpha_2}$

$$\frac{\hat{\phi} \vDash_\mathsf{S} \alpha_1 \twoheadrightarrow \theta \qquad \hat{\phi} \vDash_\mathsf{S} \alpha_2 \twoheadrightarrow \theta}{\hat{\phi} \vDash_\mathsf{S} \alpha_1 \approx\!\!\!\approx \alpha_2}$$

$\boxed{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e}$

$$\frac{\tau \in \hat{\phi}(\alpha) \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e}{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} \mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e}$$

$$\frac{\hat{\phi} \vDash_\mathsf{S} \alpha_f \approx\!\!\!\approx \alpha_x \twoheadrightarrow \mu f\!:\!\alpha_f.\lambda z\!:\!\alpha_z.e_b \in \hat{\rho}(x) \qquad \hat{\phi} \vDash_\mathsf{S} \alpha_f \approx\!\!\!\approx \alpha_f \twoheadrightarrow \mu f\!:\!\alpha_f.\lambda z\!:\!\alpha_z.e_b \in \hat{\rho}(f) \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e_b \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e}{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} \mathtt{let}\ x\!:\!\alpha_x = \mu f\!:\!\alpha_f.\lambda z\!:\!\alpha_z.e_b\ \mathtt{in}\ e}$$

$$\frac{\hat{\phi} \vDash_\mathsf{S} \alpha_f \approx\!\!\!\approx \alpha_x \twoheadrightarrow \mu f\!:\!\alpha_f.\Lambda\beta.e_b \in \hat{\rho}(x) \qquad \hat{\phi} \vDash_\mathsf{S} \alpha_f \approx\!\!\!\approx \alpha_f \twoheadrightarrow \mu f\!:\!\alpha_f.\Lambda\beta.e_b \in \hat{\rho}(f) \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e_b \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e}{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} \mathtt{let}\ x\!:\!\alpha_x = \mu f\!:\!\alpha_f.\Lambda\beta.e_b\ \mathtt{in}\ e}$$

$$\frac{\forall \mu f\!:\!\alpha_f.\lambda z\!:\!\alpha_z.e_b \in \hat{\rho}(x_f)\ .\ \begin{pmatrix}\forall b_\mathsf{s} \in \hat{\rho}(x_a)\ .\ \hat{\phi} \vDash_\mathsf{S} \mathsf{TyOf}(b_\mathsf{s}) \approx\!\!\!\approx \alpha_z \twoheadrightarrow b_\mathsf{s} \in \hat{\rho}(z) \wedge \\ \forall b_\mathsf{s} \in \hat{\rho}(\mathsf{ResOf}(e_b))\ .\ \hat{\phi} \vDash_\mathsf{S} \mathsf{TyOf}(b_\mathsf{s}) \approx\!\!\!\approx \alpha_x \twoheadrightarrow b_\mathsf{s} \in \hat{\rho}(x)\end{pmatrix} \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e}{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} \mathtt{let}\ x\!:\!\alpha_x = x_f\ x_a\ \mathtt{in}\ e}$$

$$\frac{\forall \mu f\!:\!\alpha_f.\Lambda\beta.e_b \in \hat{\rho}(x_f)\ .\ \begin{pmatrix}\forall \tau \in \hat{\phi}(\alpha_a)\ .\ \tau \in \hat{\phi}(\beta)) \wedge \\ \forall b_\mathsf{s} \in \hat{\rho}(\mathsf{ResOf}(e_b))\ .\ \hat{\phi} \vDash_\mathsf{S} \mathsf{TyOf}(b_\mathsf{s}) \approx\!\!\!\approx \alpha_x \twoheadrightarrow b_\mathsf{s} \in \hat{\rho}(x)\end{pmatrix} \qquad \hat{\phi}; \hat{\rho} \vDash_\mathsf{S} e}{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} \mathtt{let}\ x\!:\!\alpha_x = x_f\ \mathtt{[}\alpha_a\mathtt{]}\ \mathtt{in}\ e}$$

$$\frac{}{\hat{\phi}; \hat{\rho} \vDash_\mathsf{S} x}$$

Figure 3. Specification-Based Formulation of TCFA

4. Flow-Graph-Based Formulation of TCFA

We give judgments pertaining to the behavior of the Flow-Graph analysis in Figure 4. These judgments are analogous to building a flow-graph of a program P where the edges are the definite flows between types, type variables, simple binders and expression variables. There is also a conditional edge between a binder and an expression variable that is guarded by the type compatibility of the two in P. Once it is learned that the types are indeed compatible, the edge is activated and belongs to the final result. After the graph is constructed, Typed- and Control-Flow Analysis is reduced to Graph Reachability across the flow-graph.

4.1 Flow-Graph Analysis

We define a series of judgments to perform a program parameterized flow-graph analysis, dependent upon the Sub-part relation in Figure 2.

The analysis uses a program based type compatibiliity, defined mutually with type variable compatibility. Any two DeBruijn indices are compatible if they are the same index number. Otherwise, for functions and forall types that have type variable components, the two types are compatible if the corresponding components are compatible under the program. Two type variables, α_1 and α_2 are compatible in a program if there is a τ_1 and τ_2 flowing to α_1 and α_2, respectively, such that τ_1 is compatible with τ_2 in the program.

The flow-graph constructed by the analysis performed on a program, P, consists of both type-flow information and value-flow information.

We know that a type, τ, flows to a given type variable, α_1, when one of two cases is true: there is an explicit `let`-α binding in the program involving τ and α_1; if not, there is some α_2 and we have learned that α_1 flows to α_2. Such an edge between type variables is constructed if there exists a type application in P where α_2 is the formal type parameter and α_1 is the supplied type argument.

For a simple binder, b_s to flow to an expression variable, x, we need to know two pieces of information. We must have already seen that the binder could possibly flow to the variable and also that the type of the binder is compatible with the type of the variable. Initially, for all λ- and Λ-abstractions, we assert that the abstraction flows to its own recursive function variable. Other conditional flow edges are added whenever the program contains a `let`-x expression using b_s and x and whenever we learn of a transitive variable flow. Whenever we see an expression application and we already know that a λ-abstraction flows to the function variable, we add a flow edge between the argument variable and the formal parameter variable dependent upon the type of the parameter and also between the return of the function and the variable being bound by the `let`-x expression. The return of Λ-abstraction also flows to a `let`-x bound variable dependent upon the bound variable's type if the binder is a type application and the Λ-abstraction flows the function variable.

4.2 Soundness

From our flow-based analysis, we prove that if we have a $\hat{\phi}$ and $\hat{\rho}$ that are "flow-induced" from a well-typed program, then they soundly model the program. Before we begin, we introduce the following lemma:

Lemma 1. *For all P, $\hat{\phi}$, if*
$\forall \alpha, \tau \,.\, \tau \in \hat{\phi}(\alpha) \Leftrightarrow P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$
then $P \vDash_{\mathsf{G}} \alpha_1 \approx\!\!\!\approx \alpha_2 \Leftrightarrow \hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx\!\!\!\approx \alpha_2$

We assert that the lemma is true without an accompanying proof. By inspection, the forward case must be true because the derivation of $P \vDash_{\mathsf{G}} \alpha_1 \approx\!\!\!\approx \alpha_2$ tells us that there exists a τ_1 flowing to α_1 and a τ_2 flowing to α_2 such that $P \vDash_{\mathsf{G}} \tau_1 \approx\!\!\!\approx \tau_2$. We assume that this can be translated to $\hat{\phi} \vDash_{\mathsf{S}} \tau_1 \twoheadrightarrow \theta$ and $\hat{\phi} \vDash_{\mathsf{S}} \tau_2 \twoheadrightarrow \theta$, thus allowing us to derive our conclusion. The backwards case is a reflection of the logic in the forward case.

Theorem 1. *For all P, $\hat{\phi}$, and $\hat{\rho}$, if*

- $\forall \alpha, \tau \,.\, \tau \in \hat{\phi}(\alpha) \Leftrightarrow P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$, *and*
- $\forall x, b_{\mathsf{s}} \,.\, b_{\mathsf{s}} \in \hat{\rho}(x) \Leftrightarrow P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$

then $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{s}} P$.

Proof. By induction on P. The interesting cases are when P is a `let`-x bound to a λ-abstraction and also when P is a `let`-x bound to a type application.

Case P of $\texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b\ \texttt{in}\ e$: Knowing that P is unconditionally a sub-term of itself, we can deduce that $\texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b\ \texttt{in}\ e$ is a subterm of P. Thus we can build a conditional edge in the flow graph from the abstraction to x. Since P is well-typed, we know that $P \vDash_{\mathsf{G}} \alpha_f \approx\!\!\!\approx \alpha_x$ and can thus derive $P \vDash_{\mathsf{G}} \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b \rightarrowtail x$ and $\hat{\phi} \vDash_{\mathsf{S}} \alpha_f \approx\!\!\!\approx \alpha_x$. Our assumption thus gives us $b_{\mathsf{s}} \in \hat{\rho}(x)$ from $P \vDash_{\mathsf{G}} \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b \rightarrowtail x$. We have $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e_b$ and $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e$ by our inductive hypothesis, and thus we have our derivation for $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b\ \texttt{in}\ e$.

Case P of $\texttt{let}\ x{:}\alpha_x = x_f\ [\alpha_a]\ \texttt{in}\ e$: We again start by asserting that $\alpha_x = x_f\ [\alpha_a]\ \texttt{in}\ e$ is a subterm of P. If there is a $\mu f{:}\alpha_f \,.\, \Lambda\beta \,.\, e_b$ that flows to x_f, then we can create a conditional edge from the result of the Λ-abstraction to x and an unconditional edge from α_a and β. The conditional edge only allows binders that are type compatible in the flow specification to flow transitively from the result variable to x. By our Lemma we can show that the binders that flow to x are type compatible with α_x and our assumption tells us that those binders are in the entry for x in $\hat{\rho}$. The graph also tells us that any τ flowing to α_a flows to β, which we can use to show that τ is in the entry for β in $\hat{\phi}$ from our assumption. Our inductive hypothesis allows us to show that $\hat{\phi}$ and $\hat{\rho}$ soundly model e, and we thus derive that $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{G}} \texttt{let}\ x{:}\alpha_x = x_f\ [\alpha_a]\ \texttt{in}\ e$. □

$\boxed{P \vDash_{\mathsf{G}} \tau_1 \approxeq \tau_2}$

$$\textsc{TyCompatArrow}\ \frac{P \vDash_{\mathsf{G}} \alpha_{a1} \approxeq \alpha_{a2} \qquad P \vDash_{\mathsf{G}} \alpha_{b1} \approxeq \alpha_{b2}}{P \vDash_{\mathsf{G}} \alpha_{a1} \to \alpha_{b1} \approxeq \alpha_{a2} \to \alpha_{b2}} \qquad \textsc{TyCompatForAll}\ \frac{P \vDash_{\mathsf{G}} \alpha_{b1} \approxeq \alpha_{b2}}{P \vDash_{\mathsf{G}} \forall.\ \alpha_{b1} \approxeq \forall.\ \alpha_{b2}} \qquad \textsc{TyCompatTyIdx}\ \frac{}{P \vDash_{\mathsf{G}} \#n \approxeq \#n}$$

$\boxed{P \vDash_{\mathsf{G}} \alpha_1 \approxeq \alpha_2}$

$$\textsc{TyVarCompat}\ \frac{P \vDash_{\mathsf{G}} \tau_1 \rightarrowtail \alpha_1 \qquad P \vDash_{\mathsf{G}} \tau_2 \rightarrowtail \alpha_2 \qquad P \vDash_{\mathsf{G}} \tau_1 \approxeq \tau_2}{P \vDash_{\mathsf{G}} \alpha_1 \approxeq \alpha_2}$$

$\boxed{P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha}$

$$\textsc{LetTyBnd}\ \frac{\texttt{let}\ \alpha = \tau\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha} \qquad \textsc{TransTyBnd}\ \frac{P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha \qquad P \vDash_{\mathsf{G}} \alpha \rightarrowtail \beta}{P \vDash_{\mathsf{G}} \tau \rightarrowtail \beta}$$

$\boxed{P \vDash_{\mathsf{G}} \alpha \rightarrowtail \beta}$

$$\textsc{TyAppArg}\ \frac{P \vDash_{\mathsf{G}} \mu f{:}\alpha_f.\Lambda\beta.e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ [\alpha_a]\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} \alpha_a \rightarrowtail \beta}$$

$\boxed{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x}$

$$\textsc{TyVarCompatExpBnd}_{\mathsf{s}}\ \frac{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x \qquad P \vdash \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x}{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x}$$

$\boxed{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x}$

$$\textsc{LetExpBnd}_{\mathsf{s}}\ \frac{\texttt{let}\ x{:}\alpha_x = b_{\mathsf{s}}\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x} \qquad \textsc{TransExpBnd}_{\mathsf{s}}\ \frac{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x \qquad P \vDash_{\mathsf{G}} x \rightarrowtail y : \alpha_y}{P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} y : \alpha_y}$$

$$\mu\lambda\textsc{ExpBnd}_{\mathsf{s}}\ \frac{\texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f.\lambda z{:}\alpha_z.e_b\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} \mu f{:}\alpha_f.\lambda z{:}\alpha_z.e_b \rightarrowtail^{?} f : \alpha_f} \qquad \mu\Lambda\textsc{ExpBnd}_{\mathsf{s}}\ \frac{\texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f.\Lambda\beta.e_b\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} \mu f{:}\alpha_f.\Lambda\beta.e_b \rightarrowtail^{?} f : \alpha_f}$$

$\boxed{P \vDash_{\mathsf{G}} x \rightarrowtail y : \alpha_y}$

$$\textsc{ExpAppArg}\ \frac{P \vDash_{\mathsf{G}} \mu f{:}\alpha_f.\lambda z{:}\alpha_z.e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ x_a\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} x_a \rightarrowtail z : \alpha_z}$$

$$\textsc{ExpAppRes}\ \frac{P \vDash_{\mathsf{G}} \mu f{:}\alpha_f.\lambda z{:}\alpha_z.e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x_f\ x_a\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} \mathsf{ResOf}(e_b) \rightarrowtail x_r : \alpha_r}$$

$$\textsc{TyAppRes}\ \frac{P \vDash_{\mathsf{G}} \mu f{:}\alpha_f.\Lambda\beta.e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ [\alpha_a]\ \texttt{in}\ e \preceq_{Exp} P}{P \vDash_{\mathsf{G}} \mathsf{ResOf}(e_b) \rightarrowtail x_r : \alpha_r}$$

Figure 4. Flow-Graph-Based Formulation of TCFA

5. Algorithm

In Figures 5, 6, and 7, we give a direct algorithm implementing the flow-graph-based formulation of the type- and control-flow analysis. The algorithm returns a result set R whose elements correspond to judgements from Figure 4 that are proven to be derivable with respect to the input program P. After an initialization phase, the algorithm uses a work-queue W to process each element that is added to R; when a newly added element is processed, all of the inference rules for which the newly added element could be an antecedent are inspected to determine if the corresponding conclusion can now be added to R. In order to achieve our desired time complexity, there is a map T from elements of the form $\alpha_1 \approxeq \alpha_2$ to a queue of conclusions that may be added to R when $\alpha_1 \approxeq \alpha_2$ is proved to be derivable; the queues in T will serve as "banks" holding credit for the amortized complexity analysis.

5.1 Commentary

5.1.1 Initialization Phase

The first initialization phase (lines 5–18) adds to R and W all elements of the form $b_\mathsf{s} \rightarrowtail^? x : \alpha_x$ that are derivable using the rules LetExpBnd$_\mathsf{s}$, $\mu\lambda$ExpBnd$_\mathsf{s}$, and $\mu\Lambda$ExpBnd$_\mathsf{s}$, rules whose conclusion follows directly from the input program. Similarly, the second initialization phase (lines 19–22) adds to R and W all elements of the form $\tau \rightarrowtail \alpha$ that are derivable using the rule LetTyBnd$_\mathsf{s}$.

The third initialization phase (lines 23–27) prepares the map T, creating an empty queue for each pair of type variables that appear in the input program.

The fourth initialization phase (lines 28–48) handles the rules TyCompatArrow, TyCompatForAll, and TyCompatTyIdx for all type binds that appear in the input program. When τ_1 and τ_2 are arrow types, then $\tau_1 \approxeq \tau_2$ is derivable using the rule TyCompatArrow when the argument type variables are known to be compatible and the result type variables are known to be compatible. Therefore, we create a counter c initialized with the value 2 and add the element $\langle c, \tau_1 \approxeq \tau_2 \rangle$ to the queues in map T for the elements $\alpha_{a1} \approxeq \alpha_{a2}$ and $\alpha_{b1} \approxeq \alpha_{b2}$. The element $\langle c, \tau_1 \approxeq \tau_2 \rangle$ indicates that τ_1 and τ_2 will be known to be compatible when two pairs of type variables are known to be compatible; when $\alpha_{a1} \approxeq \alpha_{a2}$ and $\alpha_{b1} \approxeq \alpha_{b2}$ are known to be compatible, the counter will be decremented and when the counter is zero, $\tau_1 \approxeq \tau_2$ will be added to R and W (see lines 148–156). Similarly, when τ_1 and τ_2 are forall types, then $\tau_1 \approxeq \tau_2$ is derivable using the rule TyCompatForAll when the result type variables are known to be compatible and we create a counter c initialized with the value 1 and add the element $\langle c, \tau_1 \approxeq \tau_2 \rangle$ to the queue in map T for the element $\alpha_{b1} \approxeq \alpha_{b2}$. Finally, when τ_1 and τ_2 are the same type index, then $\tau_1 \approxeq \tau_2$ is immediately derivable using the rule TyCompatTyIdx and $\tau_1 \approxeq \tau_2$ is added to R and W.

5.1.2 Work-queue Phase

The work-queue phase repeatedly pops an element from the work-queue W and processes the element (possibly adding new elements to R and W) until W is empty. To process an element, all of the inference rules for which the element could be an antecedent are inspected to determine if the corresponding conclusion can now be added to R and W.

When the work-queue element is of the form $x \rightarrowtail y : \alpha_y$ (lines 51–58), only the rule TransExpBnd$_\mathsf{s}$ need be inspected. For each $b_\mathsf{s} \rightarrowtail x$ that is already known to be derivable, then TransExpBnd$_\mathsf{s}$ may derive $b_\mathsf{s} \rightarrowtail^? y : \alpha_y$ and it is added to R and W.

When the work-queue element is of the form $b_\mathsf{s} \rightarrowtail x : \alpha_x$ (lines 59–68), only the rule TyVarCompatExpBnd$_\mathsf{s}$ need be inspected. If $\mathsf{TyOf}(b_\mathsf{s})$ and α_x are already known to be compatible, then TyVarCompatExpBnd$_\mathsf{s}$ may derive $b_\mathsf{s} \rightarrowtail x$ and it is added to R and W. If $\mathsf{TyOf}(b_\mathsf{s})$ and α_x are not yet known to be compatible, then the element $b_\mathsf{s} \rightarrowtail x$ is added to the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_\mathsf{s}) \approxeq \alpha_x)$, indicating that when $\mathsf{TyOf}(b_\mathsf{s})$ and α_x are known to be compatible, $b_\mathsf{s} \rightarrowtail x$ will be added to R and W (see lines 142–147).

When the work-queue element is of the form $b_\mathsf{s} \rightarrowtail x$ (lines 69–104), the rules TransExpBnd$_\mathsf{s}$, ExpAppArg, ExpAppRes, TyAppArg, and TyAppRes need to be inspected. For each $x \rightarrowtail y : \alpha_y$ that is already known to be derivable, then TransExpBnd$_\mathsf{s}$ may derive $b_\mathsf{s} \rightarrowtail^? y : \alpha_y$ and it is added to R and W. When b_s is of the form $\mu f : \alpha_f \,.\, \lambda z : \alpha_z \,.\, e_b$ where $x_b = \mathsf{ResOf}(e_b)$ (lines 77–89), all expression applications $\texttt{let}\ x_r : \alpha_r = x\ x_a\ \texttt{in}\ e$ in the input program are examined to determine if ExpAppArg may derive $x_a \rightarrowtail z : \alpha_z$ and if ExpAppRes may derive $x_b \rightarrowtail x_r : \alpha_r$. Similarly, when b_s is of the form $\mu f : \alpha_f \,.\, \Lambda\beta \,.\, e_b$ where $x_b = \mathsf{ResOf}(e_b)$ (lines 90–102), all expression applications $\texttt{let}\ x_r : \alpha_r = x\ [\alpha_a]\ \texttt{in}\ e$ in the input program are examined to determine if TyAppArg may derive $\alpha_a \rightarrowtail \beta$ and if TyAppRes may derive $x_b \rightarrowtail x_r : \alpha_r$.

When the work-queue element is of the form $\tau \rightarrowtail \alpha$ (lines 105–120), the rules TransTyBnd and TyVarCompat need to be inspected. For each $\alpha \rightarrowtail \beta$ that is already known to be derivable, then TransTyBnd may derive $\tau \rightarrowtail \beta$ and it is added to R and W. For each $\pi \rightarrowtail \beta$ that is already known to be derivable, if τ and π are already known to be compatible, then TyVarCompat may derive $\alpha \approxeq \beta$ and it is added to R and W.

When the work-queue element is of the form $\alpha \rightarrowtail \beta$ (lines 121–128), only the rule TransTyBnd need be inspected. For each $\tau \rightarrowtail \alpha$ that is already known to be derivable, then TransTyBnd may derive $\tau \rightarrowtail \beta$ and it is added to R and W.

When the work-queue element is of the form $\tau_1 \approxeq \tau_2$ (lines 129–138), only the rule TyVarCompat need be inspected. For each $\tau_1 \rightarrowtail \alpha_1$ and $\tau_2 \rightarrowtail \alpha_2$ that are known to be derivable, then TyVarCompat may derive $\alpha_1 \approxeq \alpha_2$ and it is added to R and W.

Finally, when the work-queue element is of the form $\alpha_1 \approxeq \alpha_2$ (lines 139–159), the rules TyVarCompatExpBnd$_\mathsf{s}$ and TyVarCompat need to be inspected. Each time that $b_\mathsf{s} \rightarrowtail x : \alpha_x$ was known to be derivable but $\mathsf{TyOf}(b_\mathsf{s})$ and α_x were not yet known to be compatible (preventing TyVarCompatExpBnd$_\mathsf{s}$ from deriving $b_\mathsf{s} \rightarrowtail x$), an element of the form $b_\mathsf{s} \rightarrowtail x$ was added to the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_\mathsf{s}) \approxeq \alpha_x)$ (see line 66); hence, processing these elements of the queue will add each $b_\mathsf{s} \rightarrowtail x$ that may be derived by TyVarCompatExpBnd$_\mathsf{s}$ to R and W. For each pair of type binds τ_1 and τ_2 whose compatibility depends upon the compatibility of α_1 and α_2 (and possibly upon the compatibility of other pairs of type variables), an element of the form $\langle c, \tau_1 \approxeq \tau_2 \rangle$, where c indicates the total number of pairs of type variables whose compatibility will establish the compatibility of τ_1 and τ_2, was added to the queue given by $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$ (see lines 33, 34, and 38); hence, processing these elements of the queue will add each $\tau_1 \approxeq \tau_2$ that may be derived by TyVarCompat to R and W.

5.1.3 Termination

Note that throughout the algorithm, whenever an element is added to the result set R, it is simultaneously added to the work-queue W. Furthermore, an element is added to R and W only after checking that the element is not already in R, except during the initialization phase when all elements added to R and W are necessarily not already in R. Hence, elements are only added to W once and the work-queue phase of the algorithm terminates because, for a given input program, there are only a finite number of elements that may be added to R and W.

Ensure: $\forall \tau_1 \preceq_{TyBnd} P \,.\, \forall \tau_2 \preceq_{TyBnd} P \,.\, \tau_1 \approx\!\!\approx \tau_2 \in R \Leftrightarrow P \vDash_{\mathsf{G}} \tau_1 \approx\!\!\approx \tau_2$
Ensure: $\alpha_1 \approx\!\!\approx \alpha_2 \in R \Leftrightarrow P \vDash_{\mathsf{G}} \alpha_1 \approx\!\!\approx \alpha_2$
Ensure: $\tau \rightarrowtail \alpha \in R \Leftrightarrow P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$
Ensure: $\alpha \rightarrowtail \beta \in R \Leftrightarrow P \vDash_{\mathsf{G}} \alpha \rightarrowtail \beta$
Ensure: $b_{\mathsf{s}} \rightarrowtail x \in R \Leftrightarrow P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$
Ensure: $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x \in R \Leftrightarrow P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$
Ensure: $x \rightarrowtail y : \alpha_y \in R \Leftrightarrow P \vDash_{\mathsf{G}} x \rightarrowtail y : \alpha_y$

1: **procedure** TCFA(P) ▷ $O(l^3 + m^4) = O(l^2) + O(m^2) + O(1) + O(m^2) + O(l) + O(m) + O(m^2) + O(m^2) + O(l^3) + O(l^2) + O(l^3) + O(m^4) + O(m^3) + O(m^4) + O(m^2)$
2: $R \leftarrow$ Set.newEmpty() ▷ $O(l^2) + O(m^2)$
3: $W \leftarrow$ Queue.newEmpty() ▷ $O(1)$
4: $T \leftarrow$ Map.newEmpty() ▷ $O(m^2)$

5: **for all** `let` $x : \alpha_x$ = b_{s} `in` $e \preceq_{Exp} P$ **do** ▷ $O(l) = O(l) \times O(1)$
6: Set.insert($R, b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$)
7: Queue.push($W, b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$)
8: **match** b_{s} **with**
9: **case** $\mu f : \alpha_f \,.\, \lambda z : \alpha_z \,.\, e_b$ **do**
10: Set.insert($R, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
11: Queue.push($W, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
12: **end case**
13: **case** $\mu f : \alpha_f \,.\, \Lambda \beta \,.\, e_b$ **do**
14: Set.insert($R, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
15: Queue.push($W, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
16: **end case**
17: **end match**
18: **end for**

19: **for all** `let` α = τ `in` $e \preceq_{Exp} P$ **do** ▷ $O(m) = O(m) \times O(1)$
20: Set.insert($R, \tau \rightarrowtail \alpha$)
21: Queue.push($W, \tau \rightarrowtail \alpha$)
22: **end for**

23: **for all** $\alpha_1 \preceq_{TyVar} P$ **do** ▷ $O(m^2) = O(m) \times O(m)$
24: **for all** $\alpha_2 \preceq_{TyVar} P$ **do** ▷ $O(m) = O(m) \times O(1)$
25: Map.set($T, \alpha_1 \approx\!\!\approx \alpha_2$, Queue.newEmpty())
26: **end for**
27: **end for**

28: **for all** $\tau_1 \preceq_{TyBnd} P$ **do** ▷ $O(m^2) = O(m) \times O(m)$
29: **for all** $\tau_2 \preceq_{TyBnd} P$ **do** ▷ $O(m) = O(m) \times O(1)$
30: **match** $\langle \tau_1, \tau_2 \rangle$ **with**
31: **case** $\langle \alpha_{a1} \rightarrow \alpha_{b1}, \alpha_{a2} \rightarrow \alpha_{b2} \rangle$ **do**
32: $c \leftarrow$ Counter.new(2)
33: Queue.push(Map.get($T, \alpha_{a1} \approx\!\!\approx \alpha_{a2}$), $\langle c, \tau_1 \approx\!\!\approx \tau_2 \rangle$) ▷ $O(1)$ credit into $T[\alpha_{a1} \approx\!\!\approx \alpha_{a2}]$ queue
34: Queue.push(Map.get($T, \alpha_{b1} \approx\!\!\approx \alpha_{b2}$), $\langle c, \tau_1 \approx\!\!\approx \tau_2 \rangle$) ▷ $O(1)$ credit into $T[\alpha_{b1} \approx\!\!\approx \alpha_{b2}]$ queue
35: **end case**
36: **case** $\langle \forall.\ \alpha_{b1}, \forall.\ \alpha_{b2} \rangle$ **do**
37: $c \leftarrow$ Counter.new(1)
38: Queue.push(Map.get($T, \alpha_{b1} \approx\!\!\approx \alpha_{b2}$), $\langle c, \tau_1 \approx\!\!\approx \tau_2 \rangle$) ▷ $O(1)$ credit into $T[\alpha_{b1} \approx\!\!\approx \alpha_{b2}]$ queue
39: **end case**
40: **case** $\langle \#n, \#m \rangle$ **do**
41: **if** $n = m$ **then**
42: Set.insert($R, \tau_1 \approx\!\!\approx \tau_2$)
43: Queue.push($W, \tau_1 \approx\!\!\approx \tau_2$)
44: **end if**
45: **end case**
46: **end match**
47: **end for**
48: **end for**

Figure 5. TCFA Algorithm

```
49:  while ¬Queue.empty?(W) do
50:    match Queue.pop(W) with

51:      case x ⇀ y : α_y do                                          ▷ O(l^3) = O(l^2) × O(l)
52:        for all b_s ⇀ x ∈ R do                                     ▷ O(l) = O(l) × O(1)
53:          if b_s ⇀^? y : α_y ∉ R then
54:            Set.insert(R, b_s ⇀^? y : α_y)
55:            Queue.push(W, b_s ⇀^? y : α_y)
56:          end if
57:        end for
58:      end case

59:      case b_s ⇀^? x : α_x do                                      ▷ O(l^2) = O(l^2) × O(1)
60:        if TyOf(b_s) ≋ α_x ∈ R then
61:          if b_s ⇀ x ∉ R then
62:            Set.insert(R, b_s ⇀ x)
63:            Queue.push(W, b_s ⇀ x)
64:          end if
65:        else
66:          Queue.push(Map.get(T, TyOf(b_s) ≋ α_x), b_s ⇀ x)        ▷ O(1) credit into T[TyOf(b_s) ≋ α_x] queue
67:        end if
68:      end case

69:      case b_s ⇀ x do                                              ▷ O(l^3) = O(l^2) × O(l)
70:        for all x ⇀ y : α_y ∈ R do                                 ▷ O(l) = O(l) × O(1)
71:          if b_s ⇀^? y : α_y ∉ R then
72:            Set.insert(R, b_s ⇀^? y : α_y)
73:            Queue.push(W, b_s ⇀^? y : α_y)
74:          end if
75:        end for
76:        match b_s with
77:          case μf : α_f . λz : α_z . e_b do
78:            x_b ← ResOf(e_b)
79:            for all let x_r : α_r = x x_a in e ⪯_Exp P do           ▷ O(l) = O(l) × O(1)
80:              if x_a ⇀ z : α_z ∉ R then
81:                Set.insert(R, x_a ⇀ z : α_z)
82:                Queue.push(W, x_a ⇀ z : α_z)
83:              end if
84:              if x_b ⇀ x_r : α_r ∉ R then
85:                Set.insert(R, x_b ⇀ x_r : α_r)
86:                Queue.push(W, x_b ⇀ x_r : α_r)
87:              end if
88:            end for
89:          end case
90:          case μf : α_f . Λβ . e_b do
91:            x_b ← ResOf(e_b)
92:            for all let x_r : α_r = x [α_a] in e ⪯_Exp P do       ▷ O(l) = O(l) × O(1)
93:              if α_a ⇀ β ∉ R then
94:                Set.insert(R, α_a ⇀ β)
95:                Queue.push(W, α_a ⇀ β)
96:              end if
97:              if x_b ⇀ x_r : α_r ∉ R then
98:                Set.insert(R, x_b ⇀ x_r : α_r)
99:                Queue.push(W, x_b ⇀ x_r : α_r)
100:             end if
101:           end for
102:         end case
103:       end match
104:     end case
```

Figure 6. TCFA Algorithm (continued)

```
105:            case τ ⇀ α do                                          ▷ O(m⁴) = O(m²) × O(m²)
106:                for all α ⇀ β ∈ R do                                   ▷ O(m) = O(m) × O(1)
107:                    if τ ⇀ β ∉ R then
108:                        Set.insert(R, τ ⇀ β)
109:                        Queue.push(W, τ ⇀ β)
110:                    end if
111:                end for
112:                for all π ⇀ β ∈ R do                                   ▷ O(m²) = O(m²) × O(1)
113:                    if τ ≋ π ∈ R then
114:                        if α ≋ β ∉ R then
115:                            Set.insert(R, α ≋ β)
116:                            Queue.push(W, α ≋ β)
117:                        end if
118:                    end if
119:                end for
120:            end case

121:            case α ⇀ β do                                          ▷ O(m³) = O(m²) × O(m)
122:                for all τ ⇀ α ∈ R do                                   ▷ O(m) = O(m) × O(1)
123:                    if τ ⇀ β ∉ R then
124:                        Set.insert(R, τ ⇀ β)
125:                        Queue.push(R, τ ⇀ β)
126:                    end if
127:                end for
128:            end case

129:            case τ₁ ≋ τ₂ do                                        ▷ O(m⁴) = O(m²) × O(m²)
130:                for all τ₁ ⇀ α₁ ∈ R do                                 ▷ O(m²) = O(m) × O(m)
131:                    for all τ₂ ⇀ α₂ ∈ R do                                 ▷ O(m) = O(m) × O(1)
132:                        if α₁ ≋ α₂ ∉ R then
133:                            Set.insert(R, α₁ ≋ α₂)
134:                            Queue.push(W, α₁ ≋ α₂)
135:                        end if
136:                    end for
137:                end for
138:            end case

139:            case α₁ ≋ α₂ do                                        ▷ O(m²) = O(m²) × O(1)
140:                while ¬Queue.empty?(Map.get(T, α₁ ≋ α₂)) do
141:                    match Queue.pop(Map.get(T, α₁ ≋ α₂)) with          ▷ O(1) credit from T[α_a1 ≋ α_a2] queue
142:                        case b_s ⇀ x do
143:                            if b_s ⇀ x ∉ R then
144:                                Set.insert(R, b_s ⇀ x)
145:                                Queue.push(W, b_s ⇀ x)
146:                            end if
147:                        end case
148:                        case ⟨c, τ₁ ≋ τ₂⟩ do
149:                            Counter.dec(c)
150:                            if Counter.get(c) = 0 then
151:                                if τ₁ ≋ τ₂ ∉ R then
152:                                    Set.insert(R, τ₁ ≋ τ₂)
153:                                    Queue.push(W, τ₁ ≋ τ₂)
154:                                end if
155:                            end if
156:                        end case
157:                    end match
158:                end while
159:            end case

160:        end match
161:    end while

162:    return R
163: end procedure
```

Figure 7. TCFA Algorithm (continued)

5.2 Complexity

5.2.1 Preliminaries

Before analyzing the time complexity of the algorithm, we first make some (standard) assumptions about the representation of the input program.

We assume that all `let`-, μ-, and λ-bound expression variables and all `let`-, and Λ-bound type variables in the program are distinct. We further assume that expression variables and type variables can be mapped (in $O(1)$ time) to unique integers (for $O(1)$ time indexing into an array) and that integers can be mapped (in $O(1)$ time) to corresponding expression variables and type variables.[2] Given the assumption that all `let`-, μ-, and λ-bound expression variables in the program are unique, each expression variable in the program is annotated with a single type variable at its unique binding occurrence. We therefore assume that expression variables can be mapped (in $O(1)$ time) to its annotating type variable. Given the assumption that all μ-bound expression variables in the program are unique, each simple expression bind in the program is unique and can be mapped (in $O(1)$ time) to and from unique integers.[3] We do not assume that each type bind in the program is unique, but we do assume that each type bind in the program can be mapped (in $O(1)$ time) to and from unique integers. Finally, we assume that $\mathsf{ResOf}(\cdot)$ can be computed in $O(1)$ time.[4]

5.2.2 Data Structures and Operations

We next consider the data structures used to implement the result set R and the map T and the cost of various operations.

The result set R is implemented as seven multi-dimensional arrays, each corresponding to one of the seven judgements from Figure 4. Given the assumptions above, it is easy to see that the arrays corresponding to $\tau_1 \approxeq \tau_2$, $\alpha_1 \approxeq \alpha_2$, $\tau \rightarrowtail \alpha$, $\alpha \rightarrowtail \beta$, and $b_\mathsf{s} \rightarrowtail x$ are simple two-dimensional arrays with $O(1)$ time indexing by mapping components to unique integers. Furthermore, the arrays corresponding to $b_\mathsf{s} \rightarrowtail x : \alpha_x$ and $x \rightarrowtail y : \alpha_y$ can also be implemented with simple two-dimensional arrays (indexed by b_s/x and x/y, respectively), because the type variable is always the single type variable at the unique binding occurrence of the expression variable and can be left implicit. Thus, queries like $b_\mathsf{s} \rightarrowtail x \notin R$ and operations like $\mathsf{Set.insert}(R, b_\mathsf{s} \rightarrowtail x)$ can be performed in constant time. Loops like "**for all** $b_\mathsf{s} \rightarrowtail x \in R$ **do**" for fixed b_s instantiating x or for fixed x instantiating b_s can be implemented as a linear scan of an array column or array row.

The map T is implemented with a simple two-dimensional array, indexed by pairs of type variables. Operations like $\mathsf{Map.set}(T, \alpha_1 \approxeq \alpha_2, q)$ and $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$ can be performed in constant time.

The work-queue W and the queues in map T are implemented with a simple linked-list queue. Queries like $\mathsf{Queue.empty?}(W)$ and operations like $\mathsf{Queue.push}(W, b_\mathsf{s} \rightarrowtail x)$ and $\mathsf{Queue.pop}(W)$ can be performed in constant time.

5.2.3 Coarse Analysis

We first argue that the algorithm is $O(n^4)$ time, where n is the size of the input program P. First, note that there are $O(n)$ type variables, $O(n)$ type binds, $O(n)$ expression variables, and $O(n)$ simple expression binds in the program. Thus, the result set R requires $O(n^2)$ space for (and is $O(n^2)$ time to create) each of the seven two-dimensional arrays and the map T requires $O(n^2)$ space for (and is $O(n^2)$ time to create) the two-dimensional array.

The first initialization phase is $O(n)$ time to traverse the program and process each simple expression bind. Similarly, the second initialization phase is $O(n)$ time to traverse the program and process each type bind. The third initialization phase is $O(n^2)$ time to process each pair of type variables. The fourth initialization phase is $O(n^2)$ time to process each pair of type binds. Altogether, the initialization phase is $O(n^2) = O(n)+O(n)+O(n^2)+O(n^2)$ time.

As noted above, elements are only added to W once. Therefore, the time complexity of the "**while** $\neg\mathsf{Queue.empty?}(W)$ **do**"-loop is the sum of the time required to process an element of each kind times the number of elements of that kind. There are $O(n^2)$ elements of the form $x \rightarrowtail y : \alpha_y$ (recall that the α_y is implicitly determined by the y) and processing an $x \rightarrowtail y : \alpha_y$ element is $O(n)$ time to scan for all $b_\mathsf{s} \rightarrowtail x \in R$. There are $O(n^2)$ elements of the form $b_\mathsf{s} \rightarrowtail^? x : \alpha_x$ and processing a $b_\mathsf{s} \rightarrowtail^? x : \alpha_x$ element is $O(1)$ time. There are $O(n^2)$ elements of the form $b_\mathsf{s} \rightarrowtail x$ and processing a $b_\mathsf{s} \rightarrowtail x$ element is $O(n)$ time to scan all $x \rightarrowtail y : \alpha_y \in R$ and $O(n)$ time to find all $\texttt{let}\ x_r\texttt{:}\alpha_r = x\ x_a\ \texttt{in}\ e \preceq_{Exp} P$ and to find all $\texttt{let}\ x_r\texttt{:}\alpha_r = x\ \texttt{[}\alpha_a\texttt{]}\ \texttt{in}\ e \preceq_{Exp} P$. There are $O(n^2)$ elements of the form $\tau \rightarrowtail \alpha$ and processing a $\tau \rightarrowtail \alpha$ element is $O(n)$ time to scan for all $\alpha \rightarrowtail \beta \in R$ and $O(n^2)$ to process all $\pi \rightarrowtail \beta \in R$. There are $O(n^2)$ elements of the form $\alpha \rightarrowtail \beta$ and processing an $\alpha \rightarrowtail \beta$ element is $O(n)$ time to scan for all $\tau \rightarrowtail \alpha \in R$. There are $O(n^2)$ elements of the form $\tau_1 \approxeq \tau_2$ and processing a $\tau_1 \approxeq \tau_2$ element is $O(n^2)$ time to scan for all $\tau_1 \rightarrowtail \alpha_1 \in R$ and $\tau_2 \rightarrowtail \alpha_2 \in R$. There are $O(n^2)$ elements of the form $\alpha_1 \approxeq \alpha_2$, processing an $\alpha_1 \approxeq \alpha_2$ element must process each element in the queue $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$, and, therefore, the time complexity to process an $\alpha_1 \approxeq \alpha_2$ element is the sum of the time required to process the elements in the queue of each kind times the number of elements of that kind; there are $O(n^2)$ elements of the form $b_\mathsf{s} \rightarrowtail x$ in the queue (since an element of the form $b_\mathsf{s} \rightarrowtail x$ are added at most once to at most one queue (see line 66)) and processing an $b_\mathsf{s} \rightarrowtail x$ element is $O(1)$ time and there are $O(n^2)$ elements of the form $\langle c, \tau_1 \approxeq \tau_2\rangle$ (since an element of the form $\langle c, \tau_1 \approxeq \tau_2\rangle$ is added at most twice to at most one queue (see lines 33–34 and line 38)) and processing a $\langle c, \tau_1 \approxeq \tau_2\rangle$ element is $O(1)$ time. Altogether, the work-queue phase is $O(n^4) = O(n^2) \times O(n) + O(n^2) \times O(1) + O(n^2) \times (O(n) + O(n) + O(n)) + O(n^2) \times (O(n) + O(n^2)) + O(n^2) \times O(n) + O(n^2) \times O(n^2) + O(n^2) \times (O(n^2) \times O(1) + O(n^2) \times O(1))$.

Thus, the entire algorithm is $O(n^4)$. Recall that algorithms for classic (untyped) control-flow analysis have been shown to be $O(n^3)$ [2, 7, 10, 11], though recently improved to $O(n^2 \log n)$ [8].

5.2.4 Refined Analysis

In order to clarify the relationship between the time complexity of algorithms for classic (untyped) control-flow analysis and our algorithm for type- and control-flow analysis, we perform a refined analysis of our algorithm.

First, note that the quartic components of the algorithm are due to the processing of elements of the form $\tau \rightarrowtail \alpha$, $\tau_1 \approxeq \tau_2$, and $\alpha_1 \approxeq \alpha_2$. Intuitively, the increased time complexity of the algorithm for type- and control-flow analysis compared to algorithms for classic (untyped) control-flow analysis is due to the computation of the type-compatibility relations.

[2] These mappings can established with a linear-time preprocessing step.

[3] In a richer language with simple-expression-bind forms that do not include a bound expression variable (e.g., $\langle x_1, x_2\rangle$ pairs), we can assume a numbering of all simple expression binds in the program, similar to the labeling found in textbook presentations of CFA [10].

[4] This can be established either by a linear-time preprocessing step (associating each result variable with its corresponding abstraction) or by changing the representation of expressions to a list of $\alpha = \tau$ and $x\texttt{:}\alpha_x = b$ bindings paired with the result variable.

Second, in typical programs of interest, we expect that the total size of the program to be dominated by the contribution of (bound) expression variables and expression binds, with the contribution of (bound) type variables and type binds significantly (asymptotically?) less. For example, a program may have many definitions of and uses of `int` $\rightarrow$ `int` functions, all of which can share the same (top-level) `let` $\alpha_{\texttt{i}}$ `=` `int` `in` `let` $\alpha_{\texttt{i}\rightarrow\texttt{i}} = \alpha_{\texttt{i}} \rightarrow \alpha_{\texttt{i}}$ `in` ... type bindings. Indeed, our ANF representation of types encourages type-level optimizations such as `let`-floating, common subexpression elimination (CSE), and copy propagation, which would further reduce the contribution of types to the total program size. Therefore, we consider it useful to distinguish l, the size of (bound) expression variables and expression binds, and m, the size of (bound) type variables and type binds, where we have $O(l) + O(m)$ is $O(n)$ and we expect $O(l) \gg O(m)$, though, in the worst-case, both $O(l)$ and $O(l)$ are $O(n)$. We further assume an $O(n)$ preprocessing step that provides an enumeration of all `let` $x{:}\alpha_x$ `=` b `in` $e \preceq_{Exp} P$ in $O(l)$ time and an enumeration of all `let` α `=` τ `in` $e \preceq_{Exp} P$ in $O(m)$ time.

We now argue that the algorithm is $O(l^3 + m^4)$ time. First, note that there are $O(m)$ type variables, $O(m)$ type binds, $O(l)$ expression variables, and $O(l)$ simple expression binds in the program. Thus, the result set R requires $O(l^2 + m^2)$ space for (and is $O(l^2 + m^2)$ time to create) the seven two-dimensional arrays and the map T requires $O(m^2)$ space for (and is $O(m^2)$ time to create) the two-dimensional array.

The first initialization phase is $O(l)$ time to process each simple expression bind. Similarly, the second initialization phase is $O(m)$ time to process each type bind. The third initialization phase is $O(m^2)$ time to process each pair of type variables. The fourth initialization phase is $O(m^2)$ time to process each pair of type binds; included in this processing time is an $O(1)$ credit "deposited" into the queues in T when pushing elements, which "pre-pays" for the processing of the elements when popped. Altogether, the initialization phase is $O(l + m^2) = O(l) + O(m) + O(m^2) + O(m^2)$ time.

The analysis of the work-queue phase is similar to that performed above: the time complexity of the "**while** $\neg\mathsf{Queue.empty?}(W)$ **do**"-loop is the sum of the time required to process an element of each kind times the number of elements of that kind; we simply refine n to l or m as appropriate. We further perform an amortized analysis of the time complexity to process an $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ element and to process an $\alpha_1 \approxeq \alpha_2$ element. Included in the time to process an $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ element is an $O(1)$ credit "deposited" into the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x)$ when pushing elements, which "pre-pays" for the processing of the elements when popped. As before, processing an $\alpha_1 \approxeq \alpha_2$ element must process each element in the queue $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$; however, an $O(1)$ credit may be "withdrawn" from the queue $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$ when popping elements and this $O(1)$ credit may be used to "pay" for the popping and processing of the element. Thus, processing an $\alpha_1 \approxeq \alpha_2$ element is (amortized) $O(1)$ time.[5] Altogether, the work-queue phase is $O(l^3 + m^4) = O(l^2) \times O(l) + O(l^2) \times O(1) + O(l^2) \times (O(l) + O(l) + O(l)) + O(m^2) \times (O(m) + O(m^2)) + O(m^2) \times O(m) + O(m^2) \times O(m^2) + O(m^2) \times O(1)$.

Thus, the entire algorithm is $O(l^3 + m^4)$.

[5] Note that without the amortized analysis, processing an $\alpha_1 \approxeq \alpha_2$ element would be $O(l^2) + O(m^2)$ time and the entire algorithm would be $O(l^3 + l^2m^2 + m^4)$.

6. Conclusion

References

[1] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In J. Hughes, editor, *FPCA'91: Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 427–447, Cambridge, Massachusetts, Aug. 1991. Springer-Verlag.

[2] A. E. Ayers. Efficient closure analysis with reachability. In M. Billaud, P. Castéran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors, *Actes WSA'92 Workshop on Static Analysis*, Bigre, pages 126–134, Bordeaux, France, Sept. 1992. Atelier Irisa, IRISA, Campus de Beaulieu.

[3] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In S. Peyton Jones, editor, *FPCA'95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, June 1995.

[4] M. Fluet. A type- and control-flow analaysis for System F. In R. Hinze, editor, *IFL'12: Post-Proceedings of the 24th International Symposium on Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, Oxford, England, 2013. Springer-Verlag. To appear.

[5] M. Fluet. A type- and control-flow analysis for System F. Technical report, Rochester Institute of Technology, February 2013. `https://ritdml.rit.edu/handle/1850/15920`.

[6] F. Gecseg and M. Steinby. *Tree Automata*. Akademiai Kiado, Budapest, Hungary, 1984.

[7] N. Heintze. Set-based program analysis of ML programs. In C. L. Talcott, editor, *LFP'94: Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 306–317, Orlando, Florida, June 1994.

[8] J. Midtgaard and D. V. Horn. Subcubic control flow analysis algorithms. Computer Science Research Report 125, Roskilde University, Roskilde, Denmark, May 2009. Revised version to appear in Higher-Order and Symbolic Computation.

[9] P. Mishra and U. S. Reddy. Declaration-free type checking. In M. S. Van Deusen, Z. Galil, and B. K. Reid, editors, *POPL'85: Proceedings of the Twelfth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 7–21, New Orleans, Louisiana, Jan. 1985. ACM, ACM.

[10] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[11] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[12] C. A. Stone. Type definitions. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

Worker/wrapper for a Better Life

Extended Abstract

Brad Torrence Mike Stees Andrew Gill

Information and Telecommunication Technology Center
The University of Kansas
{brad.torrence,mstees,andygill}@ittc.ku.edu

Abstract

In Software Engineering, an implementation, and its model, can fall out of step. If we can connect the implementation and model, then development of both artifacts can continue, while retaining confidence in the overall design and implementation. In this paper, we show it is possible in practice to connect together an executable specification of Conway's Game of Life, and a number of implementations, using the worker/wrapper transformation. In particular, we use the rewrite tool HERMIT to apply post-hoc transformations to replace a linked-list based description with other data structures. Directed optimizations allow for highly-efficient executable specifications, where the model becomes the implementation. This work is the first time programmer-directed worker/wrapper has been attempted on a whole application, rather than simply on individual functions. Beyond data representation improvement, we translate our model such that we can execute on a CPU/GPU hybrid, by targeting the accelerate DSL.

1. Introduction

The concepts of the worker/wrapper methodology introduced by Gill and Hutton in [6] are the driving force behind the examples demonstrated in this paper. We have implemented the worker/wrapper methodology in the Haskell Equational Reasoning Model to Implementation Tunnel (HERMIT) system. HERMIT has already demonstrated several examples of its capability of using the worker/wrapper theory [3, 4, 11]. However, these examples have only consisted of small transformations usually over a single function. In this paper, we scale the methodology and tool support to a larger example – an implementation of the Game of Life.

1.1 Worker/Wrapper

The worker/wrapper transformation is a technique for improving the performance of a program by changing the underlying implementation. The transformation generates two components: the "worker", which performs using the new implementation, and the "wrapper", which conceals this new implementation under the original API.

When using worker/wrapper, the programmer provides two functions, `abs` and `rep` that translate from the new representation to the original representation, and from the original representation to the new representation. When given some specific preconditions on the relationship between `abs` and `rep`, we can assume another condition that creates a provably correct worker/wrapper transformation [10]. Specifically, given a fix-point of the original computation,

```
comp = fix work
```

as well as `abs` and `rep`, and the worker/wrapper preconditions, we can rewrite comp into a worker and wrapper.

```
comp = abs worker
worker = fix (rep . work . abs)
```

Critically, the worker is now acting over a new representation. By applying laws that relate `rep`, `work`, and `abs`, we can optimize the `worker`, giving a more efficient program. The user intervention is the choice of `abs` and `rep`, and demonstrating the pre-conditions, and we want to use HERMIT to perform both of these steps, as well as optimize the result.

1.2 HERMIT

HERMIT is a framework that provides tools for interacting with and transforming GHC Core programs. The primary means of facilitating this interaction is through the HERMIT Shell, a REPL interface that allows the user to traverse a GHC Core abstract syntax tree. It is inside this REPL that the user issues commands that construct rewrites from AST to AST [3, 11]. Use of the 'unfold-rule' command in particular, in conjunction with GHC RULES pragma [8], allows the user to construct a rewrite from the given rule [3].

Adding to these capabilities are the new worker/wrapper related commands, 'split-1-beta' and 'split-2-beta'. These new commands perform the worker/wrapper split using the safe correctness conditions discussed in the previous section. These commands take a function argument, which is the target of the split, and two more arguments that comprise a transformation-pair. This pair of functions are referred to as the 'abs' and 'rep' functions [10] or 'work' and 'wrap' functions [6] in previous discussions about the worker/wrapper theory. If this transformation pair is created to meet the necessary preconditions to make 1β or 2β true, then they can be given to a "split" command. The product of a split is a worker, which implements new functionality, and a wrapper, which simply calls the new function, maintaining the original interface.

The creation of these commands have given HERMIT users the ability to transform an entire program using the worker/wrapper methodology, provided the theory can be applied effectively using automated mechanisms such as HERMIT to transform an implementation without changing the source code.

1.3 The Game of Life

To show that worker/wrapper transformations under HERMIT can be accomplished for complex programs composed of multiple source files, a suitable program had to be selected that was complex enough to have merit, but not so complex as to provide an overly involved example. An example program that contains these features is the Game of Life.

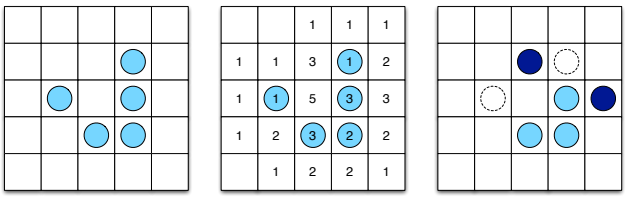
The Game of Life was created by the British mathematician John Horton Conway [5]. Technically, it's a simulation. A player creates the initial state of the board and the rules of the game determine how the board evolves, the player can only observe once the game has begun.

The game board is made of cells arranged in a two dimensional grid. A cell can either be in an alive or dead state, and depending on the state of neighboring cells a cell may change its state. The neighbors of a cell are simply those immediately adjacent to the the target cell in any direction. The rules of the game are simple and as follows:

1. Under-population – A living cell dies if it has fewer than two living neighbors.
2. Stable-population – A living cell remains alive if it has 2 or 3 living neighbors.
3. Over-population – A living cell dies if it has more than three living neighbors.
4. Reproduction – A new cell is born if it is empty and has exactly three living neighbors.

The combination of these simple rules allows for surprisingly interesting and complex patterns to emerge from the game.

The first image in the following figure shows a popular pattern in the game known as the Glider. This pattern replicates through generations. Due to the rules of the game, the pattern moves across the game board in a specific direction. The second image shows the pattern with each cell imprinted with the number of living neighbor cells. These numbers determine the pattern of the next generation, displayed in the third image. The white cells die due to under-population, the dark cells are newly born due to reproduction, and the lightly colored cells remain from the previous generation due to stable-population numbers. These five cells represent the new generation in the game and are used to calculate the next pattern.



The game has been implemented many times. Graham Hutton implemented the Game of Life in Haskell using a simple list-based implementation in a terminal console[7, p. 94-97]. It was from Hutton's original implementation that our implementation was derived.

The implementation used in this experiment still uses Hutton's original design, however it has been slightly modified. The new design restricts the implementation to use a sorted-list. This created an isomorphic structure to transform, making the task of proving code equivalence simpler. The sorted-list is made of two-tuples of integers. Each pair represents a location on the two-dimensional board structure and has a type synonym, `Pos`. The pairs featured in the list represent the position of living cells in each generation of the game. Therefore, when a cell dies that position is removed from the list, and the reverse is true when a new cell is born.

Another difference in Hutton's Life is that the dimensions of the game board were hard coded into the source code. The program has been modified in a way to allow the board configuration to be altered by user input. Through these modifications, the user can change the board dimensions, represented as an integer pair. The user can also dictate whether or not the edges of the board wrap around to the opposite edge, represented as a boolean value. This information is stored into a new type created called a `Config`. It is simply a two-tuple of an integer pair and a boolean. This configuration and the data structure containing the game data are contained in a new structure called a `LifeBoard`, and it is a new data type with a constructor and accessing functions for the board and configuration fields. It also aided in another change to the original program. Hutton's Life source code was also divided. The part of the code that calculates each generation of the game (the engine) has been separated from the part of the code that visualizes each generation (the display). The reason was to allow new engine and display mechanisms to be created and connected. The class which merges the engine and the display elements is the Life class. This abstract interface contains functions that allow the combination of any engine that implements the Life class functions with any display that utilizes the Life class interface. This section shows the module Life.Types which contains the types described.

```
type Pos = (Int,Int)
type Size = (Int,Int)
type Config = (Size,Bool)

class Life b where
    empty :: Config -> b
    diff :: b -> b -> b
    next :: b -> b
    inv :: Pos -> b -> b
    dims :: b -> Size
    alive :: b -> [Pos]

scene :: Life board => Config -> [Pos] -> board
scene = foldr inv . empty

data LifeBoard c b = LifeBoard{ config :: c, board :: b }
    deriving Show
```

The module contains the abstract definition of the Life class mentioned as well as the LifeBoard structure. In addition, the `scene` function is also defined there. It provides a standard way to transform a [Pos] into an implementation-specific board. And the following section shows the Life.Engine.Hutton module which is derived from Hutton's Life.

```
type Board = LifeBoard Config [Pos]

neighbs :: Config -> Pos -> Board
neighbs c@((w,h),warp) (x,y) = LifeBoard c $ sort $ if warp
    then map (\(x,y) -> (x `mod` w, y `mod` h)) neighbors
    else filter
        (\(x,y) -> (x >= 0 && x < w) && (y >= 0 && y < h))
        neighbors
    where neighbors =[(x-1,y-1), (x,y-1), (x+1,y-1), (x-1,y),
                      (x+1,y), (x-1,y+1), (x,y+1), (x+1,y+1)]

isAlive :: Board -> Pos -> Bool
isAlive b p = elem p $ board b

isEmpty :: Board -> Pos -> Bool
isEmpty b = not . (isAlive b)

liveneighbs :: Board -> Pos -> Int
```

```
liveneighbs b =
  length . filter (isAlive b) . board . (neighbs (config b))

survivors :: Board -> Board
survivors b = LifeBoard (config b) $
    filter (\p -> elem (liveneighbs b p) [2,3]) $
        board b

births :: Board -> Board
births b = LifeBoard (config b) $
   filter (\p -> isEmpty b p && liveneighbs b p == 3) $
    nub $ concatMap (board . (neighbs (config b))) $
        board b

nextgen :: Board -> Board
nextgen b = LifeBoard (config b) $
    sort $ board (survivors b) ++ board (births b)

instance Life Board where
    next b = nextgen b
    alive b = board b
    empty c = LifeBoard c []
    dims b = fst $ config b
    diff b1 b2 = LifeBoard (config b1) $
            board b1 \\ board b2
    inv p b = LifeBoard (config b) $
        if isAlive b p
        then filter ((/=) p) $ board b
        else sort $ p : board b
```

Although the top-level program module is loaded into HERMIT, only this engine module is targeted for transformation, leaving the original display mechanisms unchanged and still effective through worker/wrapper methodology.

2. HERMIT Worker/Wrapper Examples

The goal of this experiment is to confirm that an entire program can be transformed with HERMIT using the worker/wrapper methodology. The examples take a slighly-modified version Hutton's original implementation of the game, which uses lists as the data structure for representing the game board, and modify it using HERMIT. This was done by applying the worker/wrapper concept through HERMIT in an effort to change the underlying data structures used in the program.

The following sections describe the examples and the methods used in HERMIT to accomplish a worker/wrapper transformation. Each experiment involved changing Hutton's Life (the list-based implementation) into an implementation using another primary data structure, such as a QuadTree, a Set, and an Unboxed Vector. Along with changing the representation of the game, we found it is also possible to change the hardware used by the program with the inclusion of certain DSLs (like Accelerate, which gives program access to a GPU). First, lets explore the preparation process required to do these transformations within HERMIT.

As outlined in previous worker/wrapper example transformations using HERMIT [11], the process requires the creation of specific GHC rules and a set of transformation functions.

The module that contains this information is referred to as the transformation-module. This module must be tailored specifically to each conversion. A transformation-module requires one pair of conversion functions for each function targeted. When approaching the problem, it is best to start by creating the most basic transformations first. Transformation pairs can be stacked, using more basic pairs in their definitions.

The transformation-module is also where a series of GHC Rules will be written to allow HERMIT to equate sections of Core code between old and new implementations. This is accomplished via the use of GHC rules that are added to the conversion file using the GHC RULES pragma and compiled into the HERMIT session.

Once a worker/wrapper split is made within HERMIT, the process requires making transformations to segments of code in an effort to match the AST with a GHC rule that swaps equivalent code statements.

It is most likely that a HERMIT user will not know all the needed rules to accomplish the transformation before the process begins. At this stage in HERMIT development, it is easiest to take a more organic approach when performing a worker/wrapper conversion. By that, one should create the needed transformation-pairs and a few rules that will be known to be useful to start, then add new rules as they are needed. Adding a new GHC rule to a HERMIT session requires exiting HERMIT and reentering to add any newly created rules to the environment. This was the process used to complete these transformations. That being said the finished product is a HERMIT script that can perform the entire transformation. This script makes a perfect guide to lead through the following examples.

These examples were performed with GHC 7.8 in combination with the latest version of HERMIT. All of them target the same program described in the previous section.

2.1 Example: List-to-Set implementation transform

The first example uses the same data representation for the board structure. The goal of the transformation was to implement the game engine using the Set data structure featured in the standard library Data.Set. The transformation replaces the use of the standard Haskell list with the use of the set in the engine module. This transformation is isomorphic because we have restricted the use of the Set to maintain order of its elements. Therefore, relationship of a sorted-list to a sorted-set is trivially equivalent, swapping the containers used by the engine produces an equivalent program. The representation remains the same, both containers contain pairs of integers that correspond to living cells on the board. The containing LifeBoard structure is maintained except it is morphed to contain a set rather than a list.

2.1.1 Transform Preparation

The transformation-module should contain all the transformation functions necessary to complete the process. To make the type definitions shorter and relate to the list-based engine, the module also includes type synonyms for the `Board` (copied from the source) and `Board'` (the transformed type). These type definitions as well as all the necessary transformation pairs are displayed here for reference.

```
type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (Set Pos)

{-# NOINLINE absb #-}
absb :: Set Pos -> [Pos]
absb = toAscList

{-# NOINLINE repb #-}
repb :: [Pos] -> Set Pos
repb = fromDistinctAscList

{-# NOINLINE absB #-}
absB :: Board' -> Board
absB b = LifeBoard (config b) $ absb (board b)

{-# NOINLINE repB #-}
repB :: Board -> Board'
repB b = LifeBoard (config b) $ repb (board b)

absBx :: (Board' -> a) -> Board -> a
absBx f = f . repB
```

```
repBx :: (Board -> a) -> Board' -> a
repBx f = f . absB

absxB :: (a -> Board') -> a -> Board
absxB f = absB . f

repxB :: (a -> Board) -> a -> Board'
repxB f = repB . f

absCPB :: (Config -> Pos -> Board') -> Config -> Pos -> Board
absCPB f = absxB . f

repCPB :: (Config -> Pos -> Board) -> Config -> Pos -> Board'
repCPB f = repxB . f

absBB :: (Board' -> Board') -> Board -> Board
absBB = absBx . absxB

repBB :: (Board -> Board) -> Board' -> Board'
repBB = repBx . repxB

absPBB :: (Pos -> Board' -> Board')-> Pos -> Board -> Board
absPBB f = absBB . f

repPBB :: (Pos -> Board -> Board) -> Pos -> Board' -> Board'
repPBB f = repBB . f

absBBB :: (Board' -> Board' -> Board') -> Board -> Board -> Board
absBBB f = absBB . f . repB

repBBB :: (Board -> Board -> Board) -> Board' -> Board' -> Board'
repBBB f = repBB . f . absB
```

The purpose of the transformation is to replace the use of lists with the use of sets. The most basic transformation function pair, and the first that should be created, is the pair that transforms the data structure. In staying true to the worker/wrapper methodology the function names begin with `abs` and `rep`. These abs-rep pairs need to be designed to perform bidirectional transformations, where the abs function returns the original form and the rep function returns the new form.

The base pair are named `absb` and `repb`. Both make use of the Data.Set functions that transform a set to/from an ordered list, the `fromDistinctAscList` and `toAscList` functions. Each function of the pair should reverse the results of the other. The compiler directives above these functions are directions that notify GHC to not automatically inline these functions. GHC will do this automatically for some code in an optimization effort. Without these directives the absb and repb functions would not appear in HERMIT session.

To continue, the base pair only transforms the underlying data structure. There also needs to be a pair of functions that will transform the containing data structure `LifeBoard`. Notice how they simply replace one of the contained structures using of the predefined absb or repb functions. This stacking trend continues through all of the transformation-pairs.

Analyzing the source code of the Hutton's Life helps to know what other transformations are necessary. We start with the Life class function `dims`. This function simply returns the board dimensions that are originally entered by the user. The output of the `dims` function is a Size, which doesn't require any transformation because the new implementation retains this original structure. The absBx-repBx pair is defined is polymorphic because with its simple definition it can be used for several conversions. Since the function-types of the `alive`, `isAlive`, `isEmpty`, and `liveneighbs` functions, are similar to the `dims` function, this pair can also be used in their conversion processes.

Next we turn to the function `empty`. This function simply creates an empty board structure. To convert this function the functions, `absxB` and `repxB` would be used. This pair is polymorphic for reasons similar to the absBx-repBx pair, it can be used in several worker/wrapper splits.

Now consider the `neighbs` function. It is used to create a list of neighboring locations depending on the board configuration. This function will require the absCPB-repCPB pair for conversion. The previous polymorphic functions are used in their definitions. This aids the unfolding and code reduction process inside a HERMIT session.

Probably the most crucial function in the Life class for our purposes is the "next" function. This function is used to calculate the next generation of the game from the current board structure. It requires the functions, `absBB` and `repBB`. Upon inspecting the source code, one will note that this type of transformation will be necessary for the engine functions `nextgen`, `births`, and `survivors`. Reusing transformation pairs is useful, and since these functions have the same type, there is no need to create a polymorphic transformation function.

Now consider the `inv` function of the Life class. It takes a board position and inverts the cell status on the given board. The `absPBB` and `repPBB` functions are required for its transformation.

The last function to consider is `diff`. It is used to compare two boards and get return the differences between them in a new board. The `absBBB` `repBBB` definitions are used to convert this function. It has the most complex definition because all of its arguments are of type `Board` or `Board'`, and so requires the most transformation.

With the function pairs completed, we move on to the transformation process within HERMIT. But first, we can assume a few necessary GHC Rules that will definitely be useful, like the following.

```
{-# RULES "repB-absB" [~] forall b. repB (absB b) = b #-}

{-# RULES "LifeBoard-absb" [~] forall c b.
  LifeBoard c (absb b) = absB (LifeBoard c b) #-}

{-# RULES "config-absB" [~] forall b.
  config (absB b) = config b #-}

{-# RULES "board-absB"  [~] forall b.
  board (absB b) = absb (board b) #-}

{-# RULES "repB-LifeBoard" [~] forall c b.
  repB (LifeBoard c b) = LifeBoard c (repb b) #-}
```

These rules are known to be useful because they all simply perform a code transformation designed to move the transformation function node in the AST. For instance, the 'repB-absB' rule is used to eliminate an transformation pair once they have been syntactically juxtaposed. Since the goal is to remove unnecessary transformations, this rule is almost necessity. The other rules shown above are useful for similar reasons, they either eliminate or move a transformer in the AST. Typically one won't know all the rules needed prior to starting a worker/wrapper conversion due to the fact that one may not know what form the Core syntax will take during the process. However, the following rules were discovered and are necessary to complete this example.

```
{-# RULES "repb-null" [~] forall c.
  LifeBoard c (repb []) = LifeBoard c Set.empty #-}

{-# RULES "not-elem-absb" [~] forall p b.
  not (elem p (absb b)) = notMember p b #-}

{-# RULES "elem-absb" [~] forall p b.
  elem p (absb b) = member p b #-}
```

```
{-# RULES "length-absb" [~] forall b.
  length (absb b) = size b #-}

{-# RULES "filter-absb" [~] forall f b.
  Prelude.filter f (absb b) = absb (Set.filter f b) #-}

{-# RULES "sort-++-absb" [~] forall b1 b2.
  sort (absb b1 ++ absb b2) = absb (union b1 b2) #-}

{-# RULES "ncm-absb" [~] forall f b.
  nub (concatMap (\p -> absb (board (f p))) (absb b)) =
  absb (unions (toList (Set.map (\p -> board (f p)) b))) #-}

{-# RULES "diff-absb" [~] forall b1 b2.
  absb b1 List.\\ absb b2 = absb (b1 Set.\\ b2) #-}

{-# RULES "insertion" [~] forall b p.
  sort (p : absb b) = absb (insert p b) #-}

{-# RULES "deletion" [~] forall b p.
  Prelude.filter ((/=) p) (absb b) = absb (delete p b) #-}
```

Notice that each of these rules replaces some list-based implementation with an equivalent implementation that uses the set data-structure.

Once the transformation functions are created, a HERMIT session can be started and the worker/wrapper conversion attempted. If all of the rules are known prior to start then the conversion can be completed in one session. Most likely one will have to exit a session to create rules to continue the process. This is where HERMIT scripts are useful. Commands in HERMIT can be saved as scripts, which is especially useful for replaying sessions after modifying HERMIT.

2.1.2 Transform Process

The order that the transformation occurs is important, one should focus on the functions that have the no dependence on other target functions first. One must also consider the transformed result when deciding since the dependencies may change during the process. Transforming one function before you have transformed another function on which the first depends will only create more work, and most likely a poor transformation result.

With these facts in mind, we begin with the Life class functions for the Board type. Consider first, the `empty` function, its definition is very simple and it doesn't depend on other functions. The result that is desired would instead use an empty set rather than an empty list. The desired function will also not depend on any of the other module functions. This is a great function to start the transformation. From the worker/wrapper method we know that at some point the definition will be `repB (LifeBoard c [])`, which after unfolding repB will be `LifeBoard c (repb [])`. This indicates that the rule called 'repb-null' is necessary. Since we have a transformation rule ready for the `empty` function the process can begin.

The script used to convert the `empty` function is shown here as an example.

```
binding-of '$cempty
fix-intro
down
split-1-beta $cempty [|absxB|] [|repxB|]
{
  rhs-of 'g
  repeat (any-call (unfold ['repxB, 'repB]))
  bash
  any-call (unfold-rule repb-null)
}
let-subst
alpha-let ['$cempty']
{ let-bind ; nonrec-rhs ; unfold ; bash }
top
innermost let-float
```

The first and last few lines of each conversion script are identical, the first few commands focus HERMIT on the proper function then perform the worker/wrapper split. The last few commands move the new function to the top-level of the program so that it becomes a part of the API. Most of these commands are common HERMIT commands that are used to manipulate the Core AST. For brevity, these commands will not be covered in detail. The interesting command that may differ each script is the `split-1-beta` command.

The arguments for this command will vary with each script, here the command is `split-1-beta $cempty [|absxB|] [|repxB|]`. This command performs a worker/wrapper split on the named function (`$cempty`) using the given transformation-pair (`absxB` and `repxB`). This command is what performs the worker/wrapper split, producing the wrapper, which retains the original name `$cempty`, and the worker, which is always named g. The rest of the commands that are unique to the script are shown between the { and } commands. Except the `rhs-of g`, this command is common to all the scripts.

The first step in all the scripts is to unfold the transformation functions. How far they are unfolded depends on the particular function being converted. The `bash` command is commonly used to reduce code to a form more suitable for using GHC rules. The definition of `empty` is not complex and does not require much manipulation. That is why there are few commands present in this script. Typically following the unfolding of the transformers there are a series of Core manipulations that would put the AST in a form that matches a predefined GHC rule. However, for this conversion the `bash` command was sufficient.

The application of a GHC rule is done with the `unfold-rule` command. As seen in this script, only the 'repb-null' rule was needed and applied. After the application of this script the function `empty'` is available through the API. The equivalent definition in Haskell would be `empty' c = LifeBoard c Data.Set.empty`.

Focusing on the `alive` conversion script, the absBx-repBx pair are used to perform the worker/wrapper split, and repBx is the only function unfolded. The original `alive` function is synonymous with the `board` access function. But in our new program `board` will return a `(Set Pos)`, and `alive` still returns a `[Pos]`. So, it is necessary to leave a transform function in the new definition. This is one instance where one does not wish to eliminate all of the transformation functions from the definition. The `toAscList` function will appear by simply unfolding the `absb` function, when it is in the right position. This produces the definition `alive' b = toAscList (board b)`, completing this transform.

The `dims` function, also uses the absBx-repBx pair. And similar to the previous conversion, it only needs to unfold the repBx function. After the application of the 'config-absB' rule to eliminate the transform-function. The function is in the desired form since the `dims` definition should not change. After applying this rule the conversion is complete, and "dims'" will be accessible.

The `diff` function conversion requires the absBBB-repBBB pair to perform the worker/wrapper split. And after unfolding the repBBB function, a few rules are needed to complete the transformation, which requires swapping the use of `Data.List.\\` to `Data.Set.\\` via the 'diff-absb' rule.

The `neighbs` conversion makes use of the absCPB-repCPB pair for its split. The process is similar to the process for `alive`, in that it requires leaving a transformation in place. For this function, the

implementation is left alone and the structure is converted before being returned.

The worker/wrapper split is performed on the next three functions using the absBx-repBx pair. The function `isEmpty` is considered next. Its definition is simple but it is important that this function be converted before `isAlive` because `isEmpty` depends on it. However, when converted it will no longer have this dependency. The resulting definition of `isEmpty` uses the `Data.Set.notMember` function and is created with the 'not-elem-absb' rule.

Once that is complete the process moves to `isAlive`. The process is similar to `isEmpty`'s but uses the 'elem-absb' rule to produce a function that uses the `Data.Set.member` function.

The `liveneighbs` function uses the 'filter-absb' and 'length-absb' rules to complete its conversion which replaces `Prelude.filter` and `Prelude.length` with `Data.Set.filter` and `Data.Set.size` respectively.

The functions `survivors`, `births`, `nextgen`, and `next` all require the use of the absBB-repBB pair. The `survivors` function only needs to swap its `filter` function from the Prelude to Set implementation. But `births` requires this transformation in addition to the use of the 'ncm-absb' rule, which changes the list-based implementation to a set-based one.

The `nextgen` function requires the use of the 'sort-++-absb' rule, which changes concatenation into a set union. While `next` is a special case an only requires changing the function to call `nextgen'` rather than `nextgen`.

The final function in the conversion is `inv`. This conversion requires that `isAlive` be transformed first so that `isAlive'` is accessible. Although, `inv` is not dependent on `isAlive`, `inv'` will depend on `isAlive'`. This function is a little more complex to convert because it has two branches that must be accounted during the transformation. It requires the use of both 'insertion' and 'deletion' rules to replace the code in each branch.

When all the necessary functions have been converted the HERMIT command `continue` can be given to complete compilation of the new program.

In addition to the example above, the final paper will discuss similar transformations from List to QuadTree, and List to Unboxed Vector.

2.2 Example: List-to-Accelerate implementation transform

The Accelerate language is a Haskell embedded DSL that provides arrays and scalars, and a collection of collective operations applied to arrays. These operations are algorithmic skeletons that target CUDA implementations via the Accelerate code generator [2]. For more information about Accelerate and its implementation, consult [2, 9].

2.2.1 Transform Process

For this transformation, we again use the definition of `Board` from our earlier examples, but we choose an interesting type for `Board'`.

```
type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (Acc (Array DIM2 Int))
```

Additionally, the types and implementations of `absb` and `repb` are more involved than the previous ones we have seen.

```
repb :: Size -> [Pos] -> Acc (Array DIM2 Int)
repb (w,h) xs =
    A.reshape (A.index2 (lift w) (lift h))
              (A.scatter to def src)
    where     sz = List.length xs
        to = A.map (\pr -> let (x,y) = unlift pr
                    in (x * (lift w)) + y)
              (A.use $ A.fromList (Z :. sz) xs)
        src = A.fill (A.index1 (lift sz)) 1
        def = A.fill (A.index1 (lift $ w * h)) 0

absb :: Size -> Acc (Array DIM2 Int) -> [Pos]
absb (w,h) arr =
    let prs = A.reshape (A.index1 (lift (w * h)))
            $ A.generate (index2 (lift w) (lift h))
                (\ix -> let Z :. i :. j = unlift ix
                in lift (i :: Exp Int, j :: Exp Int))
        res = A.filter (\pr -> let (i,j) =
            unlift pr :: (Exp Int, Exp Int)
            in (arr A.! (index2 i j)) ==* 1) prs
    in toList $ run res
```

These choices, along with a comparison of this implementation to the other implementations will be discussed in the final paper.

3. Related works

The HERMIT toolkit has experienced success in a wide variety of applications. Some of those applications include applying the worker/wrapper transformation to optimize functions like `reverse` and `last` [3, 11]. Additionally, HERMIT has also been used to mechanize an optimization pass for SYB [1], and to enable stream fusion for `concatMap` [4].

The full paper will contain a detailed literature survey of related works.

4. Conclusion

Earlier work in [3, 11] showed that worker/wrapper can be successfully mechanized in the small. We extended that work to include the transformation of an entire program. In choosing our target application, Game of Life, we wanted an application that was complex enough to require several functions and potentially multiple modules, but simple enough that it could be conceptually understood quickly.

Through the course of our investigation into applying worker/wrapper to the Game of Life, we transformed the original List based version into versions that used Sets, Unboxed Vectors, and Quad Trees. In addition to changing the underlying structure, we also wanted to leverage the GPU. By targeting the Accelerate DSL, we were able to transform the List based version into a version that performed all of the population calculations on the GPU. Ultimately, our experiences in this exploration have shown us that application wide worker/wrapper transformations can in fact be mechanized, and that HERMIT is a valuable tool for doing such a mechanization.

Acknowledgments

We would like the thank Andrew Farmer for help with HERMIT. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

References

[1] M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing syb is easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 71–82, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. . URL `http://doi.acm.org/10.1145/2543728.2543730`.

[2] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[3] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proceedings of*

the ACM SIGPLAN Haskell Symposium, Haskell '12, pages 1–12. ACM, 2012. ISBN 978-1-4503-1574-6. . URL http://doi.acm.org/10.1145/2364506.2364508.

[4] A. Farmer, C. Höner zu Siederdissen, and A. Gill. The hermit in the stream: Fusing stream fusion's concatmap. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 97–108, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. . URL http://doi.acm.org/10.1145/2543728.2543736.

[5] M. Gardner. Mathematical games – the fantastic combinators of john conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[6] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(02):227–251, 2009.

[7] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[8] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[9] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 49–60. ACM, 2013.

[10] N. Sculthorpe and G. Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113–127, 2014. URL http://dx.doi.org/10.1017/S0956796814000045.

[11] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103, 2013. URL http://dx.doi.org/10.1007/978-3-642-41582-1_6.

Selected Issues in Persistent Asynchronous Adaptive Specialization for Generic Array Programming

Clemens Grelck Heinrich Wiesinger

University of Amsterdam
Informatics Institute
Amsterdam, Netherlands
C.Grelck@uva.nl H.M.Wiesinger@student.uva.nl

Abstract

Asynchronous adaptive specialization of rank- and shape-generic code for processing immutable (purely functional) multi-dimensional arrays has proven to be an effective technique to reconcile the desire for abstract specifications with the need to achieve reasonably high performance in sequential as well as in automatically parallelized execution. Since concrete rank and shape information is often not available as a matter of fact until application runtime, we likewise postpone the specialization and in turn aggressive optimization of generic functions until application runtime. As a consequence, we use parallel computing facilities to asynchronously and continuously adapt a running application to the structural properties of the data it operates on.

A key parameter for the efficiency of asynchronous adaptive specialization is the time it takes from requesting a certain specialization until this specialization effectively becomes available within the running application. We recently proposed a persistence layer to effectively reduce the average waiting time for specialized code to virtually nothing In this paper we revisit the proposed approach in greater detail. We identify a number of critical issues that partly have not been foreseen before. Such issues stem among others from the interplay between function specialization and function overloading as well as the concrete organization of the specialization repository in a persistent file system. We describe the solutions we have adopted for the various issues identified.

Categories and Subject Descriptors Software and its engineering [*Software notations and tools*]: Dynamic compilers

Keywords Array processing, Single Assignment C, runtime optimization, dynamic compilation, rank and shape specialization

1. Introduction

SAC (Single Assignment C) is a purely functional, data-parallel array language [4, 6, 7] with a C-like syntax (hence the name). SAC features immutable, homogeneous, multi-dimensional arrays and supports both shape- and rank-generic programming: SAC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions).

In software engineering practice, it is generally desirable to abstract as much as possible from concrete shapes and ranks. This is particularly true for the compositional array programming style advocated by SAC, where in the tradition of APL entire applications are constructed abstraction layer by abstraction layer from basic building blocks, which are by definition rank- and shape-generic as well as application-agnostic.

However, generic array programming comes at a price. In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and again for rank-generic code [18]. There are various reasons for this observation and often their relative importance is operation-specific, but nonetheless we can identify three categories of overhead caused by generic code: First, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, e.g. no static nesting of loops can be generated to implement a rank-generic multidimensional array operation. Second, many of the SAC compiler's advanced optimizations [8, 9] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Third, in automatically parallelized code [1, 3, 5, 13] many organizational decisions must be postponed until runtime, and the ineffectiveness of optimizations inflicts frequent synchronization barriers and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SAC compiler aggressively specializes rank-generic code into shape-generic code and shape-generic code into non-generic code. However, regardless of the effort put into compiler analysis for rank and shape specialization, this approach is fruitless if the necessary information is not available at compile time as a matter of principle. For example, the corresponding data may be read from a file, or the SAC code may be called from external (non-SAC) code, to mention only two potential scenarios.

Such scenarios and the ubiquity of multi-core processor architectures form the motivation for our asynchronous adaptive specialization framework [11, 12]. The idea is to postpone specialization, if necessary, until runtime, when complete structural information on array arguments (rank and shape) is trivially available. Asynchronous with the execution of a generic function, potentially in a data-parallel fashion on multiple cores, a *specialization controller* generates an appropriately specialized binary variant of the same

IFL'14, October 1–3, 2014, Boston, MA, USA.

http://dx.doi.org/10.1145/???

function and dynamically links the additional code into the running application program. Eligible functions are indirectly dispatched such that if the same binary function is called again with arguments of the same shapes as previously, the corresponding new and fast non-generic clone is run instead of the old and slow generic one.

The effectiveness of our approach, in general, depends on making specialized, and thus considerably more efficient, binary variants available to a running application as quickly as possible. This would normally call for fast and light-weight just-in-time compiler, but firstly the SAC compiler is everything but light-weight and rewriting it in a more light-weight style would be a gigantic engineering effort. Secondly, making the compiler faster would inevitably come at the expense of reducing its aggressive optimization capabilities, which obviously is adverse to our overarching goal: highest possible application performance.

In [10] we proposed a total of four different refinements of the original asynchronous adaptive specialization framework:

- bulk asynchronous adaptive specialization,
- prioritized asynchronous adaptive specialization,
- parallel asynchronous adaptive specialization and
- persistent asynchronous adaptive specialization

All four mutually orthogonal techniques aim at reducing the average effective time that it takes for a specialization to become available to the running application once it has been identified as needed.

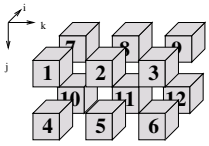
In this paper we focus on the persistence refinement. In the original asynchronous adaptive specialization framework specializations are accumulated during one execution of an application and are automatically removed upon the application's termination. Consequently, any follow-up run of the same application program starts again from scratch as far as specializations are concerned. Of course, the next run may use arrays of different shape, but in many scenarios it is quite likely that a similar set of shapes will prevail as in previous runs. The same holds across different application programs, in particular as any SAC application is heavily based on the foundation of SAC's comprehensive standard library of rank-generic array operations.

With the proposed persistent storage of specialized functions the overhead of actually compiling specializations at application runtime can often be avoided entirely. For many applications the persistent storage of specializations would in practice result in a sort of training phase, after which most required specializations have been compiled. Only in case the user runs an application on a not previously encountered array shape, does the dynamic specialization machinery become active again.

A potential scenario could be image filters. They can be applied to any image pixel format. In practice, however, users only deal with a fairly small number of different image formats. Still, the concrete formats are unknown at compile time of the image processing application. Purchasing a new camera may introduce further image formats to be used. This scenario would result in a short training phase until all image filters have been specialized for the additional image formats of the new camera.

However, persistence requires more radical changes to the dynamic specialization framework than thought at first glance. This paper is about these issues and how to solve them.

The remainder of the paper is organized as follows. In Section 2 we explain SAC in general and the calculus of multi-dimensional arrays in particular. In Section 3 we elaborate on the existing runtime specialization framework in more detail. Through Sections 4–7 we sketch out a number of issues that arise from the desire to make specializations persistent and explain how to solve them. Finally, we draw conclusions in Section 8.



rank: 3
shape: [2,2,3]
data: [1,2,3,4,5,6, 7,8,9,10,11,12]

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

rank: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]

[1, 2, 3, 4, 5, 6]

rank: 1
shape: [6]
data: [1,2,3,4,5,6]

42

rank: 0
shape: [ ]
data: [42]

Figure 1. Truly multidimensional arrays in SAC and their representation by data vector, shape vector and rank scalar

2. SAC and its Multi-Dimensional Arrays

As the name "Single Assignment C" suggests, SAC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This choice is primarily meant to facilitate familiarization for programmers who rather have a background in imperative languages than in declarative languages. Core SAC is a functional, side-effect free subset of C: we interpret assignment sequences as nested let-expressions, branching constructs as conditional expressions and loops as syntactic sugar for tail-end recursive functions. Details on the design of SAC can be found in [4, 7].

Following the example of interpreted array languages, such as APL[2, 14], J[15] and NIAL[16, 17], an array value in SAC is characterized by a triple $(r, \vec{s}, \vec{d})$. The *rank* $r \in \mathbb{N}$ defines the number of dimensions (or axes) of the array. The *shape vector* $\vec{s} \in \mathbb{N}^r$ yields the number of elements along each of the r dimensions. The *data vector* $\vec{d} \in T^{\prod \vec{s}}$ contains the array elements (in row-major unrolling), the so-called *ravel*. Here T denotes the element type of the array. Some relevant invariants ensure the consistency of array values. The rank equals the length of the shape vector while the product of the elements of the shape vector equals the length of the data vector.

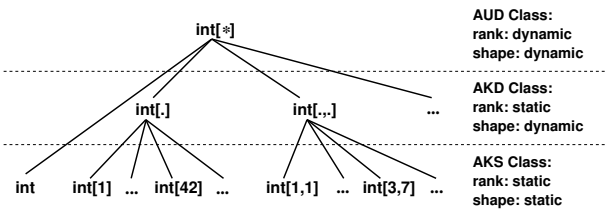


Figure 2. Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

Fig. 1 illustrates the calculus of multi-dimensional arrays that is the foundation of array programming in SAC. The array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Consequently, every value in SAC has rank, shape vector and data vector as structural properties. Both rank and shape vector can be queried by built-in functions. The data

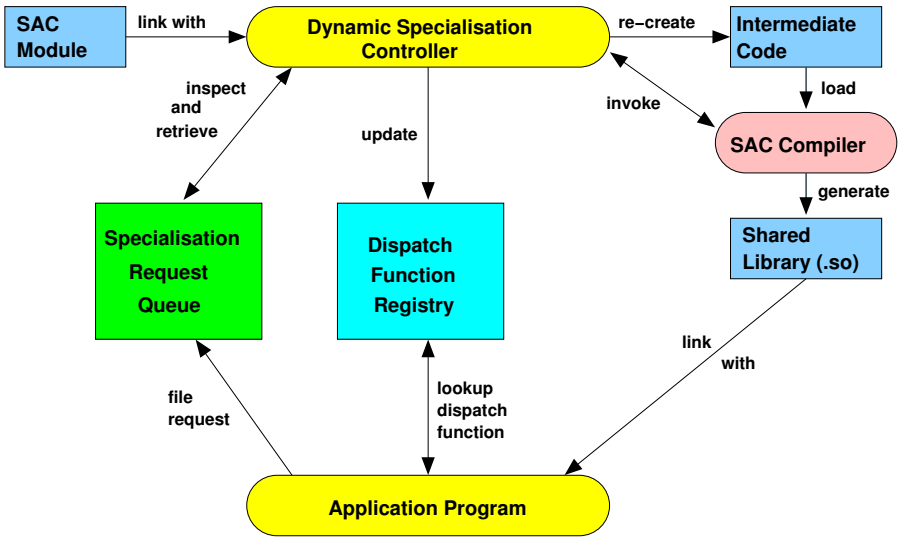


Figure 3. Software architecture of asynchronous adaptive specialization framework

vector can only be accessed element-wise through a selection facility adopting the square bracket notation familiar from other C-like languages. Given the ability to define rank-generic functions, whose argument array's ranks may not be known at compile time, indexing in SAC is done using vectors (of potentially statically unknown length), not (syntactically) fixed sequences of scalars as in most other languages. Characteristic for the calculus of multi-dimensional arrays is a complete separation between data assembled in an array and the structural properties (rank and shape) of the array.

The type system of SAC is monomorphic in the element type of an array, but polymorphic in the structure of arrays. As illustrated in Fig. 2, each element type induces a conceptually unbounded number of array types with varying static structural restrictions on arrays. These array types essentially form a hierarchy with three levels. On the lowest level we find non-generic types that define arrays of fixed shape, e.g. `int[3,7]` or just `int`. On an intermediate level of genericity we find arrays of fixed rank, e.g. `int[.,.]`. And on the top of the hierarchy we find arrays of any rank, and consequently any shape, e.g. `int[*]`. The hierarchy of array types induces a subtype relationship, and SAC supports function overloading with respect to subtyping.

The array type system leads to three different runtime representations of arrays depending on the amount of compile time structural information, as illustrated in Fig. 2. For *AKS arrays* both rank and shape are compile time constants and, thus, only the data vector is carried around at runtime. For *AKD arrays* the rank is a compile time constant, but the shape vector is fully dynamic and, hence, must be maintained alongside the data vector. For *AUD arrays* both shape vector and rank are dynamic and lead to corresponding runtime data structures.

3. Asynchronous Adaptive Specialization

In order to reconcile software engineering principles for generality with user demands for performance we have developed the asynchronous adaptive specialization framework illustrated in Fig. 3. The idea is to postpone specialization if necessary until runtime, when all structural information is eventually available no matter what. A generic SAC function compiled for runtime specialization leads to two functions in binary code: the original generic and presumably slow function definition and a small proxy function that is actually called by other code instead of the generic binary code.

When executed, the proxy function files a specialization request consisting of the name of the function and the concrete shapes of the argument arrays before calling the generic implementation. Of course, proxy functions also check whether the desired specialization has been built before, or whether an identical request is currently pending. In the former case, the proxy function dispatches to the previously specialized code, in the latter case to the generic code, but without filing another request.

Concurrent with the running application, a specialization controller (thread) takes care of specialization requests. It runs the fully-fledged SAC compiler with some hidden command line arguments that describe the function to be specialized and the specialization parameters in a way sufficient for the SAC compiler to re-instantiate the function's partially compiled intermediate code from the corresponding module, compile it with high optimization level and generate a new dynamic library containing the specialized code and a new proxy function. Eventually, the specialization controller links the application with that library and replaces the proxy function in the running application.

The effectiveness of asynchronous adaptive specialization depends on how often the dynamically specialized variant of some function is actually run instead of the original generic version. This depends on two connected but distinguishable properties. Firstly, the application itself must apply an eligible function repeatedly to arguments with the same shape. Secondly, the specialized variant must become available sufficiently quickly to have a relevant impact on application performance. In other words, the application must run considerably longer than the compiler needs to generate binary code for specialized functions.

The first condition relates to a property of the application. Many applications in array processing do expose the desired property, but obviously not all. We can only deal with unsuitable applications by dynamically analyzing an application's properties and by stopping the creation of further specialized functions at some point.

The second condition sets the execution time of application code in relation to the execution time of the compiler. In array programming, however, the former often depends on the size of the arrays being processed, whereas the latter depends on the size and structure of the intermediate code. Obviously, execution time

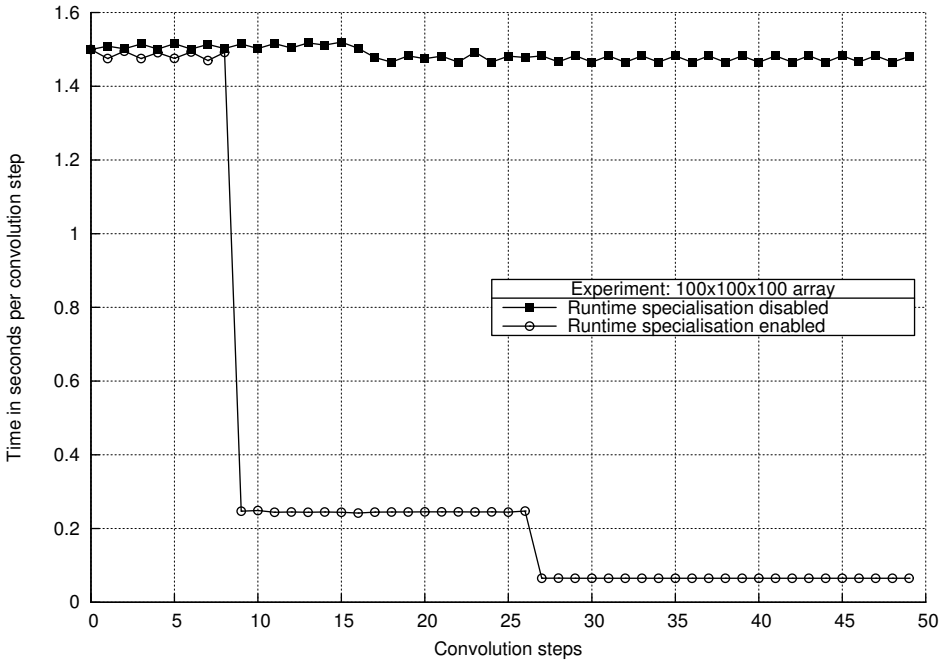


Figure 4. Case study: running a generic convolution kernel on a 3-dimensional argument array of shape $100 \times 100 \times 100$ with and without asynchronous adaptive specialization

and compile time of any code are unrelated with each other and, thus, many scenarios are possible.

In order to demonstrate the possible dynamic behaviour of asynchronous adaptive specialization and its impact on application performance, we show the measurements from one experiment in Fig. 4. The experiment was performed on an AMD Phenom II X4 965 quad-core system. The machine runs at 3.4GHz clock frequency and is equipped with 4GB DDR3 memory; the operating system is Linux with kernel 2.6.38-rc1, but we expect very similar results on different processor architectures.

The experiment is based on a rank-generic convolution kernel with convergence test. In this code two functions are run alternately for a sequence of steps: a convolution step that computes a new array from an existing one and a convergence test that checks whether the old and the new array are sufficiently similar to stop computing. Both functions are defined in rank-generic manner and appropriate measures are put in place to prevent the SAC compiler from statically optimizing either function.

Fig. 4 shows the dynamic behaviour of an application that applies this convolution kernel to a 3-dimensional array of $100 \times 100 \times 100$ double precision floating point numbers. The plot shows individual iterations on the x-axis and measured execution time for each iteration on the y-axis. The two lines show measurements with runtime specialization disabled and enabled, respectively.

With asynchronous adaptive specialization disabled the time it takes to complete one cycle consisting of convolution step and convergence check — as expected — is more or less constant. With asynchronous adaptive specialization enabled, however, we can observe two significant drops in per iteration execution time. After 8 iterations running completely generic binary code a shape-specialized version of the convolution step becomes available. Switching from a generic to a non-generic implementation reduces the execution time per iteration from about 1.5 seconds to roughly 0.25 seconds. After 26 iterations in addition to the specialized convolution step also a specialized convergence check has been compiled and linked into the running application. This reduces the execution time of a single iteration further from 0.25 seconds to 0.065 seconds.

This example demonstrates the tremendous effect that runtime specialization can have on generic array code. The main reason for this considerable performance improvement again is the effectiveness of optimizations that fuse consecutive array operations and, thus, avoid the creation of intermediate arrays. A more detailed explanation of this experiment as well as a number of further experiments can be found in [12] and in [10]. All these experiments unanimously substantiate the relevance of asynchronous adaptive specialization in practice.

4. Issue 1: specialization vs overloading

Our first issue originates from SAC's support for function overloading in conjunction with our desire to share specializations between independent applications. The combination of overloading and specialization raises the question how to correctly dispatch function applications between different function definitions of the same name. In Fig. 5 we show an example of 5 overloaded definitions of the function `foo`. The actual bodies of the function definitions are irrelevant in our context and, thus, we leave them out. Moreover, SAC is currently restricted to be monomorphic on the element type of arrays. Hence, we uniformly use type `int` throughout the example.

From a given set of overloaded function definitions the SAC compiler derives explicit dispatch code that dispatches on parameter types from left to right and for each parameter first on rank and then on type. The type system of SAC ensures that the dispatch is unambiguous [19]. More precisely, if the first parameter type of some function instance is a subtype of the first parameter type of some other overloaded instance of the same function, then the same relationship must hold for all further parameter types, etc.

```
1  int[*] foo( int[*]    a, int[*]   b) {...}
2  int[*] foo( int[.]    a, int[.]   b) {...}
3  int[*] foo( int[7]    a, int[8]   b) {...}
4  int[*] foo( int[.,.] a, int[42] b) {...}
5  int[*] foo( int[2,2] a, int[99] b) {...}
```

Figure 5. Example of shapely function overloading in SAC

Fig. 6 shows an excerpt of the wrapper code derived from the original overloading example.

For the construction of the dispatch tree it is irrelevant whether a some instance of a function definition is original user-supplied code or a compiler-generated specialization. There is, however, a significant semantic difference between the two cases that makes our life difficult as it comes to the proposed persistence layer: when dispatching between compiler-generated specializations of the some original function, it is desirable to dispatch to the most specific instance because that is arguably the most efficient one, but it is, not necessary from a correctness point of view. In contrast, when dispatching between different overloaded instances of some function, the compiler must dispatch any application to the best matching instance, no matter what.

With this in mind the obvious question is how we dispatch function applications in the case of the original asynchronous adaptive specialization framework. In fact, we can exploit an interesting feature of the SAC module system. It allows us to import possibly overloaded instances of some function and to again overload those instances with further instances in the importing module. This feature allows us to incrementally add further instances to a function, and this feature is extremely handy when it comes to implementing runtime specialization.

On every module level that adds further instances a new dispatch (wrapper) function similar to that shown in Fig. 6 is generated that implements the dispatch over all visible instances of a function regardless of where exactly these instances are actually defined. We take advantage of this design for implementing asynchronous adaptive specialization as follows: each time we generate a new specialization at application runtime we effectively construct a new module that imports all existing instances of the to be specialized function and then adds one more specialization to the module, the one matching the current function application. Without further ado the SAC compiler in addition to the new executable function instance also generates a new dispatch wrapper function that dispatches over all previously existing instances plus the one newly generated instance. All we need to do at runtime then is to appropriately replace the previously existing dispatch function by the new one.

At first glance, it seems we could continue with this scheme, and whenever we add a further specialization to the repository of specializations we replace the previous dispatch function by the new one. In other words, we would carry over the concept from a single application run to the set of all application runs in the history of the computing system installation.

Unfortunately, this would be incorrect.

The show-stopper here is the coexistence of semantically equivalent specializations and possibly semantically different overloadings of function instances. One dispatch function in the specialization repository is not good enough because any program (or module) may well contribute further overloadings to whatever function definition is available. This may shadow certain specializations in the repository and at the same time require the generation of new specializations that are semantically different from the ones in the repository, despite sharing the same function name.

```
1  int[*] foo_wrapper(int[*] a, int[*] b)
2  {
3    if (dim(a) == 1) {
4      if (shape(a) == [7]) {
5        if (dim(b) == 1) {
6          if (shape(b) == [8]) {
7            c = foo_3( a, b);
8          }
9          else {
10           c = foo_2( a, b);
11         }
12       }
13       else {
14         c = foo_1( a, b);
15       }
16     }
17     else {
18       if (dim(b) == 1) {
19         c = foo_2( a, b);
20       }
21       else {
22         c = foo_1( a, b);
23       }
24     }
25   }
26   else if (dim(a) == 2) {
27     if (shape(a) == [2,2]) {
28       if (dim(b) == 1) {
29         if (shape(b) == [99]) {
30           c = foo_5( a, b);
31         }
32         else if (shape(b) == [42]) {
33           c = foo_4( a, b);
34         }
35         else {
36           c = foo_1( a, b);
37         }
38       }
39       else {
40         c = foo_1( a, b);
41       }
42     }
43     else {
44       if (shape(b) == [42]) {
45         c = foo_4( a, b);
46       }
47       else {
48         c = foo_1( a, b);
49       }
50     }
51   }
52   else {
53     c = foo_1( a, b);
54   }
55
56   return c;
57 }
```

Figure 6. SAC wrapper function with dispatch code for the five overloaded instances of function `foo` shown in Fig. 5

A simple example illustrates the issue: let us assume a function `foo` with, for simplicity, a single argument of type `int[*]`. Again the element type, here `int`, is irrelevant. Let us further assume that the original definition of `foo` is found in module `A`. Now, some application using module `A` may have created specializations for shapes `[42]`, `[42,42]` and `[42,42,42]`, i.e. for 1-dimensional, 2-dimensional and 3-dimensional arrays of size 42 in each dimension.

In this context we write another module B that imports the original definition of function foo, i.e. the generic one, and adds one more instance: foo(int[.,.]). This new instance of foo may not be semantically equivalent to the generic function imported from module A. Of course, it would be good software engineering practice if both function instances that bear the same name are somewhat related, but firstly this cannot be enforced in any way and secondly there may be a good reason to provide the specific definition of function foo for matrices, although it does not yield the exact (bit-wise) same result as applying the original rank-generic definition to a matrix.

As a consequence of the scenario sketched out above, an application of function foo to a vector of 42 elements in module B could be dispatched to the specialized instance in the repository, same as in module A. However, an application of function foo to a matrix of 42x42 elements in module B must be dispatched to the shape-generic instance defined in module B itself. This should trigger a further runtime specialization during the execution of module B. As a consequence, two different instances of function foo both specialized for 42x42 element matrices materialize in the specialization repository.

This raises questions pertaining to the organization of the specialization repository that we elaborate on in Section 5 while we focus on the dispatch issue for now. From the above scenario it becomes clear that we need a two-level dispatch for the persistence layer. Firstly, we must dispatch within the current application. This can be done with the conventional dispatch wrapper functions as illustrated in Fig. 6. If as the result of this first level dispatch a rank- or shape-generic function instance is selected, we must interfere.

First, we focus our attention on the specialization repository. We must figure out whether or not a suitable specialization already exists. For this purpose module name, function name and the sequence of argument types with full shape information (as is always available at application runtime) suffice to identify the correct instance. If the required specialization does already exist, we can directly link it into the running application and call it. If the required specialization does not yet exist, we file the corresponding specialization request, as described in Section 3. Then we call the generic function instance. Asynchronously, the specialization controller will create an executable specialization of this specific generic function instance and likewise asynchronously will add it to the specialization repository when finished with compilation.

5. Issue 2: file system as specialization data base

So far, we have silently assumed some form of specialization collection or data base that allows us to store and retrieve function specializations in a space and time efficient way. To be more concrete now, we deem the file system to be the best option to serve as this persistent data base.

To avoid issues with write privileges in shared file systems we refrain from sharing specilazations between multiple users. While it would appear attractive to do so in particular for functions from the usually centrally stored SAC standard library from a purely technical perspective, the system administration concerns of running SAC applications in privileged mode can hardly be overcome in practice. Consequently, we store specialized function instances in the user's file system space. A subdirectory .sac2c in the user's home directory appears to be a suitable default location.

Each specialized function instance is stored in a separate dynamic library. In order to store and later retrieve specializations we make reuse of an already existing feature within the SAC compiler: to disambiguate overloaded function instances (and likewise compiler-generated specializations) in compiled code we employ a scheme that constructs a unique function name out of module name, function name and argument type specifications. We use that very same scheme, but replace the original separator token (underscore-underscore) by a slash. As a consequence, we end up with a potentially very complex directory structure that effectively implements a search tree and thus allows us to efficiently locate existing specializations as well as to identify missing specializations.

There is, however, one small pitfall that luckily can be overcome fairly easily. A module name in SAC is not necessarily unique in a file system. Like many other compilers the SAC compiler allows users to specify directory paths to locate modules in the file system. Changing the path specification from one compiler run to the next may effect the semantics of a program. Like with any other compiler, it is the user's responsibility to get it right. For our purpose this merely means that instead of the pure module name we need to use a fully qualified path name to uniquely identify a module definition.

6. Issue 3: semantic revision control

For users who merely run SAC application programs instead of writing their own the techniques described in the preceding two sections would be sufficient. Of course, forbidding users to write their own SAC code when making use of persistent asynchronous adaptive specialization is a fairly undesirable constraint.

So, what is the issue?

Let us go back to the scenario sketched out in Section 4. The user's specialization repository contains four specializations, three specializations of the function foo(int[*]) as defined in module A ([42], [42,42] and [42,42,42]) and one specialization of function foo(int[.,.]) as defined in module B (again [42,42]).

A developing user could now simply come up with the idea to change the implementation of function foo(int[.,.]) in module B and by doing so invalidate certain existing specializations in the repository. To be on the safe side, we must incorporate the entire definition of a rank- or shape-generic function into the identifier of a specialization.

For this purpose we linearize the intermediate code of a generic function instance into textual form and compute a suitable hash when generating a dynamic specialization of this generic instance. This hash is then used as the lowest directory level when storing a new specialization in the file system.

Upon retrieving a specialization from the file system repository a running application again generates a hash of a linearization of the intermediate code of its own generic definition and uses this for generating the path name to look up the existence of a specific specialization needed.

With this non-trivial solution we ensure that we never accidentally run an outdated specialization.

7. Issue 4: specialization repository size control

A rather obvious issue in persistent asynchronous adaptive specialization is the need to control the size of a specialization repository in some suitable way. Otherwise, the scheme as described so far is bound to accumulate more and more specializations over time. With today's typical disk spaces this is not an immediate problem, but of course it will become one over time, no matter what. Requiring the user to manually discard all specializations when running out of disk space is not an attractive solution.

Instead, we ask the user at installation time how much disk space he would like to give SAC for the specialization repository; of course, this could be changed later. Now, the specialization repository becomes a sot of cache memory. As long as the size limit has not been reached, we simply let it grow. When the size limit is reached, we must create space before storing a new specialization. As is common in cache organizations, we expect the least recently used specialization across all modules, functions, etc. to be the least

likely to be used in the future. This is of course just a heuristics, but it has worked reasonably well in hardware caches and in the absence of accurate prediction of the future there is not much we could do to be much smarter. Given the heuristic nature of this approach we can — just as hardware caches do — get away with a reasonable approximation of the least recently used property.

File system time stamps provide all the necessary information for free. Unfortunately, searching for the file with the oldest access time stamp in a reasonably large specialization repository can be unpleasantly time consuming. Of course, this would happen asynchronously to the running application in a specialization controller thread, but notwithstanding it makes sense to think about a smarter scheme.

Our plan is to store a small file containing the access time stamp of the least recently accessed file in the whole directory. Originally, this is the creation time of the first file in some directory (and that of the directory). Adding a new file (or subdirectory) to a directory does not affect this time stamp because that file would have a newer time stamp.

However, if a specialization is loaded from the repository the access time stamp of the corresponding file is updated. If that file is/was the oldest in the repository (i.e. its time stamp coincides with that stored in the special file), the time stamp in the special file will be updated to the now oldest time stamp found in the directory. If so, we go one directory up and check if the special file on that level contains exactly the given time stamp. If so, we must update the information of this directory level. Since directory time stamps do not accurately reflect accesses to subdirectories, we must rely on our own time stamp files. In this case we search for the special file with the oldest time stamp among all subdirectories and copy this file (or rather its contents) into the current directory. We recursively repeat this procedure until we reach the top level of the specialization repository.

If new a specialization is to be stored in an already full specialization repository, we can now efficiently locate the least recently accessed specialization in the whole repository by going top-down from the root of the directory tree always choosing the least recently accessed subdirectory based on the time stamps in the special files. After deleting the least recently used specialization, we recursively go up the directory tree again applying the exact same technique as described above for loading a specialization.

The advantage of the proposed scheme is that its overhead is linear in the depth of the tree, not in the size of the tree as a naive search. The scheme is, nonetheless, not fully accurate as it only recognizes when a specialization is loaded into a running application, not how often that specialization is effectively used in that application. One can think of a refinement that updates the access time stamps above whenever a specialized function instance from the repository is actually executed within an application. It is, however, not a-priori clear that the additional overhead that such a refinement would bring with it on average pays off. We consider this an area of future research to give more substantiated answers to these questions.

8. Conclusions

Asynchronous adaptive specialization is a viable approach to reconcile the desire for generic program specifications in (functional) array programming with the need to achieve competitive runtime performance under limited compile time information about the structural properties (rank and shape) of the arrays involved. This scenario of unavailability of shapely information at compile time is extremely relevant. Beyond potential obfuscation of shape relationships in user code data structures may be read from files or functional array code could be called from less information-rich environments in multi-language applications. Furthermore, the scenario is bound to become reality whenever application programmer and application user are not identical, which rather is the norm than the exception in (professional) software engineering.

In the past we proposed several improvements and extensions to asynchronous adaptive specialization that generally broaden its applicability by making specialized binary code available quicker [10]. One key proposal was to make specializations persistent. Persistent asynchronous adaptive specialization aims at sharing runtime overhead across several runs of the same application or even across multiple independent applications sharing the same core library code (e.g. from the SAC standard library).

In the ideal case required specializations of some function do not need to be generated on demand in a time- and resource-consuming way at all. Instead, following some learning or setup period the vast majority of required specializations have already been generated in preceding runs of the same application or even other applications that share some of the code base, as for example parts of SAC's comprehensive standard library. If so, these pre-generated specializations merely need to be loaded from a specialization repository and linked into the running application. In many situations the proposed persistence layer may effectively reduce the average overhead of asynchronous adaptive specialization to close to nothing.

What appeared to be very attractive but mainly an engineering task at first glance has proven to be fairly tricky in practice. In this paper we identified a number of issues related to correct function dispatch in the presence of specialization and overloading, use of the file system as code data base, revision control in the potential presence of semantically different function definitions and, last not least, control of the specialization repository size. We sketched out our solutions found for each of the four issues.

Currently, we are busy implementing the various proposed solutions. In the near future we expect to run experiments that demonstrate how we can reconcile abstract specifications with high sequential and parallel execution performance, seemingly without observable overhead.

References

[1] M. Diogo and C. Grelck. Heterogenous computing without heterogeneous programming. In K. Hammond and H. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*. Springer, 2013.

[2] A. Falkoff and K. Iverson. The Design of APL. *IBM Journal of Research and Development*, 17(4):324–334, 1973.

[3] C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany, 2001. Logos Verlag, Berlin, 2001.

[4] C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.

[5] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[6] C. Grelck and S.-B. Scholz. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In N. Glew and G. Blelloch, editors, *2nd Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France*, pages 25–33. ACM Press, 2007.

[7] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[8] C. Grelck and S.-B. Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.

[9] C. Grelck and S.-B. Scholz. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[10] C. Grelck and H. Wiesinger. Next generation asynchronous adaptive specialization for data-parallel functional array processing in sac. In R. Plasmeijer, editor, *Implementation and Application of Functional Languages, 25th International Symposium, IFL 2013, Nijmegen, Netherlands, Revised Selected Papers*. ACM, 2014.

[11] C. Grelck, T. van Deurzen, S. Herhut, and S.-B. Scholz. An Adaptive Compilation Framework for Generic Data-Parallel Array Programming. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*. Vienna University of Technology, Vienna, Austria, 2010.

[12] C. Grelck, T. van Deurzen, S. Herhut, and S.-B. Scholz. Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. *Concurrency and Computation: Practice and Experience*, 24(5):499–516, 2012.

[13] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the gpu programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.

[14] International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.

[15] K. Iverson. *Programming in J*. Iverson Software Inc., Toronto, Canada, 1991.

[16] M. Jenkins. Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. *Software Practice and Experience*, 19(2):111–126, 1989.

[17] M. Jenkins and J. Glasgow. A Logical Basis for Nested Array Data Structures. *Computer Languages Journal*, 14(1):35–51, 1989.

[18] D. Kreye. A Compilation Scheme for a Hierarchy of Array Types. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers*, volume 2312 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2002.

[19] S.-B. Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

Abstract machines for higher-order term sharing

Connor Lane Smith
University of Kent
cls204@kent.ac.uk

Abstract

In this paper we take the Krivine machine, a simple abstract machine for weak β-reduction, and augment it with the σ-calculus to unlock strong reduction. We demonstrate that this abstract machine can be used to drive higher-order rewriting, and with some alterations can be used for 'higher-order term sharing': rather than normalising a term at each rewrite step, we view the lambda calculus itself as a sophisticated sharing mechanism, and make use of it so as to avoid needless duplication of rewrite steps.

1. Introduction

Higher-order term rewriting [9] is a powerful generalisation of first-order term rewriting in which rewrite steps are performed modulo the simply-typed λ-calculus. Terms are generally assumed to be normalised after each rewrite step, but doing so loses the sharing of subterms present in the unreduced term. For example, the normalisation $(\lambda A\underline{00})B \to_\beta ABB$ *unshares* B from one instance to two, meaning if we wish to rewrite B to C then we must now do it twice instead of only once.

In this paper we explore two known mechanisms for reducing λ-terms: the Krivine machine [6], a simple abstract machine for weak β-reduction; and the σ-calculus [1], an explicit substitution calculus. We then introduce a new 'Kσ-machine' combining these two approaches into an abstract machine for strong β-reduction.

By further extending this machine, and tracing the provenance of the subterms forming a rewrite redex in a term's normal form, we can compute a reduction that reduces only the part of the term needed to reveal that redex. This allows us to maintain during reduction the sharing present in λ-terms, rather than β-normalising rewritten terms as is generally done. This sharing scheme parallels Wadsworth's [11] 'first-order' mechanism for sharing using dags. Comparatively, we make steps towards 'higher-order term sharing'.

2. Preliminaries

We will assume familiarity with the simply-typed λ-calculus [2], and will avoid the complexities of named variables by using De Bruijn indices [3] exclusively.

Definition 1. The set *simple types* is the closure of a fixed set of *type atoms* under the binary function type constructor $\to$. Notation: $\to$ is right-associative: $\alpha \to \beta \to \gamma = \alpha \to (\beta \to \gamma)$.

Definition 2. We use ϵ for the empty list, and :: for the 'cons' operator — i.e. datatype $\text{List}(\alpha) = \epsilon \mid \alpha :: \text{List}(\alpha)$. We write $\alpha \cdot \beta$ for the concatenation of α and β.

Definition 3. A *basis* Γ is a list of simple types, with which we may derive a simply-typed *term* $t \in \mathbb{T}$, written $\Gamma \vdash t : \tau$.

$$\begin{aligned} \sigma :: \Gamma \vdash \underline{0} : \sigma \\ \Gamma \vdash \underline{n} : \sigma \implies \rho :: \Gamma \vdash \underline{n{+}1} : \sigma \\ \sigma :: \Gamma \vdash t : \tau \implies \Gamma \vdash \lambda t : \sigma \to \tau \\ \Gamma \vdash t_1 : \sigma \to \tau \wedge \Gamma \vdash t_2 : \sigma \implies \Gamma \vdash t_1 t_2 : \tau \\ K : \tau \implies \Gamma \vdash K : \tau \end{aligned}$$

Definition 4. A *context* C is a λ-term containing a single unique 'hole' symbol $\Box$. The hole in C may be 'filled' by a term t, written $C[t]$, replacing the hole with the term t with no variable adjustment. Contexts may be composed, such that $(C_1\,;C_2)[t] \equiv C_2[C_1[t]]$.

Definition 5. A *substitution* θ is a mapping from variables to terms of the same type, which may be lifted to a homomorphism over terms, written $\hat{\theta}(t)$.

Definition 6. We write β-reduction as $\to_\beta$, η-reduction $\to_\eta$, and their union, γ-reduction, $\to_\gamma = \to_\beta \cup \to_\eta$. All are closed under contexts and substitutions. There are a number of subrelations of β-reduction:

- Weak reduction – as β-reduction, but reductions cannot be performed under a lambda. Full β-reduction may in contrast be called 'strong reduction'.
- Head reduction – β-reduction of only the leftmost subterm, such that the *head* — the leftmost atom (constant or variable) applied to a 'spine' of argument terms — is found.
- Weak head reduction – head reduction that is also weak, such that the *weak head* of a term may alternatively be a lambda that does not form a β-redex.

A normal form — be it weak, strong, head, or weak head — is a term that cannot be reduced by the respective relation. Any simply-typed term has a unique normal form for each of these subrelations.

3. Explicit reduction

3.1 K-machine

The *Krivine machine* [6], or 'K-machine', is an abstract machine for the weak β-reduction of λ-terms. The machine $\langle\langle t, e\rangle \mid \mathit{stack}\rangle$ has three components: the *term* t being reduced; the *environment* e, a stack of term–environment pairs with the topmost corresponding to De Bruijn index 0 and the bottom to index n; and the *stack*, likewise a stack of term–environment pairs, but from which items are taken as arguments to applied lambdas. The machine is therefore of

the following type, where the μ operator yields the least fixpoint of a type.

$$(\mu\alpha.\,\mathbb{T} \times \text{List}(\alpha)) \times \text{List}(\mu\alpha.\,\mathbb{T} \times \text{List}(\alpha))$$

The mechanics of the K-machine are defined in terms of a set of *transition rules*, detailed in Figure 1. The machine simulates weak β-reduction, in that, given two machine configurations m and m' and terms t and t', if $m \sim t$ and $m' \sim t'$, then $m \to^* m' \implies t \to^*_\beta t'$, for the relation $\sim$:

$$\langle\langle t_0, e_0\rangle \mid \langle t_1, e_1\rangle :: \ldots :: \langle t_n, e_n\rangle :: \epsilon\rangle \sim \hat{e}_0(t_0)\hat{e}_1(t_1)\ldots\hat{e}_n(t_n)$$

When reducing a term t, we run the machine $\langle\langle t, \epsilon\rangle \mid \epsilon\rangle$ until it halts. There are two possible configurations in which the machine halts: $\langle\langle \underline{n}, \epsilon\rangle \mid \mathit{stack}\rangle$ and $\langle\langle K, e\rangle \mid \mathit{stack}\rangle$. In either case, $t_0 \equiv \hat{e}_0(t_0)\downarrow_\beta$, so the weak head normal form has been found. We can continue on to full weak normal form (i.e. all redexes not under a lambda have been reduced) by recursing over the terms and environments in the stack of the halted machine.

The reason the K-machine cannot perform strong β-reduction is there is no way to represent in its stack the offset in De Bruijn indices that would be required of the environment if the machine were to move under a lambda. For this we need a more sophisticated data structure, and for that we look towards explicit substitution.

3.2 $\lambda\sigma$-calculus

The *$\lambda\sigma$-calculus* [1] is a 'substitution calculus' that renders the higher-order λ-calculus into a first-order term rewriting system, thus formalising the mechanisms of substitution. σ-substitutions may be thought of as a data structure for representing arbitrary λ-calculus substitutions on De Bruijn indices. σ-substitutions have the following constructors:

$$\begin{aligned}
\text{id} &= \{n \mapsto n\} && \text{Identity}\\
\uparrow &= \{n \mapsto n{+}1\} && \text{Shift}\\
t \cdot \sigma &= \{0 \mapsto t, n{+}1 \mapsto \sigma(n)\} && \text{Cons}\\
\rho\,;\sigma &= \{n \mapsto \rho(n)[\sigma]\} && \text{Compose}
\end{aligned}$$

Note that I use the notation $\rho\,;\sigma$ where Abadi, et al. have used $\rho \circ \sigma$, so as to avoid any confusion between left-to-right and right-to-left composition. I also use the De Bruijn indices over the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ rather than $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$. Neither of these notational changes have any effect on the calculus itself.

Definition 7. A σ-substitution σ is applied to a λ-term t, written $t[\sigma]$, like so:

$$\begin{aligned}
(t_1 t_2)[\sigma] &= t_1[\sigma]t_2[\sigma]\\
(\lambda t)[\sigma] &= \lambda(t[\underline{0} \cdot (\sigma\,;\uparrow)])\\
\underline{n}[\text{id}] &= \underline{n}\\
\underline{n}[\uparrow] &= \underline{n{+}1}\\
\underline{0}[t \cdot \sigma] &= t\\
\underline{n{+}1}[t \cdot \sigma] &= \underline{n}[\sigma]\\
\underline{n}[\rho\,;\sigma] &= \underline{n}[\rho][\sigma]\\
K[\sigma] &= K
\end{aligned}$$

β-reduction is defined as the relation $(\lambda s)t \to_\beta s[t \cdot \text{id}]$ closed under contexts and substitution.

Definition 8. We extend simply-typed λ-terms to the $\lambda\sigma$-calculus by introducing a typing for σ-substitutions. We write $\Gamma \vdash \sigma \triangleright \Gamma'$ to say that in the environment Γ the substitution σ has the environment Γ', by the following rules:

$$\begin{aligned}
\Gamma &\vdash \text{id} \triangleright \Gamma\\
\tau :: \Gamma &\vdash \uparrow \triangleright \Gamma\\
\Gamma \vdash t : \tau \wedge \Gamma \vdash \sigma \triangleright \Gamma' &\implies \Gamma \vdash t \cdot \sigma \triangleright \tau :: \Gamma'\\
\Gamma \vdash \rho \triangleright \Gamma' \wedge \Gamma' \vdash \sigma \triangleright \Gamma'' &\implies \Gamma \vdash \rho\,;\sigma \triangleright \Gamma''
\end{aligned}$$

We can then use this to derive simple types for σ-closures.

$$\Gamma \vdash \sigma \triangleright \Gamma' \wedge \Gamma' \vdash t : \tau \implies \Gamma \vdash t[\sigma] : \tau$$

3.3 Kσ-machine

We will now introduce the *Kσ-machine*, an extension of the K-machine in which the environment stack has been generalised to a σ-substitution. This unlocks reduction under lambda, as necessary for strong β-reduction. The Kσ-machine is similar to the machine described in [1], but the terms themselves are not modified in any way, much like the original Krivine machine. With this approach the structures of terms and substitutions are kept entirely separate, which means that the type of the machine's environment and stack is independent from that of the term's closures, if any. This will prove important in §4.2, where the machine's thunks' closures are labelled, but the terms' are not.

In order to keep the machine definition simple, we assume that σ-substitutions are always of the pattern $(\sigma_1\,;(\sigma_2\,;\ldots\,;(\sigma_n\,;\text{id})))$. The initial substitution (id) satisfies this pattern, and the rules of the machine maintain it. Additionally, we parametrise the substitution type, Subst(α), so that instead of 'cons' being of the type $\mathbb{T} \to \text{Subst} \to \text{Subst}$, it is of the type $\alpha \to \text{Subst}(\alpha) \to \text{Subst}(\alpha)$. These substitutions then form maps of the type $\mathbb{N} \to \alpha + \mathbb{N}$. In the case of the K$\sigma$-machine, they are of the type $\mu\alpha.\,\text{Subst}(\mathbb{T} \times \alpha)$. We call these term–substitution pairs *thunks*.

These σ-substitutions alone do not quite give us full strong reduction, however. Analogous to the K-machine halting at weak head normal form, we would expect the Kσ-machine to halt at (strong) head normal form $\lambda\ldots\lambda t_0(t_1[\sigma_1])\ldots(t_n[\sigma_n])$, where t_0 is a variable or constant. But if we are to pass under a lambda, we must discard it from the term, and we cannot know how many lambdas are in the head normal form. The solution used in [1] is to suspend the machine at this point, and to handle the 'Lambda' case external to the machine definition. We take a different tack: as well as generalising environments to substitutions, we generalise the stack to a context. This deviation will prove useful in §4.3, where it drastically simplifies the operation of the machine.

Definition 9. A *zipper* [5] is a term representation of a context, or a suspended traversal through a term. A zipper for the λ-calculus is a first-order term with the following signature:

$$\begin{aligned}
@_l^\alpha &: \text{Context}(\alpha) \times \alpha \to \text{Context}(\alpha)\\
@_r^\alpha &: \alpha \times \text{Context}(\alpha) \to \text{Context}(\alpha)\\
\Lambda^\alpha &: \text{Context}(\alpha) \to \text{Context}(\alpha)\\
\top^\alpha &: \text{Context}(\alpha)
\end{aligned}$$

We omit the superscript type parameter where it may be inferred. Each symbol has a meaning as a context, and may be thought of as a traversal through a term.

- $@_l(C, t)$ represents the context $(\Box t\,; C)$.
- $@_r(t, C)$ represents the context $(t\Box\,; C)$.
- $\Lambda(C)$ represents the context $(\lambda\Box\,; C)$.
- $\top$ represents the trivial context $\Box$.

Example 1. *A context $(\lambda\Box)K$ is represented by the zipper term $\Lambda(@_l(\top, K))$, and $(\lambda\underline{0})\Box$ the zipper term $@_r(\lambda\underline{0}, \top)$.*

Where the K-machine has $\langle t, e\rangle :: \mathit{stack}$, the K$\sigma$-machine has $\Box(t[\sigma])\,; C$, i.e. $@_l(C, t[\sigma])$. Yet we can also add lambdas through

Figure 1. K-machine

$$\langle\langle t_1 t_2, e\rangle \mid \mathit{stack}\rangle \to \langle\langle t_1, e\rangle \mid \langle t_2, e\rangle :: \mathit{stack}\rangle \quad \text{Apply}$$
$$\langle\langle \lambda t_1, e_1\rangle \mid \langle t_2, e_2\rangle :: \mathit{stack}\rangle \to \langle\langle t_1, \langle t_2, e_2\rangle :: e_1\rangle \mid \mathit{stack}\rangle \quad \text{Beta}$$
$$\langle\langle \underline{0}, \langle t, e_2\rangle :: e_1\rangle \mid \mathit{stack}\rangle \to \langle\langle t, e_2\rangle \mid \mathit{stack}\rangle \quad \text{Head}$$
$$\langle\langle \underline{n{+}1}, \langle t, e_2\rangle :: e_1\rangle \mid \mathit{stack}\rangle \to \langle\langle \underline{n}, e_1\rangle \mid \mathit{stack}\rangle \quad \text{Tail}$$

Figure 2. Kσ-machine

$$\langle (t_1 t_2)[\sigma] \mid C\rangle \to \langle t_1[\sigma] \mid \Box(t_2[\sigma])\,;C\rangle \quad \text{Left}$$
$$\langle (\lambda t)[\sigma] \mid \Box(u[\rho])\,;C\rangle \to \langle t[(u[\rho]\cdot\sigma)\,;\mathrm{id}] \mid C\rangle \quad \text{Beta}$$
$$\langle (\lambda t)[\sigma] \mid C\rangle \to \langle t[(\underline{0}[\mathrm{id}]\cdot(\sigma\,;\uparrow))\,;\mathrm{id}] \mid \lambda\Box\,;C\rangle \quad \text{Lambda}$$
$$\langle \underline{n}[(\pi\,;\rho)\,;\sigma] \mid C\rangle \to \langle \underline{n}[\pi\,;(\rho\,;\sigma)] \mid C\rangle \quad \text{Associate}$$
$$\langle \underline{0}[(u[\pi]\cdot\rho)\,;\sigma] \mid C\rangle \to \langle u[\pi\,;\sigma] \mid C\rangle \quad \text{Head}$$
$$\langle \underline{n{+}1}[(u[\pi]\cdot\rho)\,;\sigma] \mid C\rangle \to \langle \underline{n}[\rho\,;\sigma] \mid C\rangle \quad \text{Tail}$$
$$\langle \underline{n}[\uparrow\,;\sigma] \mid C\rangle \to \langle \underline{n{+}1}[\sigma] \mid C\rangle \quad \text{Shift}$$
$$\langle \underline{n}[\mathrm{id}\,;\sigma] \mid C\rangle \to \langle \underline{n}[\sigma] \mid C\rangle \quad \text{Id}$$

$$\langle (t_1 t_2)[\sigma] \mid C\rangle \to \langle t_2[\sigma] \mid (t_1[\sigma])\Box\,;C\rangle \quad \text{Right}$$
$$\langle (t[\rho])[\sigma] \mid C\rangle \to \langle t[\rho\,;\sigma] \mid C\rangle \quad \text{Closure}$$

which we have passed with $\lambda\Box\,;C$, i.e. $\Lambda(C)$. The Kσ-machine is therefore of the type,

$$(\mu\alpha.\,\mathbb{T} \times \mathrm{Subst}(\alpha)) \times \mathrm{Context}(\mu\alpha.\,\mathbb{T} \times \mathrm{Subst}(\alpha))$$

The Kσ-machine is defined in Figure 2. The machine definition has overlapping rules with which it is capable of any *standard reduction* [2], but during normal operation (β-normalisation) we assume that the 'Beta' rule is favoured over 'Lambda', and that the 'Left' rule is used instead of the optional 'Right'. There is also a 'Closure' rule for composing two substitutions into one; this is only necessary if the term structure itself may contain closures of the $\lambda\sigma$-calculus. Note that the machine's 'thunks' are separate from the term structure itself.

When reducing a term t, we run the machine $\langle t[\mathrm{id}] \mid \Box\rangle$ until it halts. Similar to the K-machine, the Kσ-machine simulates strong β-reduction;

$$\langle t[\sigma] \mid C\rangle \sim C[t[\sigma]]$$

4. Higher-order term sharing

Here we use Nipkow's Higher-order Rewrite Systems (HRSs) [9], or strictly speaking *higher-order pattern rewrite systems*, to which they are most commonly restricted.

Definition 10. A *higher-order pattern* [8] is a β-normal term with a constant at its head, and in which any free variable f may only in the form $f t_1 \ldots t_k$ where each t_i is η-equivalent to a distinct bound variable.

Definition 11. A Higher-order Rewrite System $\mathcal{H}$ is a set of *rewrite rules* $\langle l, r\rangle$ where l is a pattern and r a term of the same atomic type, and $\mathrm{FV}(l) \supseteq \mathrm{FV}(r)$. Each rule $\mathcal{R}$ induces a rewrite relation $t \to_{\mathcal{R}} t'$ where $\hat{\theta}(l) \leftrightarrow_\gamma^* t$ and $t' \leftrightarrow_\gamma^* \hat{\theta}(r)$. $\mathcal{H}$ then induces the union $\to_{\mathcal{H}} = \bigcup_{\mathcal{R}\in\mathcal{H}} \to_{\mathcal{R}}$.

Although HRSs rewrite modulo the simply-typed λ-calculus, it is generally assumed that the term is β-normalised after each rewrite step. But if it is not, then the β-reduction potential of the term acts as a kind of term sharing mechanism. For instance, a first-order β-redex $(\lambda t_1)t_2$ 'shares' the term t_2 amongst all instances of the variable $\underline{0}$ in t_1; a rewrite step may take place in t_2 and it will in effect have occurred in any number of positions in t_1 in the term's β-normal form.

Example 2. *Given a rewrite system $\{\langle B, C\rangle\}$, a term $(\lambda A\underline{00})B$ can be written in one step to $(\lambda A\underline{00})C$, the β-normal form of which is ACC. But if we β-normalise the term first to ABB, it takes* two *steps, via either ABC or ACB, to reach ACC. This demonstrates that B is being* shared *by the β-redex in the initial term.*

The sharing arrangement in Example 2 can also be achieved by representing a term as a dag, such that after the β-reduction of a term $(\lambda t_1)t_2$ all residual instances of t_2 in t_1 are references to the same term, which can then be rewritten. This form of sharing was formalised by Wadsworth [11]. However, the sharing offered by the simply-typed λ-calculus as a whole, higher-order β-redexes included, is in general more powerful than Wadsworth's dags. With dags, only full terms may be shared; if two terms are almost identical, but one has one term where the other has another, then they cannot be shared despite their similarities.

Example 3. *Given a term $(\lambda C(\underline{0}A)(\underline{0}B))(\lambda t)$, reducing the root β-redex would yield $C((\lambda t)A)((\lambda t)B)$; with dag sharing the two instances of λt would be shared. But further reduction to Ct_1t_2 would require t_1 and t_2 to become unshared, as the former contains A where the latter does B. That the simply-typed λ-calculus* can *share t_1 and t_2, in the single instance of t, demonstrates that it is in general more powerful than dag sharing.*

The question, then, is how to perform higher-order rewriting without unsharing unnecessarily. This is related to Lévy's *optimal reduction* [7], but is not the same: we are using the $\lambda(\sigma)$-calculus as the sharing mechanism, not as the term rewriting system being shared. The notion that rewriting between non-normalised terms

in a higher-order rewriting system can be seen as a kind of term sharing is mentioned in [10].

4.1 β-traversals

Since β-reduction introduces no term structure (its right-hand side as an HRS comprising only variables and applications), given a reduction $t \to_\beta^* t'$, every atom (variable or constant) present in t' will have originated somewhere in t, and through the process of reduction a copy will have been placed in its new position in t'. During this process, its arguments and the value to which itself is an argument may both have changed in any number of ways. If we are to understand how the atom arrived at its position in t', we would like to keep a track of the context of reductions and substitutions by which it got there.

Definition 12. A *β-traversal* may be thought of as a path through the β-reduction of a term. A β-traversal is a first-order term with the following signature:

$$\begin{aligned} @_l &: A \to A & \top &: A \\ @_r &: A \to A & \mathrm{B} &: A \to A \\ \Lambda &: A \to A & \Sigma &: A \times A \to A \end{aligned}$$

β-traversals are akin to the zipper, but the $@_l$ and $@_r$ symbols have as a subterm only a continuation of the context and not an extra α value as with $@_l^\alpha$ and $@_r^\alpha$. Furthermore, substitutions (Σ) may cause there to be more than one top ($\top$). In addition to the symbols in the signature of the zipper,

- B indicates a β-reduction having occurred in its subterm, which is itself somewhat 'inside-out', the lambda occurring outside the application with which it formed a β-redex.
- Σ indicates a substitution having occurred as a result of some β-reduction. It has two subterms, the first the traversal to the variable, the second to the substitute.

Example 4. *The β-traversal to the head of the reduct of a reduction $(\lambda\underline{0})K \to_\beta^* K$ is $\Sigma(\mathrm{B}(\Lambda(@_l(\top))), @_r(\top))$. The β-traversal $\Lambda(@_l(\top))$ represents the traversal down to the body of the lambda on the left-hand side of the application at the top, i.e. $(\lambda\Box)K$, although unlike Example 1 the right-hand side of that application is not made explicit: $(\lambda\Box)\_$.*

The B *symbol then shows that there has been a β-reduction between the application and the lambda, similar to a proof term, e.g. $\beta((\lambda\Box)\_)$. Finally, $\Sigma(\alpha, \beta)$ substitutes for the variable at traversal α the value at β, in this case effectively $\beta((\lambda\Box)K) \leftarrow_\Sigma (\lambda\underline{0})\Box$. The complete traversal term thus describes the position of the head 'through' the reduction.*

Definition 13. A β-traversal over $t \to_\beta^* t'$ may be converted into a *path*, a list of symbols identifying a subterm. Two paths of a traversal are particularly distinct: the *dynamic path*, the destination of the traversal relative to t'; and the *static path*, that relative to t. The dynamic path for a traversal α may be obtained by $path_1(\alpha, \epsilon)$, the static path $path_2(\alpha, \epsilon)$:

$$\begin{aligned} path_i(\top, p) &= p \\ path_i(@_l(\alpha), p) &= path_i(\alpha, @_l :: p) \\ path_i(@_r(\alpha), p) &= path_i(\alpha, @_r :: p) \\ path_i(\Lambda(\alpha), p) &= path_i(\alpha, \Lambda :: p) \\ path_i(\mathrm{B}(\alpha), p) &= path_i(\alpha, p) \\ path_1(\Sigma(\alpha, \beta), p) &= path_1(\alpha, p) \\ path_2(\Sigma(\alpha, \beta), p) &= path_2(\beta, p) \end{aligned}$$

$t|_p$ represents the subterm of the term t reached via path p.

$$\begin{aligned} t|_\epsilon &= t \\ (t_1 t_2)|_{@_l :: p} &= t_1|_p \\ (t_1 t_2)|_{@_r :: p} &= t_2|_p \\ (\lambda t)|_{\Lambda :: p} &= t|_p \end{aligned}$$

We can also refer to a subterm reached *modulo β-reduction*, $t|_{p/\beta}$, which is the same as $t|_p$ except that β-redexes along the path are reduced. This means that $t \to_\beta^* t{\downarrow_\beta}[u]_p \iff t \to_\beta^* t[u]_{p/\beta}$. Also note that $t|_{p\cdot q} \equiv t|_p|_q$.

In order for a pattern to match in a reduct of t we require that the reductions necessary to assemble its (strict) subterms in the right positions have all been performed. However, not all reductions that can be done need be done, and if we identify the *horizon* of the match — the outermost point at which a substitution contributes to the atoms comprising the match — then we can ignore all reductions beyond that point.

Definition 14. Two paths p and q are *compatible* if there exists a third path r such that both p and q are prefixes of r. The *horizon* of a β-traversal α is the supremum of the lengths of the paths compatible with all elements of the set $reach(\alpha)$:

$$\begin{aligned} reach(\top) &= \{path_1(\top, \epsilon)\} \\ reach(@_l(\alpha)) &= \{path_1(@_l(\alpha), \epsilon)\} \cup reach(\alpha) \\ reach(@_r(\alpha)) &= \{path_1(@_r(\alpha), \epsilon)\} \\ reach(\Lambda(\alpha)) &= \{path_1(\Lambda(\alpha), \epsilon)\} \cup reach(\alpha) \\ reach(\mathrm{B}(\alpha)) &= reach(\alpha) \\ reach(\Sigma(\alpha, \beta)) &= reach(\alpha) \cup reach(\beta) \end{aligned}$$

If all paths in the set are compatible with one another then the set and the length will both be infinite. Thus the horizon is a member of the set of "tropical natural numbers" $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$.

The horizon for the match $t[\hat{\theta}(u)]_{p/\beta}$ of a pattern u is then the minimal horizon of all β-traversals corresponding to non-free-variable atoms in u, treating the head of the match as having the horizon ∞ (since it does not need to be in any particular position for the pattern to match).

Theorem 1. *If the β-traversal α over $t \to_\beta^* t{\downarrow_\beta}[\hat{\theta}(u)]_{p\cdot q}$ has the horizon $|p|$, and $\bar{p}$ is the static path $path_2(\alpha)$, then,*

$$t \to_\beta^* t[\hat{\theta}(u)]_{(p\cdot q)/\beta} \implies t \to_\beta^* t[\hat{\theta}'(u)]_{\bar{p}\cdot(q/\beta)}$$

With this in mind, given a rewrite rule $\mathcal{R} = \langle l, r\rangle$, we see that,

$$\begin{aligned} & t[\hat{\theta}(l)]_{(p\cdot q)/\beta} \to_\mathcal{R} t[\hat{\theta}(r)]_{(p\cdot q)/\beta} \\ \implies & t[\hat{\theta}'(l)]_{\bar{p}\cdot(q/\beta)} \to_\mathcal{R} t[\hat{\theta}'(r)]_{\bar{p}\cdot(q/\beta)} \end{aligned}$$

What is needed is a process by which to calculate the values of $\bar{p}$ and q for a given β-traversal, requiring an alteration to the Kσ-machine. We then need to be able to, given these values, reveal the match so that it may be rewritten; this requires one final alteration to the machine.

4.2 K$\sigma^\ell_\Uparrow$-machine

A variant of the Kσ-machine is the K$\sigma^\ell_\Uparrow$-machine, in which each thunk is labelled with (an algebraic interpretation of) its β-traversal, which is computed as the machine runs. As it stands, introducing labels into the Kσ-machine reveals a potential problem in the σ-calculus: when a substitution passes under a lambda, its variable is not left untouched, but is rather replaced with a new variable constructed from whole cloth and given the appropriate De Bruijn index. Variables in the $\lambda\sigma$-calculus are nothing but their index, so no troubles arise, but if this is not the case — such as in our new

Figure 3. K$\sigma^{\ell}_{\Uparrow}$-machine

$$\langle (t_1 t_2)[\sigma]^{\alpha} \mid C\rangle \to \langle t_1[\sigma]^{@_l(\alpha)} \mid \Box(t_2[\sigma]^{@_r(\alpha)})\,;C\rangle \quad \text{Left}$$
$$\langle (\lambda t)[\sigma]^{\alpha} \mid \Box(u[\rho]^{\beta})\,;C\rangle \to \langle t[(u[\rho]^{\beta}\cdot\sigma)\,;\mathrm{id}]^{\mathrm{B}(\Lambda(\alpha))} \mid C\rangle \quad \text{Beta}$$
$$\langle (\lambda t)[\sigma]^{\alpha} \mid C\rangle \to \langle t[\Uparrow(\sigma)\,;\mathrm{id}]^{\Lambda(\alpha)} \mid \lambda\Box\,;C\rangle \quad \text{Lambda}$$
$$\langle \underline{n}[(\pi\,;\rho)\,;\sigma]^{\alpha} \mid C\rangle \to \langle \underline{n}[\pi\,;(\rho\,;\sigma)]^{\alpha} \mid C\rangle \quad \text{Associate}$$
$$\langle \underline{0}[(u[\pi]^{\beta}\cdot\rho)\,;\sigma]^{\alpha} \mid C\rangle \to \langle u[\pi\,;\sigma]^{\Sigma(\alpha,\beta)} \mid C\rangle \quad \text{Head}$$
$$\langle \underline{n{+}1}[(u[\pi]^{\beta}\cdot\rho)\,;\sigma]^{\alpha} \mid C\rangle \to \langle \underline{n}[\rho\,;\sigma]^{\alpha} \mid C\rangle \quad \text{Tail}$$
$$\langle \underline{0}[\Uparrow(\rho)\,;\sigma]^{\alpha} \mid C\rangle \to \langle \underline{0}[\sigma]^{\alpha} \mid C\rangle \quad \text{Naught}$$
$$\langle \underline{n{+}1}[\Uparrow(\rho)\,;\sigma]^{\alpha} \mid C\rangle \to \langle \underline{n}[(\rho\,;\uparrow)\,;\sigma]^{\alpha} \mid C\rangle \quad \text{Lift}$$
$$\langle \underline{n}[\uparrow\,;\sigma]^{\alpha} \mid C\rangle \to \langle \underline{n{+}1}[\sigma]^{\alpha} \mid C\rangle \quad \text{Shift}$$
$$\langle \underline{n}[\mathrm{id}\,;\sigma]^{\alpha} \mid C\rangle \to \langle \underline{n}[\sigma]^{\alpha} \mid C\rangle \quad \text{Id}$$

machine, where a closure has a label — then a variable with one label is replaced with one with another label, which is unsound. In order to fix this, we will introduce the *lift* operator from the $\lambda\sigma_{\Uparrow}$-calculus of Hardin, et al. [4]. The $\sigma_{\Uparrow}$-substitution $\Uparrow(\sigma)$ is equivalent to the σ-substitution $\underline{0}\cdot(\sigma\,;\uparrow)$, except that it allows us to genuinely not substitute the variable, and so its label is also left unchanged.

The K$\sigma^{\ell}_{\Uparrow}$-machine is defined in Figure 3. If the term structure may contain $\sigma_{\Uparrow}$-closures, those closures' substitutions are unlabelled and have to be mapped into the labelled term space; they cannot just be composed with a labelled substitution. This process is fairly straightforward, mirroring the standard traversal over the term structure, but it is tedious so we shall not discuss it here.

Definition 15. A *cord* is a structure of type $\mathcal{P} \times \mathrm{List}(\mathcal{P}) \times \mathbb{N}^{\infty}$. It acts as a map from steps along the path $p \cdot q$ of a β-traversal, to their static path counterparts, together with the traversal's horizon. The cord may be *split* into a pair of paths corresponding to $\bar{p}$ and q.

$$\begin{aligned}
\mathit{split}(\langle q, ps, \infty\rangle) &= \langle \epsilon, q\rangle \\
\mathit{split}(\langle q, \bar{p} :: ps, 0\rangle) &= \langle \bar{p}, \epsilon\rangle \\
\mathit{split}(\langle F :: q, _ :: ps, k{+}1\rangle) &= \langle \bar{p}, F :: q'\rangle \\
&\quad \text{where } \langle \bar{p}, q'\rangle = \mathit{split}(\langle q, ps, k\rangle)
\end{aligned}$$

By running the K$\sigma^{\ell}_{\Uparrow}$-machine with the algebraic interpretation of β-traversals described in Figure 4, when the machine halts we are left with the cord belonging to an atom of the reduct. When recursing over thunks in the context, we must re-initialise each cord label $\langle q, ps, k\rangle$ to $\langle q, ps, \infty\rangle$, as the horizon of an atom is always relative to the head of the spine to which it belongs.

If in the reduct we find a match for a pattern l (the method for which is outside the scope of this paper, though any higher-order matching algorithm should do), we collect together the cords of all atoms corresponding to non-free-variables in the matching pattern. We can then calculate the cord for the whole match, $\langle q, ps, k\rangle$ where q and ps are from the cord for head of the match and k is the horizon of the match, calculated as described in Definition 14. By splitting this cord we get the paths $\bar{p}$ and q for Theorem 1.

At this point we will not yet have modified the term structure in any way, but rather 'peeked' into the term's β-normal form in order to find a term matching a pattern to reduce. What we need to do now is to take the paths corresponding to that match and with them compute $t \to^{*}_{\beta} t[\hat{\theta}'(l)]_{\bar{p}\cdot(q/\beta)}$.

4.3 K$\sigma^{\sharp}_{\Uparrow}$-machine

The final K$\sigma^{\sharp}_{\Uparrow}$-machine is a simple extension of the Kσ-machine (albeit with the *lift* operator to bring it in line with the calculus of the K$\sigma^{\ell}_{\Uparrow}$-machine), taking a 'track' as an extra component. This track is a path telling the machine the position of the subterm we wish to evaluate to. The K$\sigma^{\sharp}_{\Uparrow}$-machine is described in Figure 5. As an example, we might initialise the machine with the configuration $\langle t|_{\bar{p}}[\mathrm{id}] \mid \Box\rangle^{q}$. Running this machine until it halts will result in a configuration $\langle t'[\sigma] \mid C\rangle^{\epsilon}$, with the components $t'[\sigma] \equiv t|_{\bar{p}\cdot(q/\beta)}$ and $C \equiv t|_{\bar{p}}[\Box]_{q/\beta}$.

If we intend to perform a rewrite step we may now replace the matching $\hat{\theta}'(l)$ with its contractum $\hat{\theta}'(r)$ at this location, as is conventionally done at β-normal form. Assembling these components into the term $t[C[\hat{\theta}'(r)]]_{\bar{p}}$ then completes the full rewrite step.

5. Conclusion

We have introduced an abstract machine, the Kσ-machine, for evaluating terms of the λ-calculus. Building on this, we have introduced a pair of variants, K$\sigma^{\ell}_{\Uparrow}$ and K$\sigma^{\sharp}_{\Uparrow}$, which together enable higher-order term sharing by 'peeking' into a term's normal form to find a subterm matching a pattern, and then reducing enough of the term to reveal the match for rewriting without fully normalising the term. In this way the term sharing described by the simply-typed λ-calculus is made use of, instead of being lost by normalisation, or otherwise restricted to sharing with dags, which are equivalent to first-order β-redexes alone.

The value of the machinery being used here is that, as it in no way modifies the term structure itself, it does not reduce the term as it travels, but instead navigates through the unmodified term by building up a substitution environment. This parallels common approaches to sharing first-order terms with dags, in which a redex is found 'modulo sharing' and then the necessary components are unshared to make the rewrite possible. However, the simply-typed λ-calculus provides more sophisticated sharing than dags, and we make steps towards its treatment as a mechanism for 'higher-order term sharing'.

The way we make use of the sharing of the λ-calculus here is, however, relatively primitive. Although it is necessary to determine the horizon of a match, as the K$\sigma^{\ell}_{\Uparrow}$-machine does, in order to avoid rewriting in one location when it could be done in multiple, sharing information is still lost when subsequent β-reductions are performed by the K$\sigma^{\sharp}_{\Uparrow}$-machine. This is not about optimality *à la* Lévy [7], but about managing higher-order sharing in the same

Figure 4. An algebraic interpretation of a β-traversal

$$\begin{aligned}
\top &= \langle \epsilon, \epsilon :: \epsilon, \infty \rangle && \text{Top} \\
\mathrm{B}(\langle \alpha, ps, k \rangle) &= \langle \alpha, ps, k \rangle && \text{Beta} \\
\Sigma(\langle \alpha, p :: ps, k \rangle, \langle @_r :: \beta, q :: qs, k' \rangle) &= \langle \alpha, q :: ps, \mathit{min}(k, |qs|) \rangle && \text{Substitute} \\
@_l(\langle \alpha, p :: ps, k \rangle) &= \langle @_l :: \alpha, (@_l :: p) :: p :: ps, k \rangle && \text{Left} \\
@_r(\langle \alpha, p :: ps, k \rangle) &= \langle @_r :: \alpha, (@_r :: p) :: p :: ps, k \rangle && \text{Right} \\
\Lambda(\langle \alpha, p :: ps, k \rangle) &= \langle \Lambda :: \alpha, (\Lambda :: p) :: p :: ps, k \rangle && \text{Lambda}
\end{aligned}$$

Figure 5. K$\sigma^{\sharp}_{\Uparrow}$-machine

$$\begin{aligned}
\langle (t_1 t_2)[\sigma] \mid C \rangle^{@_l :: \alpha} &\to \langle t_1[\sigma] \mid \Box(t_2[\sigma]) \,;\, C \rangle^{\alpha} && \text{Left} \\
\langle (t_1 t_2)[\sigma] \mid C \rangle^{@_r :: \alpha} &\to \langle t_2[\sigma] \mid (t_1[\sigma])\Box \,;\, C \rangle^{\alpha} && \text{Right} \\
\langle (\lambda t)[\sigma] \mid \Box(u[\rho]) \,;\, C \rangle^{\Lambda :: \alpha} &\to \langle t[(u[\rho] \cdot \sigma) \,;\, \mathrm{id}] \mid C \rangle^{\alpha} && \text{Beta} \\
\langle (\lambda t)[\sigma] \mid C \rangle^{\Lambda :: \alpha} &\to \langle t[\Uparrow(\sigma) \,;\, \mathrm{id}] \mid \lambda\Box \,;\, C \rangle^{\alpha} && \text{Lambda} \\
\langle \underline{n}[(\pi \,;\, \rho) \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle \underline{n}[\pi \,;\, (\rho \,;\, \sigma)] \mid C \rangle^{\alpha} && \text{Associate} \\
\langle \underline{0}[(u[\pi] \cdot \rho) \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle u[\pi \,;\, \sigma] \mid C \rangle^{\alpha} && \text{Head} \\
\langle \underline{n{+}1}[(u[\pi] \cdot \rho) \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle \underline{n}[\rho \,;\, \sigma] \mid C \rangle^{\alpha} && \text{Tail} \\
\langle \underline{0}[\Uparrow(\rho) \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle \underline{0}[\sigma] \mid C \rangle^{\alpha} && \text{Naught} \\
\langle \underline{n{+}1}[\Uparrow(\rho) \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle \underline{n}[(\rho \,;\, \uparrow) \,;\, \sigma] \mid C \rangle^{\alpha} && \text{Lift} \\
\langle \underline{n}[\uparrow \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle \underline{n{+}1}[\sigma] \mid C \rangle^{\alpha} && \text{Shift} \\
\langle \underline{n}[\mathrm{id} \,;\, \sigma] \mid C \rangle^{\alpha} &\to \langle \underline{n}[\sigma] \mid C \rangle^{\alpha} && \text{Id}
\end{aligned}$$

fashion as Wadsworth [11] does first-order sharing. By comparing the behaviour of Wadsworth's dags, and first-order β-redexes in our sharing scheme, we can see that there is still work to be done on this front. This is the subject of ongoing research.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(04):375–416, 1991.

[2] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1984.

[3] Nicolaas de Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church–Rosser theorem. In *Indagationes Mathematicae*, volume 75, pages 381–392, 1972.

[4] Thérèse Hardin and Jean-Jacques Lévy. A confluent calculus of substitutions. In *France–Japan Artificial Intelligence and Computer Science Symposium*, volume 106, 1989.

[5] Gérard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

[6] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[7] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191, 1980.

[8] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[9] Tobias Nipkow. Higher-order critical pairs. In *Proceedings of the 6th Annual IEEE Symposium of Logic in Computer Science*, 1991.

[10] Vincent van Oostrom. *Confluence for abstract and higher-order rewriting*. PhD thesis, Vrije Universiteit, 1994.

[11] Christopher Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, University of Oxford, 1971.

Church Encoding of Data Types Considered Harmful for Implementations

– Functional Pearl –

Pieter Koopman Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
pieter@cs.ru.nl rinus@cs.ru.nl

Jan Martin Jansen
Netherlands Defence Academy (NLDA)
The Netherlands
jm.jansen.04@nlda.nl

Abstract

From the λ-calculus it is known how to represent (recursive) data structures by ordinary λ-terms. Based on this idea one can represent algebraic data types in a functional programming language by higher-order functions. Using this encoding we only have to implement functions to achieve an implementation of the functional language with data structures. In this paper we compare the famous Church encoding of data types with the less familiar Scott and Parigot encoding.

We show that one can use the encoding of data types by functions in a Hindley-Milner typed language by adding a single constructor for each data type. In an untyped context, like an efficient implementation, this constructor can be omitted. By collecting the basic operations of a data type in a type constructor class and providing instances for the various encodings, these encodings can coexist in a single program. This shows the differences and similarities of the encodings clearly. By changing the instance of this class we can execute the same algorithm in a different encoding.

We show that in the Church encoding selectors of constructors yielding the recursive type, like the tail of a list, have an undesirable strictness in the spine of the data structure. The Scott encoding does not hamper lazy evaluation in any way. The evaluation of the recursive spine by the Church encoding makes the complexity of these destructors $O(n)$. The same destructors in the Scott encoding requires only constant time. Moreover, the Church encoding has serious problems with graph reduction. The Parigot encoding combines the best of both worlds, but in practice this does not offer an advantage.

Categories and Subject Descriptors D [*1*]: 1; D [*3*]: 3Data types and structures

Keywords Implementation, Data Types, Church Numbers, Scott Encoding, Parigot Encoding

1. Introduction

In the λ-calculus it is well-known how to encode data types by λ-terms. The most famous way to represent data types by functions in the λ-calculus is based on the encoding of Peano numbers by Church numerals [2, 3]. In this paper we review the Church encoding of data types and show that it causes serve problems complexity problems and spoils laziness. These problems can be prevented by using the far less known Scott encoding of data types.

Based on this approach we can transform the algebraic data types from functional programming language like Clean [30] and Haskell [17] to functions. These algebraic data types are first introduced in the language HOPE [10]. The algebraic data types are equivalent to the polynomial data types used in type theory. Representing all data types by plain functions simplifies the implementation of the programming language. The implementation only has to cope with plain higher order functions, instead of these functions and data types. Most abstract machines [22] used to implement programming languages contain special instructions to handle data types, e.g., the SECD machine [24], the G-machine [21, 29] and the ABC-machine [23]. When we represent data types by functions the transformed programs only contain functions. Since the implementation of the transformed programs does not have to cope with algebraic data types, they are easier to implement.

Barendregt [3] explains in chapter 6 how the Church encoding was transformed to the typed λ-calculus in papers of Böhm and various coauthors [5–9]. The Church encoding adapted to the typed λ-calculus is also known as the Böhm-Berarducci encoding. Barendregt introduces an alternative encoding called the *standard representation* in [2]. This is a two step approach to represent λ-terms. First, the λ-terms are represented as Gödel numbers. Next, these Gödel numbers are represented as Church Numerals.

The oldest description of an alternative encoding is in a set of unpublished notes of Dana Scott, see [12] page 504. This encoding is reinvented many times in history. For instance, the implementation of the language Ponder is based on this idea [13]. Also the language TINY from Steensgaard-Madsen uses the Scott encoding [32]. The representation of data types by Mogensen is an extension of the Scott encoding that enables reflection [3, 25]. For the implementation of the simple programming language SAPL we reinvented this encoding [18, 19]. SAPL is used to execute Clean code of an iTask program in the browser [31]. Naylor and Runciman used the ideas from SAPL in the implementation of the Reduceron [26]. Despite these publications the Scott encoding is still much less famous than the Church encoding. Moreover, we do not know a paper in which these encodings are thoroughly compared. Parigot proposed an encoding for data types that is a combination of the Church and Scott encoding [27, 28].

Using one additional constructor for each type and encoding, the encodings of the data types become valid types in the Hindley-Milner type system. This enables experiments with the encodings in typed functional languages like Clean and Haskell.

In this paper we use a type constructor class to capture all basic operations on algebraic data types, constructors as well as matching and selector functions. All manipulations of the data type in the source language are considered to be expressed in terms of the functions in this type class. By switching the instance of the type class, we obtain another encoding of the data type. We add a single constructor to the implementation of an algebraic data type by functions. This constructor is necessary to make the encoding typable in the Hindley-Milner type system. This is necessary in order to experiment with these encodings in a strongly typed language like Haskell or Clean. Moreover, the constructor enables us to distinguish the various encodings as different instances of a type class. Just like a `newtype` in Haskell there is no reason to use these constructors in an actual implementation based on these encodings. The uniform encoding based on type classes helps us to compare the various encodings of the data type.

In the next section we review the encoding of non-recursive data types in the λ-calculus. In Section 3 the λ-calculus is replaced by named higher-order functions. The named function makes the notation of recursive algorithms easier. We show how recursive data types can be represented by named function in Section 4. The Church and Scott encodings will be different instances of the same type constructor class. This makes the differences in the encoding very clear. Optimisations of the encodings are discussed in Section 5. In Section 6 we conclude that algorithmic complexity of the Church encoding is for recursive selector functions higher than the complexity of the Scott encoding.

2. Data Types in the λ-calculus

Very early in the development of λ-calculus it was known how to use λ-terms for manipulations handled nowadays by algebraic data types in functional programming languages. Since one does not want to extend the λ-calculus with new language primitives, λ-terms achieving the effect of the data types were introduced.

For an enumeration type with n constructors without arguments, the λ-term equivalent to constructor C_i, selects argument i from the given n arguments: $C_i \equiv \lambda\ x_1\ \ldots x_n\ .\ x_i$. For each function over the enumeration type we supply n arguments corresponding to results for the constructors. The simplest nontrivial example is the type for Boolean values.

2.1 Booleans in λ-calculus

The Boolean values True and False can be represented by a data type having just two constructors T and F[1]. We interpret the first argument as true and the second one as false.

$$T \equiv \lambda\ t\ f\ .\ t$$
$$F \equiv \lambda\ t\ f\ .\ f$$

For the Boolean values the most famous application of these terms is probably the conditional.

$$cond \equiv \lambda\ c\ t\ e\ .\ c\ t\ e$$

Using Currying the conditional can also be represented as the identity function: $cond\ \equiv\ \lambda\ c\ .\ c$. We can even apply the Boolean value directly to the then- and else-part. This is used in the following definition of the logical and-operator.

$$and \equiv \lambda\ x\ y\ .\ x\ y\ F$$

If the first argument x is true the result of the and-function is determined by the argument y. Otherwise, the result of the and-function is F (false).

2.2 Pairs in λ-calculus

This approach can be directly extended to data constructors with non-recursive arguments. The arguments of the constructor in the original data type are given as arguments to the function representing this constructor. A pair is a data type with a single constructor. This constructor has two arguments. The functions `e1` selects the first element of such a pair and `e2` the second element. The λ-terms encoding such a pair are:

$$Pair \equiv \lambda\ x\ y\ p\ .\ p\ x\ y$$
$$e1 \equiv \lambda\ p\ .\ p\ (\lambda x\ y\ .\ x)$$
$$e2 \equiv \lambda\ p\ .\ p\ (\lambda x\ y\ .\ y)$$

With these terms we can construct a term that swaps the elements in such a pair as:

$$swap \equiv \lambda\ p\ .\ Pair\ (e2\ p)\ (e1\ p)$$

3. Representing Data Types by Named Functions

It seems to be straightforward to transform the λ-terms representing Booleans and pairs to functions in a functional programming language like Clean or Haskell. In this paper we will use Clean, but any modern lazy functional programming language with type constructor classes will give very similar results.

3.1 Encoding Booleans by Named Functions

The functions form Section 2.1 for the Booleans become[2][3]:

```
T :: a a → a
T t f = t

F :: a a → a
F t f = f

cond :: (a a → a) a a → a
cond c t e = c t e
```

3.2 Encoding Pairs by Named Functions

The direct translation of the λ-terms in Section 2.2 for pairs yields[4]:

```
Pair‘ :: a b → (a b → c) → c
Pair‘ x y = λp.p x y

e1‘ :: ((a b → a) → a) → a
e1‘ p = p (λx y.x)

e2‘ :: ((a b → b) → b) → b
e2‘ p = p (λx y.y)

swap‘ :: ((a a → a) → a) → (a a → b) → b
swap‘ p = Pair‘ (e2‘ p) (e1‘ p)
```

Unfortunately, the type for `swap‘` requires that both elements of the pair have the same type `a`. This is due to the fact that the type for `Pair‘` states the type of the access function as `a b→c` and

[1] In this paper we will use names starting with a capital for functions mimicking constructors and names starting with a lowercase letter for all other functions. Hence a boolean is a function with two arguments.

[2] For typographical reasons we generally prefer a function like `T t f = t` over the equivalent `T = λt f.t`.

[3] In Haskell the type `a a→a` is written as `a→a→a`.

[4] In Haskell the anonymous function `λp.p x y` is written as `\p->p x y`.

the Hindley-Milner type system requires a single type for such an argument [3]. The encoding with additional constructors and type constructor classes developed below in Section 4.1 will remove this limitation.

This restriction only limits the possibility to execute the encoding in functions in a strongly typed language. In an untyped context these functions will behave correctly.

4. Recursive Data Types

For recursive types we show two different ways to represent data types by functions. The first approach is a generalisation of the well-known Church numbers [2]. Here the recursive occurrences of a function representing a constructor are all equal. This implies that a recursive data structure mirrors an expanded fold as pointed out by Hinze in [16]. This is especially convenient in λ-calculus since recursive functions require that the function itself is passed around as an additional argument. This recursion is usually achieved by an application of the Y-combinator.

The second approach does nothing special for recursive arguments of constructors. Hence, it uses explicitly recursive manipulation functions for recursive data types, just like the algebraic data types in functional programming languages like Haskell and Clean. This arbitrary recursion pattern of these functions is not limited to folds of the Church encoding. The oldest source of this approach is a set of unpublished notes from Dana Scott, hence this approach is called the Scott encoding.

Since we have named functions, arbitrary recursion is no problem at all. For this reason we state both encodings of the data type and the associated manipulation functions directly as named functions. In order to compare both encodings easily and to be able to write functions that work for both encodings we construct type (constructor) classes for the data types in our description. The type constructor class will contain the constructors and the selection functions for elements of the constructors.

These type classes require a constructor in the functions representing data types. We already want to insert constructors to enable the use of the functional encodings in a strongly typed language. The constructors are still unnecessary when we fix the encoding and work in an untyped setting.

4.1 Pairs Revisited

In our new approach `Pair` is a type constructor class with a constructor `Pair` and two destructors `e1` and `e2`. These destructors select the first and second argument.

```
class Pair t where
    Pair :: a b → t a b
    e1   :: (t a b) → a
    e2   :: (t a b) → b
```

Using the primitives from this class, the `swap` function becomes[5]:

```
swap :: (t a b) → t b a | Pair t
swap p = Pair (e2 p) (e1 p)
```

Note that the use of the type class yields better readable types and eliminates the problem with the types of the arguments in the function `swap`. Here it is completely valid to use different types for the elements of the pair. In fact, it is even possible to yield a pair that is a member of another instance of the type constructor class `swap :: (t a b) → (u b a) | Pair t & Pair u`.

[5] The class restriction `| Pair t` in the type of the function `swap` states that this function works for any type `t` that is an instance of the type constructor class `Pair`. This ensures that `Pair`, `e1` and `e2` are defined for `t`. In Haskell such a class constraint is written as `swap :: (Pair t) ⇒ (t a b) → t b a`.

4.1.1 Pairs with Native Tuples

The instance of `Pair` for 2-tuples is completely standard.

```
instance Pair (,) where
    Pair a b  = (a, b)
    e1 (a, b) = a        // equal to the function fst from StdEnv
    e2 (a, b) = b        // equal to the function snd from StdEnv
```

4.1.2 Pairs with Functions

Since `Pair` is not recursive, both encodings of such a pair with functions coincide. We introduce a placeholder type `FPair` to satisfy the type class system. This also allows use to introduce a universally quantified type variable `t` for the result of manipulations of the type.

```
:: FPair a b = FPair (∀t: (a b→t) → t)

instance Pair FPair where
    Pair a b     = FPair λp.p a b
    e1 (FPair p) = p λa b.a
    e2 (FPair p) = p λa b.b
```

In order to give `FPair` the kind required by the type constructor class `Pair` the type of the result `t` is an universal quantified type in this definition. This makes the definition of `swap` typable in a Hindley-Milner type system, even without a type constructor class, while the definition of `swap‘` in Section 3.2 restricts the elements `e1` and `e2` of the pair to have identical types.

4.2 Peano Numbers

The simplest recursive data type is a type `Num` for Peano numbers. This type has two constructors. `Zero` is the non recursive constructor that represents the value zero. The recursive constructor `Succ` yields the successor of such a Peano number, it has another `Num` as argument. There are two basic manipulation functions, a test on zero and a function computing the predecessor of a Peano number.

```
class Num t where
    Zero   :: t
    Succ   :: t → t
    isZero :: t → Bool
    pred   :: t → t
```

It is of course possible to replace the basic type `Bool` by the type representing Booleans in this format introduced in Section 3. We use here a basic type to show that both type representation perfectly mix.

4.2.1 Peano Numbers with Integers

An implementation of these numbers based on integers is:

```
instance Num Int where
    Zero     = 0
    Succ   n = n+1
    isZero n = n == 0
    pred   n = if (n > 0) (n - 1) undef
```

4.2.2 Peano Numbers with an Algebraic Data Type

The implementation of `Num` with an ordinary algebraic data type `Peano` has no surprises. We use case expressions instead of separate function alternatives to make the definitions a little more compact.

```
:: Peano = Z | S Peano

instance Num Peano where
    Zero     = Z
    Succ   n = S n
    isZero n = case n of Zero = True; _ = False
    pred   n = case n of S m = m; _ = undef
```

4.2.3 Peano Numbers in the Church Encoding

The first encoding with functions is just the encoding of Church numbers in this format. The type `Peano` has two constructors. Hence, each constructor function has two arguments. The first argument represents the case for zero, the second one mimics the successor. A nonnegative number n is represented by n applications of this successor to the value for zero. The constructor `Succ` adds one application of this function.

The test for zero yields `True` when the given number `n` is `Zero`. Otherwise, the result is `False`. The predecessor function in the Church encoding is somewhat more challenging. The number n is represented by n applications of some higher order function `s` to a given value `z`. The predecessor must remove one of the function `s`. The first solution for this problem is found by Kleene while his wisdom teeth were extracted at the dentist [11]. The value zero is replaced by a tuple containing `undef`, the predecessor of zero[6], and the predecessor of the next number: `zero`. The successor function recursively replaces the tuple `(x,s x)` by `(s x,s (s x))` starting at `z`. The result of this construct is the tuple `(pred n,n)`. The predecessor function selects the first element of this tuple. Since the predecessor is constructed from the value zero upwards to n, the complexity of this operation is $O(n)$.

Just like in the representation of pairs we add a constructor `CNum` to solve the type problems of this representation in the Hindley-Milner type system of Clean. The universally quantified type variable `b` ensures that the functions representing the the constructors can yield any type without exposing this type in `CNum`. This type is very similar to the type $\mathsf{cnat} := \forall X.X \to (X \to X) \to X$ used for Church numbers in polymorphic λ-calculus, $\lambda 2$ [14].

The pattern `FPair _` in the `pred` function is an artefact of our encoding by type classes, it solves the overloading of `Pair`.

```
:: CNum = CNum (∀b: b (b→b) → b)

instance Num CNum where
  Zero   = CNum λzero succ.zero
  Succ n = CNum λzero succ.succ ((λ(CNum x).x) n zero succ)
  isZero (CNum n) = n True λx.False
  pred   (CNum n)
   = CNum λz s.e1 (n (Pair undef z)
                     (λp=:(FPair _).Pair (e2 p) (s (e2 p))))
```

Notice that the successor itself passes the values for `zero` and `succ` recursively to the given number `n`. Removing the constructor `CNum` from the argument of `Succ` is done in its right-hand side to prevent this argument that the argument is strict. This enables the proper evaluation of expressions like `isZero (Succ undef)`.

The function `Succ` has to add one application of the argument `succ` to the given number. Since all functions are equal this can be done in two ways. Above we add the additional application of `succ` to the encoding of `n`. It is in this encoding also possible to replace the given value of `zero` in the recursion by `succ zero`. That is `Succ (CNum n) = CNum λzero succ.n (succ zero) succ`. The direct access to both side of the sequence of applications of `succ` is unique for the Church encoding. We will use this below in Section 5 to optimise fold like operations over recursive types in the Church encoding.

4.2.4 Peano Numbers in the Scott Encoding

The type `SNum` to represent the Scott representation of numerals uses a constructor and a universally quantified type variable for exactly the same reasons as the Church encoding. This type is related to the types assigned to Scott numerals by Abadi et al., [1]. This type is again similar to $\mathsf{snat} := \forall X.X \to (\mathsf{snat} \to X) \to X$ used for Scott numbers in $\lambda 2\mu$ [14]. Since we have recursive functions in our core language, the external recursion in this problem is no problem. In λ-calculus we need for instance a fixed point-combinator for the recursion.

The Scott encoding for the non-recursive cases `Zero` and `isZero` is equal to the Church encoding. For the recursive functions `Succ` and `pred` the Scott encoding is simpler than the Church encoding. The recursion pattern of the Scott encoding is very similar to the definitions for the type `Peano`.

```
:: SNum = SNum (∀b:b (SNum→b) → b)

instance Num SNum where
    Zero           = SNum λzero succ.zero
    Succ   n       = SNum λzero succ.succ n
    isZero (SNum n) = n True λx.False
    pred   (SNum n) = n undef λx.x
```

Since the implementation of `pred` in this Scott encoding is a simple selection of an element of a constructor its complexity is $O(1)$. This is much better than the $O(n)$ complexity of the same operator in the Church encoding.

4.2.5 Peano Numbers in the Parigot Encoding

Parigot proposed a different encoding of data types in an attempt to enable reasoning about algorithms as well as an efficient implementation of these algorithms [27, 28]. These papers do not mentioning the Scott encoding. In addition for a recursive type the constructors contain the Church-Style fold argument, as well as the Scott-style plain recursive argument. For numbers this reads:

```
:: PNum = PNum (∀b:b (PNum b→b) → b)

instance Num PNum where
    Zero           = PNum λz s.z
    Succ    p      = PNum λz s.s p ((λ(PNum n).n) p z s)
    isZero (PNum n) = n True λp x.False
    pred   (PNum n) = n undef (λp x.p)
```

It will be no surprise that this type resembles the type in $\lambda 2\mu$: $\mathsf{pnat} := \forall X.X \to (\mathsf{pnat} \to X \to X) \to X$ used for Parigot numbers in [14] (called Church-Scott numbers there).

Notice that `pred` is implemented here in the more efficient Scott way. The second argument of `Succ` is more suited for a fold-like operation.

4.2.6 Using the Type Class Num

Using the primitive from the class `Num` we can define manipulation functions for these numbers. The transformation of any of these number encodings to one of the other encodings is given by `NumToNum`. This uniform transformation is a generalisation of the transformations between Church and Scott numbers in [20]. The context determines the encodings `n` and `m`. Using a very similar recursion pattern we can define addition for all instances of `Num` by the function `add`.

```
NumToNum :: n → m | Num n & Num m
NumToNum n | isZero n
    = Zero
    = Succ (NumToNum (pred n))

add :: t t → t | Num t
add x y | isZero x
    = y
    = add (pred x) (Succ y)
```

[6] Every now and then people use `zero` instead of `undef` as predecessor of `Zero`. This prevents runtime errors, but it does not correspond to our intuition of numbers and complicates reasoning. For instance, the property `pred n == pred m ⇒ n == m` does not hold since `pred (succ zero)` becomes equal to `pred zero`.

Using details of the encoding it is possible to optimise these functions. Although the definitions work for all instances, the algorithmic complexity depends on the encoding selected. In particular the processor function `pred` is $O(n)$ in the Church encoding and $O(1)$ for the other implementations of `Num`. In Section 5 we discuss how this can be improved for these examples.

4.3 Lists

In the Peano numbers all information is given by the number of applications of `Succ` in the entire data structure. Recursive data types that contain more information are often needed. The simplest extension of the Peano numbers is the list. The `Cons` nodes of a list corresponds to the `Succ` in the Peano numbers, but in contrast to the Peano numbers a `Cons` contains an element stored at that place in the list. This is modelled by type class `List`. Compared to `Num` there is an additional argument `a` in the constructor for the recursive case, and there is an additional primitive access function `head` to select this element from the outermost `Cons`.

```
class List t where
    Nil   :: t a
    Cons  :: a (t a) → t a
    isNil :: (t a) → Bool
    head  :: (t a) → a
    tail  :: (t a) → t a
```

4.3.1 List with the Native List Type

The instance for the native lists in Clean is very simple[7]

```
instance List [] where
    Nil       = []
    Cons a x  = [a:x]
    isNil  xs = case xs of [] = True; _ = False  // isEmpty
    head   xs = case xs of [a:x] = a; _ = undef  // hd
    tail   xs = case xs of [a:x] = x; _ = undef  // tl
```

4.3.2 List in the Church Encoding

The instance inspired on the Church numbers is rather similar to the instance for `CNum`. The definition for `Nil` is completely similar to the instance for `Zero`. The constructor `Cons` has the list element to be stored as additional argument. Here it does matter whether we insert the new element at the head or the tail of the list. It is actually quite remarkable that we can add an element to the tail of the list without explicit recursion. Note that the arguments for `nil` and `cons` are passed recursively to the tail `x` of the list. The manipulation functions `isNil` and `head` directly yield the desired result by applying the function `xs` to the appropriate arguments. The implementation of `tail` is more involved. We use the approach known from `pred`. From the end of the list upwards a new list is constructed that is the tail of this list. Note that this is again $O(n)$ work with n the length of the list. Moreover, it spoils lazy evaluation by requiring a complete evaluation of the spine of the list. This also excludes the use of infinite list as arguments of this version of the `tail`.

```
:: CList a = CList (∀b: b (a→b→b) → b)

instance List CList where
 Nil      = CList λnil cons.nil
 Cons a x = CList λn c.c a ((λ(CList l).l) x n c)
 isNil  (CList l) = l True λa x.False
 head   (CList l) = l undef λa x.a
 tail   (CList l)
  = CList λnil cons.e1 (l (Pair undef nil)
         (λa p=:(FPair _).Pair (e2 p) (cons a (e2 p))))
```

[7] In Haskell the list `[a:x]` is written as `(a:x)`. The expression `[a:x]` is valid Haskell, but it is a singleton list containing the list `(a:x)` as its element.

4.3.3 List in the Scott Encoding

The implementation of lists based on Scott numbers differs at the recursive argument of the `Cons` constructor. Here we use a term of type `SList a`. In the list based on Church numbers this argument has type `b`, the result type of the list manipulation. As a consequence, we do not pass the arguments `nil` and `cons` as arguments to the tail `x` in the definition for the constructor `Cons`. This makes the access function `tail` a simple $O(1)$ access function.

```
:: SList a = SList (∀b: b (a→(SList a)→b) → b)

instance List SList where
    Nil             = SList λnil cons.nil
    Cons a x        = SList λnil cons.cons a x
    isNil (SList xs) = xs True λa x.False
    head  (SList xs) = xs undef λa x.a
    tail  (SList xs) = xs undef λa x.x
```

4.3.4 List in the Parigot Encoding

Just as for numbers the Parigot encoding of `Cons` contains an argument for Scott type of recursion (i.e. `x`), as well as for the Church type recursion (i.e. `((PList l).l) x n c`).

```
:: PList a = PList (∀b: b (a (PList a) b→b) → b)

instance List PList where
    Nil      = PList λnil cons.nil
    Cons a x = PList λn c.c a x ((λ(PList l).l) x n c)
    isNil (PList l) = l True λa t x.False
    head  (PList l) = l undef (λa t x.a)
    tail  (PList l) = l undef (λa t x.t)
```

4.3.5 Using the List Type Class

Using the primitives from `List` the list manipulations fold-right and fold-left can be defined in the well-known way.

```
foldR :: (a b→b) b (t a) → b | List t
foldR op r xs | isNil xs
    = r
    = op (head xs) (foldR op r (tail xs))

foldL :: (a b→a) a (t b) → a | List xs
foldL op r xs | isNil xs
    = r
    = foldL op (op r (head xs)) (tail xs)
```

Due to the $O(n)$ complexity of `tail` in the Church representation, the folds have $O(n^2)$ complexity in the Church encoding when the operators `op` is strict in both arguments. In the other instances of `List` the complexity is only $O(n)$. In Section 5 we show how this complexity problem can be fixed for `foldR`.

The transformation from one list encoding to any other instance of `List` is done in `ListToList` by an application of this `foldR`. The summation of a list is done by a fold-left since the use of an accumulator enables a constant memory implementation. This function works for any argument type `a` having an addition operator `+` and a unit element `zero`.

```
ListToList :: (t a) → u a | List t & List u
ListToList xs = foldR Cons Nil xs

suml :: (t a) → a | List t & +, zero a
suml xs = foldL (+) zero xs
```

4.3.6 Measuring Execution Time

Using these definitions we can easily verify the described behaviour of the implementations of the class `List`. Our first example is extremely simple; it takes the `head` of the `tail` of a list.

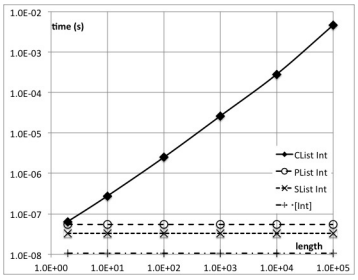
By just changing the type at a strategic place, here the function `headTail`, we enforce another implementation of `List`. When we replace `CList` in this function by `SList`, `PList` or `[]` that type instance of `List` is used. The length of the list is controlled by the definition of `m`.

```
headTail :: !(CList Int) → Bool
headTail l = head (tail l) == 2

fromTo :: Int Int → t Int | List t
fromTo n m | n > m
    = Nil
    = Cons n (fromTo (n+1) m)

Start = headTail (fromTo 1 m)
```

All experiments are done with 32-bit Clean 2.4 running on windows 7 in a virtual box on a MacBook Air with 1.8 GHz Intel Core i5 under OS X version 10.9.4. For reliable measurements the computation is repeated such that the total execution time is at least 10 seconds.



head (tail (fromTo 1 m))

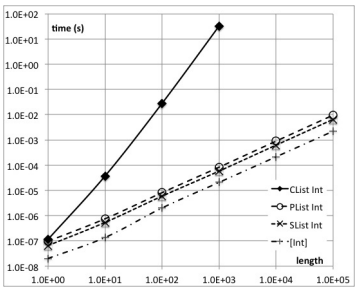
m	CList Int	SList Int	PList Int	[Int]
2	$6.5\,10^{-8}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10	$2.8\,10^{-7}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^2	$2.5\,10^{-6}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^3	$2.6\,10^{-5}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^4	$2.8\,10^{-5}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^5	$4.7\,10^{-5}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$

Figure 1. Execution time in seconds as function of the length for the encodings of `List`. Note the double logarithmic scale.

It is no surprise that the execution time for `SList Int` and `[Int]` is completely independent of the upper bound, `m`, of the list. Due to lazy evaluation the list is only evaluated until its second element. As predicted the execution time for `CList Int` is linear in the length of the list since `tail` enforces evaluation until the `Nil`. For very long lists in the Church representation, e.g. 10^5 elements, garbage collection causes an additional increase of the execution time.

In the second experiment we enforce evaluation of the entire list by computing the sum of the numbers 1 to m and check whether this sum is indeed $m(m-1)/2$ for various values of m. We use a tail recursive definition for the function `Sum`:

```
sum :: (t Int) → Int | List t
sum l | isNil l
    = 0
    = head l + sum (tail l)
```



sum (fromTo 1 m)

m	CList Int	SList Int	PList Int	[Int]
1	$1.2\,10^{-7}$	$6.6\,10^{-8}$	$9.7\,10^{-8}$	$2.0\,10^{-8}$
10	$3.7\,10^{-5}$	$5.4\,10^{-7}$	$7.5\,10^{-7}$	$1.4\,10^{-7}$
10^2	$2.8\,10^{-2}$	$5.9\,10^{-6}$	$8.2\,10^{-6}$	$2.0\,10^{-6}$
10^3	$3.3\,10^{+1}$	$5.8\,10^{-5}$	$8.3\,10^{-5}$	$2.1\,10^{-5}$
10^4		$6.2\,10^{-4}$	$9.0\,10^{-4}$	$2.1\,10^{-4}$
10^5		$6.5\,10^{-3}$	$9.6\,10^{-3}$	$2.2\,10^{-3}$

Figure 2. Execution time in seconds as function of the length for the encodings of `List`.

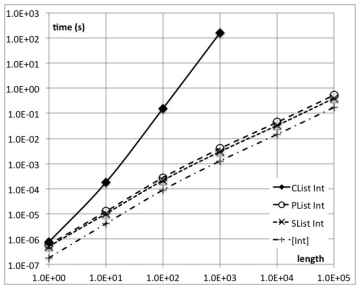
As expected the execution time for `SList Int` and `[Int]` grows linear with the the number of elements in the list. The version for `CList Int` is again much slower. Since the `tail` inside `sum` is $O(n)$ for a list in the Church representation, the `sum` itself is at least $O(n^2)$. The measurements show that the actual execution time grows as $O(n^3)$ in the Church representation. Since the `tail` in the Church representation yields a function application, the reduction of an application `tail l` cannot be shared. This expression is re-evaluated for each use of the resulting list. Each of these `tail` functions is $O(n)$. This makes the total complexity of `sum` $O(n^3)$ in the Church representation and $O(n)$ in the Scott representation and in the native lists of Clean.

In the final example we apply the quick-sort algorithm to a lists of pseudo random integers in the range $0..999$. Quick-sort is implemented for all list implementations in the class `List` by the function `qs`.

```
qs :: (l a) → (l a) | List l & <, == a
qs l | isNil l
    = Nil
    = append (qs (fltr (λx.x <  y) l))
         (append (fltr (λx.x == y) l)
             (qs (fltr (λx.y <  x) l))) where y = head l

append :: (t a) (t a) → (t a) | List t
append l1 l2 | isNil l1
    = l2
    = Cons (head l1) (append (tail l1) l2)

fltr :: (a→Bool) (l a) → l a | List l
fltr p l | isNil l
    = Nil
    | p x
        = Cons x (fltr p (tail l))
        =        fltr p (tail l)       where x = head l
```



qs (take m randomInts))

m	CList Int	SList Int	PList Int	[Int]
1	$7.5\,10^{-7}$	$5.2\,10^{-7}$	$6.0\,10^{-7}$	$1.8\,10^{-7}$
10	$1.8\,10^{-4}$	$1.0\,10^{-5}$	$1.3\,10^{-5}$	$4.1\,10^{-6}$
10^2	$1.5\,10^{-1}$	$2.1\,10^{-4}$	$2.8\,10^{-4}$	$8.9\,10^{-5}$
10^3	$1.6\,10^{+2}$	$3.0\,10^{-3}$	$3.9\,10^{-3}$	$1.3\,10^{-3}$
10^4		$3.3\,10^{-2}$	$4.4\,10^{-2}$	$1.4\,10^{-2}$
10^5		$3.9\,10^{-1}$	$5.3\,10^{-1}$	$1.7\,10^{-1}$

Figure 3. Relation between execution time in seconds and length of the list for four different implementations of `List`.

The measurements show the expected $O(n \log n)$ growth with the length of the list for the Scott representation of lists and native lists in Clean. The Church representation shows again a $O(n^3)$ growth with the length of the list. Since Quick-sort requires more list operations than `sum`, the increase in execution time is even bigger than for `sum`.

The Scott representation, `SList Int`, is on average a factor 2.8 slower than the native Clean lists, `[Int]`. This additional execution time is caused by the additional constructors `SList` need to convince the type system of correctness of this representation. This factor is independent of the size of the lists.

Functions like `sum`, `append` and `fltr` used in these examples can be expressed as applications of `fold`. It is possible to optimise a `fold` in the Church representation as outlined in Section 5. Since we study the representation of data structures by functions for a *simple* compiler, it is unrealistic to expect such a compiler to perform the

required transformations. Moreover, this does not solve the problems with laziness, in examples like `head (tail (fromTo 1 m))`, and the complexity problems in situations where the `tail` function is not part of a `foldr`, like `qs_o2`.

4.4 Tree

We demonstrate how the approach is extended to data types with multiple recursive arguments such as binary trees. The constructor for nonempty trees, `Fork`, has now two recursive instances as argument instead of just one. There are now two selectors for the recursive arguments, `left` and `right`, instead of just `tail`.

```
class Tree t where
    Leaf   :: t a
    Fork   :: (t a) a (t a) → t a
    isLeaf :: (t a) → Bool
    elem   :: (t a) → a
    left   :: (t a) → t a
    right  :: (t a) → t a
```

4.4.1 Tree with an Algebraic Data Type

The instance for a two constructor algebraic data type is again standard.

```
:: Bin a = Empty | Node (Bin a) a (Bin a)

instance Tree Bin where
    Leaf       = Empty
    Fork x a y = Node x a y
    isLeaf t   = case t of Empty = True; _ = False
    elem   t   = case t of (Node x a y) = a; _ = undef
    left   t   = case t of (Node x a y) = x; _ = undef
    right  t   = case t of (Node x a y) = y; _ = undef
```

4.4.2 Tree in the Church Encoding

The instance based on Church numbers passes the arguments for `leaf` and `node` now to two recursive occurrences in the constructor `Fork` for nonempty trees. The selector functions for the recursive arguments, `left` and `right`, use the same pattern as `pred` and `tail`. The difference is that there are two recursive cases. Fortunately, they can be handled with a single function. For readability we use tuples from Clean instead of a `Pair` as introduced in Section 2.2. Like above this selection function visits all n nodes in the subtree. Hence, its complexity is $O(n)$ while the version using ordinary algebraic data types does the job in constant time, $O(1)$.

```
:: CTree a = CTree (∀t: t (t a t→t) → t)

instance Tree CTree where
 Leaf = CTree λleaf fork
 Fork (CTree x) a (CTree y)
  = CTree λleaf fork. fork (x leaf fork) a (y leaf fork)
 isLeaf (CTree t) = t True λx a y.False
 elem   (CTree t) = t undef λx a y.a
 left   (CTree t)
  = CTree λe f.e1 (t (undef,e) (λ(s,t) a (x,y).(t,f t a y)))
 right  (CTree t)
  = CTree λe f.e1 (t (undef,e) (λ(s,t) a (x,y).(y,f t a y)))
```

4.4.3 Tree in the Scott Encoding

The version based on the Scott encoding of numbers is again much more similar to the implementation based on plain algebraic data types. In the constructor `Fork` for nonempty trees the arguments `leaf` and `node` are not passed to the recursive occurrences of the tree, `x` and `y`. This makes the selection of the recursive elements identical to the selection of the non recursive argument, `elem`. The complexity of the three selection functions is the desired $O(1)$.

```
:: STree a = STree (∀t: t ((STree a) a (STree a)→t) → t)

instance Tree STree where
    Leaf = STree λleaf node.leaf
    Fork x a y = STree λleaf node.node x a y
    isLeaf (STree t) = t True λx a y.False
    elem   (STree t) = t undef λx a y.a
    left   (STree t) = t undef λx a y.x
    right  (STree t) = t undef λx a y.y
```

4.4.4 Using the Tree Type Class

Using the primitives from `Tree` we can express insertion for binary search trees by the function `insertTree`.

```
insertTree :: a (t a) → t a | Tree t & < a
insertTree a t
    | isLeaf t  = Fork Leaf a Leaf
    | a < x     = Fork (insertTree a (left t)) x (right t)
    | a > x     = Fork (left t) x (insertTree a (right t))
    | otherwise = t // x == a
where x = elem t
```

Note that the recursion pattern in this function is dependent of the value of the element to be inserted, `a`, and the element in the current node, `elem t`.

When the selectors `left` and `right` operate in constant time the average cost of an insert in a balanced tree are $O(\log\ n)$, and in worst case the insert is $O(n)$ work. For the implementation based on Church numbers however, the complexity of the selectors `left` and `right` is $O(n)$. This makes the average complexity of an insert in a balanced tree $O(n \log\ n)$, in worst case the complexity is even $O(n^2)$. In testing basic properties of trees this is very well noticeable. Even with small test cases tests with the data structures based on the Church numbers take at least one order of magnitude more time than all other tests for the definitions in this paper together.

4.5 General Transformations

The examples in the previous sections illustrate the general transformation scheme from a constructor based encoding to a function based encoding. In the transformations below we omit the additional type and constructors used in this paper to handle the various versions in a single type constructor class. A type T with a arguments and n constructors named $C_1 \ldots C_n$ will be represented by n functions. The transformation $\mathcal{T}$ specifies the functions needed to represent a type T.

$$\begin{aligned}\mathcal{T}[\![\, T\ x_1 \ldots x_a = C_1\ a_{1_1} \ldots a_{1_m} | \ \ldots\ | C_n\ a_{n_1} \ldots a_{n_m} \,]\!] \\ = \mathcal{C}[\![\, C_1\ a_{1_1}\ \ldots\ a_{1_m} \,]\!]\ n \cdots \mathcal{C}[\![\, C_n\ a_{n_1}\ \ldots\ a_{n_m} \,]\!]\ n\end{aligned}$$

The function for constructor C_i with m arguments has the same name as this constructor and has $n + m$ arguments. $\mathcal{C}$ yields the function for the given constructor.

$$\begin{aligned}&\mathcal{C}[\![\, C_i\ a_{i_1} \ldots a_{i_m} \,]\!]\ n \\ &\quad = C_i\ a_1 \ldots a_m\ x_1\ \ldots\ x_n = x_i\ \mathcal{A}[\![\, a_1 \,]\!]\ n\ \ldots\ \mathcal{A}[\![\, a_m \,]\!]\ n\end{aligned}$$

For all arguments in the Scott encoding and the non recursive arguments in the Church encoding, the transformation $\mathcal{A}$ just produces the given argument.

$$\mathcal{A}[\![\, a \,]\!]\ n = a$$

For recursive arguments in the Church encoding however, all arguments of the function C are added:

$$\mathcal{A}_2[\![\, a \,]\!]\ n = (a\ x_1\ \ldots\ x_n)$$

These definitions show that a constructor is basically just a selector that picks the continuation corresponding to its constructor

number. In a function over type T we provide a value for each constructor, similar to a case distinction in a switch expression.

$$\begin{aligned}&\mathcal{S}[\![\,\mathsf{case}\ e\ \mathsf{of}\\&\quad C_1\ a_{1_1} \ldots a_{1_m} = r_1;\\&\quad \ldots\\&\quad C_n\ a_{n_1} \ldots a_{n_m} = r_n;\,]\!]\\&= e\ (\lambda\, a_{1_1}\ \ldots\ a_{1_m}\,.\,r_1)\ \ldots\ (\lambda\, a_{n_1}\ \ldots\ a_{n_m}\,.\,r_n)\end{aligned}$$

Due to recursive passing of arguments in the Church encoding, a recursive argument a_j will be transformed to an expression of the result type R of the case expression. In the Scott encoding it will still have type T. This implies that we can still decide in the body r_i whether we want to apply the function recursively or not. In the Church encoding the function is alway applied recursively.

5. Optimisations

In a real implementation of a functional language that represents data types by functions, the type constructor classes introduced here and the constructors required by those type classes should be omitted. They are only introduced in this paper to allow experiments with the various representations.

A version of the Church encoding for lists without additional constructors can be expressed directly in Clean.

```
:: ChurchList a r := r (a r→r) → r

cnil :: (ChurchList a r)
cnil = λn c.n

ccons :: a (ChurchList a r) → (ChurchList a r)
ccons a x = λnil cons.cons a (x nil cons)

ctail :: (ChurchList a (r,r)) → (ChurchList a r)
ctail xs = λn c.fst (xs (undef,n) (λa (x,y).(y,c a y)))

clToStrings::(ChurchList a [String]) → [String]|toString a
clToStrings xs = xs ["[]"] λa x.[toString a, ": ": x]
```

Although these function are accepted by the compiler, there are severe limitations to this approach. The type system rejects many combinations of these function. This is due to the monomorphism constraint on function arguments. These problems are very similar those encountered by the function `swap'` in Section 3.2.

The Hindley-Milner type system does not accept the Scott encoding of data types without additional constructors, see Barendregt [3] or Barendsen [4] for a proof. Geuvers shows that this can be typed in $\lambda 2\mu$: $\lambda 2$ + positive recursive types [14]. Nevertheless, these functions work correctly in an untyped world with higher order functions.

```
snil      = λnil cons.nil
scons a x = λnil cons.cons a x
stail xs  = xs undef (λa x.x)
```

5.1 Using the Structure of the Type Representations

The destructors of the implementations of data constructors based on Church numbers are all very expensive operations, typically $O(n)$ where n is the size of the recursive data structure. When the shape of the computation matches the recursive structure of the Church encoding we can achieve an enormous optimisation by replacing definitions based on the interface provided by the type constructor classes by direct implementations. Since the encoding based on Church numbers is basically a `foldr`, as explained by Hinze in [16], these optimisations will work for manipulations that can be expressed as a `foldr`.

5.2 Peano Numbers

Many operations on Peano numbers can be expressed as a fold operation. For instance, a Peano number of the form `λzero succ.succ (succ .. zero)` can be transformed to an integer by providing the argument 0 for `zero` and the increment function, `inc`, for integers for `succ`. For the same operation on number in the Scott encoding we need to specify the required recursion explicitly. This is reflected in the tailor made instances of the class `toInt` for these number encodings.

```
instance toInt CNum where toInt (CNum n) = n 0 inc
instance toInt SNum where toInt (SNum n) = n 0 (inc o toInt)
```

The complexity of both transformations is $O(n)$. For the Church encoding this is a serious improvement compared to using `NumToNum` to transform a `CNum` to `Int`. Due to the $O(n)$ costs of `pred` for `CNum` the complexity of this transformation is $O(n^2)$. For the other encoding we can at best gain a constant factor. This can be generalised in a translation of `CNum` to other instances of `Num`.

```
CNumToNum :: CNum → n | Num n
CNumToNum (CNum n) = n Zero Succ
```

Similar clever definitions are known for the addition and multiplication of Church numbers. We express the optimised addition as an instance of the operator `+` for `CNum`. We achieve addition by replacing the `zero` of `x` by the number `y zero succ`. In exactly the same way we can achieve multiplication by replacing the successor such of `x` by `λz.y z succ`.

```
instance + CNum where
 (+) (CNum x) (CNum y) = CNum λz s.x (y z s) s
instance * CNum where
 (*) (CNum x) (CNum y) = CNum λz s.x z (λz2.y z2 s)
```

It is obvious that this reduces the complexity of these operations significantly. However, the addition is not the constant $O(1)$ manipulation it might seem to be. The result is a function and any application determining a value with this function will be $O(n \times m)$ work. This is of course a huge improvement to $O(n^2 \times m)$ for the addition for `CNum` using the function `add` from Section 4.2.6.

Those optimisations are only possible when the structure of the manipulation can be expressed by the structure of the encoding of `CNum`. No solutions of this kind are known for operations like predecessor and subtraction. For the predecessor it might look attractive to transform the encoding from `CNum` to `SNum` and perform the predecessor here in $O(1)$ instead of in $O(n)$ in `CNum`. Unfortunately, this transformation itself is $O(n)$, even using the optimised `CNumToNum` function. Nevertheless, such a transformation is worthwhile when we need to do more operation with higher cost in the `CNum` encoding that in the `SNum` encoding. This occurs for instance in subtraction m from n by repeated applications of the predecessor function, here the complexity drops from $O(n \times m)$ to $O(n + m)$. This is still more expensive that the $O(m)$ for the other encodings.

5.3 Lists

The Church encoding of lists is based on a fold-right. Also the Parigot encoding contains such a fold. By making an optimised fold function instead of the simple recursive version from Section 4.3.5 we can take advantage of this representation.

```
class foldR_o t :: (a b→b) b !(t a) → b

instance foldR_o CList where
 foldR_o f r (CList l) = l r f

instance foldR_o PList where
foldR_o f r (PList l) = l r λa t x.f a x
```

```
instance foldR_o SList where
 foldR_o f r (SList l) = l r λa x.f a (foldR_o f r x)

instance foldR_o []     where
 foldR_o f r l = case l of []=r; [a:x]=f a (foldR_o f r x)
```

For the Church and Parigot encoding of lists we directly use the given function `f` the the fold of this representation. The Scott encoding and the native lists of Clean does not have such a direct fold. Hence, we define an explicit recursive function.

5.4 Effect of the Optimisations

In order to determine the effect of the optimised fold implementation we replaced the functions `append` and `filter` in the function `qs` by their fold based variant shown above.

```
qs_o :: (t a) → (t a) | <, == a & foldRo, List t
qs_o l | isNil l
    = Nil
    = appendo (qs_o (fltro (λx.x < h) l))
              (appendo (fltro (λx.x == h) l)
                 (qs_o (fltro (λx.h < x) l))) where h = head l

append_o :: (t a) (t a) → t a | foldRo, List t
append_o l1 l2 = foldR_o Cons l2 l1

fltr_o :: (a→Bool) (t a) → t a | foldRo, List t
fltr_o p l = foldR_o (λa x.if (p a) (Cons a x) x) Nil l
```

The results of this experiment are listed in Figure 4.

qs_o (take m randomInts))				
m	CList Int	SList Int	PList Int	[Int]
1	$3.2\,10^{-7}$	$4.5\,10^{-7}$	$3.5\,10^{-7}$	$1.1\,10^{-7}$
10	$5.8\,10^{-6}$	$6.2\,10^{-6}$	$6.5\,10^{-6}$	$2.1\,10^{-6}$
10^2	$1.2\,10^{-4}$	$1.3\,10^{-4}$	$1.3\,10^{-4}$	$4.7\,10^{-5}$
10^3	$1.7\,10^{-3}$	$1.9\,10^{-3}$	$1.9\,10^{-3}$	$7.2\,10^{-4}$
10^4	$1.8\,10^{-2}$	$2.0\,10^{-2}$	$2.0\,10^{-2}$	$7.9\,10^{-3}$
10^5	$2.4\,10^{-1}$	$2.9\,10^{-1}$	$2.9\,10^{-1}$	$1.0\,10^{-1}$

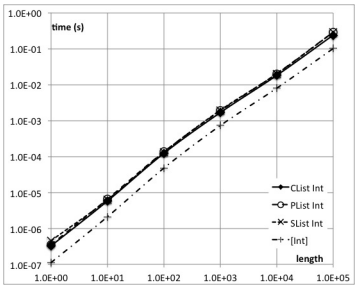


Figure 4. Execution time in seconds as function of the length for the encodings of `List`.

When all recursive list operations are replaced by an optimised fold-right all encodings of lists show very similar execution results. The small differences are explained by the additional arguments that have to be passed around in the Parigot encoding, and the additional constructors needed by this simulation of the Scott encoding.

These gains work only properly when every list manipulation is a fold-right. Introducing a single other operation can completely spoil the performance. Consider for instance a somewhat different formulation of our Quick-sort algorithm.

```
qs_o2 :: (t a) → (t a) | <, == a & foldRo, List t
qs_o2 l | isNil l
    = Nil
    = appendo (qs_o2 (fltro (λx.x < h) t))
      (Cons h (qs_o2 (fltro (λx.h ≤ x) t)))
where h = head l; t = tail l
```

The measurements in Figure 5 show that behaves similar to fold-based Quick-sort for most encodings. For small lists two instead of three filters over the list yields a gain. For long lists there will be many duplicates of the numbers between 0 and 999, hence the additional of equal elements yields a small gain. The overal complextity is $O(n\ \log n)$. For the Church-lists however, the single `tail` is a complete party breaker. For the same reasons as before the complexity is $O(n^3)$ in this Church encoding.

qs_o2 (take m randomInts))				
m	CList Int	SList Int	PList Int	[Int]
1	$2.1\,10^{-7}$	$2.2\,10^{-7}$	$1.8\,10^{-7}$	$7.2\,10^{-8}$
10	$1.2\,10^{-5}$	$4.3\,10^{-6}$	$3.7\,10^{-6}$	$2.2\,10^{-6}$
10^2	$5.6\,10^{-3}$	$9.4\,10^{-5}$	$9.4\,10^{-5}$	$5.8\,10^{-5}$
10^3	$5.6\,10^{0}$	$1.5\,10^{-3}$	$1.5\,10^{-3}$	$1.1\,10^{-3}$
10^4	$6.0\,10^{+3}$	$2.3\,10^{-2}$	$2.3\,10^{-2}$	$1.8\,10^{-2}$
10^5		$9.1\,10^{-1}$	$8.2\,10^{-1}$	$5.8\,10^{-1}$

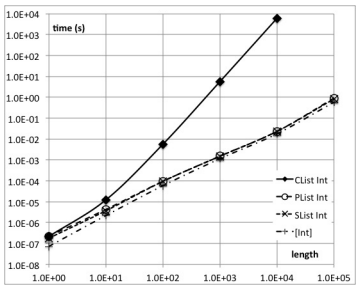


Figure 5. Execution time in seconds as function of the length for the encodings of `List`.

5.5 Optimization of other Operations

This kind of optimisation works only for operations that can be expressed as a fold-right. This implies that we cannot use it for many common list manipulations like, fold-left (`foldl`), `take`, `drop`, and insertion in a sorted list. For many operations that require a repeated application of `tail` it is worthwhile to transform the `CList` to a `SList`, perform the transformation on this `SList`, and finally transform back to the `CList` when we really want to use the Church encoding. This route via the lists in Scott encoding still force the evaluation of the spine of the entire list.

In some programming tasks it is possible to construct also in the Church encoding an implementation that executes the given task with a better complexity than obtained by applying the default deconstructs. For instance, the function to take the first `n` elements of a list using the functions from the class `List` is:

```
takeL :: Int (t a) → t a | List t
takeL n xs | n > 0 && not (isNil xs)
    = Cons (head xs) (takeL (n - 1) (tail xs))
    = Nil
```

For the Church encoding this has complexity $O(n \times L)$ where n is the number of elements to take and L is the length of the list. The complexity for the Scott encoding is $O(n)$, just like a direct definition in Clean. In contrast with the Church encoding, the Scott encoding is not strict in the spine of the list. Using a tailor-made instance of the fold in the Church encoding, the complexity of the take function can be reduced to $O(L)$:

```
takeC :: Int (CList a) → CList a
takeC n (CList xs)
 = fst (xs (Nil, xs 0 (λa x.x + 1) - n)
           (λa (ys, m).(if (m > 0) Nil (Cons a ys), m - 1)))
```

The expression `xs 0 (λa x.x+1)` computes the length of the lists. The outermost fold produces $length\ xs - n$ times `Nil`. When the counter `m` becomes non-positive, the fold starts copying the list elements. For finite lists, this is the same result as `takeL m`. It is not obvious how to derive such an optimised algorithm from an arbitrary function using the primitives from a class like `List`. Hence, constructing an optimised version requires in general non-trivial human actions. Even when this problem would be solved, the Scott encoding still has a better complexity and more appealing strictness properties.

5.6 Deforestation of Lists

Function fusion is a program transformation that tries to achieve the result of two separate functions f and g applied after each other by a single function h. That is we try to generate a function h such that $f \cdot g \equiv h \Rightarrow \forall x\,.\,f(g\ x) = h\ x$. Especially when fusion manages to eliminate a data structure this transformation will reduce the execution time significantly.

Wadler introduced a special form of fusion called deforestation [33]. In deforestation we recognise *producers* and *consumers*. For lists, any function forcing evaluation and processing the list like `foldr` is a good consumer. A producer is a similar recursive function that generates a list. When there is an immediate composition of a producer and a consumer these functions can be fused and the intermediate list is not longer needed. That is a function composition like `foldR g r (foldR (Cons o f) Nil xs)` can be fused to `foldR (g o f) r xs`. This is a standard transformation in many advanced compilers for functional programming languages, e.g. the GHC as described by Gill et. al. [15].

Since any `CList` is essentially a `foldR` that forces evaluation, all Church lists are good consumers. When the list is generated by a `foldRC` the function applied to as `Cons`-constructor can be written as `Cons o f`. Hence functions like `mapC` are good producers. This implies that there will be relatively many opportunities for deforestation in a program based on Church lists. Although the gain of deforestation can be a substantial factor, it does not change the complexity of the algorithm.

5.7 Optimisation of Tree Manipulations

The results from lists immediately carry over to trees. Any fold over a tree in the Church encoding can be optimised by to call of `foldTC`.

```
foldTC :: (b a b→b) b (CTree a) → b
foldTC f r (CTree t) = t r f
```

Using this fold we can collect all elements in a search tree by a single in-order traversal of the Church tree in a Church list by `inorderC`.

```
inorderC :: (CTree a) → CList a
inorderC t = foldTC (λx a y.appendC x (Cons a y)) Nil t
```

This works only for tree manipulations that can be expressed efficiently by a fold over the tree. As a consequence it does not solve the complexity problems for insertion, lookup and deletion from binary search trees.

6. Summary

For simple implementations of functional programming languages it is convenient to transform the data types to functions. By translating all data types to functions, we only need to implement functions on the core level in our implementation. The implemented language still provides algebraic data types like lists and trees to the user, but there is no runtime notion of these types needed. Such transformations are well known in λ-calculus. In this paper we use a language with named functions and basic types like numbers and characters, instead of pure λ-calculus. The named functions enable us to use recursion in an easier way than in the λ-calculus. In this paper we showed and compared three different implementation strategies for recursive data types by functions. The first strategy is an extension of the well known Church numerals. These Church numerals are treated in nearly all introductory texts about the λ-calculus. The second encoding is based on an idea originating from unpublished notes of Scott. Due to the lack of a good reference, this encoding is reinvented several times in history. The third encoding in a combination of the previous two known as the Perigot encoding.

The difference between these encodings is in the way they handle recursion. In the Church encoding the functions representing the data type contain a fold-like recursion pattern to process the list. In the Scott encoding the recursive manipulations are done by a recursive function that resembles the recursive functions in ordinary functional programming languages much closer.

By using some additional constructors we were able to implement instances of a type constructor class capturing the basic operations of lists or trees for an algebraic data type, an encoding based on Church numerals and an encoding of Scott numbers.

The comparison shows us that the functions producing the recursive branch in a constructor, like the tail of a list or a subtree of in a binary tree, are troublesome in the Church encoding. These operations become spine strict in the recursion. This undesirable strictness of the Church encoding ruins lazy evaluation and gives the selection operators an undesirable high complexity. The amount of work to be done is proportional to the size of the data structure instead of constant. This is caused by fold-based formulation of the selector functions. In the Scott encoding the selectors are simple non recursive λ-expressions, hence they do not have the strictness and complexity problems of the Church encoding.

The complexity problems of the Church representation are increased by the fact that the reduction of a recursive selector is not shared in graph reduction. A selector in the Church representation is a higher order function that needs the next manipulation as argument before it can be evaluated.

When the manipulation used is essentially a fold it is possible to optimise the functions implementing the data structure in the Church encoding to achieve the required complexity. For manipulations that are not a fold, the fold-like recursion pattern enforced by the Church encoding really hampers. Since many useful programs are not only executing folds over their recursive data structures, we consider the Church encoding of data structures harmful for the implementation purposes discussed in this paper.

The Perigot encoding contains both a Scott encoding and a Church encoding. Compared with the Church encoding it solves the complexity problems of selecting the recursive branch and it prevents the undesired strict evaluation. However, the Perigot encoding does not bring us the best of both worlds. The additional effort and space required to maintain both encodings spoils the potential benefits of the native fold-right recursion compared with Scott encoding.

Acknowledgments

Special thanks to Peter Achten from the Radboud University for useful feedback on draft versions of this work and stimulating discussions. Aaron Stump form the university of Iowa stimulated us to look again at the Parigot encoding. The feedback of anonymous referees helped us to improve the reading frame of this paper.

References

[1] M. Abadi, L. Cardelli, and G. D. Plotkin. Types for the scott numerals. Unpublished note, 1993. URL `http://lucacardelli.name/Papers/Notes/scott2.pdf`.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985. ISBN 9780080933757.

[3] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. ISBN 9780521766142.

[4] E. Barendsen. An unsolvable numeral system in lambda calculus. *J. Funct. Program.*, 1(3):367–372, 1991.

[5] A. Berarducci and C. Böhm. Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science*, 39 (820076097):135–154, 1985.

[6] A. Berarducci and C. Böhm. A self interpreter of Lambda-calculus having a normal form. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M. M. Richter, editors, *Computer Science Logic. 6th Workshop, CSL '92*, volume 702 of *LNCS*, pages 85–99. Springer, 1993. ISBN 978-3-540-56992-3.

[7] C. Böhm. The CUCH as a formal and description language. In T. Steel, editor, *Formal Languages Description Languages for Computer Programming*, pages 179–197. North-Holland, 1966.

[8] C. Böhm and W. Gross. Introduction to the CUCH. In E. Caianiello, editor, *Automata Theory*, pages 35–65, London, UK, 1966. Academic Press.

[9] C. Böhm, A. Piperno, and S. Guerrini. Lambda-definition of function(al)s by normal forms. In D. Sannella, editor, *ESOP*, volume 788 of *LNCS*, pages 135–149. Springer, 1994. .

[10] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 136–143. ACM, 1980.

[11] J. Crossley. Reminiscences of logicians. In J. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 1–62. Springer, 1975. ISBN 978-3-540-07152-5.

[12] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II*. North-Holland, 1972.

[13] J. Fairbairn and U. of Cambridge. Computer Laboratory. *Design and Implementation of a Simple Typed Language Based on the Lambda-calculus*. Computer Laboratory Cambridge: Technical report. University of Cambridge, Computer Laboratory, 1985.

[14] H. Geuvers. The Church-Scott representation of inductive and coinductive data. Types 2014, Paris, Draft, 2014. URL `http://www.cs.ru.nl/~herman/PUBS/ChurchScottDataTypes.pdf`.

[15] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232. ACM, 1993.

[16] R. Hinze. Theoretical pearl church numerals, twice! *J. Funct. Program.*, 15(1):1–13, Jan. 2005. ISSN 0956-7968.

[17] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):1–, 1992.

[18] J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Revised Selected Papers of the 7th TFP'06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.

[19] J. Jansen, P. Koopman, and R. Plasmeijer. From interpretation to compilation. In Z. Horváth, editor, *Proceedings of the 2nd CEFP'07*, volume 5161 of *LNCS*, pages 286–301, Cluj Napoca, Romania, 2008. Springer.

[20] J. M. Jansen. Programming in the λ-calculus: From Church to Scott and back. In P. Achten and P. Koopman, editors, *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer, 2013. ISBN 978-3-642-40354-5.

[21] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1987.

[22] W. Kluge. *Abstract computing machines: a lambda calculus perspective*. Texts in theoretical computer science. Springer, 2005. ISBN 3-540-21146-2.

[23] P. W. M. Koopman, M. C. J. D. V. Eekelen, and M. J. Plasmeijer. Operational machine specification in a functional programming language. *Software: Practice and Experience*, 25(5):463–499, 1995. ISSN 1097-024X.

[24] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[25] T. Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[26] M. Naylor and C. Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In O. Chitil, Z. Horváth, and V. Zsók, editors, *IFL*, volume 5083 of *LNCS*, pages 129–146. Springer, 2007. ISBN 978-3-540-85372-5.

[27] M. Parigot. Programming with proofs: A second-order type theory. In *Proc. ESOP '88*, LNCS 300, pages 145–159. Springer, 1988.

[28] M. Parigot. Recursive programming with proofs. *Theor. Comput. Sci. 94*, pages 335–336, 1992.

[29] S. L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201. ACM, 1989. ISBN 0-89791-328-0.

[30] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

[31] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *Proceedings of the ICFP'07*, pages 141–152, Freiburg, Germany, 2007. ACM.

[32] J. Steensgaard-Madsen. Typed representation of objects by functions. *ACM Trans. Program. Lang. Syst.*, 11(1):67–89, Jan. 1989. ISSN 0164-0925.

[33] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

Bidirectional parsing

a functional/logic perspective

Peter Kourzanov
NXP Eindhoven/TU Delft, Netherlands
kourzanov@acm.org

Abstract

We introduce PURE[3], a pure declarative approach to implementing declarative transformations with declarative tools. This Domain-Specific Language (DSL), inspired by the Definite Clause Grammar (DCG) and Parsing Expression Grammar (PEG) formalisms, is implemented using the Revised[5] Report on the Algorithmic Language Scheme (R5RS). Thanks to its use of the MINIKANREN logic programming system it supports fully reversible and extensible syntax-semantics relations. In this paper we highlight the usability and simplicity of PURE[3]'s approach, address the problem of *left-recursion* and show how its features help in defining custom and extensible typing systems for JavaScript Object Notation (JSON).

Categories and Subject Descriptors D.1.6 [*Programming techniques*]: Logic programming; D.3.2 [*Language Classifications*]: Applicative (Functional) languages; D.3.4 [*Processors*]: Parsing

General Terms (embedded) domain specific languages, automatic program generation, type systems/checking/inferencing

1. Introduction

The declarative approach to programming unifies logic/relational and functional communities in the shared vision of tools that need only be told *what* should be done rather than *how* that must be accomplished.[1] Ideally, these tools should (inter)actively participate in the creative process, i.e., *art* of computer programming by performing *parsing* of the human input, *checking* and *inference* [DM82] of the various properties of such inputs,[2] manipulation, refactoring and optimization of programs [BD77], compilation to machine code [App06] and last but not least, should provide feedback to the user.

Because humans inevitably are still in the loop of this development cycle, it is important that each stage remains palpable - that is, can be understood *semantically* and manipulated using *syntactically* simple terms. First and foremost this concerns parsing, a well-researched domain where many well-established methods exist [ALSU06] and [GBJL02], and yet, very few practical tools possess that elusive mathematical elegance that can immediately appeal to practitioners. Further down the transformation chain, complexity quickly rises and at the level of inference already presents formidable challenges [Wel94] to human understanding.

In this paper we present PURE[3] as a declarative approach to declarative *transformations* using declarative tools. The focus is on parsing as a particular kind of transformation of a linear stream of tokens into a set of Abstract Syntax Tree (AST) instances containing terminals (literal values), non-terminals (expressions, statements), types, assembly etc.

The approach is declarative in that we take the Backus-Naur Formalism (BNF) as a starting point and do not restrict ourselves to a specific way of codifying it. The transformations are declarative because they stay largely independent from the evaluation strategy. The tools are declarative in that we take MINIKANREN [FBK], a logic programming system embedded in R5RS [ABB+98] and abstain from overusing the extra-logical features that are available.

We shall first use a running example of an expression grammar/parser to explain our technical contributions and then switch to an *extensible* JSON grammar/parser to successively illustrate parsing, type checking, type inference and generation of syntax-semantics pairs constrained by types, all within a single framework.

Our main contributions are: the clean-room declarative implementation of PURE[3] (using the hygienic `syntax-rules` macro system and MINIKANREN) relying on *naturally* declarative semantics:

- featuring logical laziness,
- (full) reversibility-by-default,
- on-line behavior for left-recursion, and
- binding schemes for controlled (weak) hygiene "breaking"

This paper is structured as a flow that first addresses the background aspects in the introduction, explains the ideas and the implementation of the new formalism in section 2 and then highlights the use of the formalism by specifying an admittedly simple, yet concise and flexible typing system in section 3. The problems in implementing and using extensible transformations are addressed in section 4. Related work is reviewed in section 5, while the conclusions can be found in section 6. Full implementation using BIGLOO and conforming to R5RS plus two relevant SRFI libraries is available at github [Kou].[3]

1.1 Definite Clause Grammars

DCG is a technique originating in Prolog that allows one to embed a parser for a context-sensitive language into logic programming,

[1] we set machine learning community aside for now

[2] this is commonly known as *type*-checking and *type*-inference

[3] note that our use of R5RS is flavored by macro-expressible pattern-matching as well as a few syntactic liberties for recursive (`def`) and non-recursive (`defn`) bindings, brackets and lexical syntax (viz. *reader-macros*)

via Horn clauses. Logic programming languages such as Prolog and MINIKANREN also support relational programming. Instead of *functions* and *procedures* there are *predicates* that specify relations between terms. Rather than enforcing a particular way of evaluation, these languages specify a *resolution* (i.e., a search) procedure that can be applied and controlled in many ways. We explain the way how this is done in MINIKANREN in section 2.1. These features imply that a carefully designed grammar/parser can be run forwards (i.e., generating semantics from syntax), backwards (i.e., generating syntax from semantics) and sideways (e.g., constrained generation of syntax-semantics pairs).

A particularly nice feature of DCGs is its declarative nature and yet *executable* semantics [PW80]. This can be seen in the BNF specification as well as in the following Prolog code for a trivial context-free grammar/recognizer with precedence below.

```
<factor> ::= <literal> | <factor> '^' <literal>
<term> ::= <factor> | <term> '*' <factor>
                    | <term> '/' <factor>
<expr> ::= <term> | <expr> '+' <term>
                  | <expr> '-' <term>
```

Assuming a suitable definition of the `literal` predicate, the BNF can be automatically converted to the corresponding Prolog DCG rules, or, as shall be shown in section 2.2, to R5RS and MINIKANREN using the `syntax-rules`. Both kinds of encodings are "almost" directly executable, modulo *left-recursion* - a problem that plagues many recursive descent systems, and which we address by a novel technique of *logical laziness* in section 2.4.

```
%% An ideal Prolog DCG for a trivial expression grammar
factor --> factor, [^], literal.
factor --> literal.
term --> term, [*], factor.
term --> term, [/], factor.
term --> factor.
expr --> expr, [+], term.
expr --> expr, [-], term.
expr --> term.
```

As shall be become apparent shortly, the DCGs are more powerful than just Chomsky Type-2 systems (context-free grammars, or non-deterministic push-down automata) and in fact can express attribute grammars by allowing the predicates to take arguments that are used to compute variables bottom-up (i.e., a feature identical to synthesized attributes) or to generate and pass around non-instantiated variables (i.e., a feature identical to inherited attributes). This opens the door to concise [FBK05], declarative specification of *typing* systems, relational interpreters [Byr10] as well as a way towards a *practical* DSL for bidirectional transformations.

1.2 Parsing Expression Grammars

This grammar formalism [For04] dispenses with complexities of LL/LR grammars, takes a step back to recursive descent, and then extends it with a few combinators inspired by Type-3, *regular* languages. In addition, the PEG formalism introduces syntactic *and*- and *not*-predicates as well as *prioritized* choice (used for grammar *disambiguation*). This is an improvement over plain recursive descent because explicit recursion is often avoided (by turning it into *primitive* recursion via the Kleene-$*$ and $+$ operators).

One nice aspect of PEGs is better surface syntax for common patterns of programming parsers and transformations. For example, the `expr` predicate from the section above can be concisely specified as the following recognizer.

```
(pcg expr ([] ⇔ [term] [(['+ / '-] : [term]) *]))
```

Looking ahead, we might define a recognizer for a context-sensitive language (using our `pcg` rules introduced in the next section) with PEG combinators and syntactic predicates as follows:

```
;; A context-sensitive grammar with PEG combinators
(defn a^n b^n a^n (pcg ⇔ S
  (S ([] ⇔ when([A] 'a) ('a +) [B] unless(['a / 'b])))
  (A ([] ⇔ 'a ([A] ?) 'b))
  (B ([] ⇔ 'b ([B] ?) 'a))
  ))
```

It is apparent that PEGs and DCGs share many of the same benefits and shortcomings. Syntactic predicates as well as the prioritized/ordered choice of PEG are naturally expressible as *committed* choice in logic programming. Left-recursion, however, is still troublesome and has to be either *avoided*, *eliminated* or solved by *ad-hoc* methods such as *cancellation tokens*, *curtailment* or *memoing* (see section 5).

2. Parsing Clause Grammars

In this chapter we introduce our implementation of DCGs which we dub the Parsing Clause Grammar (PCG) as a tribute to the other source of inspiration, the PEG. First, we introduce MINIKANREN and provide a way to specify first-order *predicates* concisely using only the `syntax-rules` of R5RS. Then we show a few examples of the usefulness of *higher-order* predicates. Controlled access to hygiene (i.e., weak hygiene "breaking") is then used to let `syntax-rules` implement an equational theory for name bindings across disparate code fragments. Finally, we highlight PCG's support for left-recursion in a pure, on-line fashion.

We make use of the macro-expressible pattern-matching (introduced in [KS13]) in BIGLOO - a practical implementation of R5RS [SW95]. The Scheme reader is extended with reader macros via `set-sharp-read-syntax!` (see the Chicken Scheme wiki: Unit library [Sch]) and provides a handler for `#h` form (see section 4 for a few examples) in order to obtain a stream of lexical tokens unconstrained by the conventional Scheme syntax.

```
;; A recognizer, each clause a separate predicate
(pcg Factor
   ([] ⇔ [Factor] ^ [literal])
   ([] ⇔ [literal]))
(pcg Term
   ([] ⇔ [Term] * [Factor])
   ([] ⇔ [Term] / [Factor])
   ([] ⇔ [Factor]))
(pcg Expr
   ([] ⇔ [Expr] + [Term])
   ([] ⇔ [Expr] - [Term])
   ([] ⇔ [Term]))
```

The translation of the expression grammar from the previous section is straightforward with the `pcg` macro and is given above. It defines several clause groups and binds a given name to the predicate/function implementing a disjunction for each group. Please see the code in section 2.4 for an illustration of MINIKANREN code that is automatically generated for the `Expr` part of this recognizer.

- we assume that the Scheme `read` procedure has performed lexical analysis on the input, that is, we deal only with syntactic and semantic analysis of tokens produced by the reader
- BNF *terminals* are assumed to be interned Scheme atoms such as literals (`#true` and `#false`), numbers, "strings" and `'symbols`, which might include characters such as `([,]{.})` when wrapped in `|`vertical bars`|`. Terminals are *auto-quoted*.
- BNF *non-terminals* are translated to MINIKANREN predicates (which are just regular, pure Scheme functions), where the first

two arguments represent PCG monadic state, the *difference-list*. Note that unlike original DCGs we *prepend* the pair of `Lin`/`Lout` variables comprising the diff-list at the beginning of the argument list because our predicates are possibly *variadic*

2.1 Declarative logic programming with MINIKANREN

In this section we briefly introduce the way in which we use MINIKANREN's primitives [FBK05] such as *success* and *failure* (`#s` and `#u`), binding of logic variables (`fresh`), unification ($\equiv$), disjunctions (fair choice `conde`, *soft-cutting* `conda`, committed choice `condu`), conjunctions (`all`), impure predicates (`project` for reifying variables) and finally `run`/`run*` that provide the interface between Scheme and the non-determinism monad that lies at the heart of MINIKANREN.

```
;; the swiss army knife of logic programming
(def append⁰ (predicate
 ([_ `() b b])
 ([_ `(,x . ,a1) b `(,x . ,c1)] :- [append⁰ a1 b c1])
))
```

Predicates are introduced by either `predicate` or `pcg` macros (these share many design aspects), and may have many clauses inside. Each clause contains a *head* followed by an optional *body*. We borrow the syntax from Prolog, separate the head from the body by a (:-) form and introduce an *implicit* disjunction between all clauses. By convention shared with `syntax-rules`, `predicate` clause heads may begin with any *tag* identifying the clause or with just a wildcard `[_]` while `pcg` clause heads (e.g., for recognizers) may be empty `[]`, in which case they don't unify any passed arguments. If the `pcg` head is not empty but contains only the `[_]` tag then the (thus variadic) predicate will unify exactly one argument with each consumed token in the input, point-wise.

```
;; e is somewhere in t ;; using explicit disjunction
(def member⁰ (predicate  (def member⁰ (predicate
 ([_ e `()] :- #u)         ([_ e `()] :- #u)
 ([_ e `(,e . ,t)])        ([_ e `(,h . ,t)] :-
 ([_ e `(,h . ,t)] :-       ([≡ e h] / [member⁰ e t]))
  [member⁰ e t])))          ))
```

By design shared with MINIKANREN, juxtaposition of goals (in the body) and clause attributes (in the head) corresponds to the conjunction. As is observed from 2 versions of the `member`^{0} predicate above, *explicit* PEG-style disjunction in clause bodies is often essential,[4] avoiding duplication of clause bodies and heads.

In contrast to MINIKANREN, PURE^3 advocates Prolog-style *automatic* inference of bindings. Unlike Prolog, however, in all `predicate` examples, variable names are extracted from clause heads and then are equated with the corresponding bindings from clause bodies using the Term-Rewriting System (TRS) equational theory that is explained in section 2.3.

Because of this, no binding can be used in a clause body without it being mentioned first in the clause head, which enforces fully reversible predicates which are "correct-by-construction". For some predicates, there may be *fresh* bindings introduced in the head but not used in the body (e.g., `fresh`^{0} in section 2.4) or there may be bindings (see `locals:` spec) that are not explicitly named in the head but used in the body to build some synthesized attribute that is mentioned in the head (e.g., the `prefix`^{0} in section 2.4)

2.2 Macro-expressibility of PCG rules

The `pcg` macro builds upon the structure introduced in the previous section and provides (1) *natural* representation of the syntax for terms of the expression grammar - to the right of $\Leftrightarrow$, (2) *natural* representation of semantics, i.e., an AST - to the left of $\Leftrightarrow$, (3) direct-style operator *associativity* and *precedence* and (4) inverse for free (note that we separate the clause head from the clause body by $\Leftrightarrow$ to indicate full reversibility). Our final version of a reversible syntax-semantics relation for expressions is given in figure 1.

```
(pcg
(Factor
 ([_ `(^ ,x ,y)] ⇔ [Factor x] ^ [literal y])
 ([_ x] ⇔ [literal x]))
(Term
 ([_ `(* ,x ,y)] ⇔ [Term x] * [Factor y])
 ([_ `(/ ,x ,y)] ⇔ [Term x] / [Factor y])
 ([_ x] ⇔ [Factor x]))
(Expr
 ([_ `(+ ,x ,y)] ⇔ [Expr x] + [Term y])
 ([_ `(- ,x ,y)] ⇔ [Expr x] - [Term y])
 ([_ x] ⇔ [Term x])
))
```

Figure 1. Pure, declarative PCG parser analyzer

The design of PCG (see figure 2) is centered around a set of `syntax-rules` macros: `seq` for processing clause bodies, `process-args` for clause heads, `predicate` and `pcg` that glue everything together.[5]

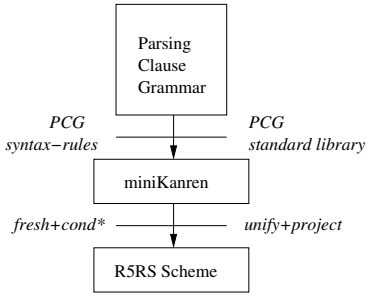


Figure 2. PURE^3 DSL architecture

The `seq` macro (see figure 5) implements the *threading* of a difference-list, per-clause sub-goal sequencing, introduction of a new logical temporary for each step and dispatching on the *shape* of forms encountered as sub-goals (non-terminals, quasi-data, atoms, escapes, ϵ, PEG combinators). By the very nature of hygienic `syntax-rules`, both components of the difference list (i.e., monadic state bindings) as well as all logical temporaries can not leak to user code, making the PCG formalism safe. This macro also performs a few optimizations such as skipping the introduction of a new logical temporary at the end of the sub-goal list.

Since each temporary is introduced by a different invocation of the `seq` macro, and yet 2 bindings get referred to by the generated code at each step (see section 2.4 for an example), an expander[6] compatible with the Scheme Request for Implementation (SRFI)#46: "Basic Syntax-rules Extensions" [Cam05] shall automatically rename it, while gratuitous bindings thus introduced shall be removed by the BIGLOO compiler's constant β-reduction pass, as they are immediately shadowed by the `fresh` binder. The `syntax-rules` therefore give us gratis, *pure*, declarative `gensym`!

Both `predicate` and `pcg` flavors of our `syntax-rules` macros support *named* (see the `pcg expr` from section 1.2) as well as

[4] note that our disjunction is pure (`conde`) by default. Soft-cut resp. committed/ordered choice are introduced explicitly by `*->` resp. `->` combinators

[5] due to space limits we can only refer to snippets of these in the appendix

[6] we use the ALEXPANDER library ported to, and integrated with BIGLOO as it still lacks a native and compatible `syntax-rules` expander

anonymous (see e.g., section 2.1) predicate abstractions. In addition, `pcg` macros allow specification of a group of possibly mutually recursive predicates where each is named and visible from the top-level (see figure 1), as well as a group where a distinguished predicate is selected as a *start* predicate (see the $\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n$ recognizer from section 1.2) with the rest hidden from the top-level.

2.2.1 Higher-order rules

Since MINIKANREN predicates are represented by normal Scheme functions, all the benefits of working in a Functional Programming (FP) language are retained. The `ne-list` predicate shown below supports repeated matching of the user-supplied `elem` predicate, with the literal represented by the value of the `comma` argument matched as a list separator.[7]

```
;; ... Passing functions into predicates ...
;; Monomorphic lists (for JSON), used in section 3.1
(defn [ne-list comma elem] (pcg ⇔ s
 (s ([_ `(,v)] ⇔ [elem v])
    ([_ `(,v . ,vs)] ⇔
      [elem v] [idem comma] [s vs])
)))
```

The recursion works out of the box for predicates such as `ne-list`, which employ right-recursion. However, some grammars such as the one from the notorious expression parser of figure 1, need to use left-recursion if the associativity of operators and the naturality of the parser representation is to be maintained. We shall present a "logical" solution for this problem in section 2.4.

```
;; ... Returning functions from predicates ...
;; Left-recursion avoidance (higher-order patching)
(pcg Factor
 ([_ π(λ (z) (y (if [null? z] x `[^ ,z ,x])))]
  ⇔ [literal x] ^ [Factor y])
 ([_ π(λ (z) (if [null? z] x `[^ ,z ,x]))]
  ⇔ [literal x])
)
```

An example of a "functional" solution would be left-recursion avoidance by returning functions from higher-order predicates [For02]. A `pcg` representation of the `Factor` fragment of the expression grammar illustrating this technique is shown above. Note the similarity of this to the emulation of `fold-left` by `fold-right` (see e.g., [Hut99]) and the technique introduced in [DG05]. The use of the impure `project` (π) form,[8] which requires that variables are *grounded* (i.e., instantiated), precludes the use of this predicate in reverse.

2.3 Breaking hygiene (look ma, no `gensym`)

In section 2.1 we explained *why* PURE^3 uses inference of logic variable bindings in order to promote (full) reversibility and to avoid code clutter by explicit `fresh` introductions (see section 2.4 for a convincing case). In this section we show *how* inference can be implemented in our `process-args` macro by "breaking" the weak hygiene of `syntax-rules`.

It is well known that the promise of `syntax-rules` never to cause the capturing of bindings (hygiene) can be subverted [Kis02]. The `extract` and `extract*` macros implementing the so-called Petrofsky's extraction are typically used to capture the bindings regardless of their *color* (scope information) and pass them further to the other macros. The feature of `syntax-rules` that makes this possible is the semantics of macro literals [ABB+98].

A first step towards an equational theory of name binding across disparate code fragments using a TRS consists of extracting the *free* variables from a tree of terms (`w syntax-rules` macro). We assume the *weak* hygiene where all bindings are intended to be local and are not redefined outside of the terms being processed. This is exactly the same assumption that `extract` macro makes (see [Kis02] for further details).

```
;; Ignoring (some) Scheme primitives
(def-syntax (scheme-bindings (k a b [s ...] . d))
   (k a b [s ... if cond begin null? list first second
               pair? car cdr + - * / ^ = ≡ : ...] . d))
```

Of course, all binders must be known to this macro (and names thus introduced must be skipped in appropriate scopes), in addition to all of the *eigen* primitives that must not be considered free in given terms. This is accomplished using the macro given above, which employs the macro-level Continuation Passing Style (CPS) [HF00] to bootstrap the `w` macro by including common Scheme primitives in a list of bindings already processed.

2.3.1 Handling attributes

In the process of generating `fresh` and `projected` predicate arguments for the inferred attribute bindings we need to make sure that the bindings given to the binder correspond to the bindings captured from the body. If there are no bindings then we default to some construct like `begin` or `all`. For `project`, we verify that all attributes are grounded and vacuously succeed otherwise.

```
;; Introducing the fresh and project binders
(def-syntax make-scopes (syntax-rules (project)
  ([_ _ _ _] #s) ;; nothing to do - just succeed
  ([_ _ () default . body] (default . body))
  ([_ project (var ...) _ . body]
     (project (var ...)
       (or (and (ground? var) ... . body) #s)
   ))
  ([_ binder vars _ . body]
   (let-syntax-rule ([K args terms]
     (binder args . terms))
      (extract* vars body (K [] body))
   ))
  ))
```

Now we're ready to complete the third step: attacking the `process-args` macro, which makes sure that the resolution of the synthesized attributes in the clause head, the clause body, logical actions, and finally - evaluation of `projected` code - are all scheduled appropriately. Correct sequencing is essential for reversibility - when run in reverse the synthesized attributes actually provide the inputs, and hence must resolve first. When running forwards, resolving them first does no harm, since clause heads can only indirectly employ constructs made available through the `process-args` implementation - unification using quasi-data and conjunction using `all` (disjunctions and recursion are *not* available inside clause heads). Projections (π) can only run in forward mode and always after resolving the clause body.

Reversible logical actions (not explained in this paper in detail, but see section 3.2 for a use-case) have to run after clause body terms when running forwards but before when run in reverse. Non-reversible actions, as implemented by escapes in the `seq` macro, are left to be explicitly scheduled by the user in clause bodies.

We ignore (inherited) attributes that are explicitly bound from outside when looking for free bindings, but do include them into the list of attributes to generate using the `make-scopes` macro. Each attribute in the clause head gets unified with the respective argument of the clause function,[9] while the final *void* attribute tail gets syntax-bound (i.e., renamed) to `'()` as is customary in Scheme variadic functions.

[7] via explicit lifting using the `idem` predicate, defined in the appendix

[8] the straightforward implementation of π is elided in this paper

[9] using a technique similar to the `seq` macro described above

Now we can bring together the full reversibility-by-default and attribute bindings inference (as implemented by the `seq` and `process-args` macros), and actually complete our TRS for the predicate clause forms as it is implemented by the `pcg/predicate` macro. Each clause is translated to a separate R5RS function which implements all aforementioned aspects of the corresponding predicate logic. Despite the seeming restriction that each clause is visible from the top-level, Scheme's `define` form actually is macro-expressible by the `letrec` binder when it precedes all other forms in a block. This is useful for implementing the `pcg` variants that hide internal predicates and expose a single starting predicate function to the top-level.[10]

Synthesized attributes Similar to the *S-attributed* grammars, where inherited attributes are not allowed, PURE^3 also is tuned for seamless expression of grammars with only synthesized attributes. In fact, all examples introduced so far used no `local:` attribute specifications, inferring the attributes in clause bodies from clause heads. Together with the ban on `project` (π), this enforces the fully reversible behavior in a "correct-by-construction" way.

Inherited attributes Having no possibility of generating and passing non-instantiated logic variables severely restricts the expressiveness of the formalism. There are examples of when such a strategy for implementing semantics is essential, e.g., predicates from section 3.2. Here we illustrate PURE^3's implementation of inherited attributes using a particular way of solving left-recursion by left-factoring that is commonly used in e.g., Prolog community.

```
;; Left-recursion elimination by left-factoring
(defn exprs (pcg ⇔ expr
(factor locals: (x)
 ([_ y] ⇔ [literal x] [factor' x y]))
(factor' locals: (y)
 ([_ x z] ⇔ ^ [literal y] [factor' `(^ ,x ,y) z])
 ([_ x x] ⇔ ϵ))
(term locals: (x)
 ([_ y] ⇔ [factor x] [term' x y]))
(term' locals: (y)
 ([_ x z] ⇔ * [factor y] [term' `(* ,x ,y) z])
 ([_ x z] ⇔ / [factor y] [term' `(/ ,x ,y) z])
 ([_ x x] ⇔ ϵ))
(expr locals: (x)
 ([_ y] ⇔ [term x] [expr' x y]))
(expr' locals: (y)
 ([_ x z] ⇔ + [term y] [expr' `(+ ,x ,y) z])
 ([_ x z] ⇔ - [term y] [expr' `(- ,x ,y) z])
 ([_ x x] ⇔ ϵ))
))
```

Left-factoring is usually understood as the process of introducing additional predicates for matching common prefix terms of a number of clauses and then factoring them out from the original predicates. Although most often used for optimization, this technique proves helpful in converting left-recursion into right-recursion. One has to be careful, however, not to change the associativity of the operators when applying left-factoring.

Let us implement left-recursion elimination using PCG. Here, the parent predicate (e.g., `expr'`) binds a fresh variable for the inherited attribute using the `locals:` spec and then passes it to the child predicate (e.g., `term`) which resolves the attribute and communicates it to its sibling. The recursive call then unifies the semantics with a newly formed AST node upon completion. Note that this bears strong resemblance to the *L-attributed* grammars.

Although PURE^3 definitely supports it, this approach does not possess naturally declarative semantics, since the semantic actions are now interwoven with the syntax. Also, similarly to left-recursion avoidance in section 2.2.1, the attribute resolution is delayed until the end of the input, which prohibits on-line, incremental parsing. Section 2.4 presents a better approach to left-recursion.

[10] we refer to our github for further details about `select` and `pcg`

2.3.2 Handling binding in combinators

Consider the semantics of Scheme's `(-)` function: it is left-associative and accepts non-zero number of arguments. One might define both the syntax and semantics of `(-)` using the PEG's $*$ combinator as follows, assuming the variadic handling of the operator in the AST:

```
;; Minus in Scheme has arity >= 1
(pcg -' ([_ `(- ,t . ,ts)]⇔[term t] [('- : [term ts])*]))
```

Note that this avoids explicit recursion. However, now the $*$ combinator has to collect all elements of the input matching the `term ts` predicate invocation into a *list*, while each invocation of the `term` predicate still unifies with a single term only. This implies that representing each attribute as it is being synthesized with a single logical variable is not sufficient. In fact, we need 4 logical variables for each attribute: one for returning the final result, one when matching on each `term`, one for the accumulator and another one for the intermediate results as needed for *looping* in the `seq` rule that implements the Kleene-$*$ operator.

2.4 Pure, on-line left-recursion

Now we're well-equipped to attack left-recursion in PCG rules by applying the technique of logical laziness. This problem arises due to the infinite regress when a predicate recurses while not having consumed anything from the input. Still, direct-style associativity prevents us from applying left-recursion elimination or avoidance while the need to support left-recursion in a pure, on-line and fully reversible fashion precludes usage of impure techniques such as curtailment or memoing. Looks like we're stuck.

```
;; Diverging R5RS code generated for Expr (+) clause:
(define _head_422 (λ (Lin Lout . vars)
  (fresh (y x) (≡ vars (cons (list '+ x y) '()))
     (fresh (_temp_505 temp)
        (Expr Lin temp x)
        (≡ temp (cons '+ _temp_505))
        (Term _temp_505 Lout y)
      ))))
;; ... _head_424 and _head_426 elided ...
(def Expr (λ vs (conde ((apply ([extend] 'Expr) vs))
 ((apply _head_422 vs)) ;; ([_ `(+ ,x ,y)] ⇔ ...)
 ((apply _head_424 vs)) ;; ([_ `(- ,x ,y)] ⇔ ...)
 ((apply _head_426 vs)) ;; ([_ x] ⇔ ...)
)))
```

Looking at the diverging generated code for the `Expr` grammar fragment from figure 1 (see above) we immediately observe the problem: the base case of this non-well-founded recursion (`_head_426`) is never reached. Fortunately, the problem has a surprisingly simple solution which we dub "logical laziness". It comprises several steps all of which are automated using the `pcg` and the `seq` - both pure and declarative `syntax-rules` macros.

1. identifying (mutually) recursive clauses (section 2.2)
2. marking of such clauses using the *lift* form (can also be explicitly marked by the user if needed, as e.g., in section 4.3)
3. *untying* the recursive knot and insertion of replacement calls to `append`^{0} (section 2.1) with dummies (variable `d` below)
4. delaying of the resolution and subsequent *dropping* of recursive calls at the very end (or very beginning) of a clause body
5. *tying* the knot by unifying the difference list with `[d '()]`

In essence, here we apply a *predicate transformation* where the order of predicate resolution is adapted to suit the resolution procedure. The grammar designer is not required to apply workarounds for left-recursion and can regain a high-level view as in figure 1, as long as the set of mutually recursive predicate clauses can be automatically identified by our pcg macro. The code below that is generated for the same grammar fragment exhibits the technique.

```
;; Reversible R5RS code generated for Expr clause:
;; ([_ `(+ ,x ,y)] ⇔ [Expr x] + [Term y])
(define _head_422 (λ (Lin Lout . vars)
 (fresh (y x) (≡ vars (cons (list '+ x y) '()))
   (fresh (_temp_505 temp d)
    (project (Lin) (if (ground? Lin) #s (Expr d '() x)))
    (append⁰ d temp Lin)
    (≡ temp (cons '+ _temp_505))
    (Term _temp_505 Lout y)
    (project (Lin) (if (ground? Lin) (Expr d '() x) #s))
))))
```

This technique has the advantage of maintaining both naturality of the grammar as well as full reversibility of the resulting parser. When the input Lin is grounded (i.e., the parser is running forwards), the recursive call is delayed to the very end of the clause, effectively making it *tail-recursive*. When the parser is running backwards (i.e., the input Lin is fresh), the recursive call has to run first in the clause, because otherwise the recursion becomes non-well-founded, this time due to semantic destructuring of the vars.

```
;; (prefixed) infinite streams of logic variables
(def (fresh⁰ x)
  (conde ([≡ x '()])
     (else (fresh (y z)
       (fresh⁰ z)
       (≡ x `(,y . ,z))
      ))))
(defn (prefix⁰ a b)
   (fresh (x)
      (fresh⁰ x)
      (append⁰ a x b)
      ))
```

```
(def fresh⁰ (predicate
   ([_ '()])
   ([_ `(,y . ,z)] :-
     (fresh⁰ z)
    )))
(defn prefix⁰
 (predicate locals: (x)
  ([_ a b] :- (fresh⁰ x)
    (append⁰ a x b)
  )))
```

Does this work as promised, in an on-line fashion? Lets generate an infinite input using predicates above (where both MINIKANREN and PURE^3 versions are given) and verify by running!

```
;; Parsing prefixed infinite input stream
(verify Expr (run 1 (q)
  (fresh (l)
     (prefix⁰ '(1 * 2 + 3 * 5) l)
     (Expr l '() q)))
 ===> (+ (* 1 2) (* 3 5)))
```

One disadvantage of this technique is the *non-determinism* (and thus a quadratic-time slowdown) introduced by the append^0. In practice, however, this is not problematic because non-determinism often is and/or can be easily constrained by putting the limit to the *lookahead* by tokens that immediately follow the recursive call.

3. Type systems a la carte

Having introduced all the tools necessary to attack a more practical problem of adding types to a fully reversible JSON parser, we now turn to figure 3, which depicts a type-free implementation of the JSON syntax.[11] This PCG is our starting point for this section.

The semantics is represented by the AST where JSON literals remain strings, symbols and numbers. Name-value pairs become List Processing (LISP) pairs, while arrays and objects turn into

[11] we rely on BIGLOO reader for symbol, string and number parsing

```
(pcg
 (json-symbol ([_ x] ⇔ [strings x]))
 (json-key = json-symbol)
 (json-number = number)
 (json-bool ([_] ⇔ ('true / 'false)))
 (json-value ([_] ⇔ 'null)
  ([_ x] ⇔ [json-bool x])
  ([_ x] ⇔ ([json-symbol x] / [json-number x]))
  ([_ x] ⇔ ([json-object x] / [json-array x])))
 (json-pair ([_ `(,n . ,v)] ⇔
  [json-key n] |:| [json-value v]))
 (json-value-list ([_ '()] ⇔ ϵ)
  ([_ l] ⇔ [(ne-list |,| json-value) l]))
 (json-pair-list ([_ '()] ⇔ ϵ)
  ([_ l] ⇔ [(ne-list |,| json-pair) l]))
 (json-array
  ([_ `(arr . ,es)] ⇔ |[| [json-value-list es] |]|))
 (json-object
  ([_ `(obj . ,es)] ⇔ |{| [json-pair-list es] |}|))
)
```

Figure 3. Type-free JSON

explicitly tagged lists of JSON values. This grammar is interesting because of this recursion and the presence of various data-types.

3.1 Type checking

Base Types		Pairs, Lists	
$\frac{}{\Gamma\vdash \mathbf{null}:U}$	(UNIT)	$\frac{\mathrm{T}^0=\{U,B,S,N\},S\times\mathrm{T}^1}{\mathrm{T}^1=\mathrm{T}^0\cup\{A(\mathrm{T}^1),O(S,\mathrm{T}^1)\}}$	
$\frac{x\in\{\mathbf{true},\mathbf{false}\}}{\Gamma\vdash x:B}$	(BOOL)	$\frac{\forall t{:}\mathrm{T}^1,\Gamma\vdash v_1,\ldots v_n{:}t}{\Gamma\vdash[v_1,\ldots v_n]{:}A(t)}$	(ARRI)
$\frac{x\in\mathrm{Strings}}{\Gamma\vdash x:S}$	(STR)	$\frac{\forall t{:}\mathrm{T}^1,\Gamma\vdash k{:}S,\Gamma\vdash v{:}t}{\Gamma\vdash(k,v){:}S\times t}$	(PAIRI)
$\frac{x\in\mathbb{R}}{\Gamma\vdash x:N}$	(NUM)	$\frac{\forall t{:}\mathrm{T}^1,\Gamma\vdash v_1,\ldots v_n{:}S\times t}{\Gamma\vdash\{v_1,\ldots v_n\}{:}O(S,t)}$	(OBJI)

Table 1. Typing rules for monomorphic JSON

Thanks to the availability of unification, addition of types to the grammars using PURE^3 is easy. In figure 4 on the left-hand side (i.e., the semantics), each clause is extended with an additional attribute, while the right-hand side (i.e., the syntax) is essentially not modified. The higher-order ne-list predicate (section 2.2.1) is not touched for the *monomorphic* lists, a variant for introducing types in JSON that ensures homogeneity of objects and arrays through *static* typing.

```
;; Monomorphic JSON lists (i.e., arrays and objects)
(pcg          ;; replace these in Figure 4. below
(json-value-list
  ([_ '() `(List ,t)] ⇔ ϵ)
  ([_ l `(List ,t)] ⇔
   [(ne-list |,| (_ json-value t)) l]))
(json-pair-list
  ([_ '() `(PList (,t1 ,t2))] ⇔ ϵ)
  ([_ l `(PList (,t1 ,t2))] ⇔
   [(ne-list |,| (_ json-pair `[Pair ,t1 ,t2])) l])
))
```

Conventional typing rules are given in table 1 while their translation to PURE^3 is a straightforward extension of rules from figure 3 with typed versions of json-value-list and json-pair-list predicates. The code above implements the ARRI, PAIRI and OBJI typing rules (the rest of the rules can be found in figure 4). We rely on *sectioning* to partially apply the json-value and json-pair predicates to known types, and to introduce type schemes (containing "fresh" types) for empty containers. The logical unification ensures type correctness by applying the same types throughout the lists once they are instantiated, leading to monomorphic containers.

```
(defn tjson-value (pcg ⇔ json-value
 (json-symbol extend: extend ([_ x 'Str] ⇔
  [strings x]))
 (json-key extend: extend ([_ x t] ⇔
  [json-symbol x t]))
 (json-number ([_ x 'Num] ⇔ [number x]))
 (json-value ([_ 'null 'Unit] ⇔ 'null)
  ([_ x 'Bool] ⇔ [json-bool x])
  ([_ x t] ⇔ ([json-symbol x t] / [json-number x t]))
  ([_ x t] ⇔ ([json-object x t] / [json-array x t])))
 (json-pair ([_ `(,n . ,v) `(Pair ,tn ,tv)] ⇔
  [json-key n tn] |:| [json-value v tv]))
 (json-value-list ([_ '() `(List ,t)] ⇔ ϵ)
  ([_ l `(List ,ts)] ⇔
   [(poly-ne-list |,| json-value) l ts]))
 (json-pair-list ([_ '() `(PList (,t1 ,t2))] ⇔ ϵ)
  ([_ l `(PList . ,ts)] ⇔
   [(poly-ne-list |,| json-pair) l ts]))
 (json-array ;; promote List to an Array
  ([_ `(arr . ,es) `(Array ,t)] ⇔
   |[| [json-value-list es `(List ,t)] |]|))
 (json-object ;; promote List to an Object
  ([_ `(obj . ,es) `(Object ,ts)] ⇔
   |{| [json-pair-list es `(PList . ,ts)] |}|))
))
```

Figure 4. Typed, extensible JSON

Note that we are making the `json-symbol` and `json-key` predicates extensible via `extend:` spec. This shall prove its usefulness in section 4 where we extend the set of JSON values by proper symbols and JSON keys by numbers and booleans while preserving the modularity and compositionality of the grammar above.

3.2 Type inference

The previous section has introduced type-checking to JSON for monomorphic containers denoting objects such as arrays of numbers or objects of pairs of strings. Such types can be either checked (provided they are given as instantiated attributes to the `tjson-value` predicate), or inferred (provided they are given as free logic variables). However, for homogeneous arrays and objects, inference in this context will typically mean that the type shall be derived using the first element (pair), with the rest of the elements simply checked against the already instantiated type.

$\mathrm{T}^2{=}\mathrm{T}^1{\cup}\{\sum(\mathrm{T}^2{,}\mathrm{T}^2)\}$	$\frac{\exists i{:}t{=}t_i}{\sum(t{,}\sum t_i){=}\sum t_i}$ (SUM1)
$\mathrm{T}^3{=}\mathrm{T}^1{\cup}\{A(\mathrm{T}^2){,}O(S{,}\mathrm{T}^2)\}$	$\frac{\forall i{:}t{\neq}t_i}{\sum(t{,}\sum t_i){=}\sum t_{i+1}}$ (SUM2)
$\mathrm{T}^2{\cong}\sum(\mathrm{T}^2)$ (VACU)	
$\frac{\forall t{\in}\mathrm{T}^2}{\Gamma{\vdash}[]{:}A(t)}$ (ARR0)	$\frac{\forall t{\in}\mathrm{T}^3{,}\Gamma{\vdash}x{:}t{,}y{:}A(\sum t_i)}{\Gamma{\vdash}[x]{\oplus}y{:}A(\sum(t{,}\sum t_i))}$ (ARR*)
$\frac{\forall t{\in}\mathrm{T}^2}{\Gamma{\vdash}\{\}{:}O(S{,}t)}$ (OBJ0)	$\frac{\forall t{\in}\mathrm{T}^3{,}\Gamma{\vdash}x{:}t{,}y{:}O(S{,}\sum t_i)}{\Gamma{\vdash}(s{,}x){\otimes}y{:}O(S{,}\sum(t{,}\sum t_i))}$ (OBJ*)

Table 2. Typing rules for polymorphic JSON

In this section we develop a more general notion of type inference for JSON. Rather than insisting on monomorphic lists, we allow polymorphism. The mechanisms included in PURE^3 support the declarative specification of a larger class of *polymorphic* typing rules. With such rules, a *sum-type* can appear in type terms, expressing the set of possible value-types that might appear in a given JSON list. The polymorphic typing rules are given in table 2.

To implement the SUM1 and SUM2 rules we introduce the following predicate, which handles injection and insertion of types into a set of types. This is easily accomplished using the *soft-cut*, or logical "if-then-else" construction (also available in many Prolog implementations). In the `sigma` predicate below, the type is first checked for membership in a given union representing a set of types. If found, the set is unified with the result. Otherwise, a new set that is formed by insertion of the new type is returned.

```
;; Working with types
(defn sigma (predicate locals: (ts')
 ([_ t `(Union . ,ts') `(Union . ,ts)] :-
  ([member⁰ t ts'] *-> [≡ ts' ts]
   / [insert⁰ t ts' ts]))
 ([_ t t' ts] :-
  ([≡ t t'] *-> [≡ t' ts]
   / ([! car⁰ t' 'Union]
      :[≡ ts `(Union . ,ts')]
      :[insert⁰ t `(,t') ts'])))
))
```

This predicate also keeps unions non-vacuous by collapsing singleton sets to their isomorphic member (VACU rule) using "soft-cuts" (`*->`) and enforces *flat* unions via "negation-as-failure" (`!`).

```
;; Polymorphic lists
(defn [poly-ne-list comma elem] (pcg ⇔ s
 (s locals: (t ts')
  ([_ `(,v) t] ⇔ [elem v t])
  ([_ `(,v . ,vs) ts] <=[(sigma t ts' ts)]=>
   [elem v t] [idem comma] [s vs ts'])
 )))
```

Because the PCG predicate that matches a list of JSON values is required to compute new types as it parses the input terms, it can not remain *homomorphic* in the presence of polymorphic types, as the `ne-list` predicate. A straightforward extension that uses the `sigma` predicate above, and which implements the ARR* and OBJ* rules from table 2 is shown above (`poly-ne-list` predicate). It utilizes a reversible logical action (using the `<=[actions ...]=>` syntax) to make sure that the type inference of the JSON values is performed in a fully reversible way (see section 2.3.1 above).

3.3 Term generation

We make no distinction between the parsing and the typing phases - both kinds of semantic attributes are computed together. Looking at the figure 4, we observe that types are just like other semantic attributes, obtained by removing some details from semantic terms (the AST), in a way that parallels *abstract interpretation* [CC77]. Using PURE^3, one might even compute a number of different types or kinds of semantics via unification of PCG attributes in parallel.

```
;; Constrained generation of syntax-semantics pairs
(verify enum1 (run* (q) (fresh (x) (tjson-value
           x '() q `(Object (Pair Bool Num)))))
 =>) ;; there are no terms of this type (yet)
;; there are infinitely many terms of this type
(verify enum2 (run 5 (q) (fresh (x) (tjson-value
           x '() q `(Object (Pair Str Num)))))
---> (obj ("a" . 0)) (obj ("b" . 0)) (obj ("a" . 1))
     (obj ("a" . 0) ("a" . 0)) (obj ("a" . 2)))
```

Because of this, the constrained generation of syntax-semantics pairs based on types is just a mode of running PCG predicates.

4. Extensibility

For JSON, one of common objections is the lack of human-friendly surface syntax. CSON [Lup], for example, dispenses with the need to quote all symbols as JSON strings. It is therefore useful to allow *contained* extensibility for the specification of DSL *families*.

The PCG formalism as introduced so far supports a rather rigid specification of syntax and semantics. Referential transparency of the pure predicates implies that any *local* change done to a grammar requires redefinition of all dependent predicates. Relying on implicit reference cells as used by Scheme's `define` primitive would

not work with pcg rules that hide internals. Also, there are difficulties with this approach when running it on the BIGLOO interpreter as well as with BIGLOO's native, and Java Virtual Machine (JVM) back-ends, which disallow redefinition of procedures [Se].

In PURE^3 we would like to be able to introduce orthogonal (i.e., homomorphic) extensions to both syntax and semantics, in a compositional and modular fashion. For example, a natural extension of the `json-symbol` predicate to include proper symbols as JSON values should not require a reiteration of the full grammar from figure 4. Also, some DSLs would benefit from an external JSON-like representation of *sparse* arrays, i.e., maps where the object key (`json-key` clause) can be numeric rather than always only a string.

```
;; Allow symbols as values
(defn [tjson-ext-sym] (let ([extend' (extend)])
  (fn-with [apply extend'] | 'json-symbol =>
     (pcg ([_ x 'Sym] ⇔ [symbol x]))
  )))
;; Allow numbers and booleans as keys
(defn [tjson-ext-key] (let ([extend' (extend)])
  (fn-with [apply extend'] | 'json-key =>
     (pcg ([_ x 'Num] ⇔ [number x])
          ([_ x 'Bool] ⇔ [json-bool x]))
  )))
```

Using the anonymous functions with pattern-matching (as described in [KS13]), we can define extensions in a straightforward fashion, as shown above. Here we make use of *dynamic* binding mechanism implemented by the SRFI#39: "Parameter objects" [Fee03]. The current value of the `extend` parameter object is saved, and the tag passed to the extension call is checked. If it matches the extended predicate's name, then new clause(s) generated by the `pcg` macro are returned as the predicate extension. Otherwise, saved extension is invoked (via the `[apply extend']` handler).

This way, the predicates can be row-extended seamlessly. The `Expr` predicate function in the generated code from section 2.4 exposes the internals. As a first choice point introduced with MINIKANREN's `conde`, we invoke the `extend` parameter. By default, predicates returned by parameter objects `fail`, as below:

```
;; Implementing extensibility
(defn *def-extend* (λ (head) (λ (in out . results) #u)))
(defn extend [make-parameter *def-extend*])
```

In addition to the destructive updates expressed by calling the parameter object with a single argument, the SRFI#39 supports modifying dynamically scoped bindings with new values in a statically defined scope, specified via the `parameterize` form. This is useful for expressing local, contained changes to PCG grammars.

```
;; Using extensible syntax
(parameterize ([extend (tjson-ext-sym)])
 ;; forwards
 (verify test15.1 (run 1 (q) (fresh (x t) (tjson-value
           '#h: [{foo:quux},{bar:snarf},1] ^L '() x t)
            [≡ q `(,t ,x)])))
 ===> [(Array (Union (Object (Pair Sym Sym)) Num))
       (arr (obj (foo . quux)) (obj (bar . snarf)) 1)])
 ;; backwards
 (verify test15.2 (run* (q) (tjson-value
            q '() '[obj (a . (arr "b" "2.3"))]
            `(Object (Pair Sym (Array Str)))))
 ===> (|{| a |:| |[| "b" |,| "2.3" |]| |}|)
))
(parameterize ([extend (tjson-ext-key)])
 (verify test15.3 (run* (q) (fresh (x t) (tjson-value
           '#h: [{12: "quux"},{42: "snarf"}] ^L '() x t)
            [≡ q `(,t ,x)])))
 ===> [(Array (Object (Pair Num Str)))
       (arr (obj (12 . "quux")) (obj (42 . "snarf")))]
))
```

4.1 Chaining extensions

Often, extensions make sense only when applied together, as a group. PURE^3 supports expression of extension *dependencies* by static chaining of corresponding extension functions. This can be achieved by simply capturing the dependent extension rather than the current value of the parameter object in dependee's definition:

```
;; A contrived example of chaining ext-key to ext-sym
(defn [ext-key] (let ([extend' (ext-sym)]) ...))
```

4.2 Composing extensions

Because each extension hooks onto the current value of the `extend` parameter object, we can also compose such extensions in a natural way - simply by nesting appropriate `parameterize` scopes.

```
;; Composing JSON extensions
(parameterize ([extend (tjson-ext-sym)])
(parameterize ([extend (tjson-ext-key)])
 (verify test16 (run* (q) (fresh (x t) (tjson-value
           '#h: [{12:quux},{42:snarf}] ^L '() x t)
            [≡ q `(,t ,x)])))
 ===> [(Array (Object (Pair Num Sym)))
       (arr (obj (12 . quux)) (obj (42 . snarf)))])
))
```

While in section 4.1 the composition is static, here we apply dynamic resolution and chaining of extensions referenced by the `extend` parameter object. This improves the modularity for library-based implementation of DSL families that support localization, such as the JSON extensible parser grammar described here.

4.3 Power and danger

Combining extensions with committed choice (which can be forced by the `condo:` PCG modifier), one can use extensions to subvert existing grammar in a non-monotonic fashion. Although PCGs only seem to support row-extensibility, addition of new clauses that may take precedence over previous clauses is definitely possible. The ability to recursively refer to previously defined clauses enables extensions to the rows themselves. For example, the expression grammar can be extended with new operators as follows:[12]

```
;; Extending Term predicate
(defn Term+ (let ([extend' (extend)][T' Term])
 (fn-with [apply extend'] | 'Term =>
  (pcg ([_ π(@ x y)] ⇔ [lift T' x] @ [Factor y]))
)))
```

Because we allow impure logical code here, and because the escape to the Scheme level is made possible via MINIKANREN's `run` and `project` (π) primitives, the PCG predicates can be *extended while parsing*. For example, the `@` in the clause head above can be referring to any Scheme function or procedure, and that may perform any (effectful) computation. Since the SRFI#39 allows destructive/imperative update to the `extend` reference cell, this alone effectively makes the formalism *Turing-complete* (i.e., Chomsky Type-0). This can be easily seen by translation to vW-grammars, or 2-level grammars whereby infinite context-free grammars can be generated from a finite set of (Type-3, even) meta-rules [vW74].

5. Related work

A traditional approach to the problem of left-recursion is its effective *elimination* [HMU03]. The work that formalized PEGs *avoids* left-recursion by putting it outside of the set of well-formed grammars [For04]. With Prolog DCGs the problem is typically solved via *ad-hoc* methods such as *cancellation tokens* [NK93], *memoing*

[12] note that we need to explicitly *lift* the left-recursive call in this case

[BS08] or through elimination (see section 2.3.1 for a PCG rendering of this). Parser combinators are often applying *curtailment* [FH06] and [FHC08], an old idea to limit matches by the input length [Kun65]. The work on OMETA has also advocated *memoing* for solving left-recursion [WDM08].

Unlike prior art where programming was constrained to use only reversible compositions of bijections [RO10] or where the reverse transformations are derived from the forward transformations [Vis01] and OMETA [WP07], we maintain a *single* grammar/parser that can be used in different modes: forwards, backwards, sideways etc. Unlike the more restrictive formalism of *lenses* [FGM+05] and [BFP+08] we do not rely on carrying the original sources along but allow structural changes within the limits of the information theory.

Mode analysis, inference and scheduling of predicate resolutions has been addressed, e.g., in Prolog [DW88] and in Mercury [OSS02]. Our approach differs from those as it seamlessly integrates with an existing FP language rather than relies on an abstract interpretation framework implemented inside a dedicated compiler.

Of all proposals to improve the syntax of LISP going back to "M-expressions", "I-expressions" [MÖ5], *sweet* "t-expressions" [DAW13] and CGOL [Pra73] the method of "enforestation" [RF12] seems to be the closest to our approach. This work utilizes the Pratt operator precedence parsing (which is less general than PCG) that avoids rather than addresses issue of left-recursion.

6. Conclusions

Fully reversible syntax-semantics relations are enforced by a "correct-by-construction" inference of logical variable bindings from clause heads and equation of those with the bindings from clause bodies. The information is not *dissipated* by default, so the transformations remain reversible and in fact, might become very inexpensive to run [Lan00] in the future. Parts of the information may be *hidden*, which is useful for e.g., implementing updatable views or for keeping programs invisibly statically typed.

The novel technique of logical laziness allows us to retain the purely declarative style of on-line, left-recursive grammar specifications, without sacrificing either the direct-style associativity or the naturality of the syntax, semantics and typing specifications.

We offer one possible answer to the question about what hygiene of `syntax-rules` actually means [Kis02]: it implements the access to the name and binding in Scheme (i.e., scoping rules), whereby hygiene is maintained by default while still supporting equational theory of bindings across disparate code fragments. In effect, (weak) hygiene breaking `syntax-rules` should be seen as *specifications* of such theories [Her10] and not just as cool hacks.

PCGs are "macros no more": we see no need to use arcane, albeit elegant rewriting systems such as `syntax-rules` for programming in a homoiconic language such as Scheme. The syntax and the semantics are better specified using pure declarative methods of PURE^3, naturally expressing reversible (i.e., inferrable) types, providing declarative disambiguation operators, enabling online/incremental processing as well as providing support for essential error reporting and debugging interfaces for practical DSLs.

Acknowledgments

We would like to thank Oleg Kiselyov for noting the problem of left-recursion in on-line parsers. This paper has benefited from the discussions with Willam Byrd, Ralf Lämmel and Vadim Zaytsev.

References

[ABB+98] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998. URL: `http://doi.acm.org/10.1145/290229.290234`.

[ALSU06] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[App06] Andrew W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. URL: `http://doi.acm.org/10.1145/321992.321996`.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419, 2008. URL: `http://doi.acm.org/10.1145/1328438.1328487`.

[BS08] Ralph Becket and Zoltan Somogyi. Dcgs + memoing = packrat parsing but is it worth it? In *PADL*, pages 182–196, 2008. URL: `http://dx.doi.org/10.1007/978-3-540-77442-6_13`.

[Byr10] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Department of Computer Science, Indiana University, 2010.

[Cam05] Taylor Campbell. Srfi 46: Basic syntax-rules extensions. Internet, 2005. URL: `http://srfi.schemers.org/srfi-46`.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252. ACM, 1977. URL: `http://doi.acm.org/10.1145/512950.512973`.

[DAW13] Alan Manuel K. Gloria David A. Wheeler. Srfi 49: Sweet-expressions (t-expressions). Internet, 2013. URL: `http://srfi.schemers.org/srfi-110`.

[DG05] Olivier Danvy and Mayer Goldberg. There and back again. *Fundam. Inform.*, 66(4):397–413, 2005.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982. URL: `http://doi.acm.org/10.1145/582153.582176`.

[DW88] Saumya K. Debray and David Scott Warren. Automatic mode inference for logic programs. *J. Log. Program.*, 5(3):207–229, 1988.

[FBK] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. minikanren homepage. URL: `http://minikanren.org`.

[FBK05] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The reasoned schemer*. MIT Press, 2005.

[Fee03] Marc Feeley. Srfi 39: Parameter objects. Internet, 2003. URL: `http://srfi.schemers.org/srfi-39`.

[FGM+05] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, pages 233–246, 2005. URL: `http://doi.acm.org/10.1145/1040305.1040325`.

[FH06] Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Notices*, 41(5):46–54, 2006. URL: `http://doi.acm.org/10.1145/1149982.1149988`.

[FHC08] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL*, pages 167–181, 2008. URL: `http://dx.doi.org/10.1007/978-3-540-77442-6_12`.

[For02] Bryan Ford. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *ICFP*, pages 36–47, 2002. URL: `http://doi.acm.org/10.1145/581478.581483`.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*, pages 111–122, 2004. URL: http://doi.acm.org/10.1145/964001.964011.

[GBJL02] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. John Wiley, 2002.

[Her10] David Herman. *A theory of typed hygienic macros*. PhD thesis, Northeastern University Boston, 2010.

[HF00] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme Workshop*, 2000.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999. URL: http://journals.cambridge.org/action/displayAbstract?aid=44275.

[Kis02] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Scheme Workshop*, 2002.

[Kou] Peter Kourzanov. purecube. Internet. URL: https://github.com/kourzanov/purecube.

[KS13] Peter Kourzanov and Henk Sips. Lingua franca of functional programming (fp). In Hans-Wolfgang Loidl and Ricardo Pena, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 198–214. Springer Berlin Heidelberg, 2013. URL: http://dx.doi.org/10.1007/978-3-642-40447-4_13.

[Kun65] Susumu Kuno. The predictive analyzer and a path elimination technique. *Commun. ACM*, 8(7):453–462, 1965. URL: http://doi.acm.org/10.1145/364995.365689.

[Lan00] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 44(1):261–269, 2000. URL: http://dx.doi.org/10.1147/rd.441.0261.

[Lup] Benjamin Lupton. Coffeescript-object-notation parser. Internet. URL: https://github.com/bevry/cson.

[MÖ5] Egil Möller. Srfi 49: Indentation-sensitive syntax. Internet, 2005. URL: http://srfi.schemers.org/srfi-49.

[NK93] M.-J. Nederhof and Cornelis H.A. Koster. Top-down parsing for left-recursive grammars. Technical Report 93-10, University of Nijmegen, Department of Computer Science, 1993.

[OSS02] David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *PPDP*, pages 109–120, 2002.

[Pra73] Vaughan R. Pratt. Top down operator precedence. In *POPL*, pages 41–51, 1973. URL: http://doi.acm.org/10.1145/512927.512931.

[PW80] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artif. Intell.*, 13(3):231–278, 1980. URL: http://dx.doi.org/10.1016/0004-3702(80)90003-X.

[RF12] Jon Rafkind and Matthew Flatt. Honu: syntactic extension for algebraic notation through enforestation. In *GPCE*, pages 122–131, 2012. URL: http://doi.acm.org/10.1145/2371401.2371420.

[RO10] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Haskell*, pages 1–12, 2010. URL: http://doi.acm.org/10.1145/1863523.1863525.

[Sch] Chicken Scheme. Unit library. Internet. URL: http://wiki.call-cc.org/man/4/Unit%20library#set-sharp-read-syntax.

[Se] Manuel Serrano and et.al. Bigloo homepage. URL: http://www-sop.inria.fr/indes/fp/Bigloo.

[SW95] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *SAS*, pages 366–381, 1995. URL: http://dx.doi.org/10.1007/3-540-60360-3_50.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA*, pages 357–362, 2001. URL: http://dx.doi.org/10.1007/3-540-45127-7_27.

[vW74] Adriaan van Wijngaarden. The generative power of two-level grammars. In *ICALP*, pages 9–16, 1974. URL: http://dx.doi.org/10.1007/3-540-06841-4_48.

[WDM08] Alessandro Warth, James R. Douglass, and Todd D. Millstein. Packrat parsers can support left recursion. In *PEPM*, pages 103–110, 2008. URL: http://doi.acm.org/10.1145/1328408.1328424.

[Wel94] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *LICS*, pages 176–185, 1994. URL: http://dx.doi.org/10.1109/LICS.1994.316068.

[WP07] Alessandro Warth and Ian Piumarta. Ometa: an object-oriented language for pattern matching. In *DLS*, pages 11–19, 2007. URL: http://doi.acm.org/10.1145/1297081.1297086.

A. PCG syntax-rules

```
(def-syntax seq (syntax-rules (qq skip quote unquote
      quasiquote unquote-splicing do ϵ when unless
                              ? + * / : lift unlift)
;; handling escapes
([_ in out c acc ts hs do[acts ...] . rest]
  (seq in out c (acts ... . acc) ts hs . rest))
;; handling ϵ
([_ in out c acc tmps hs ϵ . rest]
  (seq in out c ([≡ in out] . acc) tmps hs . rest))
;; handling sequencing (recursively)
([_ in out c acc temps [h(ac...)] (: . goals) . rest]
  (let ([temp #false]);; generate a new temporary
    (seq temp out c ((all
        (seq in temp c () () [h(ac... . acc)]
           . goals)) . acc)
       (temp . temps) [h (ac ...)] . rest)
   ))
;; handling quasi-data (one-level only)
([_ in out c acc tmps heads (qq d) . rest]
  (seq in out c acc tmps heads ‘d . rest))
([_ in out c acc tmps [h(ac ...)] ‘d . rest]
  (let ([temp #false][data #false]);; new temporaries
    (seq temp out c ((qs [] ;; remove quasi-quotation
      (seq data ’() c () () [h(ac ... . acc)]) ‘d)
        [≡ in ‘(,data . ,temp)] . acc)
      (temp data . tmps) [h (ac ...)] . rest)
   ))
;; handling non-terminals
([_ in out c acc temps heads [goal . args] . rest]
  (let ([temp #false]);; generate a new temporary
    (seq temp out c ([goal in temp . args] . acc)
       (temp . temps) heads . rest)
   ))
;; handling atoms (this rule has to be the last one)
([_ in out c acc tmps heads datum . rest]
 (let ([temp #false]);; generate a new temporary
   (seq temp out c ([≡ in ‘(datum . ,temp)] . acc)
      (temp . tmps) heads . rest)
  ))
))
```

Figure 5. TRS for threading PCG monadic state (abridged)

```
;; A snippet from the process-args macro implementation
;; base-case (generate code for args, terms and project)
([process-args k acc [] goals rest (e es ...)
                                    aa res ps (locals ...)]
 (let-syntax-rule ([K . vars]    ;; collect the free vars
 (let-syntax-rule ([K wv wp wt ws] ;; use extracted vars
 (let-syntax ([K (syntax-rules ()  ;; use extracted pvars
   ;; ... other special cases elided ...
   ([_ pvars (ee . bis) pats (terms []) hots]
    (make-scopes fresh bis all ;; schedule clause terms
       (let-syntax ([ee '()]);; last ee=e must be empty
          (all . pats))     ;; when resolving arguments,
       terms                ;; clause body in the middle,
       (make-scopes project  ;; projected terms delayed
          pvars all . hots) ;; to the end of the clause
   ))
   ;; ... handlers for clauses with actions elided ...
   )])
    (extract* vars (wp wt) ;; extract all bindings
       (K () wv wp wt ws)) ;; in projected terms
    ))
    (extract* (e es ... locals ... . vars) ;; extract all
       (res goals) (K () res goals ps)) ;; fresh bindings
    ))
    (scheme-bindings (w [] (K) (locals ...) aa))
 ))
;; recursive case: collect unifiers and attributes (e's)
([process-args k acc [v . vs] goals rest (ee . es)
                              aa (res ...) ps locals]
 (let ([e #false]) ;; generate a new temporary
    (process-args k acc vs goals rest (e ee . es)
       (v . aa)
       (res ... [≡ ee '(,v . ,e)])
       ps
       locals)
))
```

Figure 6. TRS equational theory for name binding

```
;; A snippet from the predicate (pred) macro
;; ... top-level predicate generation elided ...
([_ (begin ks ..) params ([_ . args] ⇔ . body) . rest]
  (let-syntax ([head #false])      ;; generate a new head
  (let-syntax-rule ([K heads condo locals]
   (pred (begin ks ..   ;; collect all clause heads and
    (define head (λ (Lin Lout . vars) ;; deliver to the
       (process-args condo (ks ..) args   ;; top-level
         ([seq Lin Lout condo () () [heads (ks ..)]
           . body] [])
         vars
         locals)
       ))
    ) params . rest))
     (select (K) (0 0 1 1 1) . params)
  )))
;; ... elided combing params to order specifiers
;; ... such as condo:, locals: and extend: ...
```

Figure 7. Implementing PCG predicates

```
(def-syntax w (syntax-rules
   (qq quote unquote quasiquote unquote-splicing λ)
 ([_ q   (k ...) b [] . a] (k ... . a))
 ([_ q       k b 't . a] (w [qq . q] k b t . a))
 ([_ [qq . q] k b ,t . a] (w q k b t . a))
 ([_ []      k b ,t . a] (bad-unquote k b ,t))
 ([_ q       k b 't . a] (w q k b [] . a))
 ([_ []  k b [λ (var ...) . body] . a]
  (w [] k (var ... . b) body . a))
 ;; ... other binders such as let, do and project elided
 ([_ q  k b [t . ts] . a] (w q (w q k b t) b ts . a))
 ([_ [] k b t a ...]
      (symbol?? t
         (member?? t (a ... . b)
            (w [] k b [] a ...)
            (w [] k b [] a ... t))
            (w [] k b [] a ...)
            ))
 ([_ [qq . q] k b  t . a] (w q k b [] . a))
))
```

Figure 8. Extracting free variables from Scheme terms

```
...
([_ in out condo acc temps
                    [(r . heads) (ac ...)] (goals ... *)]
 (let-syntax ([K (syntax-rules .. ()
  ([_ in out vars ..]
   (let loop ([lin in][lout out] [vars '()] ..)
     (let-syntax ([K (syntax-rules ... ()
       ([_ res ...]
        (let ([res #false] ...)
          (make-scopes (res ...) begin
           (letrec-syntax ([K (syntax-rules .... ()
            ([_ gls (v v1 v2 v3) ....]
            (condo ([≡ lin lout]
                    [≡ v1 v] ....)
               ([let ([temp #false][v3 #false] ....)
                (fresh (temp v3 ....)
                   (let-syntax ([v v3] ....)
                     (seq lin temp condo () ()
                      [(r . heads) (ac ... . acc)] . gls
                     ))
                   (append^0 v1 '(,v3) v2) ....
                   (loop temp lout v2 ....))])))))]
            [K1 (syntax-rules ()
             ([_ var gls . args]
              (zip4 (K gls) var . args)
             ))]
            [K0 (syntax-rules ()
             ([_ . vs]
              (extract* vs (goals ...)
                (K1 [] (goals ...) (vars ..) (res ...) vs)
             ))])
            (scheme-bindings (w [] (K0) heads (goals ...)))
           ))))])
          (scheme-bindings (w [] (K) heads (goals ...)))
         ))))])
        (seq in out condo acc temps [(r . heads) (ac ...)]
          do[(scheme-bindings (w [] (K in out) heads
               (goals ...)))])
        ))
...
```

Figure 9. Implementing the Kleene-* combinator

B. PCG standard library

```
;; miniKanren examples
(def *digits* [make-parameter (list-tabulate 10 values)])
(def (range start end)
   (unfold [_ char>? end]
     values
     (○ integer->char
        [_ + 1]
        char->integer)
     start))
(def *letters* [make-parameter (range #a #z)])
(def [lifto pred stream] (λ (x)
   (conda ([project (x)
      (or (and (ground? x) (pred x) #s) #u)])
      ([take-from (stream) x])
      )))
(def numbers? [lifto number? *digits*])
(def symbols? [lifto symbol? (λ ()
     (map (○ string->symbol list->string list) [*letters*]))])
(def strings? [lifto string? (λ ()
     (map (○ list->string list) [*letters*]))])
(def (! p . args)
   (condu ((apply p args) #u)
      (else #s)))
(def (null^0? x) [≡ x '()])
(def (pair^0? x) (fresh (x0 x1) [≡ x `(,x0 . ,x1)]))
(def (car^0 x y) (fresh (t) [≡ x `(,y . ,t)]))
(def (cdr^0 x y) (fresh (h) [≡ x `(,h . ,y)]))
(def (cons^0 h t l) [≡ l `(,h . ,t)])
(def (number Lin Lout x) (all (cons^0 x Lout Lin) (numbers? x)))
(def (symbol Lin Lout x) (all (cons^0 x Lout Lin) (symbols? x)))
(def (strings Lin Lout x) (all (cons^0 x Lout Lin) (strings? x)))
(def (literal Lin Lout x) (conde ([symbol Lin Lout x])
                                 ([number Lin Lout x])))
(def (idem Lin Lout v) (cons^0 v Lout Lin))
```

C. Acronyms

AST	Abstract Syntax Tree
BNF	Backus-Naur Formalism
CPS	Continuation Passing Style
DCG	Definite Clause Grammar
DSL	Domain-Specific Language
FP	Functional Programming
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LISP	List Processing
NXP	Next Experience Semiconductors
PCG	Parsing Clause Grammar
PEG	Parsing Expression Grammar
R5RS	Revised^5 Report on the Algorithmic Language Scheme
SRFI	Scheme Request for Implementation
TRS	Term-Rewriting System
TU	Technical University

Type Families and Elaboration

Alejandro Serrano Jurriaan Hage
Department of Information and Computing Sciences
Utrecht University
{A.SerranoMena, J.Hage}@uu.nl

Patrick Bahr
Department of Computer Science
University of Copenhagen
paba@di.ku.dk

Abstract

Type classes and type families are key ingredients to Haskell programming. Type classes were introduced to deal with ad-hoc polymorphism, although with the introduction of functional dependencies, their use expanded to type-level programming. Type families also allow encoding type-level functions, now as rewrite rules, but they lack one important feature of type classes: elaboration, that is, generating code from the derivation of a rewriting. This paper looks at the interplay of type classes and type families, how to deal with shortcomings in both of them, and discusses further relations on the assumption that type families support elaboration.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Functional Languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs – Type Structure

Keywords Type classes; Type families; Haskell; Elaboration; Functional dependencies; Directives

1. Introduction

Type classes are one of the distinguishing features of Haskell, and are widely used and studied (Peyton Jones et al. 1997). The initial aim was to support ad-hoc polymorphism (Wadler and Blott 1989): a type *class* gives a name to a set of operations along with their types; subsequently, a type may become an *instance* of such class by giving the code for such operations. Furthermore, an instance for a type may depend on other instances (its *context*). The following is a classic example of the *Show* type class and the instance for lists which illustrate these features in action:

```
class Show a where
  show :: a → String
instance Show a ⇒ Show [a] where
  show lst = "[" ++ intersperse ',' (map show lst) ++ "]"
```

For each call to an operation such as *show*, the compiler must resolve what code corresponds to that call. Note that the search is needed to find the correct code: above, *show* for type $[a]$ depends on the code for type a. The search and combination of code performed by the compiler is called *elaboration*.

We remark at this point that we consider type classes *without* support for *overlapping instances*. Overlapping instances are used to override an instance declaration in a more specific scenario. The best example is *Show* for strings, which are represented in Haskell as $[Char]$, and for which we usually want a different way to print them:

```
instance [Char] where
  show str = ...   -- show between quotes
```

Overlapping instances make reasoning about programs more difficult, since the resolution of instances may change by later overlapping declarations. Furthermore, their common usage patterns can be express by using type families as shown in Section 4.

Type classes have been later extended to support multiple parameters: unary type classes describe a subset of types supporting an operation, multi-parameter ones describe a relation over types. For example, you can declare a *Convertible* class which describes those pairs of types for which the first can be safely converted into the second:

```
class Convertible a b where
  convert :: a → b
```

In many cases, though, parameters in such a class cannot be given freely. For example, if we define a *Collection* class which relates types of collections and the type of elements, it does not make sense to have more than one instance per collection type. Such constraints can be expressed using functional dependencies (Jones 2000), a concept borrowed from database theory:

```
class Collection c e | c → e where
  empty :: c
  add   :: e → c → c
instance Collection [a] a where
  empty = []
  add   = (:)
```

If we try to add a new instance for $[a]$, the compiler does not allow it, since for each type of collection c, you can only have one e.

Using functional dependencies, *functions* can also be defined at the level of types. Since their inception, functional dependencies have been abused in that way, and it is now common folklore how to do it: given a type level function of n parameters you want to encode, define a type class with an extra parameter (the result) and include a dependency of it on the rest. Each instance will then define a rule in the function. Here is the archetypical *Add* function defined as a type class:[1]

```
data Zero
data Succ a
```

[1] Note that this example needs the UndecidableInstances extension to work in GHC.

```
class AddC m n r | m n → r
instance AddC Zero n n
instance AddC m n r ⇒ AddC (Succ m) n (Succ r)
```

Type families (Schrijvers et al. 2007) were introduced as a more direct way to define type functions in Haskell. Each family is introduced by a declaration of its arguments (and optionally its return kind) and the rules for the function are stated in a series of **type instance** declarations. The *Add* function now becomes:

```
type family AddF m n
type instance AddF Zero     n = n
type instance AddF (Succ m) n = Succ (AddF m n)
```

Type families have one important feature in common with type classes: they are *open*. This means that in any other module, a new rule can be added to the family, given that it does not overlap with previously defined ones.

However, when thinking in terms of functions, we are not used to wear our open-world hat. In a case like *Add*, we would want to define a complete function, with a restricted domain. Eisenberg et al. (2014) introduced *closed* type families to bridge this gap. Closed families are matched in order, each rule is only tried when the previous one is assured never to match. Thus, overlapping between rules is not a problem. On the other hand, these families cannot be extended in a different declaration. In GHC, closed type families are introduced using the following syntax:

```
type family AddF' m n where
  AddF' Zero     n = n
  AddF' (Succ m) n = Succ (AddF' m n)
```

As an aside, families can be *associated* with a type class. In that way, for each class instance you need to define also a set of types local to such instance. The *Collection* class is a good candidate to be given an associated type, namely the type of elements:

```
class Collection2 c where
  type Element c
  empty2 :: c
  add2   :: Element c → c → c
instance Collection2 [a] where
  type Element [a] = a
  empty2 = []
  add2   = (:)
```

The discussion above illustrates that type classes and type families have a lot of things in common, and in many cases choosing one over the other for a task is a matter of convenience or style. In other cases, though, their features differ. The following table summarizes the similarities and differences between classes and families:

	type classes	type families
open	✓	✓
closed		✓
elaboration	✓	
context	✓	

The goal of this paper is to discuss whether it is possible to bridge the gap, and bring type classes and type families even closer in terms of functionality (Sections 2 and 3). Most of the techniques presented in the those sections are folklore or have been used as part of a larger technique, but we expect to show the tight connection between them by focusing only in the tricks without a larger problem behind it. We have already seen how to simulate type families with functional dependencies.

Our *main contribution*, discussed in Section 4, is dealing with the opposite situation: *using type families to express type classes*. We shall see that a key ingredient for making type families as powerful as type classes is to equip type families with an *elaboration* mechanism. This extension does not only level the power of type classes and families, but yields new use cases that are impossible or difficult to express in terms of type classes.

2. Shortcomings of type families

Type families are usually described as a rewriting mechanism at the level of types. By writing family instances, the compiler is able to apply equalities between types to simplify them. As discussed above, the main distinguishing feature of type families is their support for closed definitions. At first sight, they lack the useful feature of elaboration, and also the ability to depend on contexts; here we show that we can simulate both of these aspects.

2.1 Elaboration

When the compiler resolves a specific instance of a type class, it checks that typing is correct, and also generates the corresponding code for the operations in the class. This second process is called *elaboration*, and is the main reason for the usefulness of type classes. Type families, on the other hand, only introduce type equalities. Any witnesses of these equalities at the term level are erased. Is it possible, however, to trick the compiler into elaborating a term from a family application?

The solution has already been pointed out in several places, e.g. by Bahr (2014), who uses it to implement a subtyping operator for compositional data types. Let us illustrate this idea with an example: we want to define a function *mkConst* that creates a constant function with a variable number of arguments. For instance, given the type $a \to b \to Bool$, we want a function $mkConst :: Bool \to (a \to b \to Bool)$.

To start, we need a type-level function which returns the result type of a curried function type of arbitrary arity:

```
type family Result f where
  Result (s → r) = Result r
  Result r       = r
```

This is the point where, if we could elaborate a function during rewriting, deriving our *mkConst* would be quite easy. Instead, we have to define an auxiliary type family that computes the *witness* of the rewriting of *Result*. The first step is creating a data type to encode such witness. By using data type promotion (Yorgey et al. 2012) we can move a common data type "one level up" such that its constructors are turned into types, and the type itself is turned into a kind. [2]

```
data ResultWitness = End | Step ResultWitness
```

We then define the closed type family *Result′*, which is responsible for computing the witness. Note the use of a kind signature to restrict its result to the types promoted before.

```
type family Result' f :: ResultWitness where
  Result' (s → m) = Step (Result' m)
  Result' r       = End
```

Here comes the trick: using a *type class* that elaborates the desired function in terms of the witness. The witness will be supplied via a zero-data constructor *Proxy*, which serves the purpose of recording the witness information:

```
data Proxy a = Proxy
class ResultE f r (w :: ResultWitness) where
  mkConstE :: Proxy w → r → f
```

[2] In GHC, this behavior is enabled by the DataKinds extension.

Each instance of *ResultE* will correspond to a way in which *ResultWitness* could have been constructed. Note that in the recurring cases, we need to provide a specific type argument using *Proxy*:

```
instance ResultE r r End where
  mkConstE _ r = r
instance ResultE m r l ⇒ ResultE (s → m) r (Step l) where
  mkConstE _ r = λ(x :: s) → mkConstE (Proxy :: Proxy l) r
```

However, we do not want the user to provide the value of *Proxy w* in each case, because we can construct it via the *Result′* type family. The final touch is thus to create the *mkConst* function which uses *mkConstElab* by providing the correct *Proxy*:

```
mkConst :: ∀ f r w.(r ∼ Result f, w ∼ Result′ f,
                    ResultE f r w) ⇒ r → f
mkConst x = mkConstE (Proxy :: Proxy w) x
```

The main idea of this trick is to get hold of a *witness* for the type family rewriting. This is usually produced by Haskell compilers as a coercion, but the user does not have direct access to it. By reifying it and promoting its constructors to the type-level, we become able to use the normal type class machinery to define elaborated operations.

2.2 Context

Within Haskell, instances may depend on a certain context being available (for example, *Show* [*a*] holds if and only if [3] *Show a*), whereas rewriting via type families does not allow any preconditions. But once again, we can encode it with a bit more work, assuming we are using closed type families. Let us consider the case of a serialization library. As part of its functionality, the library must decide which representation to use for a specific data type. Normally, the type will remain the same in this representation, but for some special cases of "list-like" types (which are to be encoded in the same way as lists) and "function-like" (whose domain and target types must be recursively encoded). Those special cases are recognized by the following families:[4]

```
type family IsListLike l :: Maybe ∗
type instance IsListLike [e]     = Just e
type instance IsListLike (Set e) = Just e
type family IsFunctionLike f :: Maybe (∗, ∗) where
  IsFunctionLike (s → r) = Just (s, r)
  IsFunctionLike t       = Nothing
```

The type family that constructs representations is intuitively formulated by matching on the result of the previously introduced families:

```
type family Repr t where
  IsFunctionLike t ∼ Just (s, r) ⇒ Repr t = Repr s → Repr r
  IsListLike     t ∼ Just e      ⇒ Repr t = [Repr e]
                                   Repr t = t
```

But the above definition is not valid Haskell syntax. Instead we have to encode the conditional equations using a chain of auxiliary type families, each of which treats a single context. As extra arguments to the auxiliary type families, we incorporate the check that should be done next. The *Repr* type family thus becomes:

```
type family Repr t where
  Repr t = Repr1 t (IsFunctionLike t)
type family Repr1 t l where
  Repr1 t (Just (s, r)) = Repr s → Repr r
  Repr1 t f             = Repr2 t (IsListLike t)
type family Repr2 t l where
  Repr2 t (Just e) = [Repr e]
  Repr2 t l        = t
```

Even though the code becomes larger, the translation could be made automatically by the compiler. The main problem in this case is the error reporting. Let us define a simple function that only works on types which are already in their representative form:

```
alreadyNormalized :: (t ∼ Repr t) ⇒ t → t
alreadyNormalized = id
```

If we try to use it on a *Map*, the compiler will complain:

```
*> alreadyNormalized Data.Map.empty
<interactive>:7:1:
 Couldn't match expected type 'Map k0 a0'
 with actual type
  Repr2 (Map k0 a0) (IsListLike (Map k0 a0))'
 The type variables 'k0', 'a0' are ambiguous
```

The source of this problem is that we have not declared whether *Map* is *ListLike* or not. However, the inner details of our implementation now escape to the outside world in this error message. If contexts were added to type families, it would greatly benefit users to treat them especially in terms of error reporting.

2.3 Open-closed families

An interesting pattern with type families is the combination of open and closed type families to create a type-level function whose domain can be enlarged, but where some extra magic happens at each specific type. As a guiding example, let us construct a type family to obtain the spiciness of certain type-level dishes:

```
data Water
data Nacho
data TikkaMasala
data Vindaloo
data SpicinessR = Mild | BitSpicy | VerySpicy
type family Spiciness f :: SpicinessR
```

The family instances for the dishes are straightforward to write:

```
type instance Spiciness Water       = Mild
type instance Spiciness TikkaMasala = Mild
type instance Spiciness Nacho       = BitSpicy
type instance Spiciness Vindaloo    = VerySpicy
```

However, when we have lists of a certain food, we want to behave in a more sophisticated way. In particular, if one is taking a list of dishes which are a bit spicy, the final result be definitely be very spicy. To rule this special case, we defer the *Spiciness* of a list to an auxiliary type family *SpicinessL*:

```
type instance Spiciness [a] = SpicinessL (Spiciness a)
type family SpicinessL lst where
  SpicinessL BitSpicy = VerySpicy
  SpicinessL a        = a
```

This trick has been used for more mundane purposes, such as creating lenses at the type level (Izbicki 2014). The key point is that the non-overlapping rules for open type families allow us to add new instances for those types for which one is not yet defined. But by calling a closed type family at a type instance rule, you can

[3] We shall remind here that we are considering type classes without overlapping instances. If overlapping instances were allowed, the implication would hold only in one direction.

[4] In some cases, GHC needs a quote sign in front of type-level tuples to distinguish them from the term-level tuples.

refine the behaviour of a particular instance. Section 4 will show other interesting uses of this pattern.

3. Shortcomings of type classes

We have looked at one side of the coin, discussing idioms to deal with shortcomings of type families with respect to type classes. Looking back at our original table in Section 1, the only functionality unsupported by type classes is closedness. We shall see how taking into account our previous results on type families, we can handle that situation.

3.1 Closed type classes

In some cases, you know that for a certain type class only a limited and known set of instances should be available. This is a situation where Haskell does not have an inmediate solution: exposing a type class without also allowing new instances to be defined. However, this sort of functionality has received some attention in the literature: Heeren and Hage (2005) discuss a **close** type class directive with this specific purpose in the framework of better error diagnosis; and Morris and Jones (2010) illustrate that their instance chains also handle this case.

There is a handful of techniques to get closed type classes known by Haskell practitioners (StackOverflow 2013). These techniques boil down to the same idea: define a secret entity of some sort (we shall see that this entity can either be a type class or a type family), define an alias and export only this alias to the world.

Heeren and Hage (2005) present an example of closing the *Integral* type class to only admit *Int* and *Integer* as instances, which we use as a running example. Following the "secret class + type alias" idea, a first attempt is:[5]

```
module ClosedIntegral (Integral) where

class Integral' i
instance Integral' Int
instance Integral' Integer

type Integral i = Integral' i
```

Note that to create such an alias, we need the ConstraintKind extension in GHC, which allows treating instance and type equality constraints as simple elements of the kind *Constraint*. This solution works fine until the moment of writing a new instance:

```
instance Integral Char
```

At that point, synonyms are expanded, and that code effectively translates to a new instance of *Integral'*. In conclusion, this method does not work.

The core problem is that, by exposing *Integral'* via a synonym, we have given access to it. Instead, we can use another type class, and make the one we want to close be a prerequisite:

```
class Integral' i ⇒ Integral i
instance Integral' i ⇒ Integral i
```

If you now try to define a new instance of *Integral* in another file, you get an error message:

```
Not in scope: type constructor or class Integral'
```

Once again, a disadvantage of this method is that error messages are worded in terms of the internal elements, in this case *Integral'*.

Instead of another type class, you can use a type family. In that case, we combine the idea from Section 2.1 with the alias approach. The first thing to do is to write a type family *Integral'* responsible for building the witness – of type *IntegralW* – for the elaboration phase. This encoding allows us to use the fact that type families can be closed:

```
data IntegralW = None | IntW | IntegerW

type family Integral' i :: IntegralW where
  Integral' Int     = IntW
  Integral' Integer = IntegerW
  Integral' other   = None
```

Note that we have included a final catch-all case for those types which should not be in the type class. The next step is defining a new type class which takes care of elaboration. In our case, this is *IntegralE*:

```
class IntegralE i (witness :: IntegralW)

instance IntegralE Int     IntW
instance IntegralE Integer (IntegerW)
```

An important remark at this point is that we do not have any instance for the *None* case. Additionally, if the module in which *Integral* is defined does not export *IntegralE*, no new case can be added, effectively closing the set of possible cases, as we did before by hiding *Integral'*. The final step is generating the alias, the one visible to the user, which takes care of calling *Integral'* and elaborating based on the witness:

```
type Integral i = IntegralE i (Integral' i)
```

This alias connects the elaboration type class with the type family responsible of building the witness.

As previously, internals of the implementation escape to the outside world in case of error. For example, if a function *f* with an *Integral* constraint is used with a *Char* value, the message produced by GHC reads:

```
No instance for (IntegralE Char 'None)
arising from a use of 'f'
```

A solution which also involves type families, but in a different way, uses in its core the ConstraintKind extension found in GHC. Since *Constraint* is a kind like $*$ or any other promoted type, writing a type family which returns one constraint is possible. This family would work as an alias for a restricted set of types:

```
type family Integral i :: Constraint where
  Integral Int     = Integral' Int
  Integral Integer = Integral' Integer
```

For those types which are stated in the type family, having *Integral* is equivalent to *Integral'*. But for those which are not members, family rewriting gets stuck:

```
Could not deduce (Integral Float)
```

One nice effect of this type family is that error messages are termed using the *Integral* type family, so fewer internals are exposed to the programmer.

4. Type families with elaboration

In (Schrijvers et al. 2007), one of the earliest papers about type families in Haskell, the authors did already consider how to express type families using type classes and functional dependencies. Thus, the question whether both sorts of type-level programming are neccessary and desirable is posed since the very beginning. We sketch in this section a *new answer*: type classes may not be needed, given that we give type families some elaboration mechanism.

4.1 Encoding type classes

Let us skip for a moment the issue of elaborating functions in type classes, and just focus on the typing parts. The aim is to find a

[5] We elided the methods to be elaborated in order to make the presentation more concise.

translation of type classes into type families such that an instance for a type is found if and only if the corresponding type family rewrites to a certain type. For this latest type, which describes whether an instance is defined, we shall use the promoted version of *Defined*:

```
data Defined = Yes | No
```

For each type class C that we want to convert, we declare a new type family *IsC* whose result is of kind *Defined*. Throughout the section, *Eq* will be used as a guiding example:

```
type family IsEq (t :: *) :: Defined
```

Furthermore, each function which declares an instance constraint must be changed to work with the new *IsC* type family. Now, the constraint is an equality between an *IsEq* application and *Yes*. The following code declares an identity function whose domain is restricted only to those types which have *Eq*:

```
eqIdentity :: IsEq t ~ Yes => t -> t
eqIdentity = id
```

Of course, the whole point of declaring a type class is to populate it with instances. The most simple cases, such as *Char*, are dealt simply by defining a **type instance** which rewrites to *Yes*:

```
type instance IsEq Char = Yes
type instance IsEq Int  = Yes
type instance IsEq Bool = Yes
```

Those cases whose definition depend on a context, such as *Eq* on lists, can call *IsC* on a smaller argument to defer the choice:

```
type instance IsEq [a] = IsEq a
```

In the case of a more complex context, such as *Eq* on tuples, which needs to check both of its type variables, we introduce a type family *And* which checks for definedness of all its arguments:

```
type family And (a :: Defined) (b :: Defined) :: Defined where
  And Yes Yes = Yes
  And a   b   = No
type instance IsEq (a, b) = And (IsEq a) (IsEq b)
```

As with type classes, we are not constrained to ground types in our type families, we can also use type constructors. A translation of the *Functor* type class and some instances in this style reads:

```
type family IsFunctor (t :: * -> *) :: Defined
type instance IsFunctor []    = Yes
type instance IsFunctor Maybe = Yes
```

At this point it is important to remark that in some cases GHC needs explicit *kind signatures* on some of the arguments of a type class. If they are not included, GHC defaults to kind $*$ instead of giving an ambiguity error, so the problem may be unnoticed until later on. Having said so, in most of the cases where the declaration and instances of a type family are written together, the compiler is able to infer kinds correctly.

Finally, we are able to encode multi-parameter type classes in the same way, as the *Collection* class in the introduction:

```
type family Collection t e :: Defined
type instance Collection [e]     e = Yes
type instance Collection (Set e) e = Yes
```

We discuss the translation of functional dependencies into this new scheme in Section 4.6. For a formal treatment of the full translation, the reader is referred to Appendix A.

4.2 Elaboration at rewriting

The previous translation works well from a typing perspective, but does not generate any code, and we do expect so when we use a type class. Since our main goal is to get rid of classes, we cannot use the same trick as we did in Section 2. Furthermore, in that case type families rewrote to different witnesses depending on the rule that was applied. But in this case we want all instances to return the same *Yes* result. If that was not the case, we could not declare a constraint such as *IsEq t* $\sim$ *Yes* which would not depend on the type itself.

For those reasons, we propose the concept of *elaboration at rewriting*. The idea is that at each rewriting step, the compiler generates a dictionary of values (similar to the one for type classes), which may depend on values from other inner rewritings. Part of this idea is already in place when GHC generates coercions from family applications.

The shape of dictionaries must be the same across all type instances of a family. Thus, as with type classes, it makes sense to declare the signature of such dictionary in the same place within a type family. Without any special preference, we shall use the **dictionary** keyword to introduce it.[6] For example, the following declaration adds an *eq* function to the *IsEq* type family:

```
type family IsEq (t :: *) :: Defined
  dictionary eq :: t -> t -> Bool
```

A type instance declaration should now define a value for each element in the dictionary, as shown below:

```
type instance IsEq Int = Defined
  dictionary eq = primEqInt   -- the primitive Int comparison
```

In the case of calling other type families on its right-hand side, a given instance can access the value of its dictionaries to build its own. As concrete syntax, we propose using *name*@ to give a name to a dictionary in the rule itself, or to refer to an element of the dictionary in the construction of the larger one. This idea is seen in action in the declaration of *IsEq* for lists:

```
type instance IsEq [a] = e@(IsEq a) where
  dictionary eq []       []       = True
             eq (x : xs) (y : ys) = e@eq x y ∧ eq xs ys
             eq _        _        = False
```

The same syntax can be used to access the dictionary in a function which has an equality constraint. One example of this syntax is the definition of non-equality in terms of the *eq* operation in the *IsEq* family:

```
notEq :: e@(IsEq a) ~ Yes => a -> a -> Bool
notEq x y = ¬ (e@eq x y)
```

We use *e*@ prefixes to make clear which dictionary we are using, but it would be possible to drop the entire prefixes when there is only one available possibility. Another option is making *eq* a globally visible name, as type classes do.

As we have seen, elaboration at rewriting is possible and opens new possibilities for type families. It is also the only piece missing that we cannot directly encode in type families. In the rest of the paper, though, we shall just focus on the typing perspective, which in contrast with elaboration *is* available in Haskell compilers.

4.3 Type class directives

The good news about our encoding of type classes is that it brings with it ways to encode some constraints over type classes that were previously considered separate extensions of Haskell. We shall

[6] We would have preferred the **where** keyword in consonance with type classes, but this syntax is already used for closed type families.

focus first on the type class *directives* of (Heeren and Hage 2005). In short, these directives introduce new constructs to describe more sharply the set of types which are instances of a type class, with the aim of producing better error messages for the programmers.

The first of these directives is **never**: as its name suggests, a declaration of the form **never** $Eq\ (a \to b)$ forbids any instance of Eq for a function. Since by convention we translated $Eq\ t$ as $IsEq\ t \sim Yes$, we only need to ensure that $IsEq\ (a \to b)$ does not rewrite to Yes. We can do that easily with the following:

$$\textbf{type instance}\ IsEq\ (a \to b) = No$$

If we try to use Eq over a function, the compiler will complain:

```
Couldn't match type 'No with 'Yes
Expected type: 'Yes
  Actual type: IsEq (t -> t)
```

Furthermore, since compilers do not allow overlapping rules for a type family, this also disallows anybody to write an instance for any instantiation of $a \to b$, as we wanted.

The second directive is **close**, which limits the set of instances for a type class to those which have been defined until that point. We have already discussed how to deal with closed type classes in Section 3.1, but with this new encoding, it becomes even easier. We only need to define a closed type family which rewrites to No for any forbidden instance. The example used above where $Integral$ has only Int and $Integer$ is written as:

$$\begin{array}{l}\textbf{type family}\ IsIntegral\ t\ \textbf{where}\\ \quad IsIntegral\ Int\ \ \ \ = Yes\\ \quad IsIntegral\ Integer = Yes\\ \quad IsIntegral\ t\ \ \ \ = No\end{array}$$

The main difference with the **close** directive is that we need to define all instances in one place, whereas the directive defines a point after which no more instances can be added. It is possible to define a source-to-source processor which would rewrite an open type family into a closed one with a fallback default case, which would behave similarly to **close** if applied to those families which simulate type classes.

Another directive available in (Heeren and Hage 2005) is **disjoint** $C\ D$, which constraints any instance of C not to be instance of D, and vice versa. For example, we could forbid a type to be at the same instance of both $Integral$ and $Rational$. A naive encoding of this directive is done as follows for $Integral$, with a similar structure for $Rational$:[7]

$$\begin{array}{l}\textbf{type family}\ IsIntegral\ t\ \textbf{where}\\ \quad IsIntegral\ t = IsICheckR\ t\ (IsRational\ t)\\ \textbf{type family}\ IsICheckR\ t\ (isRational :: Defined) :: Defined\ \textbf{where}\\ \quad IsICheckR\ t\ Yes = No\\ \quad IsICheckR\ t\ No\ = IsIntegral'\ t\\ \textbf{type family}\ IsIntegral'\ t :: Defined\end{array}$$

The idea is that $IsIntegral$, by calling $IsICheckR$, checks whether a $Rational$ instance is present. If not, then it checks whether we have an explicit $Integral$ instance, represented by $IsIntegral'$. Thus, for adding new instances, the latter needs to be extended.

$$\begin{array}{l}\textbf{type instance}\ IsIntegral'\ Int\ \ \ \ \ = Yes\\ \textbf{type instance}\ IsIntegral'\ Integer = Yes\end{array}$$

[7] If we try to define $IsIntegral$ and $IsRational$ as type synonyms, we get a complaint of cyclic definition:

```
Cycle in type synonym declarations:
 type IsIntegral t = IsICheckR t (IsRational t)
 type IsRational t = IsRCheckI t (IsIntegral t)
```

Unfortunately, this naive encoding does not work. When trying to deduce $IsIntegral$, the compiler loops: indeed, $IsIntegral$ calls $IsRational$, which in turn calls $IsIntegral$ and so on. One possible solution is changing $IsIntegral$ to:

$$\begin{array}{l}\textbf{type family}\ IsIntegral\ t\ \textbf{where}\\ \quad IsIntegral\ t = IsICheckR\ t\ (IsRational'\ t)\end{array}$$

The objective of this change is breaking the loop by directly detecting whether we have a $Rational$ instance. This works well in the case in which we do not have an $Integral$ instance because of a $Rational$ one, as GHCi shows:

```
*> :kind! IsIntegral Float
IsIntegral Float :: Defined
= 'No
```

But in those cases where an explicit $IsIntegral$ rule is provided, the system is unable to reduce the type, since it does not know what $IsRational'$ rewrites to:

```
*> :kind! IsIntegral Int
IsIntegral Int :: Defined
= IsICheckR Int (IsRational' Int)
```

As a last attempt, we might try to check $IsIntegral$ and $IsRational$ values at the same time. For this, we introduce an $OnlyFirstDefined$ closed family which describes the disjointess condition:

$$\begin{array}{l}\textbf{type}\ IsIntegral\ t = OnlyFirstDefined\ (IsIntegral'\ t)\ (IsRational'\ t)\\ \textbf{type family}\ OnlyFirstDefined\ yes\ no :: Defined\ \textbf{where}\\ \quad OnlyFirstDefined\ Yes\ no\ = Yes\\ \quad OnlyFirstDefined\ yes\ Yes = No\end{array}$$

But once again we encounter the same problem: if the type does not have a defined $IsIntegral'$ rule, the system is not able to continue to the next branch in the type family. At this point, we admit defeat, and have not found a good way to encode **disjoint** directly as type families, as we have done for **never** and **close**.

4.4 Instance chains

Instance chains were introduced in (Morris and Jones 2010) as an extension to type classes in which to encode certain patterns that would otherwise require overlapping instances. The new features are *alternation*, that is, allowing different branches in an instance declaration, and *explicit failure*, which means that you can state negative information about instances.

One case where overlapping instances are needed in common Haskell is the definition of the $Show$ instance for lists: in this case, a special instance is used for strings, that is $[Char]$. With this extension, the exception will be handled as an instance chain:

$$\begin{array}{l}\textbf{instance}\ Show\ [Char]\ \textbf{where}\\ \quad show = \ldots\ \ \text{-- Special case for strings}\\ \textbf{else instance}\ Show\ [a]\ \textbf{if}\ Show\ a\ \textbf{where}\\ \quad show = \ldots\ \ \text{-- Common case}\end{array}$$

$Show$ also gives us an example of explicit failure: in general, we cannot make an instance for functions $a \to b$. However, if the domain of the function supports the $Enum$ class, we can give an instance which traverse the entire set of input values. In any other case, we want the system to explicitly know that no instance is possible:

$$\begin{array}{l}\textbf{instance}\ Show\ (a \to b)\ \textbf{if}\ (Enum\ a, Show\ a, Show\ b)\ \textbf{where}\\ \quad show = \ldots\\ \textbf{else instance}\ Show\ (a \to b)\ \textbf{fails}\end{array}$$

As we did for type class directives, we can encode these cases using our type family translation. The first thing we notice is that the $Show$ instance chain follows the pattern of the open-closed

type families: we must allow adding new rules for those types not already covered by other rules, but for some cases we need to make some ordered distinction, which takes the form of a closed family. We also apply the transformation of contexts as seen in Section 2.2. Putting it all together, the corresponding *IsShow* type family reads:

```
type family IsShow t :: Defined
type instance IsShow [a] = IsShowList a
type family IsShowList a where
  IsShowList Char = Yes
  IsShowList a    = IsShow a
type instance IsShow (a → b)
  = IsShowFn (IsEnum a) (IsShow a) (IsShow b)
type family IsShowFn isEnum isShowA isShowB where
  IsShowFn Yes Yes Yes = Yes
  IsShowFn e   a   b   = No
```

The family works nicely given some initial *IsShow* rules for *Bool*:

```
type instance IsShow Bool = Yes
```

```
*> :kind! IsShow (Bool -> [Char])
IsShow (Bool -> [Char]) :: Defined
= 'Yes
```

It is interesting to notice what happens if we ask for the information of a type which we have not explicitly mentioned, such as *Int*:

```
*Main> :kind! IsShow (Maybe Bool -> [Char])
IsShow (Int -> [Char]) :: Defined
= IsShowFn (IsEnum Int) (IsShow Int) 'Yes
```

The rewriting is stuck in the phase of rewriting *IsEnum Int* and *IsShow Int*. Intuitively, we may want the system to instead continue to the next branch, and return *No* as result. However, this poses a *threat to the soudness* of the system: since the type inference engine is not complete in the presence of type families, it may well be that *IsEnum Int* $\sim$ *Yes*, but the proof could not be found. If we decided to continue, and that proof finally exists, then the inference step we made is not correct. For this reason, we forbid taking the next branch until rewriting contradicts the expected results. A similar reasoning holds for the use of *apartness* to continue with the next branch in closed type families (Eisenberg et al. 2014).

Essentially, what we do by rewriting instance chains into type families is making explicit the *backtracking* needed in these cases. In principle, Haskell does not backtrack on type class instances, but by rewriting across several steps, we simulate it.

4.5 Better error messages

Until now, the only possibilities for a type family corresponding to a type class were to return *Yes* or *No*, or to get stuck. But this is very uninformative, especially in the case of a negative answer: we know that there is no instance of a certain class, but why is this the case? The solution is to add a field to the *Defined* type to keep failure information.

```
data Defined e = Yes | No e
```

We have decided to keep the error type *e* open, so each type class could have its own way to report errors. In the case of a closed one, it makes sense to have a specific closed data type. But in open scenarios, like *IsShow*, we need something more extensible. A good match is the *Symbol* kind, which is the type-level equivalent of strings, and which has special support in GHC for writing type-level literals. Thus, the *IsShow* type family is changed to:

```
import GHC.TypeLits   -- defines Symbol
type family IsShow t :: Defined Symbol
```

An instance like functions could benefit from reporting different errors depending on the constraint that failed: [8]

```
type instance IsShow (a → b)
  = IsShowFn (IsEnum a) (IsShow a) (IsShow b)
type family IsShowFn (isEnum  :: Defined Symbol)
                     (isShowA :: Defined Symbol)
                     (isShowB :: Defined Symbol) where
  IsShowFn Yes Yes Yes = Yes
  IsShowFn (No e) a b
    = No "Function with non-enumerable domain"
  IsShowFn e (No a) b
    = No "Source type must be showable"
  IsShowFn e a (No b)
    = No "Target type must be showable"
```

The interpreter will now return the corresponding message if the function is known to be not showable:

```
*> :kind! IsShow (Float -> Bool)
IsShow (Float -> Bool) :: Defined Symbol
= 'No "Function with non-enumerable domain"
```

Currently, *Symbol* values cannot be easily manipulated. In a scenario where simple functions such as concatenation are present in the standard libraries, more complete error messages could be obtained by joining information from different sources. For example, when *IsEnum* returns *No*, its message could be combined in *IsShownFn*, assuming the presence of a $(:++:)$ type family to perform string concatenation:

```
IsShowFn (No e) a b
  = No ("Function with non-enumerable domain"
        :++: "\nbecause " :++: e)
```

In conclusion, the extra control we get by explicitly describing how to search for *Show* instances via the *IsShow* type family also helps us to better pinpoint to the user where things go wrong. This is especially important in many scenarios, such as embedded domain-specific languages (Hage 2014).

4.6 Functional dependencies

There is one feature of type classes that we have not yet covered in the translation to type families, namely, *functional dependencies*. A simple functional dependency, such as that relating *c* and *e* in:

```
class Collection c e | c → e where ...
```

can be split, as shown in (Schrijvers et al. 2007), into a type class for the relation (which would in turn be translated into a type family as discussed in this section), and another type function for defining *e* in terms of *c*:

```
type family IsCollection' c e :: Defined
type instance IsCollection' [e] e = Yes
type family IsCollectionElement c
type instance IsCollectionElement [e] = e
```

However, this split does not guarantee that the types related by *IsCollection'* and *IsCollectionElement* satisfy any constraint. Of course, you want the result of *IsCollectionElement* to be the same as the *e* in *IsCollection'*. This can be enforced by defining a synonym *IsCollection* which relates both type families via an equality constraint over the element type:

[8] As we discussed earlier, GHC needs kind signatures in some cases. Here, had we not included *Defined Symbol* on *IsShowFn* arguments, GHC would expect *Defined*$*$ as its kinds, which is not correct.

```
type IsCollection c e = And (IsCollection' c e)
  (EqDef e (IsCollectionElement c))
```

The *EqDef* type family just reifies type equality into *Defined*:

```
type family EqDef a b :: Defined where
  EqDef a a = Yes
  EqDef a b = No
```

Most uses of functional dependencies can be translated by the above schema. The reason is that in most cases, functional dependencies are just used to define type-level functions with instance arguments.

Some cases are more difficult to cope with, though, like the dependencies that you may add to addition. Essentially, when you know two arguments that make up a sum, you know the other one by simply adding or by cancellation law:

```
class AddFD m n r | m n → r, r m → n, r n → m
```

Note that if you try to give instances for this type class, such as:

```
instance AddFD Zero     n     n
instance AddFD (Succ m) Zero (Succ m)
instance AddFD m n r
  ⇒ AddFD (Succ m) (Succ n) (Succ (Succ r))
```

the compiler will complain because of a conflict in functional dependencies: if the second and third arguments are given, it cannot deduce the first one, because there is always some overlap with the first rule. However, let us suppose for a moment that we could use functional dependencies in that way: how would it translate into type families?

To get a complete answer, we need to look at the two different ways in which functional dependencies influence the type system:[9]

- *FD-improvement*: if the context contains *AddFD m1 n1* r_1 and *AddFD m2 n2 r2*, and we know that $m1 \sim m2$ and $n1 \sim n2$, then we have $r_1 \sim r2$;
- *Instance improvement*: if the context contains *AddFD m n r*, and for some substitution of *m* and *n* only one instance matches, then we can use it to rewrite *r*. For example, if we have *AddFD Zero n r*, we know inmediately that $n \sim r$.

In type family terms (where we define the corresponding *IsAddFD* family as shown above), FD-improvement translates into obtaining $r_1 \sim r2$ knowing that $m1 \sim m2$, $n1 \sim n2$ and, here comes the crux of the matter, *IsAdd m1 n1* $r_1 \sim$ *IsAdd m2 n2 r2*. Thus, the functional dependency constraint becomes a partial *injectivity* constraint in the family: if the results of a function, and some of its arguments (in this case, *m* and *n*) agree for two applications, we know that remaining argument (here, *r*) must also agree. A simple form of injectivity for type families has been considered for GHC, but has not been implemented as of version 7.8.[10]

On the other hand, instance improvements correspond to the ability of defining and *inverting* type-level functions from the instance relations. The functional dependency $m\ n \to r$ on *AddFD* is doing nothing more than defining the addition function in the type level (as shown in the Introduction), if we want to encode the other two, we need to invert addition:

```
type family IsAddRNToM where
  IsAddRNToM r      Zero     = r
  IsAddRNToM (Succ r) (Succ n) = IsAddRNToM r n
type family IsAddRMToN where
  IsAddRMToN r      Zero     = r
  IsAddRMToN (Succ r) (Succ m) = IsAddRMToN r m
```

While several approaches to bidirectionalization of functional programs have been proposed (Foster et al. 2012), it is not always possible or desirable to use bidirectionalization. Looking at type classes with our type family glasses can help decide when a certain functional dependency will be useful: if you cannot get the corresponding function out of it, the instance improvement rule may never be applied.

5. Comparison

5.1 Type families as functional dependencies

Sections 2 and 3 looked at how to deal with features not readily available in type classes or families. In Section 4 we turned to type families as an integrating framework for both concepts. In previous literature (Schrijvers et al. 2007) type classes with functional dependencies were used as the integrating glue: why is our choice any better?

The answer lies in the use of *instance improvement* by functional dependencies, as discussed in 4.6. This type of improvement makes type inference brittle: it depends on the compiler proving that only one instance is available for some case, which can be influenced by the addition of another, not related, instance for a class.

Other different problems with functional dependencies have been discussed in (Schrijvers et al. 2007; Diatchki 2007), usually concluding that type-level functions are a better option. In this paper we agree with that statement, and we show that families could replace even more features of type classes by using other Haskell extensions such as data type promotion and closed type functions.

5.2 Implicit arguments

In essence, in Section 4 we are describing a new way to deal with type-level programming which needs to decide whether a certain proposition holds while elaborating some piece of code. This comes close to the *instance arguments* feature found in Agda (Devriese and Piessens 2011), which was also proposed to simulate type classes. Any argument marked as such in a function with double braces, like:

$$myFunction : \{A : Set\} \to \{\{p : Show\ A\}\} \to A \to String$$

will be replaced by any value of the corresponding type in the environment in which it was called. Thus, if you think of *Show* of a class, you can provide an instance by constructing such a value:

```
showInt : Show Int
showInt x = ...   -- code for printing an integer
```

Since these values are constructed at the term level, you can use any construct available for defining functions. In that sense, it is close to our use of type families, with the exception that in Haskell type-level and term-level programming are completely separated. A difference between both systems is that Agda does not do any proof search when looking for instance arguments, whereas our solution can simulate search with backtracking.

5.3 Tactics

The dependently type language Idris (Brady 2013) generalises the idea of Agda's instance arguments allowing the programmer to customise the search strategy for implicit arguments. Similarly to Coq, Idris has a tactic language to customise proof search. Unlike Coq, however, Idris allows the programmer to use the same machinery to customise the search for implicit arguments (The Idris Community 2014).

[9] Using the terminology from `https://ghc.haskell.org/trac/haskell-prime/wiki/FunctionalDependencies`.

[10] GHC Trac ticket on Injective type families: `https://ghc.haskell.org/trac/ghc/ticket/6018`.

For example we can write a function of the following type, where t is a tactic script that is used for searching the implicit argument of type $Show\ a$:

$$myFunction : \{\textbf{default}\ tactics\ \{t\}\ p : Show\ a\} \to a \to String$$

The tactic t itself is typically written using reflection such that it can inspect the goal type – in this case $Show\ a$ – and perform the search accordingly:

$$myFunction : \{\textbf{default}\ tactics\ \{applyTactic\ findShow; solve\}\ p : Show\ a\} \to a \to String$$

The search strategy is defined by *findShow*, which is an Idris function of that takes the goal type and the context as argument and produces a tactic to construct a term of the goal type.

This setup is similar to *closed* type families with elaboration as presented in this paper. However, *findShow* has to operate on terms of Idris core type theory *TT*, which is quite cumbersome. Moreover, there is no corresponding setup for *open* type families.

6. Conclusion

Type classes and type families in Haskell have different sets of features. However, with a little work we can support elaboration and contexts in families, and closedness in instances. This suggests that there exists a framework for integrating the two as instances of a single concept: we show how type families can serve as such a concept. By creating type families which simulate classes, we get for free features such as type class directives, instance chains and control over the search procedure. We have argued that it is possible to add an elaboration mechanisms to type families to bridge the gap for its use in ad-hoc polymorphism.

References

P. Bahr. Composing and decomposing data types: A closed type families implementation of data types la carte. 10th ACM SIGPLAN Workshop on Generic Programming, to appear, 2014.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. . URL `http://journals.cambridge.org/article_S095679681300018X`.

D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. . URL `http://doi.acm.org/10.1145/2034773.2034796`.

I. S. Diatchki. *High-level abstractions for low-level programming*. PhD thesis, OGI School of Science & Engineering, May 2007.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL `http://doi.acm.org/10.1145/2535838.2535856`.

N. Foster, K. Matsuda, and J. Voigtlnder. Three complementary approaches to bidirectional programming. In J. Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32201-3. . URL `http://dx.doi.org/10.1007/978-3-642-32202-0_1`.

J. Hage. Domain specific type error diagnosis (DOMSTED). Technical Report UU-CS-2014-019, Department of Information and Computing Sciences, Utrecht University, 2014.

B. Heeren and J. Hage. Type class directives. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*, PADL'05, pages 253–267, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24362-3, 978-3-540-24362-5. . URL `http://dx.doi.org/10.1007/978-3-540-30557-6_19`.

M. Izbicki. A neat trick for partially closed type families. Blog post available at `http://izbicki.me/blog/a-neat-trick-for-partially-closed-type-families`, 2014.

M. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67262-3. . URL `http://dx.doi.org/10.1007/3-540-46425-5_15`.

J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. . URL `http://doi.acm.org/10.1145/1863543.1863596`.

S. Peyton Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.

T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. Chakravarty. Towards open type functions for haskell. In *19th International Symposium on Implemantation and Application of Functional Languages*, 2007.

StackOverflow. Closed type classes. Question and answers available at `http://stackoverflow.com/questions/17849870/closed-type-classes`, 2013.

The Idris Community. Programming in Idris: A tutorial. Available from `http://eb.host.cs.st-andrews.ac.uk/writings/idris-tutorial.pdf`, 2014.

D. Vytiniotis, S. Peyton jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions, Sept. 2011. ISSN 0956-7968. URL `http://dx.doi.org/10.1017/S0956796811000098`.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. . URL `http://doi.acm.org/10.1145/75277.75283`.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. . URL `http://doi.acm.org/10.1145/2103786.2103795`.

A. Formal translation from classes to families

In Section 4 we looked at the translation from type classes to families, but left out the technical details. This section deals with those details and the associated soundness and termination properties. We leave functional dependencies out of this discussion, since they come with their own set of difficulties, as shown in Section 4.6.

There are three Haskell constructs to translate: classes, contexts and instances. Type class declarations are of the form $\textbf{class}\ D\ t_1 \ldots tn$. Each of them gives raise to a new type family encoded as:

$$\textbf{type family}\ IsD\ t_1 \ldots t_m :: Defined$$

Here, *Defined* is the kind which represents whether an instance is available. It was introduced in Section 4.1 and refined in Section 4.5 to get better error messages. In addition, types t_1 to t_m may include kind annotations inferred from their use in the elaborated methods.

Note that we have not spoken about superclass contexts: they do not interfere with instance resolution, just impose a constraint of having to define an instance of each superclass. In this case, given a class $S \Rightarrow D$, the constraint would translate to having to define IsS to return Yes each time IsD returns Yes. Thus, superclasses impose their conditions on a prior stage to type checking.

The second construct to translate are context declarations of the form $Q\ s_1\ ..\ s_j$, which may appear in function signatures, data types or other instance declarations. The translation is $IsQ\ s_1 \ldots s_j$.

Finally, we need to translate instance declarations. Each instance may have a number of context declarations, say n:

$$\textbf{instance}\ (Q_1, ..., Q_n) \Rightarrow D\ t_1\ ...\ t_m$$

A type family instance is defined for each of them, of the form:

$$\textbf{type instance}\ IsD\ t_1\ ...\ t_m = And_n\ Q_1\ ...\ Q_n$$

For each number n of context declarations, we have a corresponding And_n closed type family which checks that all the arguments are Yes. More formally, we have:

$$\begin{array}{l}\textbf{type family}\ And_0 :: Defined\\ \quad And_0 = Yes\\ \textbf{type family}\ And_1\ d :: Defined\\ \quad And_1\ x = x\\ \textbf{type family}\ And_n\ d_1\ ...\ d_n :: Defined\\ \quad And_n\ Yes\ ...\ Yes = Yes \quad \text{-- case everything Yes}\\ \quad And_n\ d_1\ \ ...\ d_n\ \ = No\end{array}$$

In the translation, Q_1 to Q_n refer to the translation of instance constraints Q_1 to Q_n as given above.

A.1 OUTSIDEIN(X)

The current reference for type inference for Haskell, including type classes, type families and other extensions such as generalized algebraic data types is (Vytiniotis et al. 2011). The authors describe the inference process in terms of a general framework, called OUTSIDEIN(X), which is parametrized by a constraint system X. Each constraint system defines a concrete entailment $\mathcal{Q} \Vdash W$ which gives semantics to certain constraint Q under the axioms in the set $\mathcal{Q}$. Axioms are the generic name given to declarations such as class and family instances.

In particular, we are interested in the case X = type classes and type families, that is also discussed in (Vytiniotis et al. 2011). For this case, many rules are given for the concrete entailment $\Vdash$. Many of them deal, such are those dealing with reflexivity, symmetry and transitivity are quite straightforward:

$$\frac{}{\mathcal{Q} \Vdash \tau \sim \tau}\ \text{REFL} \qquad \frac{\mathcal{Q} \Vdash \tau_1 \sim \tau_2}{\mathcal{Q} \Vdash \tau_2 \sim \tau_1}\ \text{SYM}$$

$$\frac{\mathcal{Q} \Vdash \tau_1 \sim \tau_2 \qquad \mathcal{Q} \Vdash \tau_2 \sim \tau_3}{\mathcal{Q} \Vdash \tau_1 \sim \tau_3}\ \text{TRANS}$$

The rules related to type classes and type families are:

$$\frac{\mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash F\ \overline{\tau_1} \sim F\ \overline{\tau_2}}\ \text{FCOMP}$$

$$\frac{\mathcal{Q} \Vdash D\ \overline{\tau_1} \qquad \mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash D\ \overline{\tau_2}}\ \text{DICTEQ}$$

$$\frac{\forall \overline{a}.Q_1 \Rightarrow Q_2 \in \mathcal{Q} \qquad \mathcal{Q} \Vdash [\overline{a \mapsto \tau}]Q_1}{\mathcal{Q} \Vdash [\overline{a \mapsto \tau}]Q_2}\ \text{AXIOM}$$

The first two rules define how type equality distributes over instance constraints and type family applications. The last one describes the application of axioms: if we can prove the preconditions of an axiom for an specific substitution $[\overline{a \mapsto \tau}]$, then we can conclude the postcondition in the axiom. Note than in the case of type family instances, Q_1 is always empty, so the rule in that case reads:

$$\frac{\forall \overline{a}.F\ \overline{\rho} \sim \sigma \in \mathcal{Q}}{F\ \overline{[\overline{a \mapsto \tau}]\rho} \sim [\overline{a \mapsto \tau}]\sigma}\ \text{AXIOM'}$$

A.2 Soundness of translation

In OUTSIDEIN(X), entailment relations are parametrized by a set of axioms $\mathcal{Q}$, which can be either type class or type family instances. We define $\mathcal{Q}^{trans}$ as the set of axioms obtained by translating each instance axiom as defined above.

Lemma 1. *If $\mathcal{Q} \Vdash \bigwedge_i D_i\ \overline{\tau_i} \sim Yes$, then $\mathcal{Q} \Vdash And_n\ \overline{D_i\ \overline{\tau_i}} \sim Yes$.*

Proof. By case analysis of the definition of And_n. □

Theorem 1. *If $\mathcal{Q} \Vdash D\ \overline{\tau}$, then $\mathcal{Q}^{trans} \Vdash IsD\ \overline{\tau} \sim Yes$.*

Proof. By inversion of the rule applied to get $\mathcal{Q} \Vdash D\ \overline{\tau}$. There are only two interesting cases, DICTEQ and AXIOM.

For DICTEQ, taking into account the translation, proving $\mathcal{Q}^{trans} \Vdash IsD\ \overline{\tau} \sim Yes$ boils down to proving the soundness of this rule:

$$\frac{\mathcal{Q} \Vdash IsD\ \overline{\tau_1} \qquad \mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash IsD\ \overline{\tau_2}}$$

The following derivation shows how to get it:

$$\frac{\mathcal{Q} \Vdash IsD\ \overline{\tau_1} \qquad \dfrac{\mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash IsD\ \overline{\tau_1} \sim IsD\ \overline{\tau_2}}\ \text{FCOMP}}{\mathcal{Q} \Vdash IsD\ \overline{\tau_2}}\ \text{SYM, TRANS}$$

For AXIOM, first note that instance axioms of the form $\overline{Q} \Rightarrow Q^*$ get translated into type family axioms of the form $Q^* \sim And_n\ \overline{IsQ\ \overline{q}}$. Thus, we need to prove soundness of the rule:

$$\frac{\forall \overline{a}.IsD\ \overline{\sigma} \sim And_n\ \overline{IsQ\ \overline{q}} \in \mathcal{Q} \qquad \mathcal{Q} \Vdash \bigwedge IsQ\ [\overline{a \mapsto \tau}]\overline{q} \sim Yes}{\mathcal{Q} \Vdash IsD\ [\overline{a \mapsto \tau}]\overline{\sigma} \sim Yes}$$

We can derive the first premise by using AXIOM:

$$\frac{\forall \overline{a}.IsD\ \overline{\sigma} \sim And_n\ \overline{IsQ\ \overline{q}} \in \mathcal{Q}}{IsD\ [\overline{a \mapsto \tau}]\overline{\sigma} \sim And_n\ \overline{IsQ\ [\overline{a \mapsto \tau}]\overline{q}}}\ \text{AXIOM}$$

For the second premise, first apply the induction hypothesis to convert the proofs of the context of the rule. Then, use the previous lemma to get the version with And_n:

$$\frac{\mathcal{Q} \Vdash \bigwedge IsQ\ [\overline{a \mapsto \tau}]\overline{q} \sim Yes}{\mathcal{Q} \Vdash And_n\ \overline{IsQ\ [\overline{a \mapsto \tau}]\overline{q}} \sim Yes}$$

Using SYM and TRANS we get the desired result. □

A.3 Termination

An important issue to consider is whether termination characteristics of class instances are also carried over to the translated families. The most lenient conditions imposed by GHC over class instances[11] are the so-called *Paterson conditions*. For each constraint $Q\ s_1\ ...\ s_j$ in the instance context:

1. No type variable has more occurrences in the constraint than in the instance head.
2. The constraint has fewer constructors and variables (taken together and counting repetitions) than the head.

In the case of type families $F\ t_1\ ...\ t_m = s$, the conditions imposed by GHC ask that for each type family application $G\ \ r_1\ ...\ r_k$ appearing in s, we have:

[11] If the user does not turn on the *UndecidableInstances*, which turns off any termination checking.

1. Each of the arguments $r_1 .. r_k$ do not contain any other type family applications.
2. The total number of data type constructors and variables in $r_1 .. r_k$ is strictly smaller than in $t_1 .. t_m$.
3. Each variable occurs in $r_1 .. r_k$ at most as often as in $t_1 .. t_m$.

The translation of a class instance which satisfies the Paterson conditions into a type family instance:

$$\textbf{type instance } IsD\ t_1 \ldots t_m = And_n\ Q_1 \ldots Q_n$$

satisfies the terminations conditions (2) and (3) of type families. However, condition (1) is not satisfied, because And_n contains nested family applications. Note that these are the only nested applications generated by the translation.

The key point is observing that each application of And_n adds just one extra rewriting step. If type families fulfill their termination conditions (2) and (3), And_n just adds a number of steps bounded by the size of the derivation tree. Thus, termination is still guaranteed.

Really Natural Linear Indexed Type Checking

Arthur Azevedo de Amorim
University of Pennsylvania

Marco Gaboardi
University of Dundee

Emilio Jesús Gallego Arias
University of Pennsylvania

Justin Hsu
University of Pennsylvania

Abstract

Recent works have shown the power of *linear indexed type systems* for capturing complex safety properties. These systems combine *linear type systems* with a language of *indices* that appear in the types, allowing more fine-grained analysis. For example, linear indexed types have been fruitfully applied to verify differential privacy in the *Fuzz* type system.

A natural way to enhance the expressiveness of this approach is by allowing the indices to depend on runtime information, in the spirit of dependent types. This approach is used in *DFuzz*, an extension of *Fuzz*. The *DFuzz* type system relies on an index-level language supporting real and natural number arithmetic over constants and dependent variables. Moreover, *DFuzz* uses a subtyping mechanism to semantically manipulate indices. By themselves, linearity, dependency, and subtyping each require delicate handling when performing type checking or type inference; their combination increases this challenge substantially, as the features can interact in non-trivial ways.

In this paper, we study the type-checking problem for *DFuzz*. We show how we can reduce type checking for (a simple extension of) *DFuzz* to constraint solving over a first-order theory of naturals and real numbers which, although undecidable, can often be handled in practice by standard numeric solvers.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Type structure

Keywords type checking, type inference, linear types, subtyping, sensitivity analysis

1. Introduction

Linear indexed type systems have been used to ensure safety properties of programs with respect to different kinds of resources; examples include usage analysis [24, 25], implicit complexity [4, 5, 14], sensitivity analysis [10, 23], automatic timing analysis [12, 13], and more. Linear indexed types use a type-level *index language* to describe resources and *linear types* to reason about the program's resource usage in a compositional way.

A limitation of current analysis techniques for such systems is that resource usage is inferred independently of the control flow of a program—e.g. the typing rule for branching usually approximates resources by taking the maximal usage of one of the branches, and recursion imposes even greater restrictions. To improve this scenario, some authors have proposed extending such systems with dependent types, using type indices to capture both resource usage and the *size information* of a program's input. This significantly enriches the resulting analysis by allowing resource usage to depend on runtime information. Linear dependent type systems have been used in several domains, including implicit complexity [4, 16] and sensitivity analysis [10].

Of course, there is a price to be paid for the increase in expressiveness: type checking and type inference become inevitably more complex. In linear indexed type systems, these tasks are often done in two stages: a standard Hindley-Milner-like pass, followed by a constraint-solving procedure. In some cases, the generated constraints can be solved automatically by using custom algorithms [17] or off-the-shelf SMT solvers [7, 13]. However, the constraints are specific to the index language, and richer index languages often lead to more complex constraints.

Type-checking *DFuzz*

In this paper we will focus on the type-checking problem for a particular programming language with linear dependent types, *DFuzz*. Reed and Pierce [23] recently proposed the *Fuzz* programming language, where linear indexed types are used to reason about sensitivity of programs in the context of differential privacy; the *sensitivity* of a function measures the distance between outputs on nearby inputs. In this setting, type checking and inference correspond to sensitivity analysis.

Fuzz uses real numbers as indices for the linear types. Then addition and multiplication of the indices will produce an upper bound on the sensitivity of the program. This approach gives a simple but effective sensitivity static analysis. Indeed, as shown by D'Antoni et al. [7], type-checking for *Fuzz* programs can be performed efficiently by using an SMT solver to discharge the numeric proof obligations arising from the type system. Moreover, the same approach works for type inference, which infers the minimal sensitivity of a function.

While *Fuzz* works well on a variety of simple programs, it has a fundamental limitation: sensitivity information cannot depend on runtime information, such as the size of a data structure. To get around this problem, Gaboardi et al. [10] introduced *DFuzz*, an extension of *Fuzz* with a limited form of dependent types.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

The index language in *DFuzz* combines information about the *size* of data structures with information about the *sensitivity* of functions. Technically, this is achieved by considering an index language with index variables ranging over integers (to refer to runtime sizes) and reals (to refer to runtime sensitivities). This richer index language, combined with dependent pattern-matching and subtyping, achieves increased expressiveness in the analysis, providing sensitivity bounds beyond *Fuzz*'s capabilities.

However, adding variables to the index language has a significant impact on the difficulty of type checking. Concretely, since the index language also supports addition and multiplication, index terms are now *polynomials* over the index variables. Instead of constraints between real constants like in *Fuzz*, type checking constraints in *DFuzz* may involve general polynomials.

A natural first approach is to try to extend the algorithm proposed by D'Antoni et al. [7] to work with the new index language by simply generating additional constraints when dealing with the new language constructs. This would be similar in spirit to the work of Dal Lago et al. [6] for type inference for $\mathsf{d}\ell\mathsf{PCF}$, a linear dependent type system for complexity analysis. A crucial difference between that setting and *DFuzz* is that the index language of $\mathsf{d}\ell\mathsf{PCF}$ can be extended by arbitrary (computable) functions. This makes the approach to type inference for $\mathsf{d}\ell\mathsf{PCF}$ proposed by Dal Lago and Petit the most natural, since such functions can be used as direct solutions to some of the introduced constraints.

However, such an approach does not work as well for *DFuzz*, which opts for a much smaller index language. While it may be possible to extend *DFuzz*'s index language with general functions, we opt to keep the index language simple. Instead, since the type system of *DFuzz* also supports subtyping, we consider a different approach inspired by techniques from the literature on subtyping [21] and on constraint based type-inference approaches [15, 19, 22].

The main idea is to type-check a program by inferring some set of sensitivities for it, and then testing whether the resulting type is a subtype of the desired type. To obtain completeness (relative to checking the subtype), one must ensure that the inferred sensitivities are the "best" possible for that term. Unfortunately, the *DFuzz* index language is not rich enough for expressing such sensitivities. For instance, some cases require taking the maximum of two sensitivity expressions, something that cannot be done in the language of polynomials. We solve this problem by extending the index language with three syntactic constructs, resulting in a new type system that we name *EDFuzz*. This new system has meta-theoretic properties that are similar to those of *DFuzz*, but also simplifies the search for minimal sensitivities. Using these new constructs, we design a sensitivity-inference algorithm for *EDFuzz* which we show sound and complete, modulo constraint resolution.

We now face the problem of solving the constraints generated by our algorithm. First, we show how to compile the constraints generated by the algorithmic systems to constraints in the first-order theory over mixed integers and reals. This way, we can still use a numeric solver without resorting to custom symbolic resolution. Unfortunately, the presence of universal quantification over natural numbers in the constraints leads to undecidability of constraint solving; we show that *DFuzz* type-checking is undecidable, by reduction from Hilbert's tenth problem, a standard undecidable problem.

While this result shows that we can't have a terminating type-checker that is both sound and complete, not everything is lost. We first show that by approximating the constraints, we obtain a sound and computable method to type-check *EDFuzz* programs. We show that this procedure can successfully type-check a fragment of *EDFuzz* which we call *UDFuzz*; almost all of the examples proposed by Gaboardi et al. [10] belong to this class. Of course, *UDFuzz* is a strict subset of *EDFuzz*, and it is not hard to come up with well-typed programs in *EDFuzz* that are invalid under *UDFuzz*.

Finally, we present a constraint simplification procedure that can significantly reduce the complexity of our translated constraints (measured by the number of alternating quantifiers), even when checking full *EDFuzz*.

Contributions

We briefly overview the *DFuzz* programming language in Section 2, to move to an informal exposition of the main challenges involved in Section 3. Then, we present the main contributions of the paper:

- *EDFuzz*: an extension of *DFuzz* with a more expressive sensitivity language that allows to type programs with more precise types (Section 4);
- a sound and complete algorithm that reduces type checking and inference in *EDFuzz* to constraint solving over the first-order theory of $\mathbb{N}$ and $\mathbb{R}$ (Section 5 and Section 6);
- a proof of undecidability of type checking in *DFuzz* (and *EDFuzz*) (Section 7);
- a sound translation from the previous type-checking constraints to the first-order theory of the real numbers, a decidable theory (Section 8.1); and
- a simplification procedure to make the constraints more amenable to automatic solving (Section 8.2).

2. The *DFuzz* System

DFuzz [10] is a type system for verifying differential privacy. While the precise application of *DFuzz* is somewhat beyond the scope of this paper, at a high level, *DFuzz* is a system for checking function *sensitivity*. Given a notion of *distance* between values, a function f is said to be k-*sensitive* for some number k if $dist(f(x), f(y)) \leq k \cdot dist(x, y)$. Sensitivities are expressed by the index language in a linear indexed type system; let us begin by presenting *DFuzz* in some detail before discussing the type-checking challenges.

2.1 Syntax and Types

DFuzz is an extension of PCF with indexed linear types. Indices consist of numeric constants; index-level variables, which range over *sizes* (natural numbers) or *sensitivities* (positive reals extended with ∞, denoted $\mathbb{S}$); and addition and multiplication of indices. The full syntax for *DFuzz*, including the types, terms, and the index language, is shown in Figure 1. We take a brief tour through the term language.

- Abstraction and application for index variables are captured by the $\Lambda i : \kappa.e$ and $e[R]$ terms, with κ representing the kind for i. We refer to variables of natural number kind as *size variables*, while variables of real number kind are *sensitivity variables*.
- Singleton types $\mathbb{N}[S]$ and $\mathbb{R}[R]$ are used to related type-level sizes and sensitivities with term-level sizes and sensitivities.
- Dependent pattern matching over $\mathbb{N}[S]$ types is captured by the **case** construction.
- Linear types indexed by R are written $!_R\sigma \multimap \tau$.
- Variable environments Γ carry an additional annotation for assignments $x :_{[R]} \sigma$, representing the current sensitivity R for the variable x.
- Index variable environments ϕ specify the kinding of index variables.
- Constraint environments Φ store assumptions introduced under dependent pattern matching. Often, we will think of a constraint environment as the conjunction of its constraints.

$$
\begin{array}{lcll}
\kappa & ::= & \mathrm{r} \mid \mathrm{n} & \text{(kinds)} \\
\mathbb{S} & ::= & \mathbb{R}^{\geq 0} \cup \{\infty\} & \text{(extended positive reals)} \\
S & ::= & i \mid 0 \mid S+1 & \text{(sizes)} \\
R & ::= & \mathbb{S} \mid i \mid S \mid R+R \mid R \cdot R & \text{(sensitivities)} \\
\sigma, \tau & ::= & \mathbb{R} \mid \mathbb{R}[R] \mid \mathbb{N}[S] \mid !_R \sigma \multimap \tau & \text{(types)} \\
 & \mid & \forall i : \kappa.\ \sigma \mid \sigma \otimes \tau \mid \sigma \,\&\, \tau & \\
e & ::= & x \mid \mathbb{N} \mid \mathbf{s}\ e \mid \mathbb{R}^{\geq 0} \mid \mathbf{fix}\ (x : \sigma).e & \text{(expressions)} \\
 & \mid & \lambda x :_{[R]} \sigma.e \mid e_1\ e_2 & \\
 & \mid & \Lambda i : \kappa.\ e \mid e[R] & \\
 & \mid & \langle e_1, e_2 \rangle \mid \pi_i\ e & \\
 & \mid & (e_1, e_2) \mid \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e' & \\
 & \mid & \mathbf{case}\ e\ \mathbf{of}\ 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s & \\
\Gamma, \Delta & ::= & \emptyset \mid \Gamma, x :_{[R]} \sigma & \text{(environments)} \\
\phi, \psi & ::= & \emptyset \mid \phi, i : \kappa & \text{(sens. environments)} \\
\Phi, \Psi & ::= & \top \mid \Phi, S = 0 \mid \Phi, S = i + 1 & \text{(constraints)}
\end{array}
$$

Figure 1. *DFuzz* Types and Expressions

2.2 Environment Operations

As is the case for many linear type systems, *DFuzz* defines operations on variable environments. Precisely, two environments Γ, Δ can be combined with addition, and a single environment Γ can multiplied by a sensitivity (a sort of environment scaling). Throughout, we will write $\mathrm{dom}(\Gamma)$ for Γ's domain.

We define environment multiplication $R \cdot \Gamma$ as the operation taking every element $x_i :_{[r_i]} \sigma_i$ of Γ to $x_i :_{[R \cdot r_i]} \sigma_i$. Environment addition is defined iff all the common assignments of Γ, Δ map to the same type, that is to say, forall x_i in $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$, $(x_i :_{[R_i]} \sigma_i) \in \Gamma \iff (x_i :_{[S_i]} \sigma_i) \in \Delta$. In such case:

$$
\begin{array}{rcll}
\Gamma + \Delta & = & \{x_i :_{[R_i + S_i]} \sigma & \mid x_i \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)\} \\
 & \cup & \{x_j :_{[R_j]} \sigma_j & \mid x_j \in \mathrm{dom}(\Gamma) - \mathrm{dom}(\Delta)\} \\
 & \cup & \{x_k :_{[R_k]} \sigma_k & \mid x_k \in \mathrm{dom}(\Delta) - \mathrm{dom}(\Gamma)\}
\end{array}
$$

2.3 Subtyping

DFuzz has a notion of subtyping, which intuitively corresponds to a standard property of function sensitivity: a k-sensitive function is also k'-sensitive for all $k' \geq k$. Furthermore, subtyping in *DFuzz* is the mechanism that allows types to use information from the constraint environment; in this use, subtyping allows a form of type coercion. We consider here a slightly simpler definition of subtyping than the one used in Gaboardi et al. [10]. In the environments we requires subtyping to preserve the internal type. This slight modification will allow us to simplify some rules of the type-checking algorithm.

The semantics of the subtying relation is defined by interpreting sensitivity expressions as functions that produce sensitivity values. Formally, let R be a sensitivity expression, well-typed under environment ϕ, and ρ a suitable variable *valuation* (i.e., a function that maps each variable $x : \kappa$ in ϕ to an element of $[\![\kappa]\!]$, with $[\![\mathrm{n}]\!] = \mathbb{N}$ and $[\![\mathrm{r}]\!] = \mathbb{S}$). We then define $[\![R]\!]_\rho$ as follows:

$$
\begin{array}{rcll}
[\![0]\!]_\rho & := & 0 & \\
[\![S+1]\!]_\rho & := & [\![S]\!]_\rho + 1 & \\
[\![i]\!]_\rho & := & \rho(i) & i \text{ a variable} \\
[\![r]\!]_\rho & := & r & r \text{ a constant} \\
[\![R_1 + R_2]\!]_\rho & := & [\![R_1]\!]_\rho + [\![R_2]\!]_\rho & \\
[\![R_1 \cdot R_2]\!]_\rho & := & [\![R_1]\!]_\rho \cdot [\![R_2]\!]_\rho &
\end{array}
$$

Then, the standard ordering $\geq$ on $\mathbb{S}$ (i.e., the positive real numbers with a maximal element ∞) induces an ordering on index terms, which we can then extend to a subtype relation $\sqsubseteq$ on types and environments; the rules can be found in Figure 2. Note that

$$
\frac{}{\phi; \Phi \models \sigma \sqsubseteq \sigma} \ \sqsubseteq\text{-Refl}
$$

$$
\frac{\phi; \Phi \models \sigma' \sqsubseteq \sigma \qquad \phi; \Phi \models \tau \sqsubseteq \tau'}{\phi; \Phi \models \sigma \,\&\, \tau \sqsubseteq \sigma' \,\&\, \tau'} \ (\sqsubseteq .\, \&)
$$

$$
\frac{\phi; \Phi \models \sigma \sqsubseteq \sigma' \qquad \phi; \Phi \models \tau \sqsubseteq \tau'}{\phi; \Phi \models \sigma \otimes \tau \sqsubseteq \sigma' \otimes \tau'} \ (\sqsubseteq .\otimes)
$$

$$
\frac{\models \forall \phi.\ (\Phi \Rightarrow R \leq R') \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma \qquad \phi; \Phi \models \tau \sqsubseteq \tau'}{\phi; \Phi \models !_R \sigma \multimap \tau \sqsubseteq !_{R'} \sigma' \multimap \tau'} \ (\sqsubseteq .\multimap)
$$

$$
\frac{\phi, i : \kappa; \Phi \models \sigma \sqsubseteq \tau \qquad i \text{ fresh in } \phi}{\phi; \Phi \models \forall i : \kappa.\ \sigma \sqsubseteq \forall i : \kappa.\ \tau} \ (\sqsubseteq .\forall)
$$

$$
\frac{\begin{array}{c}\forall (x :_{[R_i]} \sigma_i, x :_{[R'_i]} \sigma_i) \in (\Gamma, \Delta) \\ \mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma) \qquad \models \forall \phi.\ (\Phi \Rightarrow R_i \geq R'_i)\end{array}}{\phi; \Phi \models \Gamma \sqsubseteq \Delta} \ \sqsubseteq\text{-Env}
$$

Figure 2. *DFuzz* Subtyping Relation

checking happens under the current constraint environment Φ, so subtyping may use information recovered from a dependent match.

The leaves of the subtype derivation are either equalities that are consequences of the constraint environment Φ, or assertions $\phi \models (\Phi \Rightarrow R_1 \geq R_2)$. These are defined logically as

$$
\forall \rho.(\mathsf{dom}(\rho) = \phi \wedge \rho(\Phi)) \Rightarrow [\![R_1]\!]_\rho \geq [\![R_2]\!]_\rho,
$$

where the quantification is over all well-kinded substitutions ρ for variables specified by ϕ satisfying the constraints Φ.

2.4 Typing

Typing judgments for *DFuzz* are of the form

$$
\phi; \Phi \mid \Gamma \vdash e : \sigma
$$

meaning that term e has type σ under environments ϕ and Γ and constraints Φ; full rules are shown in Figure 3.

We highlight here just the most complex rule, the dependent pattern matching rule $(\mathbb{N}\ E)$, which allows each branch to be typed under different assumptions on the type $\mathbb{N}[S]$ of the scrutinee (e). The left branch e_0 is typed under the assumption $S = 0$, while the right branch e_s is typed under the assumption $S = i + 1$ for some i. Indeed, this rule is useful for capturing programs whose sensitivity depends on the number of iterations or number of input elements; combined with the fix rule (Fix), these features enable programs that iterate depending on a runtime parameter while still reasoning about the number of iterations. Readers interested in more details can consult Gaboardi et al. [10]; we follow their presentation closely except for a few points, which we detail in the Appendix.

2.5 Examples

We close the overview of *DFuzz* with some examples. The first example is multiplication. Usually, multiplication cannot be assigned a type as is not sensitive for any k. However, thanks to dependent types we can introduce a multiplication primitive with type:

$$
\times : \forall R_1 : \mathrm{r}. \forall R_2 : \mathrm{r}. !_{R_1} \mathbb{R}[R_1] \multimap !_{R_2} \mathbb{R}[R_2] \multimap \mathbb{R}[R_1 \cdot R_2]
$$

A function that adds ϵ noise to the output has type:

$$
\mathrm{add\_noise} : \forall \epsilon : \mathrm{r}. !_\epsilon \mathbb{R} \multimap \bigcirc \mathbb{R}
$$

where $\bigcirc \mathbb{R}$ is the type of probability distributions over $\mathbb{R}$.

$$\frac{\phi;\Phi \mid \Delta \vdash e : \sigma \qquad \phi;\Phi \models \Gamma \sqsubseteq \Delta}{\phi;\Phi \mid \Gamma \vdash e : \sigma} \quad (\sqsubseteq .\mathrm{L}) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \sigma \qquad \phi;\Phi \models \sigma \sqsubseteq \tau}{\phi;\Phi \mid \Gamma \vdash e : \tau} \quad (\sqsubseteq .\mathrm{R})$$

$$\frac{r \in \mathbb{R}}{\phi;\Phi \mid \Gamma \vdash r : \mathbb{R}} \quad (\mathrm{Const}_{\mathbb{R}}) \qquad \frac{n = [\![S]\!]}{\phi;\Phi \mid \Gamma \vdash n : \mathbb{N}[S]} \quad (\mathrm{Const}_{\mathbb{N}})$$

$$\frac{}{\phi;\Phi \mid \Gamma, x :_{[1]} \sigma \vdash x : \sigma} \quad (\mathrm{Var}) \qquad \frac{\phi;\Phi \mid \Gamma, x :_{[\infty]} \sigma \vdash e : \sigma}{\phi;\Phi \mid \infty \cdot \Gamma \vdash \mathbf{fix}\ (x : \sigma).e : \sigma} \quad (\mathrm{Fix})$$

$$\frac{\phi;\Phi \mid \Gamma, x :_{[R]} \sigma \vdash e : \tau}{\phi;\Phi \mid \Gamma \vdash \lambda x :_{[R]} \sigma.e : !_R\sigma \multimap \tau} \quad (\multimap I) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e_1 : !_R\sigma \multimap \tau \qquad \phi;\Phi \mid \Delta \vdash e_2 : \sigma}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau} \quad (\multimap E)$$

$$\frac{\phi, i : \kappa;\Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi, \Gamma}{\phi;\Phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma} \quad (\forall I) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi;\Phi \mid \Gamma \vdash e[S] : \sigma[S/i]} \quad (\forall E)$$

$$\frac{\phi;\Phi \mid \Gamma_1 \vdash e_1 : \sigma \qquad \phi;\Phi \mid \Gamma_2 \vdash e_2 : \tau}{\phi;\Phi \mid \Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \sigma \otimes \tau} \quad (\otimes I) \qquad \frac{\phi;\Phi \mid \Delta \vdash e : \sigma \otimes \tau \qquad \phi;\Phi \mid \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \vdash e' : \mu}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e' : \mu} \quad (\otimes E)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi;\Phi \mid \Gamma \vdash e_2 : \tau}{\phi;\Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \,\&\, \tau} \quad (\&\ I) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \sigma_1 \,\&\, \sigma_2}{\phi;\Phi \mid \Gamma \vdash \pi_i\ e : \sigma_i} \quad (\&\ E)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e : \mathbb{N}[S]}{\phi;\Phi \mid \Gamma \vdash \mathbf{s}\ e : \mathbb{N}[S+1]} \quad (\mathrm{S}\ I) \qquad \frac{\begin{array}{c}\phi;\Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi;\Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n};\Phi, S = i+1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \text{ fresh in } \phi\end{array}}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\ E)$$

Figure 3. *DFuzz* Typing Rules

Functions sensitive on number of iterations or size of the input are similarly typed. A function that adds noise i times to an input is:

$$\mathsf{iNoise} : \forall i : \mathrm{n}, \forall \epsilon : \mathrm{r}.!_\infty\mathbb{N}[i] \multimap !_\infty\mathbb{R}[\epsilon] \multimap !_{i\cdot\epsilon}\mathbb{R} \multimap \mathbb{R}$$

3. The Challenge of Type-checking Linear Dependent Types

Type-checking a language with linear indexed types presents several challenges, which are only compounded when dependent types and subtyping are added to the mix. In this section, we take a closer look at these challenges.

3.1 To Split, or not to Split?

The first problem we face is due to linearity. Given a term and an environment, we need a way to "split" the environment into appropriate sub-environment that can be used in the recursive calls to type check subterms.

Automatically inferring the right environments in our setting is difficult, due to the index language for *DFuzz*. Indeed, index terms are polynomials over index variables, which may range over the reals or the naturals. For instance, we may know that a particular variable x has sensitivity $i^2 \cdot j^2 + 3$ in our environment. However, it is not clear how to split such sensitivity information between two environments that share the variable x. In fact, as we will show below, in general it is not always possible to find a split. One might hope to simplify the type-checking task by requiring the programmer to provide a few type annotations, like in non-linear type systems. Unfortunately, this approach is impractical for the splitting problem because naively, the annotations must describe the split for every variable binding in the context!

To better understand this obstacle, let us consider two general approaches to type-checking linear type systems, which we call the *top-down* and *bottom-up* strategies.

The Downfall of Top-Down

For the type-checking problem, suppose we are given the environment Γ, a term e, and a purported type σ. The goal is to decide if $\Gamma \vdash e : \sigma$ is derivable. The *top-down* strategy takes a context and a term, and attempts to partition the context and recursively type the subterms of e.

The main difficulty of this approach centers around splitting the environment, a problem that is most clear in the application rule. Here is a simplified version:

$$\frac{\Gamma \vdash f : !_R\sigma \multimap \tau \qquad \Delta \vdash e : \sigma}{\Gamma + R \cdot \Delta \vdash f\ e : \tau}$$

So given a type-checking problem $\Sigma \vdash f\ e : \sigma'$ our first difficulty is to pick R, Γ, and Δ such that $\Sigma = \Gamma + R \cdot \Delta$. We could try to guess R, but unfortunately it may depend on the choice of Γ. Since our index language contains the real numbers, the number of possible splittings isn't even finite.

A natural idea is to delay the choice of this split. For instance, we may create a placeholder variable R and placeholder environments Γ', Δ', asserting $\Sigma = \Gamma' + R \cdot \Delta'$ and recursively type-checking f and e. After reaching the leaves of the derivation, we would have a set of constraints whose satisfiability would imply that the program type-checks.

Unfortunately, the constraints seem difficult to solve due to the syntactical nature of our indices. In other words, the "placeholder variables" are really *meta-variables* that range over index terms, which could potentially depend on bound index variables. In order to prove soundness of such a system with respect to the formal typing system, the solver must return success *only* if there is a solution where all the meta-variables can be instantiated to an index term—a syntactic object. This is at odds with the way most solvers work—*semantically*—finding arbitrary solutions over their domain.

It is not clear how to solve these existential constraints automatically for the specific index language of *DFuzz*.

The Rise of Bottom-Up?

A different approach is a *bottom-up* strategy: suppose we are again given an environment Γ, a term e, and a type σ, and we want to check if $\Gamma \vdash e : \sigma$ is derivable. The main idea is to avoid splitting environments by calculating the minimal sensitivities needed for typing each subexpression. For each typing rule, these minimal sensitivities can be combined to find the resulting minimal sensitivities for e. Once this is done, we just need to check whether these optimal sensitivities are compatible with Γ and σ via subtyping.

Let's consider how this works in more detail by analyzing a few important cases. At the base case, we type-check variables in a minimal context (that is, empty but for the variable) an assigning it the minimal sensitivity required:

$$\frac{}{x :_{[1]} \sigma \vdash x : \sigma}$$

Recall that we have weakening on the left so can add non-occurring variables to the context later.

Now, the key benefit of the bottom-up approach becomes evident in the application rule: we can completely avoid the splitting problem. When faced with a type-checking instance $\Sigma \vdash f\ e : \sigma$, we recursively find optimal Γ, R, and Δ for checking f and e; then, checking that $\Sigma \sqsubseteq \Gamma + R \cdot \Delta$ suffices.

Unfortunately, things don't look so easy in the additive rules. Let's examine the introduction rule for $\&$:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \,\&\, \sigma_2}$$

This rule forces both environments to have the same sensitivities, but the bottom-up idea may infer different environments for each expression:

$$\frac{\Gamma_1 \vdash e_1 : \sigma_1 \qquad \Gamma_2 \vdash e_2 : \sigma_2}{\Sigma? \vdash \langle e_1, e_2 \rangle : \sigma_1 \,\&\, \sigma_2}$$

Now we need to guess a best environment $\Sigma?$, but the *DFuzz* sensitivity language is too weak to express this value. For instance, if we consider sensitivity expressions r^2 and r depending on a sensitivity variable r, we can show that there is no minimal polynomial upper bound for them under the point-wise order on polynomials[1].

To maintain the minimality invariant, we can extend the sensitivity language with a new syntactic construct $\mathbf{max}(R_1, R_2)$ for type-inference purposes only, which should denote the maximum of two sensitivity values. We could then safely set $\Sigma? := \mathbf{max}(\Gamma_1, \Gamma_2)$, where the expression combines sensitivities for the bindings on both environments as expected.

However, there is a problem with this approach: the resulting algorithm is not sound with respect to the original type system, because it allows more terms to be typed even when sensitivities in the final type do not mention the new construct! To see this, assume that our algorithm produces a derivation $\Gamma' \vdash e : \sigma'$ using extended sensitivities. Now, soundness amounts to showing that for all Γ, σ mentioning only standard sensitivities such that $\Gamma \sqsubseteq \Gamma'$ and $\sigma' \sqsubseteq \sigma$, there exists a typing derivation $\Gamma \vdash e : \sigma$ that uses only the original sensitivity language. Let's try to sketch how this proof would work by restricting our attention to a particular instance of the application rule:

$$\frac{\phi; \emptyset \mid \emptyset \vdash f : !_{R_f}\sigma \multimap \tau \qquad \phi; \emptyset \mid x :_{[\hat{R}_x]} \mu \vdash e : \sigma}{\phi; \emptyset \mid x :_{[R_f \cdot \hat{R}_x]} \mu \vdash f\ e : \tau}$$

[1] Indeed, it can be seen that *DFuzz* does not possess minimal types. Refer to the Appendix for a more detailed proof.

where $\hat{R}_x$ is an extended sensitivity expression. By induction, we know that for all standard sensitivity expressions R_x such that $R_x \geq \hat{R}_x$, we can obtain a standard derivation $x :_{[R_x]} \mu \vdash e : \sigma$. We also have standard R_{xf} such that $R_{xf} \geq R_f \cdot \hat{R}_x$. Thus, all we need to do is to calculate from R_f, R_{xf} standard sensitivities R'_f, R'_x to be able to apply both induction hypotheses. The following result shows that this is not always possible.

Lemma 1. *Given standard sensitivities expressions R_{xf}, R_f and an extended sensitivity expression $\hat{R}_x$ such that $R_{xf} \geq R_f \cdot \hat{R}_x$, it is not the case that one can always find standard R'_f, R'_x such that $R_{xf} \geq R'_f \cdot R'_x \wedge R'_f \geq R_f \wedge R'_x \geq \hat{R}_x$.*

Proof. Take $R_{xf} = r^2 + 1$, $R_f = r$ and $\hat{R}_x = \mathbf{max}(2, r)$. As we can see, we have $r^2 + 1 \geq r \cdot \mathbf{max}(2, r)$, with a strict equality iff $r = 1$. Suppose there exist standard sensitivity expressions R'_f, R'_x such that they satisfy the statement. Because $R'_f \geq r$ and $R'_x \geq \mathbf{max}(2, r)$, we know by asymptotic analysis that the degree of R'_f and R'_x must be at least 1. Furthermore, because $r^2 + 1 \geq R'_f \cdot R'_x$, their degree must be exactly 1, with leading coefficient equal to 1. Write $R'_f = r + a$ and $R'_x = r + b$, where a, b are positive constants. The lower bound on R'_x implies $b \geq 2$. For $r = 1$, we have $R'_f \cdot R'_x \geq 3a + 3 \geq 3$. However, the lower and upper bounds for $R'_f \cdot R'_x$ coincide at that point, forcing $R'_f \cdot R'_x = 2$; contradiction. Thus, no such R'_f, R'_x can exist. □

It is not hard to adapt the above into a counterexample for the soundness of the algorithm with respect to the original system. However, we can recover soundness by extending the sensitivity language for the basic typing rules as well.

3.2 Avoiding the Avoidance Problem

After the addition of least upper bounds for sensitivities, the bottom-up approach is in a good working state for the basic system. However, other constructs in the language introduce further challenges. In particular, let's examine a simple version of the abstraction rule for sensitivity variables:

$$\frac{\phi, i : \kappa \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Gamma}{\phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma}$$

When this rule is interpreted in a top-down approach, usually no problem arises; we would just introduce the new sensitivity variable and proceed with type checking.

However, when the typechecking direction is reversed, we hit a version of the avoidance problem [8, 11, 18]. The avoidance problem usually appears in slightly different scenarios related to existential types, and could be informally stated as finding a best type free of a particular variable. In our case, we must find the "best" Γ free of i. It may not be obvious how i could have been propagated to Γ, but indeed, a function f in e could have a type like as $!_i\sigma \multimap \tau$, and applying f will introduce i into the environment in the bottom-up approach.

Fortunately, in our setting, we can easily solve the avoidance problem by further extending the sensitivity language. The "best" way of freeing a sensitivity expression R of a variable i is to take the supremum of R over all possible values of i, which we denote by $\mathbf{sup}(i, R)$[2]. Then, the minimal environment is $\mathbf{sup}(i, \Gamma)$, where the supremum is extended to each binding in the environment.

[2] Contrary to $\mathbf{max}(-,-)$, it would have been possible to define this construct as a function over sensitivity expressions, without the need to extend their syntax. This would still be true even after introducing index-level case sensitivity expression for analyzing dependent pattern matching. As the translation is somewhat intricate and leads to more complex constraints, we chose to add it directly to the syntax of sensitivity expressions.

3.3 Undependable Dependencies

The last case to consider in our informal overview is **case**, also referred as dependent pattern matching.

The dependent pattern matching can be considered as a special case of the two previous difficulties. Like the least upper bound, we must compute a least upper bound of the resources used in two branches. However, now the information coming from the successor branch may also contain sensitivities depending on the newly introduced refinement variable, which cannot occur in the upper bound; similar to the avoidance problem we just discussed. On top of that, information coming from both sides is conditional on the particular refinements induced by the match, so any new sensitivity information that we propagate cannot really depend on the refinements.

We now face a choice: we can introduce refinement types over sensitivity and size variables of the form $\{\sigma \mid P(\vec{i})\}$, which would allow us to express the sensitivty inference for **case** in term of the least upper bound and supremum operations. However, we take a simpler path and add a conditional operator on natural number expressions S, $\mathbf{case}(S, R_0, i, R_s)$, interpreted as R_0 if S is 0 or $R_s[i \mapsto S-1]$ if $S \geq 1$.

In the next sections we proceed to formally introduce the extended sensitivities and its semantics; we discuss the type-checking algorithm, which depends on solving inequality constraints over the extended sensitivities; and we study several approaches and discuss their decidability.

4. Extended *DFuzz*: *EDFuzz*

We define a conservative extension to *DFuzz*'s type system, *EDFuzz*, which is basically *DFuzz* with an extended sensitivity language for the indices. We summarize the new sensitivity terms:

- $\mathbf{max}(R_1, R_2)$ is the pointwise least upper bound of sensitivity terms R_1, R_2.
- $\mathbf{sup}(i, R)$ is the pointwise least upper bound of R over all i.
- $\mathbf{case}(S, R_0, i, R_s)$ is the conditional function on the size expression S that is valued R_0 when $S = 0$, and $R_s[i \mapsto S-1]$ when S is a strictly positive integer.

We write $\hat{R}$ for the *extended sensitivity language*, built from the standard sensitivity terms and operations and the new extended terms. The semantics of extended terms are defined as follows.

Definition 2 (Extended sensitivity semantics)**.** *For every well-kinded valuation $\phi \models \rho$ for $\phi \models R$ we have:*

$$[\![\mathbf{sup}(i:\kappa, \hat{R})]\!]_\rho := \sup_{r \in \kappa}\{[\![\hat{R}]\!]_{\rho \cup [i=r]}\}$$

$$[\![\mathbf{max}(\hat{R}_1, \hat{R}_2)]\!]_\rho := \max([\![\hat{R}_1]\!]_\rho, [\![\hat{R}_2]\!]_\rho)$$

$$[\![\mathbf{case}(S, \hat{R}_0, i, \hat{R}_s)]\!]_\rho := \begin{cases} [\![\hat{R}_0]\!]_\rho & \text{if} \quad [\![S]\!]_\rho = 0 \\ [\![\hat{R}_s]\!]_{\rho \cup [i=n-1]} & \text{if} \quad [\![S]\!]_\rho = n \geq 1. \end{cases}$$

$$[\![\hat{R}_1 + \hat{R}_2]\!]_\rho := [\![\hat{R}_1]\!]_\rho + [\![\hat{R}_2]\!]_\rho$$

$$[\![\hat{R}_1 \cdot \hat{R}_2]\!]_\rho := [\![\hat{R}_1]\!]_\rho \cdot [\![\hat{R}_2]\!]_\rho.$$

We define analogous operations on contexts in the obvious way. For instance, if $x :_{R_1} \sigma \in \Gamma_1$ and $x :_{R_2} \sigma \in \Gamma_2$, then $x :_{\mathbf{max}(R_1,R_2)} \sigma \in \mathbf{max}(\Gamma_1, \Gamma_2)$. Context operations that take two contexts Γ_1, Γ_2 are only defined if the contexts have the same skeleton, i.e., $\Gamma_1^\bullet = \Gamma_2^\bullet$.

It is not hard to show that any derivation valid in *DFuzz* remains valid in *EDFuzz*. Furthermore, *DFuzz*'s metatheory only relies on sensitivity terms having an interpretation as total function from free variables to a real number, rather than on any specific property about the interpretation itself. The extended interpretation is total, and hence the metatheory of *DFuzz* extends to *EDFuzz*.

5. Type Checking and Inference

We present a sound and complete type checking and inference algorithm for *EDFuzz*. The algorithm assumes the existence of an oracle for deciding the subtyping relation, so in that sense our algorithm is relatively complete. We defer discussion about solving subtyping constraints to the next section.

We remind the reader that our definitions of type-checking and inference assume that a regular typing derivation—that is to say, erasing all linear types and dependent terms—for an expression is already known. This can be computed, for example, by a Hindley-Milner-style pass. Here and below, we focus on handling the sensitivities. For a type σ, we write $\overline{\sigma}$ for the type where all linear types are mapped to regular function types and all the dependently typed types are mapped to their non-dependent version. This erasure operation is extended to environments in the natural way: Given an environment Γ, we define its skeleton as $\overline{\Gamma}$, containing a list of type bindings $(x : \sigma)$, but *without* the external sensitivities (i.e., the annotation on the colon).

Definition 3 (Type Checking)**.** *Given a context Γ, a term e, a type σ, and a HM derivation $\overline{\Gamma} \vdash_H e : \overline{\sigma}$, then the type-checking problem for* EDFuzz *is to determine whether a derivation $\emptyset; \emptyset; \Gamma \vdash e : \sigma$ exists.*

In our context, type inference means inferring the sensitivity annotations in both a context and a type.

Definition 4 (Type Inference)**.** *Given a context skeleton $\overline{\Gamma}$, a term e, a regular type σ, and a HM derivation $\overline{\Gamma} \vdash_H e : \sigma$, the type-inference problem is to compute a context Γ and a type τ such that a derivation $\emptyset; \emptyset; \Gamma \vdash e : \tau$ exists and $\overline{\Gamma} = \Gamma$ and $\overline{\tau} = \sigma$.*

5.1 The Algorithm

We can fulfill both goals using an algorithm that takes as inputs a term e, an environment free of sensitivity annotations $\Gamma^\bullet$ and a refinement constraint Φ. The algorithm will output an annotated environment Δ and a type σ. We write a call to the type inference algorithm as:

$$\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \sigma.$$

Figure 4 presents the full algorithm in a judgmental style. The algorithm is based on a syntax-directed version of DFuzz that enjoys several nice properties; full technical details can be found in the Appendix. Here, we just sketch how the transformation works in the proofs of soundness and completeness.

Theorem 5 (Algorithmic Soundness)**.** *Suppose $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$. Then, there is a derivation of $\phi; \Phi; \Gamma \vdash e : \sigma$.*

Proof. We define two intermediate systems: The first one internalizing certain properties of weakening and a second, syntax-directed. The algorithm is a direct transcription of the syntax-directed system and soundness can be proved by induction on the number of steps. We prove soundness of the syntax-directed system by induction on the syntax-directed derivation. □

Theorem 6 (Algorithmic Completeness)**.** *Suppose $\phi; \Phi; \Gamma \vdash e : \sigma$ is derivable. Then $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma'; \sigma'$ and $\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \wedge \sigma' \sqsubseteq \sigma$.*

Proof. We show that a "best" syntax-directed derivation can be build from any standard derivation by induction on the original derivation plus monotonicity and commutativity properties of the subtype relation. Completeness for the algorithm follows. □

$$\frac{}{\phi; \Phi; \Gamma^\bullet; r \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet); \mathbb{R}} \quad (\mathrm{Const}) \qquad \frac{n = [\![S]\!]}{\phi; \Phi; \Gamma^\bullet; n \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet); \mathbb{N}[S]} \quad (\mathrm{Const}_{\mathbb{N}})$$

$$\frac{}{\phi; \Phi; \Gamma^\bullet, x : \sigma; x \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma; \sigma} \quad (\mathrm{Var}) \qquad \frac{\phi; \Phi; \Gamma^\bullet, x : \sigma; e \Longrightarrow \Gamma, x :_{[R']} \sigma; \tau \quad \phi; \Phi \models R \geq R'_{\square\uparrow}}{\phi; \Phi; \Gamma^\bullet; \lambda(x :_{[R]} \sigma).\ e \Longrightarrow \Gamma; !_R\sigma \multimap \tau} \quad (\multimap I)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma; !_R\sigma \multimap \tau \quad \phi; \Phi; \Delta^\bullet; e_2 \Longrightarrow \Delta; \sigma' \quad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi; \Gamma^\bullet; e_1\ e_2 \Longrightarrow \Gamma + R \cdot \Delta; \tau} \quad (\multimap E) \qquad \frac{\phi; \Phi; \Gamma^\bullet, x : \sigma; e \Longrightarrow \Gamma, x :_{[R]} \sigma; \sigma' \quad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi; \Gamma^\bullet; \mathbf{fix}\, x : \sigma.\ e : \sigma \Longrightarrow \infty \cdot \Gamma; \sigma} \quad (\mathrm{Fix})$$

$$\frac{\phi, i : \kappa; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma}{\phi; \Phi; \Gamma^\bullet; \Lambda i : \kappa.\ e \Longrightarrow \mathbf{sup}(i, \Gamma); \forall i : \kappa.\ \sigma} \quad (\forall I) \qquad \frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi; \Gamma^\bullet; e[S] \Longrightarrow \Gamma; \sigma[S/i]} \quad (\forall E)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma_1; \sigma_1 \quad \phi; \Phi; \Gamma^\bullet; e_2 \Longrightarrow \Gamma_2; \sigma_2}{\phi; \Phi; \Gamma^\bullet; \langle e_1, e_2 \rangle \Longrightarrow \Gamma_1 + \Gamma_2; \sigma_1 \otimes \sigma_2} \quad (\otimes I) \qquad \frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \sigma \otimes \tau \quad \phi; \Phi; \Gamma^\bullet, x : \sigma, y : \tau; e' \Longrightarrow \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau; \mu}{\phi; \Phi; \Gamma^\bullet; \mathbf{let}(x, y) = e\ \mathbf{in}\ e' \Longrightarrow \Gamma + \mathbf{max}(R_{1\square\uparrow}, R_{2\square\uparrow}) \cdot \Delta; \mu} \quad (\otimes E)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma_1; \sigma_1 \quad \phi; \Phi; \Gamma^\bullet; e_2 \Longrightarrow \Gamma_2; \sigma_2}{\phi; \Phi; \Gamma^\bullet; \langle e_1, e_2 \rangle \Longrightarrow \mathbf{max}(\Gamma_1, \Gamma_2); \sigma_1\ \&\ \sigma_2} \quad (\&\ I) \qquad \frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma_1\ \&\ \sigma_2}{\phi; \Phi; \Gamma^\bullet; \pi_i e \Longrightarrow \Gamma; \sigma_i} \quad (\&\ E)$$

$$\frac{\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \mathbb{N}[S]}{\phi; \Phi; \Gamma^\bullet; \mathbf{s}\ e \Longrightarrow \Gamma; \mathbb{N}[S+1]} \quad (\mathrm{S}\ I) \qquad \frac{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \mathbb{N}[S] \qquad \phi; \Phi, S = 0; \Gamma^\bullet; e_0 \Longrightarrow \Gamma_0; \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i+1; \Gamma^\bullet, x : \mathbb{N}[i]; e_s \Longrightarrow \Gamma_s, x :_{[R']} \mathbb{N}[i]; \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i+1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\begin{array}{l}\phi; \Phi; \Gamma^\bullet; \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \mapsto e_0 \mid x_{[i]} + 1 \mapsto e_s \\ \Longrightarrow \mathbf{case}(S, \Gamma_0, i, \Gamma_s) + \mathbf{case}(S, 0, i, R'_{\square\uparrow}) \cdot \Delta; \sigma\end{array}} \quad (\mathbb{N}\ E)$$

Figure 4. Algorithmic Rules for *EDFuzz*

5.2 Removing Sensitivity Annotations

We briefly discuss the role annotations play in our algorithm. *DFuzz* programs have three different annotations: the type of the argument for lambda terms (including the sensitivity), the return type for case, and the type for fixpoints.

The sensitivity annotations ensure that inferred types are free of terms with extended sensitivities. This is useful for some optimizations on subtype checking (introduced later in the paper). However, the general encoding of subtyping checks works with full extended types, thus the sensitivity annotations can be safely omitted and the system will infer types containing extended sensitivities.

Due to technical difficulties in inferring the minimal sensitivity in the presence of higher-order functions, the argument type in functions (σ in $\lambda(x : \sigma)$) must be annotated, and we require the type of fixpoints to be annotated.

6. Constraint Solving over Mixed Reals/Naturals

The type-checking algorithm introduced in the previous section produces inequality constraints over the extended sensitivity language. While these extended sensitivity terms may appear complicated, we can translate them into formulas in the first-order theory of $\mathbb{S}$ and $\mathbb{N}$ in a sound and complete way.

While we will show in the next section that the kind of first-order formulas we generate here are in general undecidable, they can still be handled by numeric solvers providing mixed real/natural theories. Moreover, in Section 8.1 we will present a sound (although not complete) computable procedure to check the constraints.

Quantification over $\mathbb{S}$ should be interpreted as quantification over $\mathbb{R}^\infty$ with a non-negativity constraint; all the quantifiers in our target first-order theory will range over either $\mathbb{R}^\infty$ or $\mathbb{N}$. (In the next section, we will show that quantifying over just $\mathbb{R}$ and $\mathbb{N}$ is enough.)

The idea behind our translation is simple: we use a first-order formula to uniquely specify each extended sensitivity term. In other words, we define a predicate $T(R)$ for each extended sensitivity term R, such that $[\![T(R)(r)]\!]_\rho$ holds exactly when r is equal to the interpretation of R under the valuation ρ. For instance, consider the translation for $R_1 + R_2$:

$$T(R_1{+}R_2)(r) := \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r = r_1 + r_2.$$

For ρ a valuation for R_1, R_2, we have $r_1 = [\![R_1]\!]_\rho$ and $r_2 = [\![R_2]\!]_\rho$. Then the only r that satisfies this predicate is

$$r = r_1 + r_2 = [\![R_1]\!]_\rho + [\![R_2]\!]_\rho = [\![R_1 + R_2]\!]_\rho,$$

as desired.

For a more involved example, consider the translation of $\mathbf{max}(R_1, R_2)$:

$$\begin{aligned} &T(\mathbf{max}(R_1, R_2))(r) \\ &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge \\ &(r_1 \geq r_2 \wedge r = r_1 \vee r_2 \geq r_1 \wedge r = r_2). \end{aligned}$$

Again, for any valuation ρ of R_1, R_2, we have $r_1 = [\![R_1]\!]_\rho$ and $r_2 = [\![R_2]\!]_\rho$. The final conjunction states that r must be the larger of r_1 and r_2, which is precisely the semantics we have given $[\![\mathbf{max}(R_1, R_2)]\!]_\rho$. The full translation is in Figure 5.

We formalize our intuitive explanation of the translation with the following lemma.

$$
\begin{aligned}
\kappa &:= \mathbb{N} \mid \mathbb{S} \\
T(i)(r) &:= i = r \\
T(R_1 + R_2)(r) &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r = r_1 + r_2 \\
T(R_1 \cdot R_2)(r) &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge r = r_1 \cdot r_2 \\
T(\mathbf{max}(R_1, R_2))(r) &:= \exists r_1\, r_2 : \mathbb{S}, T(R_1)(r_1) \wedge T(R_2)(r_2) \wedge (r_1 \geq r_2 \wedge r = r_1 \vee r_2 \geq r_1 \wedge r = r_2) \\
T(\mathbf{case}(S, R_0, i, R_s))(r) &:= \exists r_s : \mathbb{N}, T(S)(r_s) \wedge (r_s = 0 \wedge T(R_0)(r) \vee \exists i : \mathbb{N}, r_s = i + 1 \wedge T(R_s)(r)) \\
T(\mathbf{sup}(i : \kappa, R))(r) &:= \mathbf{bound}(i : \kappa, R, r) \wedge \forall r', \mathbf{bound}(i : \kappa, R, r') \Rightarrow r' \geq r \\
\mathbf{bound}(i : \kappa, R, r) &:= \forall i : \kappa, \exists r' : \mathbb{S}, T(R)(r') \wedge r' \leq r
\end{aligned}
$$

Figure 5. Constraint Translation

Lemma 7. *For every sensitivity expression R and $r \in \mathbb{S}$, and for every valuation ρ whose domain contains the free variables of R, $[\![T(R)(r)]\!]_\rho \iff r = [\![R]\!]_\rho$*

Proof. By induction on R. We have already considered the $R_1 + R_2$ and $\mathbf{max}(R_1, R_2)$ cases above. □

Using the translation of terms, we can translate sensitivity constraints generated by our typing algorithm. We map each constraint of the form

$$\models \forall \phi, \Phi \Rightarrow R_1 \geq R_2$$

for R_1 a standard sensitivity term to

$$\forall \phi, \Phi \Rightarrow \exists r : \mathbb{S}, T(R_2)(r) \wedge R_1 \geq r$$

Note that since R_1 is a standard sensitivity term, the resulting formula is a first-order formula in the theory of $\mathbb{S}$ and $\mathbb{N}$. Thanks to Lemma 7, both formulas are semantically equivalent.

7. Undecidability of Type-checking over Mixed Reals/Naturals

As we have seen in the previous section, constraints over our extended sensitivity language can be translated to simple first-order formulas. Taken by itself, this is not entirely satisfactory, as the first-order theory of $\mathbb{N}$ is already undecidable. A nice illustration of this is Hilbert's tenth problem, which asks if a polynomial equation of the form $P(\vec{x}) = 0$ over several variables has any solutions over the natural numbers. After several years of investigation, this property, easily definable in first-order arithmetic, was finally shown to be undecidable.

In this section, we will show that this problem is present in *DFuzz*: type-checking is undecidable. We begin with an auxiliary lemma.

Lemma 8. *Given polynomials P, Q over n variables with coefficients in $\mathbb{N}$, checking $\forall \vec{i} \in \mathbb{N}^n, P(\vec{i}) \geq Q(\vec{i})$ is undecidable.*

Proof. We will use a solution to our problem to solve Hilbert's tenth problem. Suppose we are given a polynomial P with integer coefficients, and we want to decide whether $\exists \vec{i} \in \mathbb{N}^n, P(\vec{i}) = 0$. This is equivalent to deciding $\neg \forall \vec{i} \in \mathbb{N}^n, P(\vec{i})^2 \geq 1$. Write $P(\vec{i})^2 = P^+(\vec{i}) - P^-(\vec{i})$, where P^+ and P^- have only positive coefficients. Then our condition is equivalent to $\neg \forall \vec{i} \in \mathbb{N}^n, P^+(\vec{i}) \geq P^-(\vec{i}) + 1$. Thus, we can solve Hilbert's tenth problem by using P^+ and $P^- + 1$ as inputs to our problem, which shows that it is undecidable. □

This class of constraints is important for *DFuzz*, as they can arise when checking the subtype relation.

Corollary 9. *The subtype relation of* DFuzz *is undecidable.*

Proof. Suppose we are given P and Q as previously. Consider the types $\sigma = \forall \vec{i}, !_0\mathbb{N}^n[\vec{i}] \multimap !_{Q(\vec{i})}\mathbb{R} \multimap \mathbb{R}$ and $\tau = \forall \vec{i}, !_0\mathbb{N}^n[\vec{i}] \multimap !_{P(\vec{i})}\mathbb{R} \multimap \mathbb{R}$. Then $\sigma \sqsubseteq \tau$ is equivalent to the previous problem, hence undecidable. □

Corollary 10. DFuzz *type checking is undecidable.*

Proof. Using recursion and dependent pattern matching, it is possible to write a function that multiplies a real number by a polynomial $Q(\vec{v})$ with variables ranging over $\mathbb{N}$. Its minimal type will clearly be σ. Therefore, type-checking it against τ is equivalent to deciding $\sigma \sqsubseteq \tau$, which is undecidable by Lemma 8. □

8. Approaches to Constraint Solving

Given that type-checking *DFuzz* (and hence also *EDFuzz*) is undecidable, is there anything more we can do besides feeding the constraints to a solver and hoping for the best? In this section, we discuss two possible directions to tackle these constraints. For both of these approaches, we require that *all annotations in the term are standard sensitivities*, rather than extended. Then, we have the following lemma. (We defer the proof to the Appendix.)

Lemma 11 (Standard Annotations)**.** *Assume annotations in a term e range over standard sensitivities and $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$. Then:*

- *σ has no extended sensitivities; and*
- *all constraints required for the algorithm are of the form $\models \forall \phi.\ (\Phi \Rightarrow R \geq R')$ where R is a standard sensitivity term.*

8.1 Modifying the subtype relation

As seen in the previous section, the *EDFuzz* subtyping relation is undecidable. Here, we explore a modified version of *EDFuzz*—which we call *UDFuzz*—that enjoys decidable typechecking. The modification is simple to describe: *UDFuzz* has all the same typing rules as *EDFuzz*, except we strongly restrict the subtyping relation to force all generated constraints to be decidable, and all annotations must be standard sensitivity terms. By restricting the subtype relation of *EDFuzz*, *UDFuzz* typeable programs are a strict subset of *EDFuzz*. This subtype restriction will rule out many programs that are typeable under *EDFuzz*, but is expressive enough to cover a range of examples (including most of the examples presented in the original work on *DFuzz* [10]).

Recall that the constraints handled by our algorithmic system have the form

$$\models \forall \phi.\ \Phi \Rightarrow R \geq R',$$

where R, R' are possibly extended sensitivity terms, and ϕ consists of both natural and real index variables. As we are requiring all annotations in *UDFuzz* standard sensitivities, then by Lemma 11, R will be a standard sensitivity term in *UDFuzz*; we use this invariant

to show the subtype relation of *UDFuzz* is a subrelation of the subtype relation of *EDFuzz*.

Furthermore, we note that the first order theory over $\mathbb{S}$ is decidable: we can try all settings variables to ∞ and check the resulting constraints (with all the remaining quantifiers ranging over $\mathbb{R}$). The resulting formula is in the first order theory over $\mathbb{R}$, and is decidable (as shown by Tarski). Hence, a natural idea is to replace quantification over the naturals with quantification over $\mathbb{S}$; let us first make this idea precise.

We define the semantics for sensitivity terms, where natural-kinded free variables may now be mapped to values in $\mathbb{S}$. We call this extension the *uniform* interpretation of size and sensitivity terms, and denote it by $[\![\cdot]\!]^U$. A well-formed *uniform valuation* $\phi \models^U \rho$ maps $\mathrm{dom}(\phi)$ to $\mathbb{S}$; note that "size variables" may be interpreted as real numbers, not just natural numbers.

First, the uniform interpretation of standard size and sensitivity terms is completely identical to the standard interpretation. The extended sensitivities have slightly different interpretations: $\mathbf{sup}(i, R)$ now takes a max over all real numbers, and $\mathbf{case}(S, R_0, i, R_s)$ must now be defined when the interpretation of S is not an integer.

$$[\![\mathbf{sup}(i:\kappa, \hat{R})]\!]^U_\rho := \sup_{r\in\mathbb{S}}\{[\![\hat{R}]\!]^U_{\rho\cup[i=r]}\}$$

$$[\![\mathbf{max}(\hat{R}_1, \hat{R}_2)]\!]^U_\rho := \max([\![\hat{R}_1]\!]^U_\rho, [\![\hat{R}_2]\!]^U_\rho)$$

$$[\![\mathbf{case}(S, \hat{R}_0, i, \hat{R}_s)]\!]^U_\rho := \begin{cases} [\![\hat{R}_1]\!]^U_\rho & \text{if} \quad [\![S]\!]^U_\rho = 0 \\ 0 & \text{if} \quad [\![S]\!]^U_\rho \in (0,1) \\ [\![\hat{R}_2]\!]^U_{\rho\cup[i=r-1]} & \text{if} \quad [\![S]\!]^U_\rho = r \geq 1. \end{cases}$$

$$[\![\hat{R}_1 + \hat{R}_2]\!]^U_\rho := [\![\hat{R}_1]\!]_\rho + [\![\hat{R}_2]\!]^U_\rho$$

$$[\![\hat{R}_1 \cdot \hat{R}_2]\!]^U_\rho := [\![\hat{R}_1]\!]^U_\rho \cdot [\![\hat{R}_2]\!]^U_\rho$$

$$[\![R]\!]^U_\rho := [\![R]\!]^U_\rho \qquad \text{otherwise.}$$

We first show that this uniform semantics is an extension of the standard semantics.

Lemma 12. *Suppose R is a standard sensitivity term, typed under context ϕ. Then, for any standard valuation $\phi \models \rho$, we have*

$$[\![R]\!]^U_\rho = [\![R]\!]_\rho.$$

Proof. Immediate from the definition of the interpretation. □

Now, we can define the uniform interpretation of constraints. A constraint

$$\models^U \forall\phi.\, \Phi \Rightarrow R \geq R'$$

is true exactly when for all real-valued valuations $\phi \models^U \rho$ satisfying Φ, we have $[\![R]\!]^U_\rho \geq [\![R']\!]^U_\rho$.

We are now ready to prove that the uniform interpretation of constraints is sound with respect to the original interpretation.

Theorem 13. *Suppose R, R' are well-typed in context ϕ. Suppose that*

$$\models^U \forall\phi.\, \Phi \Rightarrow R \geq R',$$

for R a standard sensitivity term. Then,

$$\models^U \forall\phi.\, \Phi \Rightarrow R \geq R'.$$

Proof. It suffices to show that for any standard valuation $\phi \models \rho$, we have $[\![R']\!]^U_\rho \geq [\![R']\!]_\rho$. (We defer the proof of this claim to the long version.) Assuming this, the theorem assumption shows that for all standard valuation $\phi \models \rho$, we have

$$[\![R]\!]^U_\rho \geq [\![R']\!]^U_\rho \geq [\![R']\!]_\rho.$$

But R is a standard sensitivity, so $[\![R]\!]^U_\rho = [\![R]\!]_\rho$ by Lemma 12, and we are done. □

Hence, the subtype relation of *UDFuzz* is a subrelation of the subtype relation in *EDFuzz*. By reasoning analgous to Lemma 7, we can show that relaxing the first order translation of constraints captures this uniform interpretation. More formally:

Lemma 14. *For every sensitivity term R, let $T^U(R)$ be a unary predicate defined exactly as in Figure 5, but replacing quantification over $\mathbb{N}$ with quantificiation over $\mathbb{S}$ and with the modified* **case** *translation:*

$$\begin{aligned} T^U(\mathbf{case}(S, R_0, i, R_s))(r) := \\ \exists r_s : \mathbb{N}, \quad & T(S)(r_s) \wedge (r_s = 0 \wedge T(R_0)(r)) \\ \vee \quad & (0 < r_s < 1 \wedge r = 0) \\ \vee \quad & (\exists i : \mathbb{N}, r_s = i + 1 \wedge T(R_s)(r)) \end{aligned}$$

Then, $r \in \mathbb{S}$, and for every uniform valuation ρ whose domain contains the free variables of R, $[\![T^U(R)(r)]\!]^U_\rho \iff r = [\![R]\!]^U_\rho$.

By this lemma, we can give a sound, complete and decidable type-checking algorithm for *UDFuzz*.

Theorem 15. *Suppose we use our algorithmic system, with the constraints*

$$\models^U \forall\phi.\, \Phi \Rightarrow R_1 \geq R_2$$

handled by translation to the first order formula

$$\forall\phi, \Phi \Rightarrow \exists r : \mathbb{S}, T^U(R_2)(r) \wedge R_1 \geq r,$$

where all quantifiers are over $\mathbb{S}$. Since the theory of $\mathbb{S}$ is decidable, this gives an effective type-checking procedure for UDFuzz.

Proof. Note that R_1 is a standard sensitivity term, so the translated formula is indeed a first order formula over the theory of $\mathbb{S}$. By Lemma 14, the translated formula is logically equivalent to

$$[\![\Phi]\!]^U_\rho \Rightarrow [\![R_1]\!]^U_\rho \geq [\![R_2]\!]^U_\rho$$

for all uniform valuations $\phi \models^U \rho$, which in turn implies $\phi; \Phi \models R_1 \geq R_2$ by Theorem 13. This shows that the algorithmic system is sound and complete with respect to *UDFuzz*. □

Remark 16. UDFuzz *is a strict subset of* EDFuzz*; informally, it contains* EDFuzz *programs with typing derivations that do not use facts true over $\mathbb{N}$ but not over $\mathbb{R}$. One key way that subtyping is used in* EDFuzz *is for equational manipulations of the indices; for instance, subtyping may be needed to change the index expression $3(i+1)$ to $3i+3$. This reasoning is available in* UDFuzz *as well; indeed, most of the example programs in* DFuzz *are typeable under* UDFuzz *as well. (The only exception is k-medians, which extends the index language with a division function that we do not handle.)*

However, there are many programs that lie in EDFuzz *but not in* UDFuzz*—constraints as simple as $\forall i.\ i^2 \geq i$ are true when quantifing over the naturals but not when quantifying over the reals. Valid* EDFuzz *programs that use these facts in their typing derivation will not lie in* UDFuzz.

8.2 Constraint Simplification

Rather than restricting the subtype relation, we can also try to generate simpler constraints when type-checking *EDFuzz*. While the translation of extended constraints to first order real theory is conceptually simple, the translation generates complex constraints; in particular, they may have many alternating quantifiers. In this section, present a rewriting procedure for reducing extended sensitivity terms, leading to simpler constraints. We continue to require that all source annotations must be standard sensitivity terms.

To begin, we generalize our three extended constructs with a new *constrained least upper bound* (**club**) operation, with form $\mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\}$. Here, ϕ is a size and sensitivity variable context, Φ is a constraint context, and R is a sensitivity term, extended or standard. The judgment for a well-formed

club is

$$\phi \models \mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\},$$

where each R_j has kind r under $\phi, \phi_j; \Phi_j$, and $\phi, \{\phi_j\}_j$ have disjoint domain. Intuitively, **club** is a maximum over a set of sensitivities, restricting to sensitivities where the associated constraint is satisfied. Sensitivities where the constraints are not satisfied are ignored. Formally, let ϕ contain the free variables of **club**, and let $\phi \models \rho$ be any standard valuation. We can give the following interpretation of **club**:

$$(\!|\mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\}|\!)_\rho := \max_{j \in [n]} \max\{[\![R_j]\!]_{\rho \cup \rho_j} \mid \phi_j \models \rho_j \text{ and } \rho, \rho_j \models \Phi_j\}.$$

We define the maximum over an empty set to be 0.

Now, we can encode the extended sensitivity terms using only **club**, through the following translation function:

$$\begin{aligned}
C(\mathbf{max}(\hat{R}_1, \hat{R}_2)) &:= \mathbf{club}\{(\emptyset; \emptyset; C(\hat{R}_1)), (\emptyset; \emptyset; C(\hat{R}_2))\} \\
C(\mathbf{sup}(i, \hat{R})) &:= \mathbf{club}\{(i; \emptyset; C(\hat{R}))\} \\
C(\mathbf{case}(S, i, \hat{R}_0, \hat{R}_s)) &:= \mathbf{club}\{(\emptyset; S = 0; C(\hat{R}_0)), \\
&\qquad (i; S = i + 1; C(\hat{R}_s))\} \\
C(\hat{R}_1 + \hat{R}_2) &:= C(\hat{R}_1) + C(\hat{R}_2) \\
C(\hat{R}_1 \cdot \hat{R}_2) &:= C(\hat{R}_1) \cdot C(\hat{R}_2) \\
C(R) &:= R \qquad \text{otherwise.}
\end{aligned}$$

While we may now have nested **club**, we extend the interpretation in the natural way. We can show that the translation faithfully preserves the semantics of the extended terms, with the following lemma.

Lemma 17. *Suppose $\phi \models R$ and $\phi \models \rho$ is a standard valuation. Then, $(\!|C(R)|\!)_\rho = [\![R]\!]_\rho$.*

Proof. By induction on R. □

Now, we can simplify the compiled constraints. First, we can push all standard sensitivity terms to the leaves of the expression. More formally, we have the following lemma.

Lemma 18. *Suppose $\phi \models R \cdot \mathbf{club}\{(\phi_i; \Phi_i; C_i)\}_i + R'$, where R, R' are standard sensitivity terms, and C_i is an arbitrary sensitivity term possibly involving* **club**. *Then, for any standard closing valuation $\phi \models \rho$,*

$$(\!|R \cdot \mathbf{club}\{(\phi_i; \Phi_i; C_i)\}_i + R'|\!)_\rho = (\!|\mathbf{club}\{(\phi_i; \Phi_i; R \cdot C_i + R')\}_i|\!)_\rho.$$

Proof. By the definition of the interpretations, and the mathematical fact

$$a \cdot \max_i\{b_i\} + c = \max_i\{a \cdot b_i + c\},$$

for $a, b, c \geq 0$. □

Thus, without loss of generality we may reduce the compiled sensitivity constraint to an expression of the form Q, with grammar

$$Q ::= \emptyset \mid Q_1 + Q_2 \mid Q_1 \cdot Q_2 \mid \mathbf{club}\{(\phi_i; \Phi_i; Q_i)\} \mid \mathbf{club}\{(\phi_i; \Phi_i; R_i)\},$$

where R_i are standard sensitivity terms. We will use the metavariable V to denote an arbitrary (possibly empty) collection of triples $(\phi_i; \Phi_i; R_i)_i$, and the metavariable W to denote an arbitrary (possibly empty) collection of triples $(\phi_i; \Phi_i; Q_i)_i$. Throughout, we will implicitly work up to permutation of the arguments to **club**: for instance, $\mathbf{club}\{(X), (Y)\}$ will be considered the same as $\mathbf{club}\{(Y), (X)\}$. We will also work up to commutativity of addition and multiplication: $Q_1 + Q_2$ will be considered the same as $Q_2 + Q_1$, and likewise with multiplication. We present the constraint simplification rules as a rewrite relation $\mapsto$. As typical, we will write $\mapsto^*$ for the reflexive, transitive closure of $\mapsto$. The full rules are in Figure 6.

We can prove correctness of our constraint simplification with the following lemma.

Lemma 19. *Suppose $Q \mapsto Q'$, and suppose $\phi \models Q$ and $\phi \models Q'$. Then, for any standard valuation $\phi \models \rho$, we have $(\!|Q|\!)_\rho = (\!|Q'|\!)_\rho$.*

Proof. By induction on the derivation of $Q \mapsto Q'$. The cases Plus, Mult and Red are immediate by induction. The other cases all follow by the semantics of **club**; details are in the Appendix. □

The simplification relation terminates in the following particular simple form.

Lemma 20. *Let Q be a sensitivity term involving* **club**. *Along any reduction path, Q reduces in finitely many steps to a term of the form*

$$\mathbf{club}\{V\} = \mathbf{club}\{(\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)\}.$$

Proof. First, note that any reduction of Q must terminate in finitely many steps: by induction on the derivation of the reduction, it's clear that each reduction removes one **club** subterm, and no reductions introduce **club** subterms. So, suppose that Q is a term with no possible reductions.

By induction on the structure of Q, we claim that Q is of the desired form. Say if $Q = Q_1 + Q_2$, if either Q_1, Q_2 can reduce, then Plus applies. If not, then by induction, CPlus applies. The same reasoning follows for $Q = Q_1 \cdot Q_2$: either Mult applies, or CMult does. Finally, if Q is a single **club** term, if Red and Flat both don't apply, then Q is of the desired form. □

Finally, checking a constraint $\forall \phi.\, \Phi \Rightarrow R \geq \mathbf{club}\{V\}$ is simple.

Lemma 21. *Let R be a standard sensitivity term, and let V be*

$$V = (\phi_1; \Phi_1; R_1), \ldots, (\phi_n; \Phi_n; R_n)$$

where each R_j is a standard sensitivity term without **club**. *Then, $\models \forall \phi.\, \Phi \Rightarrow R \geq \mathbf{club}\{V\}$ is logically equivalent to*

$$\forall_{j \in [n]} \phi, \phi_j.\, \Phi \Rightarrow \bigwedge_{k \in [n]} (\Phi_k \Rightarrow R \geq R_k)\,.$$

Proof. Immediate by the semantics of $\mathbf{club}\{V\}$. □

Putting together all the pieces, for a constraint

$$\models \forall \phi.\, \Phi \Rightarrow R \geq R',$$

with R standard, we can transform $C(R')$ to a term of the form Q by pushing all standard sensitivity terms to the leaves. Then, we normalize $Q \mapsto^* \mathbf{club}\{V\}$ by Lemma 20 arbitrarily. By Lemma 19, the interpretation of Q and $\mathbf{club}\{V\}$ are the same, so we can reduce the constraint $\models \forall \phi.\, \phi \Rightarrow R \geq \mathbf{club}\{V\}$ to a first order formula over mixed naturals and $\mathbb{S}$, with no alternating quantifiers, by Lemma 21.

9. Related work

There is a vast literature on type checking for various combinations of indexed types, linear types, dependent types and subtyping. A distinctive feature of our approach is that our index language represents natural and real number expressions. As we have shown in the previous sections, this makes type checking non-trivial.

The work most closely related to ours is Dal Lago et al. [6], who studied the type inference problem for dℓPCF, a relatively-complete type system for complexity analysis introduced in Dal Lago and Gaboardi [4]. dℓPCF uses ideas similar to *DFuzz* but brings the idea

$$\frac{}{\mathbf{club}\{(\phi;\Phi;\mathbf{club}\{(\phi_i;\Phi_i;R_i)\}_i),V\} \mapsto \mathbf{club}\{(\phi\cup\phi_i;\Phi\wedge\Phi_i;R_i),V\}_i}\ \text{Flat}$$

$$\frac{}{\mathbf{club}\{(\phi_i;\Phi_i;R_i)\}_i + \mathbf{club}\{(\phi'_j;\Phi'_j;R'_j)\}_j \mapsto \mathbf{club}\{(\phi_i\cup\phi'_j;\Phi_i\wedge\Phi'_j;R_i+R'_j)\}_{ij}}\ \text{CPlus}$$

$$\frac{}{\mathbf{club}\{(\phi_i;\Phi_i;R_i)\}_i \cdot \mathbf{club}\{(\phi'_j;\Phi'_j;R'_j)\}_j \mapsto \mathbf{club}\{(\phi_i\cup\phi'_j;\Phi_i\wedge\Phi'_j;R_i\cdot R'_j)\}_{ij}}\ \text{CMult}$$

$$\frac{Q_1 \mapsto Q'_1}{Q_1+Q_2 \mapsto Q'_1+Q'_2}\ \text{Plus} \qquad \frac{Q_1 \mapsto Q'_1}{Q_1\cdot Q_2 \mapsto Q'_1\cdot Q_2}\ \text{Mult} \qquad \frac{Q \mapsto Q'}{\mathbf{club}\{(\phi;\Phi;Q),W\} \mapsto \mathbf{club}\{(\phi;\Phi;Q'),W\}}\ \text{Red}$$

Figure 6. **club** Reduction

of linear dependent types to the limit. Indeed, dℓPCF index language contains function symbols that are given meaning by an equational program. The equational program then plays the role of an oracle for the type system—dℓPCF is in fact a family of type systems parametrized over the equational program. The main contribution of Dal Lago et al. [6] is an algorithm that, given a PCF program, generates a type and the set of constraints that must be satisfied in order to assign the return type to the input term.

In our terminology, their work is similar to the top-down approach we detailed in Section 3. As we discussed there, the complication of this approach is that it requires solving constraints over expressions—with possible function symbols—of the index-level language. As shown by Dal Lago and Petit, a clear advantage of the dℓPCF formulation is that instead of introducing an existential variable over expressions, one can introduce a new function symbol that will then be given meaning by the equational program generated by the constraints—i.e., the constraints give a description of the semantics of the program, which can be turned in an equational program, that in turn gives meaning to the function symbols of the index language appearing in the type. Clearly, this approach cannot be reduced to numeric resolution and need instead a combination of numeric and symbolic solving technology. The authors show that these constraints can be anyway handled by using the WHY3 framework. Some constraints are discharged automatically by some of the solvers available in WHY3 while others requires an interactive resolution using Coq.

As explained in Section 3, the situation with *DFuzz* is different. Indeed, *DFuzz* can be seen as a simplified version of dℓPCF—simplifying in particular the typing for the fixpoint and without variable bindings in !-types—extended however to deal with indices representing real numbers and using quantifications over index variables. A key distinction of *DFuzz* is that the set of constructors for the language of sensitivity is *fixed*—one cannot add arbitrary functions. Moreover, the extension to real numbers gives a different behavior from how natural numbers are used in dℓPCF—e.g., our example for the lack of minimal type would make no sense in dℓPCF. These distinctions make the type checking problem very different.

For another approach that is closely related to our work, recall that *DFuzz* is an extension of *Fuzz*. The sensitivity inference and sensitivity checking problems for *Fuzz* have been studied in D'Antoni et al. [7]. These problems are simpler than the one studied here since in *Fuzz* there is no dependency, no quantification and no subtyping. Indeed, the constraints generated are much simpler and can be solved quickly by an SMT solver.

Similarly, Eigner and Maffei [9] have studied an extension of *Fuzz* for modeling protocols. In their work they also give an algorithmic version of their type system. Their type system presents challenges similar to *Fuzz*, which they handle with algebraic manipulations. More precisely, their algorithmic version uses a technique similar to the one developed in Cervesato et al. [2] for the splitting of resources: when a rule with multiple premises is encountered the algorithmic system, first allocate all the resources to the first branch and then allocate the remaining resources to the second branch. Unfortunately, this approach cannot be easily applied to *DFuzz* due to the presence of index variables and dependent pattern matching.

From a different direction, recent works [1, 13] have shown how linear indexed type systems can be made more abstract and useful to analyze abstract resources. In particular, this kind of analyses is connected to comonadic notions of computations [20]. The type inference algorithm described in Ghica and Smith [13] is parametric on an abstract notion of resource. This resource can be instantiated on a language for sensitivities similar to the one in *Fuzz*. So, this abstract type inference procedure could be also used for sensitivity analysis.

DFuzz is one of several languages combining linear and dependent types. For example, ATS [3] is designed around a dependent type system enriched with a notion of resources that is a type-level representation of memory locations; these resources are managed using a linear discipline. ATS uses these features to verify the correctness of memory and pointer management.

Even if the use of linear types in ATS is very different from the one presented here, our type checking algorithm shares some similarities with ATS's one. The main difference is that ATS uses interactive theorem proving to discharge proof obligations while, thanks to the restricted scope of our analysis, our constraints can be handled by numeric solvers. In contrast, DML [26]—a predecessor of ATS which did not use linear types—uses an approach similar to ours by solving proof obligations using automatic numeric resolution. This required limitations on the operations available in the index language, similar to *DFuzz*.

Another work considering lightweight dependent types is the one by Zhu and Jagannathan [27]. In particular they propose a technique based on dependent types to reduce the verification of higher order programs to the verification of a first order language. While the goal of their work is similar in spirit to ours, their technique has only superficial similarities with the one presented here.

Finally, our work has been informed by the wide literature on type-checking, far too large to summarize here. For instance, the problem of dealing with subtyping rules by using syntax-directed systems has been studied by Pierce and Steffen [21], and others.

10. Conclusions and Future Work

We have presented a type-checking and inference algorithm for *EDFuzz*—a simple extension of *DFuzz*—featuring a linear indexed dependent type system. While we have shown that *DFuzz* type-checking is undecidable in the general case, our approach generates constraints over the first order theory over the reals and naturals, for which there are standard (though necessarily incomplete) solvers.

We are currently experimenting with a prototype implementation;[3] more investigation is needed in order to assess the difficulty of these constraints on real examples.

Overall, our design was guided by two principles: to stay as close to *DFuzz* as possible, and to provide a practical type checking procedure. While we do require extensions to *DFuzz*, there is a clear motivation for the introduction of each new construct. The idea of making a limited enrichment of the index language in order to simplify type-checking may be applicable to other linear indexed type systems. Furthermore, designers of such systems would do well to keep implementability in mind: seemingly unimportant decisions that simplify the metatheory may have a serious impact on type-checking.

References

[1] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *European Symposium on Programming (ESOP), Grenoble, France*. Springer, 2014.

[2] I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232 (1—2):133–163, 2000.

[3] C. Chen and H. Xi. Combining programming with theorem proving. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Tallinn, Estonia*, pages 66–77, 2005. ISBN 1-59593-064-7.

[4] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *IEEE Symposium on Logic in Computer Science (LICS), Toronto, Ontario*, pages 133–142. IEEE, 2011.

[5] U. Dal Lago and U. Schöpp. Functional programming in sublinear space. In *ACM Transactions on Programming Languages and Systems*, pages 205–225. Springer, 2010.

[6] U. Dal Lago, B. Petit, et al. The geometry of types. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 167–178, 2013.

[7] L. D'Antoni, M. Gaboardi, E. J. Gallego Arias, A. Haeberlen, and B. C. Pierce. Sensitivity analysis using type-based constraints. In *Workshop on Functional Programming Concepts in Domain-specific Languages (FPCDSL)*, FPCDSL '13, pages 43–50, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2380-2.

[8] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, POPL '03, pages 236–249, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5.

[9] F. Eigner and M. Maffei. Differential privacy by typing in security protocols. In *IEEE Computer Security Foundations Symposium, New Orleans, Louisiana*, pages 272–286, 2013.

[10] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, POPL '13, pages 357–370, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7.

[11] G. Ghelli and B. Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1–2):75 – 96, 1998.

[12] D. R. Ghica and A. Smith. Geometry of synthesis III: Resource management through type inference. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, volume 46, pages 345–356. ACM, 2011.

[13] D. R. Ghica and A. Smith. Bounded linear types in a resource semiring. In *European Symposium on Programming (ESOP), Grenoble, France*. Springer, 2014.

[14] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[15] B. Heeren, B. Heeren, J. Hage, J. Hage, D. Swierstra, and D. Swierstra. Generalizing hindley-milner type inference algorithms. Technical report, 2002.

[16] U. D. Lago and B. Petit. Linear dependent types in a call-by-value scenario. In D. D. Schreye, G. Janssens, and A. King, editors, *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP), Leuven, Belgium*, pages 115–126. ACM, 2012. ISBN 978-1-4503-1522-7.

[17] U. D. Lago and U. Schöpp. Type inference for sublinear space functional programming. In K. Ueda, editor, *Asian Symposium on Programming Languages and Systems (APLAS), Shanghai, China*, volume 6461 of *Lecture Notes in Computer Science*, pages 376–391. Springer, 2010. ISBN 978-3-642-17163-5.

[18] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems. PhD thesis*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1996.

[19] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[20] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: Unified static analysis of context-dependence. In *International Colloquium on Automata, Languages and Programming (ICALP), Riga, Latvia*, pages 385–397. Springer, 2013.

[21] B. C. Pierce and M. Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, pages 511–530, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[22] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[23] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, ICFP '10, pages 157–168, New York, NY, USA, 2010. ISBN 978-1-60558-794-3.

[24] P. Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), New Haven, Connecticut*, volume 26, pages 255–273. ACM, 1991.

[25] D. A. Wright and C. A. Baker-Finch. Usage analysis with natural reduction types. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Workshop on Static Analysis (WSA) , Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 1993. ISBN 3-540-57264-3.

[26] H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 214–227. ACM, 1999.

[27] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ml. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Rome, Italy*, pages 295–314. Springer, 2013.

[3] `http://cis.upenn.edu/~emilioga/dFuzz.tar.gz`

$$\frac{\phi;\Phi \mid \Delta \vdash e : \sigma \qquad \phi;\Phi \models \Gamma \sqsubseteq \Delta}{\phi;\Phi \mid \Gamma \vdash e : \sigma}\ (\sqsubseteq .\mathrm{L}) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \sigma \qquad \phi;\Phi \models \sigma \sqsubseteq \tau}{\phi;\Phi \mid \Gamma \vdash e : \tau}\ (\sqsubseteq .\mathrm{R}) \qquad \frac{}{\phi;\Phi \mid \Gamma \vdash \mathrm{r} : \mathbb{R}}\ (\mathrm{Const}_{\mathbb{R}})$$

$$\frac{n = [\![S]\!]}{\phi;\Phi \mid \Gamma \vdash n : \mathbb{N}[S]}\ (\mathrm{Const}_{\mathbb{N}}) \qquad \frac{}{\phi;\Phi \mid \Gamma, x :_{[1]} \sigma \vdash x : \sigma}\ (\mathrm{Var}) \qquad \frac{\phi;\Phi \mid \Gamma_1 \vdash e_1 : \sigma \qquad \phi;\Phi \mid \Gamma_2 \vdash e_2 : \tau}{\phi;\Phi \mid \Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \sigma \otimes \tau}\ (\otimes I)$$

$$\frac{\phi;\Phi \mid \Delta \vdash e : \sigma \otimes \tau \qquad \phi;\Phi \mid \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \vdash e' : \mu \qquad R \neq \square}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{let}(x, y) = e \ \mathbf{in}\ e' : \mu}\ (\otimes E) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi;\Phi \mid \Gamma \vdash e_2 : \tau}{\phi;\Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \,\&\, \tau}\ (\&\ I)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e : \sigma_1 \,\&\, \sigma_2}{\phi;\Phi \mid \Gamma \vdash \pi_i\ e : \sigma_i}\ (\&\ E) \qquad \frac{\phi;\Phi \mid \Gamma, x :_{[R]} \sigma \vdash e : \tau \qquad R \neq \square}{\phi;\Phi \mid \Gamma \vdash \lambda(x :_{[R]} \sigma).e : !_R\sigma \multimap \tau}\ (\multimap I)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e_1 : !_R\sigma \multimap \tau \qquad \phi;\Phi \mid \Delta \vdash e_2 : \sigma}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau}\ (\multimap E) \qquad \frac{\phi, i : \kappa;\Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi, \Gamma}{\phi;\Phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma}\ (\forall I)$$

$$\frac{\phi;\Phi \mid \Gamma \vdash e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi;\Phi \mid \Gamma \vdash e[S] : \sigma[S/i]}\ (\forall E) \qquad \frac{\phi;\Phi \mid \Gamma, x :_{[\infty]} \sigma \vdash e : \sigma}{\phi;\Phi \mid \infty \cdot \Gamma \vdash \mathbf{fix}\, x : \sigma.\ e : \sigma}\ (\mathrm{Fix}) \qquad \frac{\phi;\Phi \mid \Gamma \vdash e : \mathbb{N}[S]}{\phi;\Phi \mid \Gamma \vdash e + 1 : \mathbb{N}[S+1]}\ (\mathrm{S}\ I)$$

$$\frac{\begin{array}{c}\phi;\Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi;\Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n};\Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \# R \qquad R \neq \square\end{array}}{\phi;\Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma}\ (\mathbb{N}\ E)$$

Figure 7. *DFuzz* $_{\square}$ Type Judgment

A. Differences Compared to Gaboardi et al. [10]

While we hew closely to the presentation of *DFuzz* in Gaboardi et al. [10], we make a few technical changes.

- The context weakening operation $\Gamma \sqsubseteq \Gamma'$ in *DFuzz* allows the types to change. That is, a binding $x :_{[R]} \sigma \in \Gamma$ can be weakened to $x :_{[R']} \sigma'$ for $\sigma \sqsubseteq \sigma'$ two syntactically different types. We take a more restricted weakening rule, where the types must be syntactically the same; we are unaware of any programs that need the more general rule.
- We take the interpretation of $\infty \cdot 0$ to be ∞, rather than 0.
- We assume some additional type annotations in the source language, as discussed in Section 5

B. The *DFuzz* $_{\square}$ system

The first system has the goal to enjoy "context" uniformity, in the sense that sensitivity information in the contexts may be missing. We denote such an assignment $x :_{\square} \sigma$. This is a subtle technical point for crucial to enable syntax-directed typability.

We modify subtyping for environments such that $\Gamma \sqsubseteq \Delta$ requires Γ, Δ to have the same domain. The new rule is:

$$\frac{\begin{array}{c}\forall (x_i :_{[R_i]} \sigma_i, x_i :_{[R'_i]} \sigma_i) \in (\Gamma, \Delta) \\ \mathrm{dom}(\Delta) = \mathrm{dom}(\Gamma) \qquad \models \forall \phi.\ (\Phi \Rightarrow R_i \geq R'_i) \vee R'_i = \square\end{array}}{\phi;\Phi \models \Gamma \sqsubseteq \Delta}\ \sqsubseteq\text{-Env}$$

This subsumes regular variable weakening. Context operations must be aware of $\square$, with $\square + i = i$, $i \cdot \square = \square$ for the annotations.

Definition 22 (Box erasure)**.** *For any context* Γ*, we define the* $\square$*-erasure operation* $|\Gamma| = \{x :_{[R]} \sigma \mid x :_{[R]} \sigma \in \Gamma \wedge R \neq \square\}$.

We introduce the $\square$ system in Figure 7.

We prove that derivations in a system with $\square$ are in direct correspondence with derivation in a system without it.

Lemma 23. *Assume* $\phi;\Phi \mid \Gamma \vdash e : \sigma$ *in the* $\square$ *system, then* $\phi;\Phi \mid |\Gamma| \vdash e : \sigma$ *in the system without it.*

Proof. By induction on the typing derivation. The base cases and cases where the context is not modified are immediate. Subtyping on the left is proven by weakening.

The rest of cases are split in two:

- All cases featuring variables in the top rule, also have the condition $R \neq \square$, this is enough.
- For the cases involving context operations, the proofs is completed by following properties:

$$|R \cdot \Gamma| = R \cdot |\Gamma| \qquad |\Gamma + \Delta| = |\Gamma| + |\Delta|$$

□

Lemma 24. *Assume* $\phi;\Phi \mid \Gamma \vdash e : \sigma$ *in the system without* $\square$*, then* $\phi;\Phi \mid \Gamma \vdash e : \sigma$ *in the system with it.*

Proof. The proof is mostly routine by induction on the derivation, but relies in the following fact of the $\Box$ system: $\phi; \Phi \mid \Gamma \vdash e : \sigma$ implies $\phi; \Phi \mid \Gamma, x :_{\Box} \tau \vdash e : \sigma$. Then, using this lemma we can adjust the contexts so that subtyping goes through in the system with $\Box$. $\Box$

A $\Box$-elimination operation $R_{\Box\uparrow}$, which sends context annotations to sensitivities will prove useful in the the syntax directed system. It is defined as $\Box_{\Box\uparrow} = 0$, $R_{\Box\uparrow} = R$ otherwise. Remember that $\Box$ doesn't belong to the sensitivity language, so any annotation that is used in places where a sensitivity is expected must be wrapped with $-_{\Box\uparrow}$.

Definition 25 (Extension to environments operations)**.** *Operations on extended sensitivites that were extended to environments in a pointwise fashion, now must take into account the presence of* $\Box$*.*

- $\mathbf{max}(R_1, R_2)$ *operates now as* $\mathbf{max}(\Box, \Box) = \Box$, $\mathbf{max}(\Box, R) = R$, $\mathbf{max}(R, \Box) = R$, *the original term otherwise.*
- $\mathbf{sup}(i, R)$ *is extended in the natural way* $\mathbf{sup}(i, \Box) = \Box$, *the original term otherwise.*
- $\mathbf{case}(S, i, R_0, R_s)$ *operates now* $\mathbf{case}(S, i, \Box, \Box) = \Box$, $\mathbf{case}(S, i, R_0, R_s) = \mathbf{case}(S, i, R_{0\,\Box\uparrow}, R_{s\,\Box\uparrow})$ *otherwise.*

C. Subtyping Proofs

From now on we can consider only contexts of similar length. We prove a few necessary facts about subtyping.

Lemma 26 (Context manipulation)**.** *Context subtyping is preserved by addition and scalar multiplication. More formally:*

- *If* $\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \wedge \Delta \sqsubseteq \Delta'$, *then* $\phi; \Phi \models \Gamma + \Delta \sqsubseteq \Gamma' + \Delta'$*; and*
- *if* $\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \wedge R \geq R'$, *then* $\phi; \Phi \models R \cdot \Gamma \sqsubseteq R' \cdot \Gamma'$.

Proof. These follow from the interpretation of subtyping assertions. Note that the subtyping relation preserves the skeleton of the environments, thus making sure that the operations are always defined. $\Box$

Lemma 27 (Properties of extended sensitivities)**.** *Extended sensitivities satisfy the following properties:*

- $\phi; \Phi \models R \geq \mathbf{max}(R_1, R_2)$ *if and only if* $\phi; \Phi \models R \geq R_1 \wedge R \geq R_2$*;*
- $\phi; \Phi \models R \geq \mathbf{sup}(i, R')$ *with* $i \# \phi$ *if and only if* $\phi, i; \Phi \models R \geq R'$*; and*
- $\phi; \Phi \models R \geq \mathbf{case}(S, i, R_0, R_s)$ *with* $i \# \phi$ *if and only if*

$$\phi; \Phi, S = 0 \models R \geq R_0 \quad \textit{and} \quad \phi, i; \Phi, S = i + 1 \models R \geq R_s.$$

As an immediate corollary, setting R *to be* $\mathbf{max}(R_1, R_2), \mathbf{sup}(i, R'), \mathbf{case}(S, i, R_0, R_s)$ *yields*

- $\phi; \Phi \models \mathbf{max}(R_1, R_2) \geq R_1 \wedge R \geq R_2$*;*
- $\phi, i; \Phi \models \mathbf{sup}(i, R') \geq R'$*; and*
- $\phi; \Phi, S = 0 \models \mathbf{case}(S, i, R_0, R_s) \geq R_0$ *and* $\phi, i; \Phi, S = i + 1 \models \mathbf{case}(S, i, R_0, R_s) \geq R_s$.

Proof. These follow from the interpretation of extended sensitivities. $\Box$

Lemma 28. *Suppose* $\phi, i : \kappa; \Phi \models \sigma \sqsubseteq \tau$ *and* $i \# \Phi$*. Then for any* $\phi \models S : \kappa$*, we have*

$$\phi; \Phi \models \sigma[S/i] \sqsubseteq \tau[S/i].$$

Proof. By induction on the subtype derivation. For the base cases, we know

$$\forall \phi, i : \kappa.\ (\Phi \Rightarrow R \geq R'),$$

and we need to prove

$$\forall \phi.\ (\Phi \Rightarrow R[S/i] \geq R'[S/i]),$$

but this is clear from the interpretation of R, R'. $\Box$

D. The Syntax-Directed system

The syntax-directed system is presented in Figure 8. It works over a uniform context, using $\Box$ annotations to "mark", variables not occurring in the original *DFuzz* derivation.

We first prove the system sound with respect the non syntax-directed one.

Lemma 29 (Syntax-directed soundness)**.** *If* $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$ *has a derivation, then* $\phi; \Phi \mid \Gamma \vdash e : \sigma$.

Proof. By induction on the derivation proving $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$.

Case: (Var)

$$\frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^{\bullet}), x :_{[1]} \sigma \vdash_{\mathcal{S}} x : \sigma} \quad (\mathrm{Var})$$

Immediate, the same rule applies.

Case: $(\otimes I)$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash_{\mathcal{S}} (e_1, e_2) : \sigma \otimes \tau} \quad (\otimes I)$$

Immediate by induction; the same rule applies.

$$\frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^\bullet) \vdash_{\mathcal{S}} \mathrm{r} : \mathbb{R}} \ (\mathrm{Const}_{\mathbb{R}}) \qquad \frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma \vdash_{\mathcal{S}} x : \sigma} \ (\mathrm{Var})$$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash_{\mathcal{S}} (e_1, e_2) : \sigma \otimes \tau} \ (\otimes I) \qquad \frac{\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau \vdash_{\mathcal{S}} e' : \mu}{\phi; \Phi \mid \Gamma + \mathbf{max}(R_{1\,\square\uparrow}, R_{2\,\square\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \ \mathbf{in} \ e' : \mu} \ (\otimes E)$$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma \ \& \ \tau} \ (\& \ I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma_1 \ \& \ \sigma_2}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \pi_i e : \sigma_i} \ (\& \ E)$$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \models \forall \phi. \, (\Phi \Rightarrow R \geq R^\bullet{}_{\square\uparrow})}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda(x :_{[R]} \sigma). \, e : !_R \sigma \multimap \tau} \ (\multimap I) \qquad \frac{\begin{array}{c}\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1 \ e_2 : \tau} \ (\multimap E)$$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma \qquad i \text{ fresh in } \Phi}{\phi; \Phi \mid \mathbf{sup}(i, \Gamma) \vdash_{\mathcal{S}} \Lambda i : \kappa. \, e : \forall i : \kappa. \, \sigma} \ (\forall I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \forall i : \kappa. \, \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e[S] : \sigma[S/i]} \ (\forall E)$$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash_{\mathcal{S}} \mathbf{fix} \, x : \sigma. \, e : \sigma} \ (\mathrm{Fix})$$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma_0 \vdash_{\mathcal{S}} e_0 : \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n :_{[R]} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) + \mathbf{case}(S, i, 0, R_{\square\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{case} \, e \, \mathbf{return} \, \sigma \, \mathbf{of} \, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \ (\mathbb{N} \, E)$$

$$\mathrm{Ectx}(\Gamma^\bullet) := \Delta \quad \text{with} \quad \begin{cases} \mathrm{dom}(\Gamma^\bullet) & = \mathrm{dom}(\Delta) \\ \Delta(b) & \equiv \_ :_\square \_ \qquad \text{for all } b \in \mathrm{dom}(\Gamma^\bullet) \end{cases}$$

Figure 8. *DFuzz* Type Judgment, Syntax-directed Version

Case: $(\otimes E)$

$$\frac{\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \tau \vdash_{\mathcal{S}} e' : \mu}{\phi; \Phi \mid \Gamma + \mathbf{max}(R_{1\,\square\uparrow}, R_{2\,\square\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \ \mathbf{in} \ e' : \mu} \ (\otimes E)$$

By induction, we have

$$\phi; \Phi \mid \Delta \vdash e : \sigma \otimes \tau \quad \text{and} \quad \phi; \Phi \mid \Gamma, x :_{[R_1]} \sigma, y :_{[R_2]} \sigma \vdash e' : \mu$$

By Lemma 27, $\phi; \Phi \models \mathbf{max}(R_{1\,\square\uparrow}, R_{2\,\square\uparrow}) \geq R_{i\,\square\uparrow}$ for $i = 1, 2$. Abbreviating $R^\bullet := \mathbf{max}(R_{1\,\square\uparrow}, R_{2\,\square\uparrow})$ and applying weakening we have:

$$\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma, y :_{[R^\bullet]} \tau \vdash e' : \mu$$

with $R^\bullet \neq \square$ so we have exactly what we need to apply $(\otimes E)$.

Case: $(\& \ I)$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma \ \& \ \tau} \ (\& \ I)$$

By induction, we have

$$\phi; \Phi \mid \Gamma_1 \vdash e_1 : \sigma \quad \text{and} \quad \phi; \Phi \mid \Gamma_2 \vdash e_2 : \tau.$$

By Lemma 27, we have

$$\phi; \Phi \models \mathbf{max}(\Gamma_1, \Gamma_2) \sqsubseteq \Gamma_1 \quad \text{and} \quad \phi; \Phi \models \mathbf{max}(\Gamma_1, \Gamma_2) \sqsubseteq \Gamma_2.$$

By weakening, we can derive

$$\phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash e_1 : \sigma \quad \text{and} \quad \phi; \Phi \mid \mathbf{max}(\Gamma_1, \Gamma_2) \vdash e_2 : \tau,$$

when we can conclude by $(\& \ I)$.

Case: $(\& \ E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma_1 \ \& \ \sigma_2}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \pi_i e : \sigma_i} \ (\& \ E)$$

Immediate; the same rule applies.

Case: $(\multimap I)$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \models \forall \phi. \, (\Phi \Rightarrow R \geq R^\bullet{}_{\square\uparrow})}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda(x :_{[R]} \sigma). \, e : !_R \sigma \multimap \tau} \ (\multimap I)$$

By induction, we have

$$\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash e : \tau$$

and we know $R \neq \square$ and:

$$\phi; \Phi \models R \geq R^\bullet.$$

By weakening, we have

$$\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \tau,$$

and we can conclude by $(\multimap I)$.

Case: $(\multimap E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1\ e_2 : \tau} \quad (\multimap E)$$

By induction, we have

$$\phi; \Phi \mid \Gamma \vdash e_1 : !_R \sigma \multimap \tau \quad \text{and} \quad \phi; \Phi \mid \Delta \vdash e_2 : \sigma'$$

and we also know

$$\phi; \Phi \models \sigma' \sqsubseteq \sigma.$$

By subtyping on the right, we can derive

$$\phi; \Phi \mid \Delta \vdash e_2 : \sigma,$$

and we can conclude with $(\multimap E)$.

Case: $(\forall I)$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma \qquad i \text{ fresh in } \Phi}{\phi; \Phi \mid \mathbf{sup}(i, \Gamma) \vdash_{\mathcal{S}} \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma} \quad (\forall I)$$

By induction, we have

$$\phi; i : \kappa; \Phi \mid \Gamma \vdash e : \sigma$$

and i fresh in Φ. By Lemma 27, we have

$$\phi; \Phi \models \mathbf{sup}(i, \Gamma) \sqsubseteq \Gamma,$$

and so by weakening, we have

$$\phi, i : \kappa; \Phi \mid \mathbf{sup}(i, \Gamma) \vdash e : \sigma.$$

Now, we can conclude with $(\forall I)$.

Case: $(\forall E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e[S] : \sigma[S/i]} \quad (\forall E)$$

Immediate; the same rule applies.

Case: (Fix)

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash_{\mathcal{S}} \mathbf{fix}\, x : \sigma.\ e : \sigma} \quad (\text{Fix})$$

By induction; we have

$$\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \sigma'.$$

But we also have $\phi; \Phi \models \sigma' \sqsubseteq \sigma$. By subtyping, we get

$$\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \sigma$$

and we can conclude with (Fix).

Case: $(\mathbb{N}\ E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma_0 \vdash_{\mathcal{S}} e_0 : \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n :_{[R]} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) + \mathbf{case}(S, i, 0, R_{\square\uparrow}) \cdot \Delta \vdash_{\mathcal{S}} \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\ E)$$

By induction, we have

$$\begin{array}{l}\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \\ \phi; \Phi, S = 0 \mid \Gamma_0 \vdash e_0 : \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma_s, n : !_R \mathbb{N}[i] \vdash e_s : \sigma_s.\end{array}$$

By Lemma 27, we have

$$\begin{array}{l}\phi; \Phi, S = 0 \models \mathbf{case}(S, i, \Gamma_0, \Gamma_s) \sqsubseteq \Gamma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \mathbf{case}(S, i, \Gamma_0, \Gamma_s) \sqsubseteq \Gamma_s \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \mathbf{case}(S, i, 0, R_{\square\uparrow}) \geq R_{\square\uparrow}\end{array}$$

with $R_{\square\uparrow} \neq \square$, and we also know

$$\phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma$$
$$\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma.$$

By subtyping on the left and right, we have

$$\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S]$$
$$\phi; \Phi, S = 0 \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s) \vdash e_0 : \sigma$$
$$\phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \mathbf{case}(S, i, \Gamma_0, \Gamma_s), n :\ !_{R^\bullet}\mathbb{N}[i] \vdash e_s : \sigma,$$

where $R^\bullet = \mathbf{case}(S, i, 0, R_{\square\uparrow})$. We can then conclude by $(\mathbb{N}\ E)$.

$$\frac{\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \# R \qquad R \neq \square}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\ E)$$

□

We now prove completeness, that is to say, for every derivation in the original system, the syntax-directed one will have a derivation, possibly even a better from a subtype point of view.

We first need a few auxiliary lemmas:

Lemma 30. *Suppose that $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$ is derivable. Then, for any logically equivalent Ψ such that $\phi \models \Phi \Leftrightarrow \Psi$, there is a derivation of $\phi; \Psi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$ with the same height.*

Proof. By induction on the derivation. The only place the constraint context is used is when checking constraints of the form

$$\phi; \Phi \models R \geq R'.$$

But since Ψ and Φ are logically equivalent, we evidently have

$$\phi; \Psi \models R \geq R'$$

as well. □

Lemma 31 (Inner Weakening for the Syntax-directed system)**.** *Assume a derivation $\Gamma, x :_{[R]} \sigma \vdash_{\mathcal{S}} e : \tau$, a type σ' such that $\sigma' \sqsubseteq \sigma$. Then, there exists a type τ' and a derivation $\Gamma, x :_{[R]} \sigma' \vdash_{\mathcal{S}} e : \tau'$ such that $\tau' \sqsubseteq \tau$.*

Proof. By induction over the typing derivation. The base cases are immediate. In the induction hypothesis we get to pick the appropriate type and we get a better type in all the cases. □

Lemma 32 (Syntax-directed completeness)**.** *If $\phi; \Phi \mid \Gamma \vdash e : \sigma$ has a derivation, then there exists Γ', σ' such that $\phi; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e : \sigma'$ has a derivation, $\phi; \Phi \models \Gamma \sqsubseteq \Gamma'$, $\phi; \Phi \models \sigma' \sqsubseteq \sigma$.*

Proof. By induction on the derivation proving $\phi; \Phi \mid \Gamma \vdash e : \sigma$.

Case: $(\sqsubseteq .\mathrm{L})$

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \qquad \phi; \Phi \models \Gamma \sqsubseteq \Delta}{\phi; \Phi \mid \Gamma \vdash e : \sigma} \quad (\sqsubseteq .\mathrm{L})$$

Immediate, by induction; the desired context is Δ.

Case: $(\sqsubseteq .\mathrm{R})$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma \qquad \phi; \Phi \models \sigma \sqsubseteq \tau}{\phi; \Phi \mid \Gamma \vdash e : \tau} \quad (\sqsubseteq .\mathrm{R})$$

Immediate, by induction; the desired subtype is σ.

Case: (Var)

$$\frac{}{\phi; \Phi \mid \Gamma, x :_{[1]} \sigma \vdash x : \sigma} \quad (\mathrm{Var})$$

Immediate; the same rule applies.

Case: $(\otimes I)$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \sigma \otimes \tau} \quad (\otimes I)$$

By induction, we have $\Gamma_1', \Gamma_2', \sigma', \tau'$ such that

$$\phi; \Phi \models \Gamma_1 \sqsubseteq \Gamma_1' \wedge \Gamma_2 \sqsubseteq \Gamma_2' \quad \text{and} \quad \phi; \Phi \models \sigma' \sqsubseteq \sigma \wedge \tau' \sqsubseteq \tau$$

and derivations

$$\phi; \Phi \mid \Gamma_1' \vdash_{\mathcal{S}} e_1 : \sigma' \quad \text{and} \quad \phi; \Phi \mid \Gamma_2' \vdash_{\mathcal{S}} e_2 : \tau'.$$

Then we can conclude by $(\otimes I)$, since Lemma 26 shows

$$\phi; \Phi \models \Gamma_1 + \Gamma_2 \sqsubseteq \Gamma_1' + \Gamma_2' \quad \text{and} \quad \phi; \Phi \models \sigma' \otimes \tau' \sqsubseteq \sigma \otimes \tau.$$

Case: $(\otimes E)$

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \otimes \tau \qquad \phi; \Phi \mid \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \vdash e' : \mu \qquad R \neq \square}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{let}(x, y) = e \ \mathbf{in} \ e' : \mu} \ (\otimes E)$$

By induction and inversion on the subtype relation, we have $\Delta', \Gamma', \sigma', \sigma'', \tau', \tau'', \mu', R_1, R_2$ such that

$$\phi; \Phi \models \Delta \sqsubseteq \Delta'$$
$$\phi; \Phi \models \Gamma, x :_{[R]} \sigma, y :_{[R]} \tau \sqsubseteq \Gamma', x :_{[R_1]} \sigma'', y :_{[R_2]} \tau''$$
$$\phi; \Phi \models \sigma' \sqsubseteq \sigma \wedge \tau' \sqsubseteq \tau$$

this implies $\sigma' \sqsubseteq \sigma''$, $\tau' \sqsubseteq \tau''$, $R \geq R_{1\square\uparrow}$, and $R \geq R_{2\square\uparrow}$. We have derivations:

$$\phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e : \sigma' \otimes \tau' \quad \text{and} \quad \phi; \Phi \mid \Gamma', x :_{[R_1]} \sigma'', y :_{[R_2]} \tau'' \vdash_{\mathcal{S}} e' : \mu'$$

By Lemma 31, we have a derivation:

$$\phi; \Phi \mid \Gamma', x :_{[R_1]} \sigma', y :_{[R_2]} \tau' \vdash_{\mathcal{S}} e' : \mu''$$

with $\mu'' \sqsubseteq \mu'$. Hence, we can produce a syntax-directed derivation now:

$$\phi; \Phi \mid \Gamma' + \mathbf{max}(R'_{1\square\uparrow}, R'_{2\square\uparrow}) \cdot \Delta' \vdash_{\mathcal{S}} \mathbf{let}(x, y) = e \ \mathbf{in} \ e' : \mu''.$$

By Lemma 27, we have that $\phi; \Phi \models R \geq \mathbf{max}(R'_{1\square\uparrow}, R_{2\square\uparrow})$ and by Lemma 26,

$$\phi; \Phi \models \Gamma + R \cdot \Delta \sqsubseteq \Gamma' + \mathbf{max}(R'_{1\square\uparrow}, R_{2\square\uparrow}) \cdot \Delta',$$

so we are done: the context $\Gamma' + \mathbf{max}(R'_{1\square\uparrow}, R'_{2\square\uparrow}) \cdot \Delta'$ and subtype τ'' suffice.

Case: $(\& \ I)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \ \& \ \tau} \ (\& \ I)$$

By induction, there exists

$$\phi; \Phi \models \Gamma \sqsubseteq \Gamma'_1 \quad \text{and} \quad \phi; \Phi \models \Gamma \sqsubseteq \Gamma'_2$$
$$\phi; \Phi \models \sigma' \sqsubseteq \sigma \quad \text{and} \quad \phi; \Phi \models \tau' \sqsubseteq \tau$$

such that

$$\phi; \Phi \mid \Gamma'_1 \vdash_{\mathcal{S}} e_1 : \sigma' \quad \text{and} \quad \phi; \Phi \mid \Gamma'_2 \vdash_{\mathcal{S}} e_2 : \tau'.$$

By $(\& \ I)$, we have

$$\phi; \Phi \mid \mathbf{max}(\Gamma'_1, \Gamma'_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma' \ \& \ \tau'.$$

We are done, since by Lemmas 26 and 27,

$$\phi; \Phi \models \sigma' \ \& \ \tau' \sqsubseteq \sigma \ \& \ \tau \quad \text{and} \quad \phi; \Phi \models \Gamma \sqsubseteq \mathbf{max}(\Gamma'_1, \Gamma'_2) \sqsubseteq \Gamma'_i.$$

So, the desired context is $\mathbf{max}(\Gamma'_1, \Gamma'_2)$, and the desired subtype is $\sigma' \ \& \ \tau'$.

Case: $(\& \ E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma_1 \ \& \ \sigma_2}{\phi; \Phi \mid \Gamma \vdash \pi_i \ e : \sigma_i} \ (\& \ E)$$

Immediate, by induction.

Case: $(\multimap \ I)$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R]} \sigma \vdash e : \tau \qquad R \neq \square}{\phi; \Phi \mid \Gamma \vdash \lambda(x :_{[R]} \sigma).e : !_R \sigma \multimap \tau} \ (\multimap \ I)$$

By induction, there exists

$$\phi; \Phi \models \Gamma, x :_{[R]} \sigma \sqsubseteq \Gamma', x : !_{R'} \sigma \quad \text{and} \quad \phi; \Phi \models \tau' \sqsubseteq \tau$$

such that

$$\phi; \Phi \mid \Gamma', x :_{[R']} \sigma \vdash_{\mathcal{S}} e : \tau'.$$

By inversion on the subtype relation, we have

$$\phi; \Phi \models R \geq R'_{\square\uparrow} \wedge \tau' \sqsubseteq \tau.$$

and we are done, since

$$\phi; \Phi \models !_{R'_{\square\uparrow}} \sigma \multimap \tau' \sqsubseteq !_R \sigma \multimap \tau \quad \text{and} \quad \phi; \Phi \models \Gamma \sqsubseteq \Gamma'.$$

$$\frac{\phi; \Phi \mid \Gamma, x :_{[R^\bullet]} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \models \forall \phi. \ (\Phi \Rightarrow R \geq R^\bullet{}_{\square\uparrow})}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda(x :_{[R]} \sigma). \ e : !_R \sigma \multimap \tau} \ (\multimap \ I)$$

Case: $(\multimap E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e_1 : !_R \sigma \multimap \tau \qquad \phi; \Phi \mid \Delta \vdash e_2 : \sigma}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau} \quad (\multimap E)$$

By induction, there exists $\Gamma', \Delta', R', \sigma', \tau', \sigma''$ such that

$$\phi; \Phi \models \Gamma \sqsubseteq \Gamma'$$
$$\phi; \Phi \models \Delta \sqsubseteq \Delta'$$
$$\phi; \Phi \models !_{R'} \sigma' \multimap \tau' \sqsubseteq !_R \sigma \multimap \tau$$
$$\phi; \Phi \models \sigma'' \sqsubseteq \sigma,$$

and derivations

$$\phi; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e_1 : !_{R'} \sigma' \multimap \tau' \quad \text{and} \quad \phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e_2 : \sigma''.$$

By inversion on the subtype relation, we have

$$\phi; \Phi \models R \geq R' \quad \text{and} \quad \phi; \Phi \models \sigma'' \sqsubseteq \sigma \sqsubseteq \sigma' \quad \text{and} \quad \phi; \Phi \models \tau' \sqsubseteq \tau.$$

By Lemma 27, the context $\Gamma' + R' \cdot \Delta'$ and subtype τ' suffice.

Case: $(\forall I)$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi, \Gamma}{\phi; \Phi \mid \Gamma \vdash \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma} \quad (\forall I)$$

By induction, there exist

$$\phi, i : \kappa; \Phi \models \sigma' \sqsubseteq \sigma \quad \text{and} \quad \phi, i : \kappa; \Phi \models \Gamma \sqsubseteq \Gamma'$$

such that

$$\phi, i : \kappa; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e : \sigma'.$$

Thus, we have the derivation

$$\phi; \Phi \mid \mathbf{sup}(i, \Gamma') \vdash_{\mathcal{S}} \Lambda i : \kappa.\ e : \forall i : \kappa.\ \sigma'$$

and

$$\phi; \Phi \models \forall i : \kappa.\ \sigma' \sqsubseteq \forall i : \kappa.\ \sigma.$$

By Lemma 27, we actually have

$$\phi; \Phi \models \Gamma \sqsubseteq \mathbf{sup}(i, \Gamma') \sqsubseteq \Gamma',$$

so the context $\mathbf{sup}(i, \Gamma')$ and subtype $\forall i : \kappa.\ \sigma'$ suffices.

Case: $(\forall E)$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \forall i : \kappa.\ \sigma \qquad \phi \models S : \kappa}{\phi; \Phi \mid \Gamma \vdash e[S] : \sigma[S/i]} \quad (\forall E)$$

By induction, there exists

$$\phi; \Phi \models \Gamma \sqsubseteq \Gamma' \quad \text{and} \quad \phi; \Phi \models \forall i : \kappa.\ \sigma' \sqsubseteq \forall i : \kappa.\ \sigma$$

such that

$$\phi; \Phi \mid \Gamma' \vdash_{\mathcal{S}} e : \forall i : \kappa.\ \sigma'.$$

So, we have a derivation

$$\phi; \Phi \mid \Gamma'' \vdash_{\mathcal{S}} e[S/i] : \sigma'[S/i].$$

By Lemma 28,

$$\phi; \Phi \models \sigma'[S/i] \sqsubseteq \sigma[S/i],$$

so the context Γ' and subtype $\sigma'[S/i]$ suffice.

Case: (Fix)

$$\frac{\phi; \Phi \mid \Gamma, x :_{[\infty]} \sigma \vdash e : \sigma}{\phi; \Phi \mid \infty \cdot \Gamma \vdash \mathbf{fix}\, x : \sigma.\ e : \sigma} \quad (\text{Fix})$$

By induction, we have

$$\phi; \Phi \models \Gamma, x : !_\infty \sigma \sqsubseteq \Gamma', x : !_R \sigma \quad \text{and} \quad \phi; \Phi \models \sigma' \sqsubseteq \sigma$$

such that

$$\phi; \Phi \mid \Gamma', x : !_R \sigma \vdash_{\mathcal{S}} e : \sigma'.$$

We can then conclude by (Fix): the desired context is $\infty \cdot \Gamma'$ and the desired type is σ.

Case: $(\mathbb{N}\ E)$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \qquad \phi; \Phi, S = 0 \mid \Gamma \vdash e_0 : \sigma \\ \phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma, n :_{[R]} \mathbb{N}[i] \vdash e_s : \sigma \qquad i \# R \qquad R \neq \square\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma} \quad (\mathbb{N}\ E)$$

By induction, there exists

$$\phi; \Phi \models \Delta \sqsubseteq \Delta' \quad \text{and} \quad \phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e : \mathbb{N}[S'] \quad \text{and} \quad \phi; \Phi \models \mathbb{N}[S'] \sqsubseteq \mathbb{N}[S].$$

By inversion, $\phi; \Phi \models S = S'$. Also by induction,

$$\begin{aligned}
&\phi; \Phi, S = 0 \models \Gamma \sqsubseteq \Gamma'_0 \\
&\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \Gamma, n : !_R \mathbb{N}[i] \sqsubseteq \Gamma'_s, n : !_{R'} \mathbb{N}[i] \\
&\phi; \Phi, S = 0 \models \sigma'_0 \sqsubseteq \sigma \\
&\phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma'_s \sqsubseteq \sigma
\end{aligned}$$

such that

$$\begin{aligned}
&\phi; \Phi, S = 0 \mid \Gamma'_0 \vdash_{\mathcal{S}} e_0 : \sigma'_0 \\
&\phi, i : \mathrm{n}; \Phi, S = i + 1 \mid \Gamma'_s, n : !_{R'} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma'_s.
\end{aligned}$$

By Lemma 30, we also have derivations

$$\begin{aligned}
&\phi; \Phi, S' = 0 \mid \Gamma'_0 \vdash_{\mathcal{S}} e_0 : \sigma'_0 \\
&\phi, i : \mathrm{n}; \Phi, S' = i + 1 \mid \Gamma'_s, n : !_{R'} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma'_s
\end{aligned}$$

since $\phi; \Phi \models S = S'$.
Hence, we have a derivation

$$\begin{aligned}
&\phi; \Phi \mid \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) + R^\bullet \cdot \Delta' \\
&\vdash_{\mathcal{S}} \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \Rightarrow e_0 \mid n_{[i]} + 1 \Rightarrow e_s : \sigma,
\end{aligned}$$

where $R^\bullet$ is $\mathbf{case}(S', i, 0, {R'}_{\square\uparrow})$. We have

$$\begin{aligned}
&\phi; \Phi, S' = 0 \models \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) \sqsubseteq \Gamma'_0 \\
&\phi, i : \mathrm{n}; \Phi, S' = i + 1 \models \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) \sqsubseteq \Gamma'_s
\end{aligned}$$

so by Lemma 27

$$\phi; \Phi \models \Gamma \sqsubseteq \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s),$$

and

$$\phi, i : \mathrm{n}; \Phi, S' = i + 1 \models R \geq R^\bullet \geq {R'}_{\square\uparrow} \quad \text{and} \quad \phi, \Phi \models R \geq R^\bullet$$

thanks to $R \neq \square$.
By weakening, we have

$$\begin{aligned}
&\phi; \Phi \mid \Delta' \vdash_{\mathcal{S}} e : \mathbb{N}[S'] \\
&\phi; \Phi, S = 0 \mid \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) \vdash_{\mathcal{S}} e_0 : \sigma \\
&\phi, i : \mathrm{n}; \Phi, S' = i + 1 \mid \mathbf{case}(S', i, \Gamma'_0, \Gamma'_s), n : !_{R^\bullet} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s : \sigma,
\end{aligned}$$

so we can conlude with $(\mathbb{N}\ E)$. The context $\mathbf{case}(S', i, \Gamma'_0, \Gamma'_s) + R^\bullet \cdot \Delta'$ and type σ suffice (recall that $\phi; \Phi \models R \geq R^\bullet$, and $\phi; \Phi \models R \cdot \Delta \sqsubseteq R^\bullet \cdot \Delta'$ by Lemma 26).

□

D.1 Algorithm Proofs

Theorem 33 (Algorithmic Soundness)**.** *Suppose* $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$*. Then, there is a derivation of* $\phi; \Phi; \Gamma \vdash_{\mathcal{S}} e : \sigma$*.*

Proof. By induction on the algorithmic derivations we see that every algorithmic step has an exact correspondence with a syntax-directed derivation. We do a few representative cases:

Case (Var)

$$\frac{}{\phi; \Phi; \Gamma^\bullet, x : \sigma; x \Longrightarrow \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma; \sigma} \quad (\mathrm{Var})$$

$$\frac{}{\phi; \Phi \mid \mathrm{Ectx}(\Gamma^\bullet), x :_{[1]} \sigma \vdash_{\mathcal{S}} x : \sigma} \quad (\mathrm{Var})$$

Case $(\multimap E)$

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; e_1 \Longrightarrow \Gamma; !_R \sigma \multimap \tau \\ \phi; \Phi; \Delta^\bullet; e_2 \Longrightarrow \Delta; \sigma' \\ \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi; \Gamma^\bullet; e_1\ e_2 \Longrightarrow \Gamma + R \cdot \Delta; \tau} \quad (\multimap E)$$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \\ \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1\ e_2 : \tau} \quad (\multimap E)$$

Case $(\otimes E)$

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma^\bullet;e \Longrightarrow \Delta;\sigma\otimes\tau \\ \phi;\Phi;\Gamma^\bullet,x:\sigma,y:\tau;e' \Longrightarrow \Gamma,x :_{[R_1]} \sigma,y :_{[R_2]} \tau;\mu\end{array}}{\phi;\Phi;\Gamma^\bullet;\mathbf{let}(x,y)=e\ \mathbf{in}\ e' \Longrightarrow \Gamma+\mathbf{max}(R_{1\square\uparrow},R_{2\square\uparrow})\cdot\Delta;\mu} \quad (\otimes E)$$

$$\frac{\phi;\Phi \mid \Delta \vdash_{\mathcal{S}} e:\sigma\otimes\tau \qquad \phi;\Phi \mid \Gamma,x :_{[R_1]} \sigma,y :_{[R_2]} \tau \vdash_{\mathcal{S}} e':\mu}{\phi;\Phi \mid \Gamma+\mathbf{max}(R_{1\square\uparrow},R_{2\square\uparrow})\cdot\Delta \vdash_{\mathcal{S}} \mathbf{let}(x,y)=e\ \mathbf{in}\ e':\mu} \quad (\otimes E)$$

□

Theorem 34 (Algorithmic Completeness)**.** *Suppose $\phi;\Phi;\Gamma \vdash_{\mathcal{S}} e:\sigma$ is derivable. Then $\phi;\Phi;\Gamma^\bullet;e \Longrightarrow \Gamma;\sigma$.*

Proof. By induction on the syntax-directed derivation. The proof is mostly direct, we show a few representative cases.

Case $(\multimap E)$

$$\frac{\begin{array}{c}\phi;\Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 :\ !_R\sigma\multimap\tau \\ \phi;\Phi \mid \Delta \vdash_{\mathcal{S}} e_2:\sigma' \qquad \phi;\Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi;\Phi \mid \Gamma+R\cdot\Delta \vdash_{\mathcal{S}} e_1\ e_2:\tau} \quad (\multimap E)$$

By induction, we have derivations

$$\phi;\Phi;\Gamma^\bullet;e_1 \Longrightarrow \Gamma;!_R\sigma\multimap\tau \quad \text{and} \quad \phi;\Phi;\Delta^\bullet;e_2 \Longrightarrow \Delta;\sigma'.$$

Note that $\Gamma^\bullet=\Delta^\bullet$ for the syntax-directed derivation to be defined, so we can apply the algorithmic rule $(\multimap E)$:

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma^\bullet;e_1 \Longrightarrow \Gamma;!_R\sigma\multimap\tau \\ \phi;\Phi;\Delta^\bullet;e_2 \Longrightarrow \Delta;\sigma' \\ \phi;\Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi;\Phi;\Gamma^\bullet;e_1\ e_2 \Longrightarrow \Gamma+R\cdot\Delta;\tau} \quad (\multimap E)$$

Case (Fix)

$$\frac{\phi;\Phi \mid \Gamma,x :_{[R]} \sigma \vdash_{\mathcal{S}} e:\sigma' \qquad \phi;\Phi \models \sigma' \sqsubseteq \sigma}{\phi;\Phi \mid \infty\cdot\Gamma \vdash_{\mathcal{S}} \mathbf{fix}\,x:\sigma.\ e:\sigma} \quad (\text{Fix})$$

By induction, we have

$$\phi;\Phi;\Gamma^\bullet,x:\sigma;e \Longrightarrow \Gamma,x :_{[R]} \sigma;\sigma'$$

and we can apply the algorithm rule (Fix):

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma^\bullet,x:\sigma;e \Longrightarrow \Gamma,x :_{[R]} \sigma;\sigma' \\ \phi;\Phi \models \sigma' \sqsubseteq \sigma\end{array}}{\phi;\Phi;\Gamma^\bullet;\mathbf{fix}\,x:\sigma.\ e:\sigma \Longrightarrow \infty\cdot\Gamma;\sigma} \quad (\text{Fix})$$

Case $(\otimes E)$

$$\frac{\phi;\Phi \mid \Delta \vdash_{\mathcal{S}} e:\sigma\otimes\tau \qquad \phi;\Phi \mid \Gamma,x :_{[R_1]} \sigma,y :_{[R_2]} \tau \vdash_{\mathcal{S}} e':\mu}{\phi;\Phi \mid \Gamma+\mathbf{max}(R_{1\square\uparrow},R_{2\square\uparrow})\cdot\Delta \vdash_{\mathcal{S}} \mathbf{let}(x,y)=e\ \mathbf{in}\ e':\mu} \quad (\otimes E)$$

We know that $\Gamma^\bullet=\Delta^\bullet$. By induction, we know that:

$$\begin{array}{l}\phi;\Phi;\Gamma^\bullet;e \Longrightarrow \Delta;\sigma_1'\otimes\sigma_2' \\ \phi;\Phi;\Gamma^\bullet,x_1:\sigma_1,x_2:\sigma_2;e' \Longrightarrow \Gamma,x :_{[R_1]} \sigma_1,y :_{[R_2]} \sigma_2;\tau\end{array}$$

and we know $\phi;\Phi \models \sigma_1' \sqsubseteq \sigma_1 \wedge \sigma_2' \sqsubseteq \sigma_2$, so we apply the algorithmic case $(\otimes E)$:

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma^\bullet;e \Longrightarrow \Delta;\sigma\otimes\tau \\ \phi;\Phi;\Gamma^\bullet,x:\sigma,y:\tau;e' \Longrightarrow \Gamma,x :_{[R_1]} \sigma,y :_{[R_2]} \tau;\mu\end{array}}{\phi;\Phi;\Gamma^\bullet;\mathbf{let}(x,y)=e\ \mathbf{in}\ e' \Longrightarrow \Gamma+\mathbf{max}(R_{1\square\uparrow},R_{2\square\uparrow})\cdot\Delta;\mu} \quad (\otimes E)$$

Case $(\mathbb{N}\ E)$

$$\frac{\begin{array}{c}\phi;\Phi \mid \Delta \vdash_{\mathcal{S}} e:\mathbb{N}[S] \qquad \phi;\Phi,S=0 \mid \Gamma_0 \vdash_{\mathcal{S}} e_0:\sigma_0 \\ \phi,i:\mathrm{n};\Phi,S=i+1 \mid \Gamma_s,n :_{[R]} \mathbb{N}[i] \vdash_{\mathcal{S}} e_s:\sigma_s \\ \phi;\Phi,S=0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi,i:\mathrm{n};\Phi,S=i+1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\phi;\Phi \mid \mathbf{case}(S,i,\Gamma_0,\Gamma_s)+\mathbf{case}(S,i,0,R_{\square\uparrow})\cdot\Delta \vdash_{\mathcal{S}} \mathbf{case}\,e\,\mathbf{return}\,\sigma\,\mathbf{of}\,0\Rightarrow e_0 \mid n_{[i]}+1\Rightarrow e_s:\sigma} \quad (\mathbb{N}\ E)$$

We know that $\Gamma^\bullet=\Delta^\bullet$. By induction, we know that:

$$\begin{array}{l}\phi;\Phi;\Gamma^\bullet;e \Longrightarrow \Delta;\mathbb{N}[S] \\ \phi;\Phi,S=0;\Gamma^\bullet;e_0 \Longrightarrow \Gamma_0;\sigma_0 \\ \phi,i:\mathrm{n};\Phi,S=i+1;\Gamma^\bullet,x:\mathbb{N}[i];e_s \Longrightarrow \Gamma_s,x :_{[R']} \mathbb{N}[i];\sigma_s\end{array}$$

and we know

$$\phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \quad \text{and} \quad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma.$$

We can conclude with the algorithmic rule ($\mathbb{N}\ E$):

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Delta; \mathbb{N}[S] \qquad \phi; \Phi, S = 0; \Gamma^\bullet; e_0 \Longrightarrow \Gamma_0; \sigma_0 \\ \phi, i : \mathrm{n}; \Phi, S = i + 1; \Gamma^\bullet, x : \mathbb{N}[i]; e_s \Longrightarrow \Gamma_s, x :_{[R']} \mathbb{N}[i]; \sigma_s \\ \phi; \Phi, S = 0 \models \sigma_0 \sqsubseteq \sigma \qquad \phi, i : \mathrm{n}; \Phi, S = i + 1 \models \sigma_s \sqsubseteq \sigma\end{array}}{\begin{array}{c}\phi; \Phi; \Gamma^\bullet; \mathbf{case}\, e\, \mathbf{return}\, \sigma\, \mathbf{of}\, 0 \mapsto e_0 \mid x_{[i]} + 1 \mapsto e_s \\ \Longrightarrow \mathbf{case}(S, \Gamma_0, i, \Gamma_s) + \mathbf{case}(S, 0, i, R'_{\square\uparrow}) \cdot \Delta; \sigma\end{array}} \quad (\mathbb{N}\ E)$$

□

E. Auxiliary Lemmas

Lemma 35 (Standard Annotations)**.** *Assume annotations in a term e range over regular sensitivities and $\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \sigma$. Then:*

- *σ has no extended sensitivities; and*
- *all the constraints are of the form $\models \forall \phi.\ (\Phi \Rightarrow R \geq R')$ where R is a standard sensitivity term.*

This directly implies Lemma 11.

Proof. The first point is clear by inspecting the rules in Figure 8: by induction, the type of any expression has only regular sensitivities. The second point is also clear: in all subtype checks in Figure 8, both types have no extended sensitivities by the first point. The only place where we check against an extended sensitivity is in rule ($\multimap I$), with constraint

$$\models \forall \phi.\ (\Phi \Rightarrow R \geq R').$$

Here, the R is a standard sensitivity term since it is an annotation, but the R' may be an extended sensitivity. □

Editlets: type based client side editors for iTasks

László Domoszlai
Radboud University Nijmegen, Netherlands, ICIS, MBSD
dlacko@gmail.com

Bas Lijnse Rinus Plasmeijer
Radboud University Nijmegen, Netherlands, ICIS, MBSD
b.lijnse@cs.ru.nl,rinus@cs.ru.nl

Abstract

The iTask framework is for the construction of distributed systems where users work together on the internet. It offers a domain specific language for defining applications, embedded into the lazy functional language Clean. From the mere declarative specification a complete multi-user web application is generated. Although the generated nature of the user interface entails a number of benefits for the programmer, it suffers from the lack of possibility to create custom UI building blocks. In a precursory work we proposed *tasklets* for the development of custom, interactive web components. However, as tasklets are implemented as a computational element, a *task*, they lack some fundamental properties limiting their usability; these are compositionality and the capability of two-way communication between the clients and the server. In this paper, we introduce *editlets* to overcome these limitations. In addition, editlets also provide a general way to communicate in *edits* instead of exchanging the whole data; it does not just help with reducing the communication cost, but also enables multiple clients to work on the same shared data with minimizing the risk of conflicting updates.

1. Introduction

Task Oriented Programming [5, 7] (TOP) is a paradigm that is designed to construct multi-user, distributed, web-applications. The *iTask system* [6] (iTasks) is a TOP framework that offers a domain specific language embedded in the pure, lazy functional language Clean.

According to the TOP paradigm, the unit of application logic is a task. Tasks are abstract descriptions of interactive persistent units of work that have a typed value. When a task is executed, it has an opaque persistent value, which can be observed by other tasks in a controlled way. In iTasks, complex multi-user interactions can be programmed in a declarative style just by defining the tasks that have to be accomplished. The specification of the tasks is given by a domain specific language (DSL). Furthermore, the specification is given on a very high level of abstraction and does not require the programmer to provide any user interface definition. Merely by defining the workflow of user interaction, a complete multi-user web application is generated, all the details e.g. the generation of web user interface, client-server communication, state management etc. are automatically taken care of by the framework itself.

The iTask system uses generic programming [1, 4] and a hybrid static-dynamic type system [8, 9] to generate the user interface. From the programmers perspective, it is achieved in two levels. In the most basic level, the iTasks engine can be asked to generate user interface for any conceivable first order model type. iTasks uses a predefined set of primitive user interface elements to generate the GUI, a client side *editor*, for the given type, then dynamically creates an associated primitive task. On the higher level, additional user interface elements are generated as tasks are combined together. These elements reflect the actual combinators in use and express the "flow" of the application.

Developing web applications such a way is straightforward in the sense that the programmers are liberated from these cumbersome and error-prone jobs, such that they can concentrate on the essence of the application. The iTask system makes it very easy to develop interactive multi-user applications. The down side is that one has only limited control over the customization of the generated user interface. In real world applications, it is often necessary to develop custom user interface elements to achieve special functionality.

To overcome this limitation, in a previous work we introduced *tasklets* [2], a special primitive task type, for the development of custom, interactive web components. Tasklets are written in Clean and executed in a web browser using a Clean to JavaScript compilation technique [3]. In the browser, they have unlimited access to browser resources through some library functions while on the server they behave like ordinary iTasks tasks.

Using tasklets, we have successfully developed many interactive components for a wide range of applications, but we also experienced certain limitations of the technology. These are the following:

1. Tasklets cannot work with shared data. As an example, it is not possible to create an interactive map, and enable multiple users to make concurrent modifications to that (e.g. add marks). This limitation goes against the main principle of iTasks.
2. There is no way for two-way communication between the client and the server part. Tasklets implement task interface which enables the inspection of task values, the behavior of a task cannot be influenced after its evaluation is started.
3. There is only limited compositionality at task level. Generated editors cannot contain custom elements as they are primitive tasks which cannot contain other tasks.

In this paper we rethought how to create interactive components. We found that attaching the presentation logic to a type has many advantages over our previous approach. The new type of interactive elements are called *editlets* as they work on the lower editor level instead of task level as *tasklets* do. Editlets solve all the

IFL '14, October 1–3, 2014, Boston, Massachusetts, US.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

aforementioned limitations while preserving compatibility: in the most basic use cases they give back the functionality of tasklets.

Editlets also have the property that the client-server communication is done in *edits*, which means that the value of the editlet is communicated through changes instead of exchanging the whole value at every update. In certain cases it does not just reduces drastically the communication cost (just think of a source code editor component), but also allows us to avoid update conflicts when working on shared data.

In this paper we show how editlets can be defined, how they work and interact with the other part of the iTask system. This is done in a number of steps:

1. We extend iTask with editlets. An editlet consists of the type of the value it holds, a type of the edits in use, a description of the behavior of the component on the client side, and the logic of creating and applying edits from and to its current value.
2. We develop a simple, but still realistic example of a drawing applications, where multiple people can work on the same image on the same shared image to give a taste of editlets.
3. We explain the technical background of editles along with additional remarks how they fit the iTasks architecture.

References

[1] A. Alimarine. *Generic functional programming: conceptual design, implementation and applications*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, 2005.

[2] L. Domoszlai and R. Plasmeijer. Tasklets: Client-side evaluation for iTask3. In *Domain specific languages, summer school*, DSL'13, 2014. Accepted for publication.

[3] L. Domoszlai, E. Bruël, and J. Jansen. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae*, 3:76–98, 2011. URL `http://www.acta.sapientia.ro/acta-info/C3-1/info31-4.pdf`.

[4] R. Hinze. A new approach to generic functional programming. In T. Reps, editor, *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00, Boston, MA, USA*, pages 119–132. ACM Press, 2000.

[5] B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands , 2013. ISBN 978-90-820259-0-3.

[6] R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. Van Noort, and J. Van Groningen. iTasks for a change: Type-safe run-time change in dynamically evolving workflows. In *PEPM '11 : Proceedings Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*, pages 151–160, New York, 2011. ACM.

[7] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, pages 195–206, Leuven, Belgium, Sept. 2012. ACM. ISBN 978-1-4503-1522-7.

[8] T. van Noort. *Dynamic Typing in Type-Driven Programming*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands , May 2012. ISBN 978-94-6108-279-4.

[9] A. v. Weelden. *Putting types to good use*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, Oct. 17, 2007.

Task Oriented Programming with Purely Compositional Interactive Vector Graphics

Peter Achten
Radboud University Nijmegen, Netherlands, ICIS, MBSD
P.Achten@cs.ru.nl

Jurriën Stutterheim László Domoszlai
Rinus Plasmeijer
Radboud University Nijmegen, Netherlands, ICIS, MBSD
j.stutterheim@cs.ru.nl,dlacko@gmail.com,rinus@cs.ru.nl

1. Abstract

Task Oriented Programming [24, 29] (TOP) is a paradigm that is designed to construct multi-user, distributed, web-applications. The *iTask system* [28] (iTasks) is a TOP framework that offers three core concepts for software developers.

- *Tasks* which are abstractions of the *work* that needs to be performed by (teams of) human(s) and software components. A task is a value of parameterized type `(Task a)`. The type parameter `a` models the *task value* the task is currently processing. This value can be inspected by other tasks.
- *Shared data sources* (SDS) which are abstractions of *information* that is shared between tasks. A SDS is a value of parameterized type `(ReadWriteShared r w)`. The type parameters `r` and `w` model the *read* and *write* values.
- *Combinator functions* that compose tasks and SDSs into more complex tasks and SDSs and combinations of them.

The iTask system is a domain specific language (DSL) that is shallowly embedded in the strongly typed, lazy, purely functional programming language Clean [27, 30]. When developing an iTask application, the task developer can concentrate on identifying the tasks, the shared data sources, and their interrelation. The iTask system uses generic programming [5, 21] and a hybrid static-dynamic type system [31, 32] to generate all required machinery to create an executable. Among the plethora of concerns, the iTask system automatically generates a graphical user interface (GUI) for any conceivable first order model type. For this purpose the iTask system offers a comprehensive set of data types that model common user interface elements. In this way the task developer needs no working knowledge of JavaScript, HTML 5.0, handling (de)serialization and events. However, this knowledge *is* required whenever the comprehensive set of model types does not cover a particular interface element. This is unfortunate because it breaks the level of abstraction that is offered by the iTask system.

In this paper we show how the level of abstraction of iTask can remain intact when task developers define new user interface elements. This is done in a number of steps:

- We extend iTask with *Images* which are *vector graphics* based renderings. An image of type `(Image m)` is a rendering of a model value of type `m`. Images have a span to specify their dimensionality and local coordinate-system (traditional running from left-to-right and top-to-bottom), but there is no global coordinate system in which they are positioned or global canvas on which they are painted.
- We add *combinator functions* to compose images into more complex images. The absence of a global coordinate system or global canvas allows us to provide only three layout primitives: the *overlay* (placing images on top of each other), *grid* (two-dimensional structured layout), and *collage* (two-dimensional arbitrary layout). Each layout primitive has an optional *host* image that determines a reference span that is used for layout.
- We obtain *interactivity* by integrating images in iTask. Any (composite) image of type `(Image m)` can react to user events and define its behaviour via a pure function of type `(m -> m)` that alters the image's model value. This is in accordance to the philosophy of tasks: behaviour only needs to be defined in terms of how tasks depend on the model value of tasks and images.

The implementation of images and its combinator functions in iTask is based on the Scalable Vector Graphics (SVG) standard [10]. The low-level integration of these images in iTask is structured by means of *editlets*, and the high-level integration is done via the iTask *step* task combinator function.

Compositional Images

The full paper contains a detailed explanation and motivation for the compositional image library. Figure 1 displays the key elements of the API. For this abstract, we state the key properties of the API.

- Think of a basic image as an overhead-projector slide that is infinitely large. This slide can be rotated, scaled, and skewed. A finite portion of the basic image has visual content, the extent of which is defined by its *span*. The *x-span* always extends from left to right, and the *y-span* always extends from top to bottom.
- Think of a composite image as a stack of overhead-project slides. This stack can be rotated, scaled, and skewed. When composing images, their span is used to control their relative location. There are three core image combinator functions: *overlay* to stack images, *grid* to stack images row-by-row or column-by-column, and *collage* to stack images and arrange

them to your liking. The commonly occurring layouts *beside* and *above* are direct specializations of *grid*.

- The layout combinators have an optional *host image* parameter. Think of the host image as the background image relative to which the other images are to be arranged in terms of alignment.
- Images can have *tags*. This is needed when expressing spans in terms of the span(s) of other parts of the image.
- A (composite) image of type `(Image m)` can be made *interactive* by attributing it with a pure function of type `(m -> m)`, thus resulting in a change of image model value. This function is evaluated whenever the user clicks in the image (regardless of the location and transformation of the image).

```
:: Image m      // Opaque type
:: Span         // Opaque type
:: Host  m      :== Maybe (Image m)
:: ImageTag     :== String
:: FontDef      :== String
:: ImageOffset :== (Span, Span)

:: XAlign        = AtLeft | AtMiddleX | AtRight
:: YAlign        = AtTop  | AtMiddleY | AtBottom
:: ImageAlign  :== (XAlign, YAlign)

:: GridDimension = Rows Int | Columns Int
:: GridLayout  :== (GridXLayout, GridYLayout)
:: GridXLayout   = LeftToRight | RightToLeft
:: GridYLayout   = TopToBottom | BottomToTop

:: ImageLayout m :== [ImageOffset] [Image m] (Host m) -> Image m
overlay  :: [ImageAlign]                          -> ImageLayout m
beside   :: [YAlign]                              -> ImageLayout m
above    :: [XAlign]                              -> ImageLayout m
grid     :: GridDimension GridLayout [ImageAlign] -> ImageLayout m
collage  ::                                          ImageLayout m

empty    :: Span Span           -> Image m
text     :: FontDef String      -> Image m
circle   :: Span                -> Image m
ellipse  :: Span Span           -> Image m
rect     :: Span Span           -> Image m

:: Slash = Slash | Backslash

xline    :: Span                -> Image m
yline    :: Span                -> Image m
line     :: Slash Span Span     -> Image m
polygon  :: [ImageOffset]       -> Image m
polyline :: [ImageOffset]       -> Image m

rotate   :: Real      (Image m) -> Image m
fit      :: Span Span (Image m) -> Image m
fitx     :: Span      (Image m) -> Image m
fity     :: Span      (Image m) -> Image m
skewx    :: Real      (Image m) -> Image m
skewy    :: Real      (Image m) -> Image m

px         :: Real              -> Span
ex         :: FontDef           -> Span
descent    :: FontDef           -> Span
textxspan  :: FontDef String    -> Span
imagexspan :: [ImageTag]        -> Span
imageyspan :: [ImageTag]        -> Span
columnspan :: [ImageTag] Int    -> Span
rowspan    :: [ImageTag] Int    -> Span
```

Figure 1. The key elements of the Image API.

Integration in iTask

The integration of interactive, compositional images in iTask concerns the following components:

- The images are mapped to SVG. We face two major hurdles: *(i)* SVG adopts an imperative-style rendering model, so we must take care to unravel the declarative image specifications and paint them in the right order in SVG; *(ii)* text dimensions can only be computed at the client-side of the application, so the layout of images can not be performed entirely on the server-side of iTask.
- To establish the server-client side communication, we use iTask *editlets*.

These will be described in detail in the full paper.

Case studies

We demonstrate the new iTask approach by means of the following case studies:

- a 1-person pocket calculator,
- a 2-person, distributed, *tic-tac-toe* game,
- a 2-person, distributed, *trax* game [1],
- a N-person, distributed, *ligretto* card game.

Related work

Functional programming and GUIs share a long research history [2–4, 6–9, 11–19, 22, 23, 25, 26]. The full paper compares and discusses these approaches in more detail. For this abstract we restrict ourselves to the following observations:

- Regarding compositional images, the work by Henderson [19, 20] has been influential to many compositional approaches, as well as ours. Similar to Henderson's approach, we abstract from absolute location, but we do not from size. In the context of scalable vector graphics, the latter is not an issue because at any time images can be resized to any demanded size.
- Regarding compositional GUIs, Haggis [15] is similar in their approach to layout and transform GUIs. A difference is that Haggis has a monadic flavour: the GUI elements that are to be combined need to be declared *before* their handles can be used to arrange them inside layouts. In our approach, the iTask system 'collects' the offered images in the task specifications.
- Regarding 'completeness', we have not yet made use of all graphics elements that are offered by SVG. Concepts that are currently missing but are intended to be included in the iTask system are Bézier curves, multi-line text blocks, gradients, generalized clipping, and filtering. The layout combinators that we propose were inspired by the Racket image API [14]. The three core layout primitives *overlay*, *grid*, and *collage* of our approach can model them. The current proposal's event model is certainly incomplete as it covers only user-mouse clicks. We expect that extending the model to deal with the usual set of mouse and keyboard events follows the same approach.

Conclusions

In the TOP iTask framework multi-user, distributed, web-applications can be developed on a high level of abstraction because the task developer can concentrate on identifying and specifying the required tasks, information, and how they relate, knowing that the iTask framework can generate a suitable web application. The paper shows how this property can also be satisfied when developing applications that require custom built user interface(element)s. Because images are compositional, the task developer can concentrate on identifying and specifying the required graphical elements,

knowing that the image library generates a suitable SVG rendering. Via editlets graphically customized tasks are integrated seamlessly in the TOP paradigm.

References

[1] P. Achten. Why functional programming matters to me. In P. Achten and P. Koopman, editors, *The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday, Festschrift*, number 8106 in LNAI, pages 79–96. Springer, August 2013. ISBN ISBN 978-3-642-40354-5.

[2] P. Achten and S. Peyton Jones. Porting the Clean Object I/O library to Haskell. In M. Mohnen and P. Koopman, editors, *Selected Papers of the 12th International Workshop on the Implementation of Functional Languages, IFL '00*, volume 2011 of *LNCS*, pages 194–213. Springer-Verlag, Sept. 2001.

[3] P. Achten and R. Plasmeijer. The ins and outs of Concurrent Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[4] P. Achten and R. Plasmeijer. Interactive functional objects in Clean. In C. Clack, K. Hammond, and T. Davie, editors, *Selected Papers of the 9th International Workshop on the Implementation of Functional Languages, IFL '97*, volume 1467 of *LNCS*, pages 304–321. Springer-Verlag, Sept. 1998.

[5] A. Alimarine. *Generic functional programming: conceptual design, implementation and applications*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, 2005.

[6] M. Carlsson and T. Hallgren. Fudgets - a graphical user interface in a lazy functional language. In *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture, FPCA '93*, Kopenhagen, Denmark, 1993.

[7] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proceedings of the 2nd International Conference on Functional Programming, ICFP '97*, volume 32(8), pages 251–262, Amsterdam, The Netherlands, 9-11, June 1997. ACM Press.

[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, Nov. 2006. Springer-Verlag.

[9] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Proceedings of the 5th Haskell Workshop, Haskell '01*, Sept. 2001.

[10] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt. Scalable vector graphics (svg) 1.1 (second edition). Technical Report REC-SVG11-20110816, W3C Recommendation 16 August 2011, 2011.

[11] A. Dwelly. Functions and dynamic user interfaces. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA '89*, pages 371–381, Sept. 1989.

[12] C. Elliot. Tangible functional programming. In *Proceedings of the 12th International Conference on Functional Programming, ICFP '07*, pages 59–70, Freiburg, Germany, 1-3, Oct. 2007. ACM Press. ISBN 978-1-59593-815-2.

[13] M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Proceedings of the 5th International Symposium on the Practical Aspects of Declarative Programming, PADL '03*. New Orleans, LA, USA, Springer-Verlag, Jan. 2003.

[14] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System * or, Fun for Freshman Kids. In *Proceedings International Conference on Functional Programming, ICFP '09*, Edinburgh, Scotland, UK, 2009. ACM Press.

[15] S. Finne and S. Peyton Jones. Composing Haggis. In *Eurographics Workshop on Programming Paradigms in Graphics*, pages 85–101, Maastricht, the Netherlands, 1995. Springer.

[16] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In P. Degano, editor, *Proceedings of the 12th European Symposium on Programming, ESOP '03*, volume 2618 of *Lecture Notes in Computer Science* , pages 238–252, 7-11, Apr. 2003.

[17] M. Hanus. High-level server side web scripting in Curry. In *Proceedings of the 3rd International Symposium on the Practical Aspects of Declarative Programming, PADL '01*, pages 76–92. Springer-Verlag, 2001.

[18] M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th International Conference on Principles and Practice of Declarative Programming, PPDP '06*, pages 27–38. ACM Press, 2006.

[19] P. Henderson. Functional geometry. In D. Friedman and D. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 179–187, Pittsburgh, Pennsylvania, 1982. ACM Press. URL `http://www.ecs.soton.ac.uk/~ph/funcgeo.pdf`.

[20] P. Henderson. Functional geometry. *Higher-Order and Symbolic Computation*, 15:349–365, 2002.

[21] R. Hinze. A new approach to generic functional programming. In T. Reps, editor, *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00, Boston, MA, USA*, pages 119–132. ACM Press, 2000.

[22] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In J. Jeuring and S. Peyton Jones, editors, *Proceedings of the 4th International Summer School on Advanced Functional Programming, AFP '03*, volume 2638 of *Lecture Notes in Computer Science* , pages 159–187. Oxford, UK, Springer-Verlag, 2003.

[23] D. Leijen. wxHaskell: a portable and concise GUI library for Haskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, 2004. ACM. . URL `http://doi.acm.org/10.1145/1017472.1017483`.

[24] B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands , 2013. ISBN 978-90-820259-0-3.

[25] F. Loitsch and M. Serrano. Hop client-side compilation. In *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP '07*, pages 141–158, New York, NY, USA, 2-4, Apr. 2007. Interact.

[26] M. Morazán. Functional Video Games in the CS1 Classroom. In R. Page, Z. Horváth, and V. Zsók, editors, *Proceedings of the 11th Symposium on Trends in Functional Programming, TFP '10*, volume 6546 of *LNCS*, pages 166–183, 2010.

[27] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

[28] R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. Van Noort, and J. Van Groningen. iTasks for a change: Type-safe run-time change in dynamically evolving workflows. In *PEPM '11 : Proceedings Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*, pages 151–160, New York, 2011. ACM.

[29] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, pages 195–206, Leuven, Belgium, Sept. 2012. ACM. ISBN 978-1-4503-1522-7.

[30] J. van Groningen, T. van Noort, P. Achten, P. Koopman, and R. Plasmeijer. Exchanging sources between Clean and Haskell: a double-edged front end for the Clean compiler. In J. Gibbons, editor, *Haskell'10 : proceedings of the third ACM Haskell symposium on Haskell*, pages 49–60. ACM, 2010.

[31] T. van Noort. *Dynamic Typing in Type-Driven Programming*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands , May 2012. ISBN 978-94-6108-279-4.

[32] A. v. Weelden. *Putting types to good use*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, Oct. 17, 2007.

An Iterative Compiler for Implicit Parallelism

Extended Abstract

José Manuel Calderón Trilla Colin Runciman
University of York, York UK
{jmct|colin}@cs.york.ac.uk

Abstract

Advocates of lazy functional programming languages often cite easy parallelism as a major benefit of abandoning mutable state [1]. This idea drove research into the theory and implementation of compilers that take advantage of implicit parallelism in a functional program. Using static analysis techniques compilers can attempt to identify where a program can benefit from parallelism and ensure that those expressions are executed concurrently with the main thread of execution [2, 3]. These techniques can produce improvements in the runtime performance of a program, but are limited by the static analyses' poor prediction of runtime performance. Our work is on the development of a system that uses feedback from runtime profiling *in addition to* well-studied static analysis techniques in order to achieve higher performance gains than through static analysis alone.

Keywords Implicit Parallelism, Lazy Functional Languages, Automatic parallelism, Strictness Analysis, Projections, Iterative Compilation, Feedback Directed Compilation

1. Introduction

The amenability of functional languages to parallelism has long been advertised [4, 5] but the ultimate goal of writing a program in a functional style and having the compiler find the implicit parallelism still requires work. Static analysis, when used alone, has underperformed in this endeavor [2, 3, 6, 7]. Our thought is that the compiler should incorporate runtime profile data into decisions about parallelism the same way a programmer would manually tune a parallel program.

By using runtime feedback we can have the compiler be generous when introducing parallelism into the program. The profiling data will then point to the `par` annotations that under-perform and the compiler will disable the parallelism they introduce.

1.1 Contributions

The main focus of our work has been the design and implementation of an experimental compiler that allows for implicit parallelism. The source language of the compiler is an enriched lambda calculus which is suitable for use as a functional core language in a larger compiler. The contributions of our work are as follows:

- The use of *switchable* `par` annotations[1]
- An implementation of Hinze's projection based strictness analysis [8]
- Utilising the correspondence between projections and strategies to introduce parallelism into a program
- Using search strategies to improve upon the initial `par` placement

This paper presents an overview of the design of our compiler and some of the design decisions that were made. As we are now beginning to run experiments, this paper also serves as a documented hypothesis for our results.

1.2 Compiler Pipeline

The compiler is composed of 5 main phases, illustrated in Figure 1

1. Parsing
2. Defunctionalisation
3. Projection based Strictness Analysis
4. Generation of strategies
5. Placement of `par` annotations
6. G-Code Generation
7. Execution
8. Feedback and iteration

The parsing and G-Code generation are done in the standard way and will not be discussed further. The rest of the paper is organised by following the compiler pipeline as shown in figure 1. In §2 we explain the advantages of performing defunctionalisation. We motivate our use of a projection based strictness analysis [9] in §3. §4 is a description of the correspondence between projections and strategies [10] which allows us to generate parallel strategies based on the projections provided by the strictness analysis. The technique used for utilising the runtime profiling to switch off some of the introduced parallelism is described in§5 along with possible additional search techniques. Lastly, §6 contains our conclusions and thoughts on possible future work.

2. Defunctionalisaton

As mentioned above, the design of the compiler utilises a defunctionalising transformation on the input programs. Defunctionalisation

[1] Our `par` annotations take the familiar form of `par a b = b`, where the first argument is 'sparked off' in parallel and the function returns its second argument.

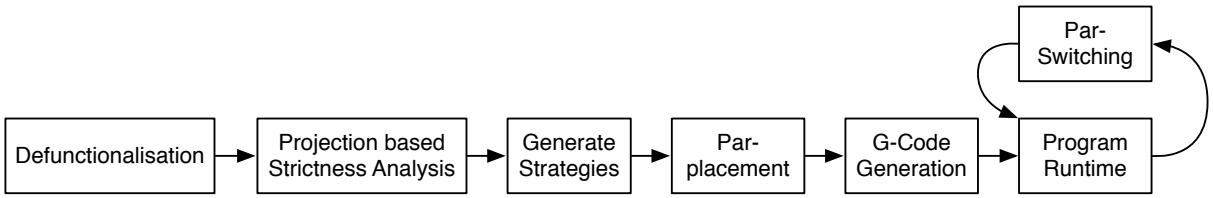


Figure 1. Compiler Pipeline After Parsing

specialises higher-order functions to the instances of their function arguments. Here we give our motivation for introducing this transformation.

Central to our design is the concept of `par` placement within a program. Each `par` is identified by its *position* in the AST. Due to the higher-order nature of our language, basing our parallelism on the location of a `par` can lead to undesirable circumstances. For example, a common pattern in parallel programs is to introduce a parallel version of the `map` function

```
parMap :: (a -> b) -> [a] -> [b]
parMap f []     = []
parMap f (x:xs) = let y = f x
                  in y `par` y : parMap f xs
```

This function allows us to use a common technique (mapping) with the possibility of performance gains through parallelism. However, when the computation `f x` is inexpensive, the parallelism may not provide any benefit or could even be detrimental. As `parMap` may be used throughout a program it is possible that there are both useful and detrimental instances of the function. For instance, `parMap f` may provide useful parallelism while `parMap g` may cost more in overhead than we gain from any parallelism. Unfortunately when this occurs we are unable switch off the `par` for `parMap g` without losing the useful parallelism of `parMap f`. This is because the `par` annotation is within the body of `parMap`. By specialising these functions we create two separate `parMap` functions: `parMapf` and `parMapg`. This now provides us with `par` annotations in *each* of the instances of `parMap`.

```
parMapf []     = []
parMapf (x:xs) = let y = f x
                 in y `par` y : parMapf xs

parMapg []     = []
parMapg (x:xs) = let y = g x
                 in y `par` y : parMapg xs
```

Because of defunctionalisation we are able to deactivate the `par` for the inexpensive computation, `g x`, without affecting the parallelism of the worthwhile computation, `f x`.

3. Strictness Analysis

The view of lazy languages (evaluation should only occur when necessary) can be at odds with the goals of performance through parallelism (do as much work as possible for faster execution time) [6]. Call-by-need semantics forces the compiler to take care in deciding which sub-expressions can safely be executed in parallel.

Having a simple parallelisation heuristic such as 'compute all arguments to functions in parallel' can alter the semantics of a non-strict language, introducing non-termination or runtime errors that would not have occurred during a sequential execution. For example, in a strict language, the function below would not terminate due to having to evaluate $\perp$ before entering the function, while lazy languages can compute the correct result since they only evaluate expressions when they are needed:

```
squareFirst :: Int -> Int -> Int
squareFirst x ⊥ = x * x
```

The problem of knowing which arguments are required for a function is known as *strictness analysis* [11] and forms the core of the static analysis phase of the compiler. In this section we provide a brief overview of the two predominant techniques for strictness analysis, ideal[2] analysis and projection based analysis. We then motivate our decision to use a projection based analysis.

3.1 Ideal Analysis

The main idea behind abstract interpretation is that you can throw away information about your program that is not necessary for the property you are analysing. When dealing with the `Integer` type, it may not be necessary to know the actual value of an integer, but instead only *some* of the information about that integer. Mycroft's "The Theory and Practice of Transforming Call-by-need into Call-by-value" [11] introduced the use of abstract interpretation for performing strictness analysis on call-by-need programs.

In the case of strictness analysis, we only require information about how defined a value is, and do not need to know about its concrete value.

In short, when performing ideal analysis we only concern ourselves with the definedness of values when analysing the strictness properties of programs.

With the strictness information in hand we can annotate our program to execute strict arguments in parallel with the function. In short, the strictness analysis informs the *initial* placement of `par` annotations in a program. Basing the initial placement on strictness information is important because we aim for our compiler to maintain the semantics of the initial sequential program.

When using a safe analysis we may not be able to determine all of the needed arguments for a given function. However, we can be certain that any argument the analysis determines is needed is definitely needed. This safety is crucial in avoiding the introduction of $\perp$ where it would not have occurred in a sequential lazy implementation [12].

The strictness properties of a function can be defined more formally as follows: A function of the form

$$f\ n_1\ n_2\ \ldots\ n_n = e$$

is said to be strict in argument n_m iff

$$f\ \ldots\ \perp_m\ \cdots = \perp$$

3.2 Projections

Strictness analysis as originally described by Mycroft is only capable of dealing with a two-point domain (values that are definitely needed,

[2] This terminology is used by Hinze in [8] to differentiate between the two methods.

ID: accepts all lists

T (tail strict): accepts all finite lists

H (head strict): accepts lists where the head of the list is defined (recursively)

HT (H and T strict): accepts finite lists where every member is defined

Figure 2. Four contexts on lists as described in [9].

and values that may or may not be needed). This works well for types that can be represented by a flat domain (Integer, Char, Bool, etc.)[3] but falls short on more complex data structures. For example, if a list argument is needed for a function to terminate, we can only evaluate up to the first `cons` safely. However, there are many functions on lists where evaluating the entire list (or even just the spine) can be safe. The canonical examples are `length` and `sum`. When evaluating the length of the list it would be safe to have evaluated the spine (and only the spine) of the list beforehand. This makes intuitive sense, if the evaluation of the spine is non-terminating, then the evaluation of `length` would be non-terminating as well. The function `sum` extends the same premise to the spine *and* the elements of a list.

In order to accommodate this type of reasoning, Wadler developed the well known four-point domain for lists [12]. While this work allowed for analysis to be performed on functions accepting lists, it was not easily extended to functions on other data structures.

Another approach involved *projections* from domain theory. Projection based analysis provides two benefits over ideal based analysis: The ability to analyse functions over arbitrary structures, and a correspondence with parallel strategies [10, 13]. This allows use to use the projections provided by our analysis to produce an appropriate function to compute the strict arguments in parallel.

Ideally we could generate and utilise strategies on any arbitrary type. This would allow to compiler to annotate the needed expression with the maximal safe amount of reduction. This requires us to use a more sophisticated form of strictness analysis: projections [9].

Projections asks a slightly different question than the ideal analysis described above. If the above asks "When passing this argument as $\bot$ is the result of the function call $\bot$?" then projections ask "If there is a certain degree of demand for the result of this function, what degree of demand is there on its arguments?".

First let us explain what is meant by 'demand'. The function `length` requires that the input list be finite, but no more. We can therefore say that `length` *demands* the spine of the argument list. The function `append` is a more interesting example

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

By studying the function we can tell that the first argument must be defined to the first cons, but we cannot know whether the second argument is ever needed. However, what if the *result* of `append` needs to be a finite list? In other words the function calling `append` requires that its input list be finite.

A simple example of this is the following program

```
lengthOfBoth :: [a] -> [a] -> Int
lengthOfBoth xs ys = length (append xs ys)
```

In this case *both* arguments to `append` must be finite. Projections allow us to make this distinction with the use of contexts [8, 9].

For lists we have the following contexts:

We can now say more about the strictness properties of `append`:

```
ID (append xs ys) = ID!(xs); ID(ys)
T (append xs ys)  = T!(xs); T!(ys)
H (append xs ys)  = H!(xs); H(ys)
HT (append xs ys) = HT!(xs); HT!(ys)
```

Here we use the convention from [8] of using `!` to denote the strictness of a context. `ID!` requires the list be defined to the first cons, whereas an expression in an `ID` context may not be needed.

Hinze's Work on Projections

Much of the work on strictness analysis as a means to achieve implicit parallelism focused on the ideal analysis approach. This was mostly an accident of timing, the work on projections had not been fully developed when implicit parallelism was a more active research area. In particular, the wonderful work on the "Automatic Parallelization of Lazy Functional Programs" [3] only used two and four-point domains (as described in [12]) in their strictness analysis. This limits the ability of the compiler to determine the neededness of more complex structures.

While projections were known as a possible technique for strictness analysis, the theory was much more complex and many of the details regarding the generality of the approach were not yet worked out. The work of Hinze [8] shows how projections can be used to determine the strictness information on complex data-types and sets the technique on a solid theoretical foundation that ensures its generality (in particular when working with polymorphic functions).

Using results from domain theory we are able to construct projections for every user-defined type, and furthermore each projection represents a specific strategy for evaluating the structure [8]. This provides us with the ability to generate appropriate parallel strategies for arbitrary types.

4. Projections and Strategies

As mentioned in the previous section, one of the reasons that projections were chosen for our strictness analysis is their correspondence to parallel strategies. The main idea behind strategies is that it is possible to write functions whose sole purpose is to force the evaluation of specific parts of a structure [10, 13]. An important point is that all strategies return `()`, having the type `type Strategy a = a -> ()`, which tells us that strategies are not used for their computed result but for the evaluation they force along the way.

The simplest strategy, named `r0` in [13], which performs no reductions is defined as `r0 x = ()`. The strategy for weak head normal form is only slightly more involved: `rwhnf x = x ‘seq‘ ()`

Neither of these strategies are of much interest. The real power comes when strategies are used on nested data-structures. Take lists for example, evaluating a list sequentially or in parallel provides us with the following two strategies

```
seqList :: Strategy a -> Strategy [a]
seqList s []     = ()
seqList s (x:xs) = s x ‘seq‘ (seqList s xs)

parList :: Strategy a -> Strategy [a]
parList s []     = ()
parList s (x:xs) = s x ‘par‘ (parList s xs)
```

First notice that each strategy takes another strategy as an argument. The provided strategy is what determines how much of each element to evaluate. If the provided strategy is `r0` the end result would be that only the spine of the list is evaluated. On the other end of the spectrum, providing a strategy that evaluates a value of type `a` fully would result in list's spine *and* elements being evaluated. Already we can see a correspondence between these strategies and

[3] Any type that can be represented as an enumerated type.

the contexts shown in figure 2. The `T` context (tail strict) corresponds to the strategy that only evaluates the spine of the list, while the `HT` context corresponds to the strategy that evaluates the spine and all the elements of a list.

This correspondence allows us to generate strategies based on the results of our strictness analysis. Because the projection based approach gives us the ability to describe different levels of demand on arbitrary data-types, we then get all of the corresponding strategies to evaluate up to that demand, but no more.

One aspect of strategies that does not directly correspond a context is choice between `seq` and `par`. Every context can be fully described by both sequential and parallel strategies. One goal of our work is to determine when it is appropriate to use parallelism in a strategy. Every field of a constructor has the potential to be evaluated in parallel. When a constructor has one field, it is not usually beneficial to do so, but when the constructor has two or more fields, it can be beneficial to evaluate *some* of the fields in parallel. It is not clear, generally, which fields should be parallelised and which should be evaluated in sequence. We currently rely on heuristics but we believe that performing a path analysis would aid in this task [14].

5. Iterative Compilation

We now have all of the building blocks for what we see as our contribution. We believe there are several reasons why previous work into implicit parallelism has not achieved the results that researchers have hoped for. Chief amongst those reasons is that the static placement of parallel annotations is not sufficient for creating well-performing parallel programs.

Imagine that you were writing a parallel program. When writing the source code you may study the structure and then decide where to place `par` annotations. When the program is compiled and executed you find that the performance was not satisfactory. Normally, one would return to the source for the program and adjust the placement of parallel annotations. This is the approach advocated by [15] and [16]. However, many of the previous attempts at implicit parallelism only analyse the program statically and do not adjust any parallel annotations after runtime data is gathered. This would be equivalent to a programmer never adjusting annotations after profiling the program.

There is one significant exception to this. In 2007 Harris and Singh published their results on a feedback directed implicit parallelism compiler [7]. The results were mostly positive (in that most benchmarks saw an improvement in performance) but were not to the degree desired. Since this research was published we have seen no other attempt in this line of research within the functional programming community.

The work in [7] attempted to use runtime profile data to introduce parallel annotations into the program based on heap allocations. In short, when viewing the parallel execution of a program as a tree, their method seeks to expand the tree based on previous executions of the program. Our goal is to develop a system that begins with a program that perhaps has *too much* parallelism and uses runtime data to prune the execution tree. We have implemented a few mechanisms to make this possible.

5.0.1 Logging:

The runtime system maintains records of the following global statistics:

- Number of reduction cycles
- Number of sparks
- Number of blocked threads
- Number of active threads

These statistics are useful when measuring the overall performance of a parallel program, but tells us very little about the usefulness of the threads themselves.

In order to ensure that the iterative feedback system is able to determine the overall 'health' of a thread, it is important that we collect some statistics pertaining to each individual thread. For this reason we have used a similar system as that outlined in [16]. With the following metrics being recorded *for each thread*:

- Number of reduction cycles
- Number of sparks
- Number of blocked threads
- Which threads have blocked the current thread

This allows us to reason about the productivity of the threads themselves. An ideal thread will perform many reductions, block very few other threads, and be blocked rarely. A 'bad' thread will perform few reductions and be blocked for long periods of time.

5.0.2 Transformation:

In order for the iterative feedback to be able to change the parallelization of a program, it must be able to determine which expressions can be transformed. The method we have devised is based on the idea that a specific `par` in the source program can be deactivated and therefore no longer create parallel tasks, while maintaining the semantics of the program. The method has two basic steps:

- `par`'s are identified via the G-Code instruction `PushGlobal "par"` and each par is given a unique identifier.
- When a thread creates the heap object representing the call to `par` the runtime system looks up the status of the `par` using its unique identifier. If the `par` is 'on' execution follows as normal. If the `par` is off the thread will ignore the G-Code instruction `Par`.

5.0.3 Iteration:

Using the runtime profile data we can experiment with different search methods. We can represent a program's `par` switches as a bit string with each `par`'s current setting stored in a bit.

One technique would be to ignore the runtime data altogether! There are lots of algorithms used for searching for optimal configurations of bit strings. In our case the fitness-function would simply be the overall running time of the program when run with a specific `par` setting.

However, while we feel that blind search techniques are worth exploring, guided search is more likely to produce results quickly. The first search heuristic could be very simple: After every execution, turn off the `par` site whose threads have the lowest average reduction count. Repeat this process until switching a `par` off increases the overall runtime of the program.

Another possibility is to determine an overhead penalty for sparking parallel threads. If the average reduction count for all the threads from a `par` site is less than the overhead penalty, the `par` is switched off. Other forms of penalties could also be introduced. Blocking other threads, being blocked for extended periods of time, creating too many parallel threads (or not enough) could all be measures that incur a penalty.

6. Conclusions

We hope we have motivated the key design choices and ideas behind our compiler: Utilising defunctionalisation in §2, and the use of projections over other strictness analysis methods §3. And that we have shown that there is a natural correspondence between projections and strategies §4 that allows us to generate parallel strategies from the results of our strictness analysis.

6.1 Future Work

One area that we expect to explore is the use of other forms of specialisation. Defunctionalisation specialises higher-order functions to first-order ones. Other possibilities include specialising polymorphic functions into their monomorphic versions and specialising functions based on their call-depth.

The first of these allows for the possibility that a polymorphic function that introduces parallelism may only provide a benefit when applied to arguments of a certain type. The depth-specialisation confronts the common granularity problem when writing recursive algorithms that introduce parallelism. The top-level call of the function may see huge benefits from its parallelism, but the lower level calls may not be as worthwhile (the `nfib` function is a good example of this, parallelising the recursive calls of `nfib 25` may be worthwhile, but not for `nfib 2`).

References

[1] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A History of Haskell: Being Lazy with Class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

[2] K. Hammond and G. Michelson, *Research Directions in Parallel Functional Programming*. Springer-Verlag, 2000.

[3] G. Hogen, A. Kindler, and R. Loogen, "Automatic Parallelization of Lazy Functional Programs," in *ESOP'92*. Springer, 1992, pp. 254–268.

[4] R. J. M. Hughes, "The Design and Implementation of Programming Languages," Ph.D. dissertation, Programming Research Group, Oxford University, July 1983.

[5] S. L. Peyon Jones, "Parallel implementations of functional programming languages," *Comput. J.*, vol. 32, no. 2, pp. 175–186, Apr. 1989.

[6] G. Tremblay and G. R. Gao, "The Impact of Laziness on Parallelism and the Limits of Strictness Analysis," in *Proceedings High Performance Functional Computing*. Citeseer, 1995, pp. 119–k133.

[7] T. Harris and S. Singh, "Feedback Directed Implicit Parallelism," *SIGPLAN Not.*, vol. 42, no. 9, pp. 251–264, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1291220.1291192

[8] R. Hinze, "Projection-based Strictness Analysis: Theoretical and Practical Aspects," 1995, Inaugural Dissertation, University of Bonn.

[9] P. Wadler and R. J. M. Hughes, "Projections for Strictness Analysis," in *Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 385–407.

[10] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones, "Algorithm + Strategy = Parallelism," *J. Funct. Program.*, vol. 8, no. 1, pp. 23–60, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1017/S0956796897002967

[11] A. Mycroft, "The Theory and Practice of Transforming Call-by-Need Into Call-by-Value," in *International symposium on programming*. Springer, 1980, pp. 269–281.

[12] P. Wadler, "Strictness Analysis on Non-Flat Domains," in *Abstract interpretation of declarative languages*. Ellis Horwood, 1987, pp. 266–275.

[13] S. Marlow, P. Maier, H. Loidl, M. Aswad, and P. Trinder, "Seq No More: Better Strategies for Parallel Haskell," in *Proceedings of the third ACM Haskell symposium on Haskell*. ACM, 2010, pp. 91–102.

[14] A. Bloss, "Path Analysis and the Optimization of Nonstrict Functional Languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 328–369, 1994.

[15] D. Jones, Jr., S. Marlow, and S. Singh, "Parallel Performance Tuning for Haskell," in *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, ser. Haskell '09. New York, NY, USA: ACM, 2009, pp. 81–92. [Online]. Available: http://doi.acm.org/10.1145/1596638.1596649

[16] N. Charles and C. Runciman, "An Interactive Approach to Profiling Parallel Functional Programs," in *Implementation of Functional Languages*. Springer, 1999, pp. 20–37.

Branch and Bound in a Data Parallel Setting

Extended Abstract

Sven-Bodo Scholz
Heriot-Watt University
S.Scholz@hw.ac.uk

Abstract

This paper investigates how branch and bound algorithms can be implemented in a functional, data parallel setting. We identify a general programming pattern for such algorithms and we discuss compilation and runtime aspects when it comes to mapping the programming pattern into parallel code. We use the maximum clique problem in undirected graphs as a running example and we present first experiences in the context of SaC.

1. Introduction

Branch and bound algorithms (which, in the sequel, we will refer to as *BBA*s) play an important role for many combinatorial search and optimisation problems. Typically these problems are NP-complete and require, at least in principle, the inspection of a search tree of exponential size. The key idea of branch and bound algorithms is the identification of certain bounds that allow pruning the search tree. That way, the overall runtime in many real-world applications can be brought down to a level where useful results are feasible despite the NP-complete nature of the underlying problem.

Application areas for these algorithms are vast including many areas that gain importance in the context of *big data* such as bio-informatics, computational chemistry, or social network analytics. The wide range of applications combined with the desire to deal with ever increasing amounts of data creates a demand for attempts to scale these applications to many-core systems. However, the challenges of successfully parallelising BBAs are many-fold. While a first cut seems rather obvious, i.e., spawning several threads that investigate separate branches of the search tree, achieving a parallel performance that scales well is far from trivial: The search tree typically is not well balanced, it may not even be statically known. The effectiveness of the bounding process often depends on knowledge gained by previous search space exploration and it may differ depending on where in the search space the exploration happens.

For many application areas, there exists a large body of work which investigates the effectiveness of different BBAs for individual problem instances. Often the perceived best solutions depend not only on the executing hardware, whether the algorithm is executed sequentially or in parallel, but they also depend on the given data itself. Low-level implementations of these algorithms are tedious, error-prone and typically require a lot of fine-tuning to achieve reasonable runtime performance.

This appears to be a setting where a declarative approach might help, be it in the form of a DSL or in the form of special language constructs. This paper presents our results when looking at BBAs from a data parallel angle. While a data-parallel approach may at first glance seem counter-intuitive for this seemingly inherent task-parallel class of algorithms, it turns out that nested reductions (folds) are a very apt vehicle for formulating BBAs. They provide an easy way to conveniently specifying the need for branching and, at the same time, they enable a compilation into effectively executable parallel code.

The main challenge, as in the manual case discussed extensively in the literature, is an effective declaration of the bounding needs. To our surprise, it turns out that very few language mechanisms suffice to express the bounding needs elegantly. We discuss what these mechanisms are and we argue their versatility. Furthermore, we show that SaC already provides suitable mechanisms. We use a classical problem from graph theory as running example to present and contrast several different specifications in SaC. This allows us to obtain initial performance figures and to relate these to the implementation features used.

Acknowledgments

This work was supported in part by grant EP/L00058X/1 from the UK Engineering and Physical Sciences Research Council (EPSRC).

Stream Processing for Embedded Domain Specific Languages

Markus Aronsson Emil Axelsson Mary Sheeran
Chalmers University of Technology
mararon@student.chalmers.se, emax@chalmers.se, ms@chalmers.se

Abstract

We present a library for expressing digital signal processing (DSP) algorithms using a deeply embedded domain-specific language (EDSL) in Haskell. The library supports definitions in functional programming style, reducing the gap between the mathematical description of streaming algorithms and their implementation. The deep embedding makes it possible to generate efficient C code without any overhead associated with the high-level programming model. The signal processing library is intended to be an extension of the Feldspar EDSL which, until now, has had a rather low-level interface for dealing with synchronous streams. However, the presented library is independent of the underlying expression language, and can be used to extend any pure EDSL for which a C code generator exists with efficient stream processing capabilities. The library is evaluated using example implementations of common DSP algorithms and the generated code is compared to its handwritten counterpart.

1. Introduction

In recent years, the amount of traffic passing through the global communications infrastructure has been increasing at a rapid pace. Worldwide, total Internet traffic is estimated to grow at an average rate of 32% annually, reaching approximately eighty million terabytes per month by the end of next year [16]. Mobile communications in particular have been growing at a phenomenal rate, which can be largely attributed to the rising popularity of mobile terminals.

For telecommunications infrastructure, the consequences of such a rapid growth rate have been a dramatic increase in the demand for network capacity and computational power [1]. At the same time, telecom carriers are faced with an increasing need to deliver new services faster, while simultaneously adapting the the recent diversification in available architecture. These factors, while positively influencing the available computational power, have also significantly increased the complexity of developing new solutions for telecommunication systems.

Today, digital signal processing software is typically implemented in low level C, which forces designers to focus on low-level implementation details rather than the mathematical specification of the algorithms. Our group is developing an embedded domain-specific language (EDSL), Feldspar [3], that aims to raise the abstraction level of signal processing software by expressing algorithms as pure functional programs.

However, signal processing is more than just pure computations – it is also about how to connect those functions in a network that operates on streaming data. A suitable programming model for reactive systems that process streams of data is *synchronous dataflow* (SDF) [19], which offers natural, high-level descriptions of streaming algorithms, while still permitting the generation of efficient code. Feldspar does have a library for programming with synchronous streams, but that library is quite low-level and tedious to use.

This paper describes a library for extending an existing Haskell EDSL with support for SDF. The underlying EDSL is used to represent pure functions (which, of course, can be arbitrarily complicated), and our library gives a means to connect such functions using an SDF programming model. If the underlying EDSL provides a C code generator with a given interface, our library is capable of emitting C code for SDF programs. We are interested in using Feldspar as the expression language; however, the library is not dependent on Feldspar, and so may be of interest to other EDSL developers.

This paper makes the following contributions:

- We present a simple EDSL for synchronous dataflow programming in Haskell. Practically, the result is a useful addition to Feldspar.
- We make use of observable sharing [7] to achieve a deep embedding without relying on combinators to express sharing or cycles. This technique has long been used in the hardware description EDSL Lava [6, 11], but our work now permits the combination not just of simple gates but of arbitrarily complex EDSL programs.
- We abstract away from the underlying expression language by establishing an interface for the underlying expression compiler.
- We show how to generate C code from our library, and evaluate the results on examples.

1.1 Synchronous dataflow programming

Dataflow programming is a paradigm which internally models an application as a directed graph [17, 21], similar to a

dataflow diagram. Nodes in the graph are then executable blocks, representing the different components of an application: they receive input, apply some transformation, and forward it to the other connected nodes. A dataflow application is then, simply stated, a composition of such blocks, with one or more source and sink blocks.

A later extension to dataflow programming is the introduction of *synchronous* dataflow. SDF is a subset of pure dataflow, in which the number of tokens produced or consumed by nodes during each step of evaluation is known at compile-time. Restricting the dataflow model in this way has the advantage that it can be statically scheduled [18], which, in turn, allows for generation of efficient code.

Lucid Synchrone [8, 20] is a member of the family of synchronous languages and is designed to model reactive systems. It was introduced as an extension of LUSTRE [13], and demonstrated that the language could be extended with new and powerful features. For instance, automatic clock and type inference was introduced, and a restricted form of higher-order functions was added. However, Lucid Synchrone is a standalone language which cannot easily be integrated with EDSLs such as Feldspar. For this reason, we chose to implement a library, partly inspired by Lucid Synchrone, that brings an SDF programming model to existing EDSLs, such as Feldspar.

2. Signal

This library is based on the concept of a *signal*, which represents an infinite sequence of values in some pure expression language. Signals are constructed by the following interface:

```
map    :: (exp a → exp b)
          → Signal exp a
          → Signal exp b

repeat :: exp a → Signal exp a

zip    :: Signal exp a
          → Signal exp b
          → Signal exp (a, b)

fst    :: Signal exp (a, b)
          → Signal exp a
```

where `exp` is the pure expression language. The `map` function promotes a pure function to operate over signals; `repeat` makes a constant-valued signal; `zip` and `fst` are used to make nodes with multiple incoming or outgoing signals.

Sequential operations are supported through the following functions, which manage a signal's phase and frequency:

```
delay  :: exp a
          → Signal exp a
          → Signal exp a

sample :: exp Int
          → Signal exp a
          → Signal exp a
```

While few in number, these sequential functions are quite general and allow for arbitrary feedback networks to be expressed.

The need to implement particular signal functions may place demands on the underlying expression language, in that support for common data types or functionality may be required. For instance, in order to implement a signal version of Haskell's `zipWith` function, the expression language needs to support tuples:

```
class TupExp exp
  where
    tup :: exp a → exp b → exp c
    fst :: exp (a,b) → exp a
    snd :: exp (a,b) → exp b

zipWith :: (TupExp exp, Signal exp ~ sig)
           ⇒ (exp a → exp b → exp c)
           →  sig a → sig b → sig c
zipWith f s u = map (λp → f (fst p) (snd p))
                $ zip s u
```

Classes such as `TupExp` provide a suitable interface with the expression language, but without forcing a particular language to be hardwired into the system.

2.1 Example: FIR Filter

Consider the mathematical definition of a finite impulse response filter of rank N:

$$y_n = \sum_{i=0}^{N} b_i * x_{n-i}$$

This description is convenient for software realization, as it can be deconstructed into three main components: a number of successive unit delays, multiplication with coefficients and a summation. We can represent the decomposed filter graphically, as in Figure 1.

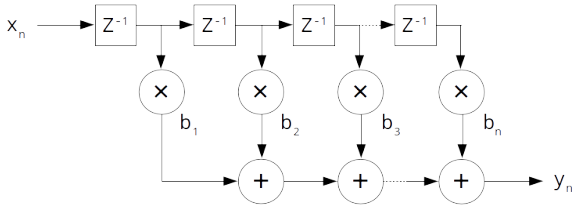


Figure 1. A direct form discrete-time FIR filter of order N

Support for such numerical operations over signals is implemented by instantiating their corresponding classes in Haskell:

```
instance (TupleExp exp, Num (exp a)) ⇒
    Num (Signal exp a)
  where
    fromInteger = repeat . fromInteger
    (+)         = zipWith (+)
    (-)         = zipWith (-)
    ...
```

where similar instance declarations can be made for fractional and floating point arithmetic.

Using Haskell's standard classes in this way simplifies the construction of signals by providing a homogeneous user interface. Furthermore, as the pure Haskell code is separated from our signal library in this way, it introduces a meta-level of computation, helping us to reason about program

correctness. The ability to use pure Haskell in this way presents several benefits, as it improves the syntax and ease of programming signals significantly. For instance, in order to define complex networks, the user is only required to be versed in Haskell's standard library operators, thereby further reducing the complexity of developing new networks.

Given this support for numerical operations, we now create helper functions, modeling the three main components of the FIR filter: summation and multiplication of signals and successive delaying. Using Haskell's standard library functions, summation can be neatly expressed as a single fold operation:

```
import qualified Prelude as P

sums :: (TupExp exp, Num (exp a))
        ⇒ [Signal exp a]
        →  Signal exp a
sums = P.foldr1 (+)
```

Similarly, both of the remaining components can be expressed using standard Haskell functions:

```
muls :: (TupExp exp, Num (exp a))
        ⇒ [exp a]
        → [Signal exp a]
        → [Signal exp a]
muls = P.zipWith (λc s → repeat c * s)

delays :: [exp a]
       → Signal exp a
       → [Signal exp a]
delays as s = P.tail
            $ P.scanl (P.flip delay) s as
```

The FIR filter can now be neatly expressed as:

```
fir :: (TupleExp exp, Num (exp Float))
        ⇒ [exp Float]
        → Signal exp Float
        → Signal exp Float
fir bs = sums . muls bs . delays ds
  where
    ds = P.replicate (P.length bs) 0
```

This description is close to the filter's graphical representation, a beneficial attribute since domain experts in DSP tend to be comfortable with composing sub-components in this way.

2.2 Example: IIR Filter

Infinite impulse response (IIR) filters are digital filters with an infinite impulse response and, unlike FIR filters, contain feedback. These filters will therefore serve as an example of how the signal library handles recursively defined signals, that is, signals whose output depends on a combination of previous input and output values.

The IIR filter is often described and implemented in terms of a difference equation, which defines how the output signal is related to the input signal:

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i * x_{n-j} \; - \; \sum_{j=1}^{Q} a_j * y_{n-j}\right)$$

where P and Q are the feedforward and feedback filter orders, respectively, and a_j and b_i are the filter coefficients. We can represent the decomposed filter graphically, as in Figure 2.

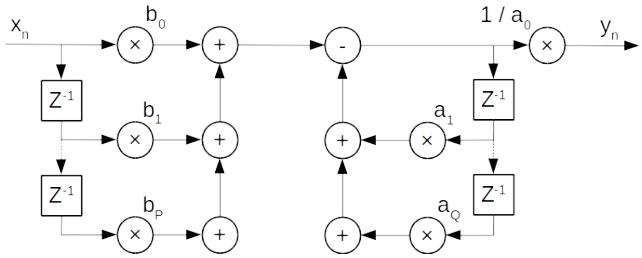


Figure 2. A direct form discrete-time IIR filter of order P and Q

This description, besides the subtraction and division of signals, is quite similar to the previous FIR filter when deconstructed. This similarity seems to imply that they share computational components. As it turns out, the previously defined helper functions can indeed be reused to implement the IIR filter as well:

```
iir :: ( TupleExp exp
       , Num (exp Float)
       , Fractional (exp Float))
       ⇒ [exp Float]
       → [exp Float]
       → Signal exp Float
       → Signal exp Float
iir (a:as) bs s = repeat (1 / a) * (l - r)
  where
    l = sums $ muls bs $ delays (inits bs) s
    r = sums $ muls as $ delays (inits as) r

    inits = P.flip P.replicate 0 . P.length
```

where the rightmost summation, here called `r`, is defined in terms of itself rather than the input signal. Recursive definitions like this are made possible by the lazy nature of the delay operator. The general idea is that any recursion expressed using the signal library introduces feedback, while recursion introduced by pure Haskell code produces repetitive code instead.

3. Implementation

Signals are implemented on top of the following `Stream` data type:

```
data Stream exp a =
     Stream (Prog exp (Prog exp a))
```

The `Prog exp` monad is a deep embedding of an imperative programming language that uses `exp` to represent pure expressions. In the above definition, the outer monad is used to initialize the stream, and the inner action is used to retrieve the next value of the stream. For example, the `head` function, which retrieves the first element of a stream is defined as follows:

```
head :: Stream exp a → Prog exp a
head (Stream init) = do
    next ← init
    next
```

The first line in the `do` block initializes the `next` action, and the second line uses `next` to get the first element.

Our model of streams is essentially the same as in Feldspar's `Stream` library, except that the `Prog exp` monad used here is a standalone monad that adds imperative programming on top of any pure expression EDSL.

The problem with `Stream` is that is quite low-level and cumbersome to program with. For example, in order to define a filter that refers to previous values of some signal, one has to manually create a mutable buffer and update it on every iteration. Feldspar exports a few combinators that hide the details of creating such networks, but these combinators are rather ad hoc, and can only handle a few predefined cases.

Our `Signal` type can be seen as a front end to `Stream` that offers a much more convenient interface. In the end, functions on signals are compiled to functions on streams, as seen in the type of the compile function:

```
compile
  :: (Typeable a, Typeable b)
  ⇒ (Signal exp a → Signal exp b)
  → IO (Stream exp a → Stream exp b)
```

The `IO` in the result type comes from the `data-reify` package that performs observable sharing [10].

The basic way to create nodes in a signal network is by lifting a stream function to a signal function:

```
lift :: (Stream exp a → Stream exp b)
        → Signal exp a
        → Signal exp b
```

We can, for example, use `lift` to define `map` from Section 2:

```
map f = lift (Stream.map f)
```

Lifting is however not enough to define generators, as those are supposed to create signals from nothing. Another signal construct is therefore introduced, modeling the signals identity morphism:

```
bot :: Signal exp a
```

which allows us to define `repeat` as:

```
repeat e = lift (const $ Stream.repeat e) bot
```

Here, `Stream.map` and `Stream.repeat` are the corresponding functions defined for streams.

In this extended abstract, we will not show the definition of `Signal`, `lift` and `compile`, but here is an outline of how it all works:

- `Signal` is a tree type, whose nodes consist of lifted stream functions, `delay` and `sample`. Observable sharing is used to turn this tree into a DAG.
- The compiler assigns a mutable reference (supported by the imperative `Prog` monad) for each node in the graph and creates a program that executes all nodes in sequence, reading and writing data to the corresponding references.
- By analyzing the graph, certain references can be eliminated, and chains of `delay` nodes can be turned into efficient cyclic buffers.

While the stream type is kept abstract in signals, basing them on the co-iterative approach [5] allows us to ensure that no unused computations are performed. Co-iteration is a concept for reasoning about infinite streams and allows one to handle such streams in a strict manner. This strictness in turn enables stream transformers to pick and remove parts of streams as they please – ensuring that no unsued part of a stream is ever computed. This property is kept for signals, as lifted nodes are fused during compilation.

Furthermore, signals offer optimization for a common concept in DSP: *feedback networks*. As the recursively defined streams in feedback networks may reference a number of previous values, memory efficient buffers are introduced for storing the delayed values in these signals. These buffers have memory proportional in size to the number of delays and are used to minimize the number of read/write operations used during execution. Detecting such feedback in signal networks is made possible by using observable sharing [7], which allows us to reify signal networks into graphs were the back-edges between nodes are visible.

4. Evaluation

In order to evaluate the signal library we will look at both its expressiveness and the code it generates. As the actual library is being finished at the time of writing, we delay most of the evaluation until the final report and only include a comparison between the generated code of an example and its hand-written counterpart. Note, too, that the compiler has some notable limitations: it currently only handles pure signal functions, that is, we can only compile types of `Signal exp a → Signal exp b`. Also, while signals support the notion of buffers, the compiler does not. The following example will therefore not make use of circular arrays, but will do so in the final version.

Disregarding the current state of our compiler, the produced code has room for several potential improvements. Firstly, too many references are used during the compilation process, which then spills into the compiled code and produces a number of unused variables. There are also some leftover pairs found in the produced code, remnants from the zipping constructor. Both the numerous variables and the pairs can however be optimized away, and should not be present in the final report. There is also room for more subtle improvements; for example, a dedicated initialization function could remove the need for some if-statements. Another interesting improvement is to make use of C specific memory management functions, such as `memset` or `memcopy`, for cases when the data is neatly ordered – as it is in the case of the FIR filter, for example.

For the actual comparison, we generated code from the previous FIR filter and compared it to a hand-written version. However, as the compiler only accepts signal functions, we feed the FIR filter a one-element list of ones before compiling it – effectively making it a rank-1 FIR filter. This produces the following code:

```
typedef struct {
  float first;
  float second;
} pair;

int main()
{
  FILE *v0;
  v0 = fopen("test", "r");
  FILE *v1;
  v1 = fopen("test2", "w");

  int i = 0;
  while (i < 3)
  {
    float v3;
    float v4;
    fscanf(v0,"%f",v4);
    float v5 = v4;
    float v6;
    pair v7;
    float v8;
    float v9;
    bool v10 = true;
    float v11 = v5;
    bool v12;
    if (v12 = v10)
    {
      v10 = false;
      v8  = 0.0;
    }
    else
    {
      float v13 = v9;
      v8 = v13;
    }
    v9 = v11;
    float v14;
    v14 = 1.0;
    float v15 = v14;
    float v16 = v8;
    v7.first = v15;
    v7.second = v16;
    pair v17 = v7;
    v6 = v17.first * v17.second;
    float v18 = v6;
    v3 = v18;
    float v19 = v3;
    fprintf(v1, "%f", v19);
    i++;
  }

  return 0;
}
```

While the hand-written code is for a more general FIR filter, where the number of coefficients hasn't been fixed yet, the comparison should however still be valid as the underlying ideas have not changed.

```
double insamp[ ... ];

void firInit( void )
{
    memset( insamp, 0, sizeof( insamp ) );
}

void fir( double *coeffs,
       double *input, double *output,
       int length, int filterLength )
{
    double acc;     // accumulator for MACs
    double *coeffp; // pointer to coefficients
    double *inputp; // pointer to input samples
    int n;
    int k;

    // put the new samples at the high
    // end of the buffer
    memcpy( &insamp[filterLength - 1], input,
            length * sizeof(double) );

    // apply the filter to each input sample
    for ( n = 0; n < length; n++ ) {
        // calculate output n
        coeffp = coeffs;
        inputp = &insamp[filterLength - 1 + n];
        acc = 0;
        for ( k = 0; k < filterLength; k++ ) {
            acc += (*coeffp++) * (*inputp--);
        }
        output[n] = acc;
    }

    // shift input samples back in time
    // for next time
    memmove( &insamp[0], &insamp[length],
            (filterLength - 1) * sizeof(double) );

}
```

While benchmarking would clearly be of interest here, we delay it to the final paper.

5. Related Work

Lava is a family of simple hardware description languages embedded in Haskell [6, 11]. What look like circuit descriptions are actually circuit generators that are run to produce internal representations, which can be used to produce further useful artifacts such as netlists or formulas for use in formal verification. Later versions of Lava have used observable sharing to provide a more user-friendly approach to capturing feedback in these circuit descriptions. There is a close link between this approach to hardware description and SDF languages like LUSTRE. One way to view the present work is as a beefed up Lava in which the building blocks are general data-processing functions rather than just Boolean gates.

Lucid Synchrone is another functional language for SDF and is hosted in OCaml, importing every ground type from the host language and lifting them to corresponding stream versions [8, 20]. Sequential operations over the imported stream are also offered, similar to those from our signal library. Lucid Syncrhone incorporates several type systems

(including clock inference) that guarantee safety properties of the generated code. In addition, special syntax for defining automata is provided. Our work is inspired by Lucid Synchrone and we will investigate the inclusion of these features in our signal library.

Functional reactive programming is another common paradigm for modeling continuous signals, and its libraries are typically implemented using Haskell's arrow or applicative classes, see for example Yampa [9, 12], reactive-banana [2] or Sodium [4]. Although we cannot support the promotion of abstract functions required by these otherwise attractive interfaces, we have still drawn inspiration from the lifting functions of FRP.

The use of pure Haskell for modeling signals [22] has also been investigated, demonstrating that functional programming and lazy evaluation can directly model common signal problems quite satisfactorily. Other related work includes Atom [14], Ivory and Tower [15], but we delay the discussion of these to the final paper.

6. Discussion

Functional programming encourages a style of programming in which combinators or higher order functions capture common patterns of computation. It is when we combine SDF with a sufficiently general value type that the question of how to design an appropriate set of combinators becomes an interesting one. For instance, the ability to pass arrays, or any similar data type, as values over signals means that the programmer is concerned with processing chunks of data rather than just individual values. This change of perspective is necessary if we are to implement key functions like FFT on signals. This generality does however come at a cost, as ill-defined signals could be expressed and type-checked due to a lack of constraints on sequential signals

The benefits of a general lifting constructor come into full effect when the underlying expression language has powerful features of its own. In the case of Feldspar, an expression language which already supports a plethora of different algorithms, several complex signal functions can often be obtained by simply lifting the regular ones. For instance, as Feldspar already contains an FFT algorithm over vectors, a signal version can be obtained by simply lifting the existing one.

The simplicity of the current signal library does mean that some ill-defined signals can be expressed. Sequential operations in particular are quite susceptible to grammatical errors, as one can easily create signals with an undefined behavior when using delay/sample. For instance, consider the following function:

```
f :: Signal exp a → Signal exp a
f sig = sample 2 sig + sig
```

The clocks of these two streams are obviously not equal, but there are at present no constraints in place to keep such signals for being defined. Recursively defined signals suffer from a similar problem: there is no constraint in place to check whether the recursive signal has been delayed or not before it is used. Signals with undefined initial values can therefore be expressed by, for example, writing:

```
g :: Num (exp a) ⇒ Signal exp a
g = o where o = map (+1) o
```

Such ill-defined signals could however be identified during the compilation process, by inspecting the signal's reified syntax tree.

Our plan is to develop this library futher, based on ideas from Lucid Synchrone and FRP.

Acknowledgments

This research was funded by the Swedish Foundation for Strategic Research (in the RAW FP project) and by the Swedish Research Agency (Vetenskapsrådet).

References

[1] Cisco visual networking index: Forecast and methodology, 2012–2017, May 2013. URL http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf.

[2] H. Apfelmus. Reactive-banana. *Haskell library available at http://www. haskell. org/haskellwiki/Reactive-banana*, 2012.

[3] E. Axelsson, K. Claessen, G. Dèvai, Z. Horvàth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178, July 2010. .

[4] S. Blackheath. Sodium reactive programming (frp) system, March 2014. URL https://hackage.haskell.org/package/sodium.

[5] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *Electronic Notes in Theoretical Computer Science*, 11(0):1 – 21, 1998. ISSN 1571-0661. . URL http://www.sciencedirect.com/science/article/pii/S1571066104000507. {CMCS} '98, First Workshop on Coalgebraic Methods in Computer Science.

[6] K. Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.

[7] K. Claessen and D. Sands. Observable sharing for functional circuit description. In P. Thiagarajan and R. Yap, editors, *Advances in Computing Science — ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66856-5. . URL http://dx.doi.org/10.1007/3-540-46674-6_7.

[8] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 230–239, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. . URL http://doi.acm.org/10.1145/1017753.1017792.

[9] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM. ISBN 1-58113-758-3. . URL http://doi.acm.org/10.1145/871895.871897.

[10] A. Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. . URL http://doi.acm.org/10.1145/1596638.1596653.

[11] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing kansas lava. In M. Morazán and S.-B. Scholz, editors, *Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16477-4. . URL http://dx.doi.org/10.1007/978-3-642-16478-1_2.

[12] G. Giorgidze and H. Nilsson. Switched-on yampa. In P. Hudak and D. Warren, editors, *Practical Aspects of Declarative*

Languages, volume 4902 of *Lecture Notes in Computer Science*, pages 282–298. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-77441-9. . URL http://dx.doi.org/10.1007/978-3-540-77442-6_19.

[13] N. Halbwachs. *Synchronous programming of reactive systems*. Number 215. Springer, 1992.

[14] T. Hawkins. Controlling hybrid vehicles with haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, 2008.

[15] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury. Building embedded systems with embedded DSLs. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 3–9. ACM, 2014.

[16] ITU. Global itc development, 2001-2014, 2014. URL http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2014/stat_page_all_charts_2014.xls.

[17] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/1013208.1013209.

[18] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24–35, Jan 1987. ISSN 0018-9340. .

[19] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987. ISSN 0018-9219. .

[20] M. Pouzet. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006.

[21] T. B. Sousa. Dataflow programming: Concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, 2012.

[22] H. Thielemann. *Audio processing using Haskell*. Zentrum für Technomathematik, 2004.

Flipping Fold, Reformulating Reduction

An Exercise in Categorical Design

Gershom Bazerman
S&P/CapitalIQ
gershomb at gmail

1. Introduction

We begin this paper by considering the Haskell ‘Foldable‘ typeclass, a stalwart of the standard libraries. Unlike many other typeclasses, most famously Monad, Foldable itself has been equipped with no required laws. This is rather surprising, as folds themselves are some of the most well understood and studied aspects of functional programming, and the universal properties of folds, in general, are what we often use to prove laws. We will explore why it is hard to give laws to Foldable on its own. From there we will define a naturally arising class, adjoint to Foldable, which we name Buildable. In turn, we will explore how Foldable and Buildable in conjunction, each individually lawless, nonetheless are mutually constrained by an elegant set of laws arising from categorical principles. We will then explore Buildable as an independently useful class that allows us to compose systems of streaming and parallel computation, and explore its relationship to a prior, similar formulation. The aim of this paper is then threefold; to provide laws to Foldable, to provide a new, useful class of Buildable types, and along the way, to illustrate a way in which categorical thinking can give rise to practical results.

2. Recalling Foldable

The ‘Data.Foldable‘ library, written by Ross Paterson, and part of the standard libraries that ship with the Glasgow Haskell Compiler, provides a Foldable typeclass. While it has many methods, all methods can be derived by the user defining only one of foldr or foldMap. So we consider the cleaner interface given below.

```
class Foldable t where
    foldr ::
        (a -> b -> b) -> b -> t a -> b Source
    foldMap ::
        Monoid m => (a -> m) -> t a -> m Source
```

In fact, there is a further function, not in the class, but included in the file, which also provides a complete implementation of ‘foldable‘. We can consider its definition as follows:

```
toList :: Foldable t => t a -> [a]
toList t = foldr (:) [] t
```


Without much work, one can see how ‘foldr‘, ‘foldMap‘, and ‘build‘ are all interdefinable and hence equal in expressive power. When one considers folds in general, one typically expects them to universally characterize the meaning of a particular data structure in terms of all operations possible on it – in fact that is the very definition of a proper fold. However, we can observe that the ‘foldr‘ given here in fact characterizes all operations possible on a data structure ***when considered as a list***. So the laws of folds themselves follow naturally from our usual constructions, and are given directly. However, what it means to consider a data structure to a list is left completely undefined. For example, we could equip all type constructors of arity one with the Foldable instance who acts as the empty list. This would violate user expectations, but not any particular given typeclass law. Clearly something must be done.

3. Enter Buildable

If the laws of Foldable won’t come from the class itself, then they must come from interaction with other classes. This is the pattern we have seen elsewhere, recently where work by Jaskelioff, and later Bird and Gibbons has provided ‘Traversable‘ functors with laws as given by their interrelationship with Applicative actions.[6][1] Much earlier, of course, we had to define the relationship of ‘Eq‘ and ‘Ord‘ instances such that they agreed. Other examples also abound.

What class shall we use to interact with ‘Foldable‘? A clue is provided in the genuine definition of ‘toList‘, which in turn is defined in terms of ‘build‘, imported from ‘GHC.Exts‘.

```
toList :: Foldable t => t a -> [a]
toList t = build (\ c n -> foldr c n t)

build ::
    forall a.
    (forall b. (a -> b -> b) -> b -> b)
    -> [a]
build g = g (:) []
```

Why this indirection? Well, as the documentation tells us, ”GHC’s simplifier will transform an expression of the form ‘foldr k z (build g)‘, which may arise after inlining, to ‘g k z‘, which avoids producing an intermediate list.”. This is an instance of ”shortcut fusion” as introduced by Gill, Launchbury, and Peyton Jones.[2] Recent work by Hinze[**?**] has explored the relationship between shortcut fusion and the categorical notion of an ***adjoint***, which we will come back to. In any case, in the special case of ‘foldr‘ and ‘build‘ on lists, we observe that they correspond to providing a full isomorphism between lists and the partial application of the fold function to lists, which is to say between lists seen ”initially” and lists seen ”finally” as characterized by their universal property.

Just as as the ‘Foldable‘ typeclass simply wraps up ‘fold‘, we now introduce a ‘Buildable‘ typeclass to wrap up ‘build‘. As all Foldables can provide a ‘toList‘, we also provide a ‘fromList‘ to help examine the behaviour of Buildables.

```
class Buildable f a where
    build :: ((a -> f a -> f a) -> f a -> f a) -> f a
    build g = g insert unit

    singleton :: a -> f a
    singleton x = build (\c n -> c x n)

    unit ::  f a
    unit = build (\cons nil -> nil)

    insert :: a -> f a -> f a
    insert x xs = build (\cons nil -> x `cons` xs)

fromList :: Buildable f a => [a] -> f a
fromList xs = foldr insert unit xs
```

A minimal complete definition is given by ‘build‘, or by ‘insert‘ coupled with ‘unit‘. The ‘build‘ function can be seen as providing the concrete constructors to a partially applied fold, and the ‘insert‘ and ‘unit‘ functions as just introducing the two constructors (the binary and unary operations) explicitly.

There are a few design decisions here worth justifying. First, the choice to use a multi-parameter typeclass, and second the choice (only implicitly present here) not to require any sort of monoidal behaviour, and instead a looser notion of “adjoint” laws. Both decisions can be justified by examining a standard type that clearly should be buildable, but nonetheless is not isomorphic to list – ‘Set‘. We can write a Buildable instance for ‘Set‘ like so:

```
instance Ord a => Buildable Set a where
    unit = Set.empty
    insert = Set.insert
```

Here the purpose of the extra type variable becomes clear – while the ‘Ord‘ constraint is not necessary to “tear down” a set, it certainly is necessary to build one up, and thus must be included in our typeclass. While this costs us in terms of verbosity, at least it introduces no loss in expressiveness.

Now we consider the behaviour of the interaction of fromList and toList on ‘Set‘. Whatever laws we introduce must surely not rule out such a basic instance. Clearly we expect ‘`thereBack xs :: toList . Set.fromList`‘ to reorder our elements. Furthermore, we expect it to merge duplicate elements. However, we also know that if we iterate ‘thereBack‘ repeatedly, it is idempotent. In this case, ‘toList‘ is a retraction of ‘fromList‘, and the composition ‘fromList . toList‘ is a split idempotent. More generally, we can consider the functorial nature of ‘Foldable‘ and ‘Buildable‘ to produce a set of laws claiming that when both instances exist, they should be ***adjoint***.

4. Folds, Builds, and Adjunctions

The connection of adjointness to folds, unfolds and fusion laws has been explored in the recent work of Ralf Hinze[4,5]. In general, fusion laws are about moving to an “adjoint space” where composition is directly given, and then shifting back to the original space to present the result. Although the movement between regular and church-encoded lists given in fold/build fusion is an isomorphism, in general there is no such restriction. Streams including ‘yield‘ are a bigger space than lists, etc. The purpose behind such adjunctions is, loosely speaking, to allow us to capture “only what matters” about a computation. When “moving across” the two functors which make up an adjoint, we are able to transport where the work of functions occurs.

To examine how this plays out in the terrain of ‘Foldable‘ and ‘Buildable‘ we can translate a version of the adjoint laws to our specialized usecase. In the ”hom-set adjunction” formulation, for two categories C and D and two functors $F : D \to C$ and $G : C \to D$, we have the formula:

$$C(FX, Y) \cong D(X, GY)$$

Take C to be some ‘Buildable‘ and ‘Foldable‘ functor f, and D to be ‘List‘, and we arrive at the following Haskell claim: For all functions ‘`f : f a -> f b`‘, there is a function ‘`g : [a] -> [b]`‘ such that ‘`f . fromList :: [a] -> f b`‘ is isomorphic to ‘`toList . g :: [a] -> f b`‘. That is to say, all functions on ‘Foldables‘ can be translated to functions on ‘Buildables‘, and vice versa, such that even if they do not actually coincide, when we ”move across” the types appropriately, they will.

When our ‘Buildable‘ and ‘Foldable‘ instances are lawful, we can in fact write functions to witness this directly, if not efficiently.

```
f2g f = toList . f . fromList
g2f g = fromList . g . toList
```

And so we see that functions on lists may be seen as functions on functors adjoint to list ”factored through” lists, and dually that functions on functors adjoint to list may be viewed as actions on lists ”factored” through the adjoint, and that such notions coincide. In the specific case of ‘Set‘, this means that there is no function on sets that cannot be written as a function on the list underlying a set, and furthermore that there is no function yielding a list that underlies a Set that cannot be transformed directly into a function on sets.

5. Reducers as Buildables

Hinze and Jeuring introduced a predecessor class to ‘Foldable‘ named ‘Reduce‘.[5] However, it is in fact ‘Buildable‘ that really provides the ”reduction” component directly – with ‘Foldable‘ describing the ”shape” of a reduction but ‘Buildable‘ providing the actual ***target semantics*** of any given fold. ‘Foldable‘ describes how to fold, but it is ‘Buildable‘ that fixes a fold to a concrete meaning. In fact, ‘Buildable‘ provides a very close analog, though more theoretically motivated, to the ‘Monoidal Reducers‘ available in Edward Kmett’s reducers package.

The following code listing demonstrates the “basic” functionality that all notions of reduction should share – the ability to define multiple aggregations such as sum and count, and the ability to zip them into one pass. Here the aggregations we define happen to be in fact monoidal. But in general, no such restriction applies.

```
newtype Sum a = Sum {getSum :: a}
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x + y)

newtype Count = Count Int deriving Show
instance Monoid Count where
    mempty = Count 0
    mappend (Count x) (Count y) = Count (x + y)

newtype Const m a = Const m

instance Num a => Buildable (Const (Sum a)) a where
    unit = Const (Sum 0)
    insert x (Const xs) = Const (Sum x `mappend` xs)

instance Buildable (Const Count) a where
```

```
    unit = Const (Count 0)
    insert x (Const xs) = Const (Count 1 `mappend` xs)

newtype Product f g a = Product (f a, g a)

instance (Buildable f a, Buildable g a) =>
               Buildable (Product f g) a where
    unit = Product (unit, unit)
    insert x (Product (xs,ys)) =
                  Product (insert x xs, insert x ys)
```

The listing contains two items of particular interest. First, we introduce a 'Const' type to carry around explicit information about what should be "fed in" to a 'Buildable', and more generally to lift an aggregation into a functorial context. Second, we introduce a traditional product of functors, and give it a 'Buildable' instance directly. By construction our builds only require one pass, and so we can introduce concurrent reductions while operating in constant space.

6. Composing Buildables Horizontally and Vertically

7. Extensions and Transformations

8. Serial and Parallel Computation

9. Relating Builds to Traversals

10. Related Work

As discussed, the closest analogue to the work presented here is Edward Kmett's monoidal reducers package. The concrete difference is that rather than generalize over things of kind '`* -> *`', Monoidal Reducers are equipped with two type parameters, each of kind '`*`' – the things that reducers "accept", and the things that reducers "reduce to." Furthermore, these reducers, as one would infer from the name, are required to operate as a monoid does, i.e. associatively. (Less importantly for our purposes, Monoidal Reducers, as one would not infer from the name, are in fact generalized as to work over semigroups [i.e. they do not require an "empty" value equivalent to 'unit' as presented here]). In the absence of any other constraints, requiring associative structure is about the minimal law one can require such a structure to hold. However, as have seen, in the presence of an interaction with 'Foldable', we can get a looser but still sufficient notion of a lawful structure even without requiring associativity – and in fact, there are very good reasons we should not!

Rich Hickey also arrived at similar formulations to Kmett's, though in an untyped context, in the 'reducers' library for Clojure. The inspiration for both lines of work is owed to Guy Steele's 2009 ICFP invited talk "Organizing Functional Code for Parallel Execution."

11. Conclusion

Acknowledgments

References

[1] Bird, R., Gibbons, J., et al. Haskell 2013: 25-36

[2] Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993) A short cut to deforestation. Proceedings, Conference on Functional Languages and Computer Architecture, 223-232.

[3] Hinze, R. Adjoint Folds and Unfolds. MPC 2010: 195-228

[4] Hinze, R. Type Fusion. AMAST 2010: 92-110

[5] Hinze, R. and Jeuring, J. Generic Haskell: Practice and theory. Technical Report UU-CS-2003-15, Department of Computer Science, Utrecht University, 2003.

[6] Jaskelioff, M., Rypacek, O. MSFP 2012: 40-49

Parametric lenses: change notification for bidirectional lenses

László Domoszlai
Radboud University Nijmegen, Netherlands, ICIS, MBSD
dlacko@gmail.com

Bas Lijnse Rinus Plasmeijer
Radboud University Nijmegen, Netherlands, ICIS, MBSD
b.lijnse@cs.ru.nl,rinus@cs.ru.nl

Abstract

In most complex applications it is inevitable to maintain dependencies between the different subsystems based on some shared data. The subsystems must be able to inform the dependent parties that the shared information is changed. As every actual notification has some communication cost, and every triggered task has associated computation cost, it is crucial for the overall performance of the application to reduce the number of notifications as much as possible. To achieve this, one must be able to define, with arbitrary precision, which party is depending on which data. In this paper we offer a general solution to this general problem. The solution is based on an extension to bidirectional lenses, called *parametric lenses*. With the help of parametric lenses one can define compositional *parametric views* in a declarative way to access some shared data. Parametric views, besides providing read/write access to the shared data, also enable to *observe* changes of some parts, given by an explicit parameter, the *focus domain*. The focus domain can be specified as a *type-based query language* defined over one or more resources using predefined combinators of parametric views.

1. Introduction

Complex applications commonly have to deal with shared data. It is often confined to the use of a relational database coupled with a simple concurrency control method, e.g., optimistic concurrency control [2]. In other cases, when a more proactive behavior is required, polling or some ad hoc notification mechanism can be invoked. At the farther end of the range there are some very involved applications (multi-user applications, workflow management systems, etc.), which are based on interdependent tasks connected by shared data. In the most general case, one has to deal with complex task dependencies defined by shared data coming from diverse sources, e.g. different databases, shared memory, shared files, sensors, etc.

As an example, consider the following case which is based on a prototype we have developed for the Dutch Coastguard [3]; it will be used throughout the paper to introduce the problem, and the concepts of the proposed solution. We have a small database which acts as a source of data of ships: name, cargo capacity, last known position, etc. The positions of the ships are updated repeatedly as the ships move; ships have a transponder on board which send their latest position on a regular basis. As a basic task, we simply want to show the positions of the ships on a map, of which users are allowed to select an area to view, the *focus* of their interest, the *focus domain*. In this setting we can think of map instances and update processes as interdependent tasks that are connected by the data of ships they share. When the position of a ship is updated in the database, the map instances, of which focus domain covers the old or the new coordinates, must be refreshed.

From a theoretical perspective, it would be correct behavior to notify every map instance on ship movement. However, this leads to huge efficiency issues in practice. There are many thousands of ships in the North Sea constantly moving around. Only those map instances need to be refreshed on which area the position of a ship is changed. As every actual notification has some communication cost, and every triggered task has associated computation cost, it is crucial for the overall performance of the application to reduce the number of notifications as much as possible. Thus, we need a notification system which, for efficiency reasons, is as accurate as possible.

As the problem described above is a very common computational pattern, we would like to offer a general, reusable solution. In addition, we would like to solve it *efficiently* enough to be comparable with the ad-hoc solutions.

From the computational perspective, focusing on a specific domain of the underlying data can be achieved by creating and working with one of its abstract views. Lenses [1] are commonly used for creating abstract views. They can be used to support partial reading and writing, for access restriction or to provide a specific view of the data. Lenses enable to define bi-directional transformations. In a nutshell, a lens describes a function that maps the input to an output (called *get*) and backwards (called *put*). The *get* function maps the input to some output, while the *put* function maps the modified output, together with the original input, to a modified input:

$$get \in X \to Y$$
$$put \in Y \times X \to X$$

In our example two kind of abstract views are needed for serving different processes: one to show the ships located in a given area of the map, and another one for the update process, which periodically updates the coordinates of a ship in the database.

The general notification problem is as follows. Given is a set of shared data sources of any type (*A* and *B* in the picture) holding a set of data (D_A, D_B). There are also given some lenses defined on top of the data sources and on each other. These are L_1, L_2, L_3 and L_4 in the picture. One typical question can be, e.g., whether a given update through L_4 affects the L_1 view or not? What about the other way around?

Unfortunately, lens theory does not say anything about how to discover when an update get issued through some lens may effect

IFL '14, October 1–3, 2014, Boston, Massachusetts, US.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

the data seen through some other. In this paper we present a general extension to lenses as a solution for this general problem. In this extension, called *parametric lenses*, lenses are partially defunctionalized to extract a first-order parameter (the focus domain: ϕ, ψ) that groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the source domain is mapped to the view domain by the lens. Parametric lenses also return a predicate in the *put* direction. This predicate, called the *invalidation function*, encodes some semantic information associated with the actual focus domain, and enables the engine to decide which domains are affected by a change of data. It tells whether the particular update of focus $p \in \phi$ affects a *get* operation of a given focus $q \in \phi$. With other words, it says whether the value of a previous read of some focus *q* is still valid or not. It returns *true* to indicate that the given focus must be re-read to be up to date.

$$get_F \in \phi \times X \to Y_\phi$$
$$put_F \in \phi \times Y_\phi \times X \to X \times (\phi \times Bool)$$

In our example, the focus domain is a type which enables to specify the area of the map one wants to focus on; the invalidation function then would predicate whether two values of the focus domain, two areas, overlap or not.

Pure parametric lenses cannot be applied to some shared data directly, therefore they attached to the shared data through a *non-pure* abstract interface called *parametric view*. The parametric views are allowed to be composed using predefined combinators. Using these combinators, one is able to specify the focus domain as a *type-based query language* defined over one or more resources. With the query language, one can focus on a specific part of the underlying shared data during reading, writing, or it can be used for notification purposes.

We offer the following contributions in the paper:

1. We introduce parametric lenses as a general extension to lenses which enables to develop efficient notification libraries for them;
2. Parametric lenses are embedded into compositional parametric views which are defined over shared data;
3. The executable semantics, using Haskell [4], of the combinators and an underlying notification engine is provided and explained. The complete Haskell implementation along with the example developed in the paper can be found at https://wiki.clean.cs.ru.nl/File:Pview.zip;
4. Parametric lenses have been introduced in the iTask coastguard case study described above.

References

[1] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TPL*, 29(3), May 2007. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/1232420.1232424`.

[2] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915. . URL `http://doi.acm.org/10.1145/319566.319567`.

[3] B. Lijnse, J. Jansen, R. Nanne, and R. Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In D. Mendonca and J. Dugdale, editors, *Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management, ISCRAM '11*, Lisbon, Portugal, May 2011.

[4] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 013453333X.

Making a Century in HERMIT

Extended Abstract

Neil Sculthorpe
Computer Science Department
Swansea University
{N.A.Sculthorpe}@swansea.ac.uk

Andrew Farmer Andrew Gill
Information and Telecommunication Technology Center
The University of Kansas
{afarmer,andygill}@ittc.ku.edu

Abstract

A benefit of pure functional programming is that it encourages equational reasoning. However, the Haskell language currently lacks direct tool support for such reasoning. Consequently, reasoning about Haskell programs is either performed manually, or in another language that does provide tool support (e.g. Agda). HERMIT is a Haskell-specific toolkit designed to support equational reasoning and user-guided program transformation, and to do so as part of the GHC compilation pipeline. This extended abstract presents a detailed case study of HERMIT usage in practice: mechanising Bird's classic "Making a Century" pearl. We also use the mechanised pearl to introduce recent HERMIT developments for supporting for equational reasoning.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords HERMIT, Equational Reasoning, Optimisation

1. Introduction

Currently, most equational reasoning on Haskell programs is performed manually, using pen-and-paper or text editors, because of the lack of up-to-date tool support. While some equational-reasoning tools do exist for Haskell [14, 35], they either target Haskell 98 or some subset thereof, and have not attempted to keep pace with the (frequently advancing) GHC-extended version of Haskell that is widely used in practice. They also work at the syntactical level, without access to the results of the type inference performed by the Haskell compiler. This is unfortunate, as pen-and-paper reasoning is slow, error prone, and allows the reasoner to neglect details of the semantics. For example, a common mistake is to neglect to consider partial and infinite values, which are notoriously tricky [7]. This was recently demonstrated by Jeuring et al. [18], who showed that the standard implementations of the state monad do not satisfy the monad laws.

This work was performed while Neil Sculthorpe was at the University of Kansas.

An alternative approach is to transliterate a Haskell program into a language or proof assistant that does provide support for equational reasoning, such as Agda [22] or Coq [33]. The desired transformations and proofs can then be performed in that language, and the resultant program or property transliterated back into Haskell. However, the semantics of these languages differ from Haskell, sometimes in subtle ways, so the transformations and proofs used may not carry over to Haskell. Again, partial and infinite values are a particular concern.

To address this situation, we have implemented a GHC plugin called HERMIT [9, 10, 29]. HERMIT is a toolkit that supports interactive equational reasoning on Haskell programs, and the mechanical verification of proof scripts. HERMIT operates on GHC's internal core language, part-way through the compilation process. User proofs of program properties are checked, and user-specified transformations are applied to the program being compiled. By performing proof checking during compilation, HERMIT ensures not only that the proof is correct, but that it corresponds to the current implementation of the program, in the context of the language extensions currently being used.

The initial HERMIT implementation [9] only supported equational reasoning that was *transformational* in nature; that is, HERMIT allowed the user to apply a sequence of correctness-preserving transformations to the Haskell program, resulting in an equivalent but (hopefully) more efficient program. This was sufficient to allow some specific instances of known program transformations to be mechanised [29], as well as for encoding prototypes of new optimisation techniques [1, 11]. However, some of the transformation steps used were only valid in certain contexts, and HERMIT had no facility for checking the necessary preconditions. Thus these preconditions had to verified by hand. Furthermore, it was not possible to state and prove auxiliary lemmas, or to use inductive proof techniques. This extended abstract describes the addition of these facilities to HERMIT, and discusses our experiences of using them on a case study. Specifically, the two main contributions of this extended abstract are:

- We describe the new equational reasoning infrastructure provided by HERMIT, discussing the challanges that arose during implementation, and the design choices we made for providing equational-reasoning support to Haskell programmers. (Section 2).
- Using our new infrastructure, we present a case study of equational reasoning in HERMIT, by mechanising a chapter from *Pearls of Functional Algorithm Design* [2], a recent textbook about deriving Haskell programs by calculation (Section 3).

2. Equational Reasoning using HERMIT

HERMIT is a GHC plugin that allows a user to apply custom transformations to a Haskell program amid GHC's optimisation passes. HERMIT operates on the program after it has been translated into GHC Core, GHC's internal intermediate language. GHC Core is an implementation of System $F_C^{\uparrow}$, which is System F [16, 25] extended with let-binding, constructors, and first-class type equalities [32]. Type checking is performed during the translation, and GHC Core retains the typing information as annotations.

HERMIT provides commands for navigating a GHC Core abstract syntax tree, applying transformations, version control, selecting different pretty printers, and invoking GHC analyses and optimisation passes. To direct and combine transformations, HERMIT uses the strategic programming language KURE [30] to provide a family of rewriting combinators. HERMIT offers three main interfaces:

- An interactive read-eval-print loop (REPL). This allows a user to view and explore the program code, as well as to experiment with transformations.
- HERMIT scripts. These are sequences of REPL commands, which can either be loaded and run from within the REPL, or automatically applied by GHC during compilation.
- A domain-specific language for transformation [30], embedded in Haskell. This allows the user to construct a custom GHC plugin using all of HERMIT's capabilities. The user can run transformations in different stages of GHC's optimisation pipeline, and add custom transformations to the REPL. New transformations can be encoded by defining Haskell functions directly on the Haskell data type representing the GHC Core abstract syntax, rather than using the more limited (but safer), monomorphically typed combinator language available to the REPL and scripts.

This extended abstract describes HERMIT's new equational reasoning infrastructure, but will not otherwise discuss its implementation or existing commands. Interested readers should consult the previous HERMIT publications [9, 29], or try out the HERMIT toolkit [10] for themselves.

2.1 Stating and Proving Lemmas

As discussed in Section 1, the HERMIT toolkit initially only supported program transformation, and any equational reasoning had to be structured as a sequence of transformation steps applied to the original source program [e.g. 29]. This was limiting, as it is often necessary to state and prove auxiliary lemmas, which can then be used to validate the transformation steps.

To address this, we have added support for auxiliary lemmas in HERMIT. As encoding a complete logic in HERMIT is a substantial task, we have begun with the simplest form of lemma that allows us to perform some interesting equational reasoning. A HERMIT lemma is (currently) an equality between two GHC Core expressions, which may contain universally quantified variables. For example:

Map Fusion
$\forall\, f\ g\,.\ map\ f \circ map\ g \equiv map\ (f \circ g)$

HERMIT maintains a set of lemmas, and records which have been proven and which have not. Proven lemmas can be applied as transformations (left-to-right or right-to-left), or used to validate transformation steps that have preconditions. A user can prove a lemma by providing a sequence of transformations on either (or both) sides of the lemma. HERMIT then checks the proof by comparing both sides of the transformed lemma using alpha equality. This proof can either be performed interactively, or loaded from a script.

Currently, we generate lemmas by exploiting GHC rewrite-rule pragmas [23]. For example, the Map Fusion lemma above can be expressed using the following RULES pragma:

```
{-# RULES "map-fusion" [~]
      ∀ f g. map f ∘ map g = map (f ∘ g)
 #-}
```

HERMIT previously allowed any rewrite rule in scope to be utilised directly as a unidirectional HERMIT transformation [9]. Any such rule can now also be converted into a HERMIT lemma, and thence proved. A side-benefit of this is that a user can experiment with applying GHC rewrite rules backwards, which was not possible previously. Note that there are some restrictions on the form of the left-hand-side of a GHC rewrite rule [23, Section 2.2], so this approach can only generate a subset of all possible lemmas.

Rewrite rules that are not intended to be used by GHC's optimiser can be annotated with the notation `[~]`, as we did for map-fusion above. This causes GHC to consider the rule as always inactive, and never attempt to use it for optimisation [12, Section 7.21.1]. In the long term, we aim to add a specific HERMIT pragma to GHC, allowing HERMIT lemmas to be stated in the source file yet be clearly distinguished from any rewrite rules. We could then be more liberal with the lemmas that can be stated than the limitations of GHC rewrite rules.

2.2 Structural Induction

Haskell programs usually contain recursive functions defined over (co)inductive data types. Proving even simple properties of such programs often requires the use of an induction principle. For example, consider this standard definition of list concatenation:

$$\begin{array}{l}(+\!\!+) :: [a] \to [a] \to [a] \\ [\,] \quad\ +\!\!+ ys = ys \\ (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)\end{array}$$

While $[\,] +\!\!+ xs \equiv xs$ can be proved simply by unfolding the definition of $+\!\!+$, proving the similar property $xs +\!\!+ [\,] \equiv xs$ requires reasoning inductively about the structure of xs.

Inductive reasoning cannot be expressed as a sequence of transformation steps: both the source and target expression must be known in advance, and the validity of rewriting one to the other is established by verifying the inductive and base cases. There are several induction principles that are relevant to Haskell programs. Thus far, we have encoded only one such principle in HERMIT: structural induction. This is implemented as a built-in proof technique that can be used to prove a lemma. Structural induction has been sufficient to prove all of the lemmas in our cases studies, but we anticipate that we will need to add other forms of induction when attempting more complex examples. The remainder of this section will formalise the structural-induction inference rule that HERMIT provides.

We first introduce some notation. Given $a_1, a_2 :: A$ for any type A, then let $a_1 \equiv a_2$ denote that a_1 and a_2 are semantically equivalent. We write $\overrightarrow{vs}$ to denote a sequence of variables, and $\forall (C\ \overrightarrow{vs} :: A)$ to quantify over all constructors C of the data type A, fully applied to a sequence $\overrightarrow{vs}$ of length matching the arity of C. Let $\mathbb{C} : A \rightsquigarrow B$ denote that $\mathbb{C}$ is an expression context containing one or more holes of type A, and having an overall type B. For any expression $a :: A$, then $\mathbb{C}[\![a]\!]$ denotes the context $\mathbb{C}$ with all holes filled with the expression a.

The structural-induction inference rule provided by HERMIT is defined in Figure 1. The conclusion of the rule is called the *induction hypothesis*. Informally, the premises require that:

- the induction hypothesis holds for undefined values;

Given contexts $\mathbb{C}, \mathbb{D} : A \rightsquigarrow B$, for any types A and B, then structural-induction provides the following inference rule:

$$\frac{\mathbb{C}[\![\bot]\!] \equiv \mathbb{D}[\![\bot]\!] \qquad \forall(C\ \overrightarrow{vs} :: A)\,.\,(\forall(v \in \overrightarrow{vs}, v :: A)\,.\,\mathbb{C}[\![v]\!] \equiv \mathbb{D}[\![v]\!]) \Rightarrow (\mathbb{C}[\![C\ \overrightarrow{vs}]\!] \equiv \mathbb{D}[\![C\ \overrightarrow{vs}]\!])}{\forall(a :: A)\,.\,\mathbb{C}[\![a]\!] \equiv \mathbb{D}[\![a]\!]}\ \text{STRUCTURAL INDUCTION}$$

Figure 1: Structural induction.

Given contexts $\mathbb{C}, \mathbb{D} : [A] \rightsquigarrow B$, for any types A and B, then:

$$\frac{\mathbb{C}[\![\bot]\!] \equiv \mathbb{D}[\![\bot]\!] \qquad \mathbb{C}[\![[\,]]\!] \equiv \mathbb{D}[\![[\,]]\!] \qquad \forall(a :: A, as :: [A])\,.\,(\mathbb{C}[\![as]\!] \equiv \mathbb{D}[\![as]\!]) \Rightarrow (\mathbb{C}[\![a : as]\!] \equiv \mathbb{D}[\![a : as]\!])}{\forall(xs :: [A])\,.\,\mathbb{C}[\![xs]\!] \equiv \mathbb{D}[\![xs]\!]}\ \text{LIST INDUCTION}$$

Figure 2: Structural induction on lists.

- the induction hypothesis holds for any fully applied constructor, given that it holds for any argument of that constructor (of matching type).

As a concrete example, specialising structural induction to the list data type gives the inference rule in Figure 2.

This form of structural induction is somewhat limited in that it only allows the induction hypothesis to be applied to a variable one constructor deep. While this is sufficient for the case study we describe in this extended abstract, it does not allow inductive proofs over recursive types where the recursion is deeper in the data type. The following type of *rose trees* is one such type, having a recursive call that is two constructors deep:

$$\mathbf{data}\ \mathit{RoseTree}\ a = \mathit{Node}\ a\ [\mathit{RoseTree}\ a]$$

As future work we need to generalise HERMIT's structural induction principle to n constructors deep.

3. Case Study: Making a Century

To assess how well HERMIT supports general-purpose equational reasoning, we decided to mechanise some existing textbook reasoning as a case study. We selected the chapter *Making a Century* from the textbook *Pearls of Functional Algorithm Design* [2, Chapter 6]. The book is entirely dedicated to reasoning about Haskell programs, with each chapter calculating an efficient program from an inefficient specification program. Additionally, many of the transformation steps used have preconditions, and thus there are several proof obligations along the way.

The program in *Making a Century* computes the list of all arithmetic expressions formed from ascending digits, where juxtaposition, addition, and multiplication evaluate to 100. For example, one possible solution is

$$100 = 12 + 34 + 5 \times 6 + 7 + 8 + 9$$

The details of the program are not overly important to the case study, and we refer the interested reader to the textbook for details [2, Chapter 6]. What is important, is that the derivation of an efficient program involves a substantial amount of equational reasoning, and the use of a variety of proof techniques, including fold/unfold transformation [4], structural induction (Section 2.2), fold fusion [21], and numerous auxiliary lemmas.

We will not present the entire case study here. Instead, we will give a representative extract, and then discuss the aspects of the mechanisation that proved challenging. The complete case study is available on the authors' web pages.

3.1 HERMIT Scripts

Our approach to mechanisation was to first state any auxiliary lemmas as (inactive) rewrite rules in the Haskell source file (as discussed in Section 2.1). To verify these lemmas, we first worked in HERMIT's interactive mode until the proof was successful, and then saved the final proof to a script that could be invoked thereafter. Finally, we developed the main transformation interactively, invoking these auxiliary proof scripts as necessary. For this case study the proofs were simply transliterations of the proofs in the textbook, but we expect developing new proofs would proceed in a similar manner, but with more experimentation and backtracking during the interactive phases.

As an example, we present the proof of Lemma 6.8, comparing the textbook proof with the HERMIT script. Figure 3a presents the proof extracted verbatim from the textbook [2, Page 36], and Figure 3b presents the corresponding HERMIT script. Note that lines beginning "--" in a HERMIT script are comments, and for readability we have typeset them differently to the (monospace) HERMIT code. These comments represent the current expression between transformation steps, and correspond to the output of the HERMIT REPL when performing the proof interactively. We manually added these comments to the HERMIT proof scripts to help readability and maintenance of the scripts.

When translating the textbook proof into HERMIT, we decided to split the middle step into two steps, as we felt that made the proof easier to read, but this is purely stylistic. Otherwise, the main difference between the two calculations is that in HERMIT we must specify where, and in which direction, to apply a lemma, whereas in the textbook the lemma is merely named, relying on the reader to be able to deduce how it was applied. Here, `one-td` (once, traversing top-down) and `any-td` (anywhere, traversing top-down) are *strategy combinators* from KURE [30], the strategic programming language that underlies HERMIT.

In this proof, and most others in the case study, we think that the HERMIT scripts are as clear, and not much more verbose, than the textbook calculations. There is one notable exception though, which involves manipulating terms containing adjacent occurrences of the function composition operator.

3.2 Associative Operators

On paper, associative binary operators such as function composition are typically written without parentheses. However, in HERMIT, a term is represented as an abstract syntax tree, with no special representation for associative operators. Terms that are equivalent semantically because of associativity properties can thus be represented by different trees. Consequently, it is sometimes necessary to perform a tedious restructuring of the abstract syntax tree before a transformation can match the term.

One way to avoid this is to work with eta-expanded terms and unfold all occurrences of function composition, as this always produces an abstract syntax tree consisting of a left-nested sequence of

$unzip \cdot map\ (fork\ (f, g))$

$=$ {definition of $unzip$ }

$fork\ (map\ fst, map\ snd) \cdot map\ (fork\ (f, g))$

$=$ {(6.6) and $map\ (f \cdot g) = map\ f \cdot map\ g$ }

$fork\ (map\ (fst \cdot fork\ (f, g)), map\ (snd \cdot fork\ (f, g)))$

$=$ {(6.5) }

$fork\ (map\ f, map\ g)$

(a) Textbook extract.

```
-- unzip · map (fork (f, g))
   one-td (unfold 'unzip)
-- fork (map fst, map snd) · map (fork (f, g))
   forward (lemma "6.6")
-- fork (map fst · map (fork (f, g)), map snd · map (fork (f, g)))
   any-td (forward (lemma "map-fusion"))
-- fork (map (fst · fork (f, g)), map (snd · fork (f, g)))
   one-td (forward (lemma "6.5a"))
   one-td (forward (lemma "6.5b"))
-- fork (map f, map g)
```

(b) HERMIT script.

Figure 3: Comparison of HERMIT script with textbook calculations for Lemma 6.8 ($fork\ (map\ f, map\ g) \equiv unzip \circ map\ (fork\ (f, g))$).

applications. However, we did not do so for this case study, as the textbook proofs are written in a point-free style, and we wanted to match those proofs as closely as possible.

More generally, rewriting terms containing associative (and commutative) operators is a well-studied problem [e.g. 3, 8, 19], and it remains as future work to provide better support for manipulating such operators in HERMIT.

3.3 Proofs Omitted in the Textbook

During mechanisation we discovered that several auxiliary properties in the textbook (Lemmas 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.10, and several minor unnamed properties) are stated as assumptions without proof. The lack of proofs is not commented on in the textbook, but we suspect that they are deemed either "obvious" or "uninteresting", as is common practice with pen-and-paper proofs. While performing reasoning beyond that presented in the textbook was not intended to be part of the case study, we decided to investigate how easy it is to prove these auxiliary properties.

In most cases, these properties had fairly straightforward inductive proofs, which were easy to encode in HERMIT. This mostly consisted of inlining definitions and then simplifying the resultant expressions as much as possible. Systematic proofs such as these are ripe for mechanisation, and HERMIT provides several strategies that perform a suite of basic simplifications to help with this. Consequently, the proof scripts were short and concise.

Assumption 6.2 also had a simple proof, but it relied on arithmetic properties of Haskell's built-in Int type (specifically, that $m \equiv n \Rightarrow m \leqslant n$). HERMIT does not yet provide any support for reasoning about built-in types, so we were not able to encode this proof. This is a clear deficiency of HERMIT, and adding such support is important future work. We found that assumptions 6.3 and 6.4 were non-trivial properties, without (to us) obvious proofs.

Additionally, the simplification of the definition of $expand$ is stated in the textbook without presenting the transformation steps [2, Page 40]. This simplification is non-trivial, and involves changing the type of an auxiliary function, and we did not find an easy way to encode this in HERMIT.

3.4 Proof Techniques Unsupported by HERMIT

Two proof techniques are used in the textbook that HERMIT does not directly support. The first is the *fold fusion* law [21]. Specialised to lists, fold fusion gives the following inference rule:

$$\frac{f\ \bot \equiv \bot \qquad f\ a \equiv b \qquad \forall x, y.\ f\ (g\ x\ y) \equiv h\ x\ (f\ y)}{f \circ foldr\ g\ a \equiv foldr\ h\ b}$$

This cannot be expressed as a HERMIT lemma, as HERMIT (currently) only supports equality lemmas, not implications. We therefore encoded *foldr-fusion* as a new primitive transformation using HERMIT's transformation DSL. We were able to reuse a substantial amount of existing HERMIT infrastructure to encode this rule, and so the encoding of the rule was only 20 lines of Haskell code. The plugin as a whole took another 30 lines of Haskell code, but that involved reusable auxiliary functions and plugin infrastructure that would be shared with any other user-added transformations. While this approach is only recommended for experienced HERMIT users, we think this is a viable approach for encoding custom transformations in HERMIT.

The second (unsupported) proof technique that the textbook uses is to postulate the existence of an auxiliary function ($expand$), use that function in the foldr-fusion rule, and then calculate a definition for that function starting from the foldr-fusion pre-conditions. This style of reasoning is not supported by HERMIT, nor is there an easy way to encode it. However, we were able to *verify* the calculation by working in reverse: starting from the definition in the textbook, we proceeded to prove the foldr-fusion pre-condition and thus validate the use of fold-fusion.

3.5 Calculation Sizes

As demonstrated by Figure 3, the HERMIT proof scripts are roughly the same size as the textbook calculations. It is difficult to give a precise comparison, as the textbook uses both formal calculation and natural language. We present some statistics in Table 1, but we don't recommend extrapolating anything from them beyond a rough approximation of the scale of the proofs. We give the size of the two main calculations (transforming $solutions$ and deriving $expand$), as well as the named auxiliary lemmas. In the textbook we measure lines of natural language reasoning as well lines of formal calculation, but not definitions, statement of lemmas, or surrounding discussion. In the HERMIT scripts, we measure the number of transformations applied, and the number of navigation and strategy combinators used to direct the transformations to the desired location in the term. We do not measure HERMIT commands for stating lemmas, loading files, switching between transformation and proof mode, or similar, as we consider these comparable to the surrounding discussion in the textbook. To get a feel for the scale of the numbers given, we recommend that the user compares Lemma 6.8 in Table 1 to the calculation in Figure 3.

Calculation	Textbook Lines	HERMIT Commands		
		Transformation	Navigation	Total
$solutions$	16	12	7	19
$expand$	19	18	20	38
Lemma 6.5	not given	4	4	8
Lemma 6.6	not given	2	1	3
Lemma 6.7	not given	2	0	2
Lemma 6.8	7	5	8	13
Lemma 6.9	1	4	4	8
Lemma 6.10	not given	23	13	36
Total	43	70	57	127

Table 1: Comparison of calculation sizes.

3.6 Summary

Our overall experience was that mechanising the textbook calculations was fairly straightforward, and it was pleasing that we could translate most steps of the textbook reasoning into an equivalent HERMIT command. The only annoyance was the need to manually apply associativity occasionally (see Section 3.2), so that the structure of the term would match the transformation we were applying.

While having to specify where in a term each lemma must be applied does result in more complicated proof scripts than in the textbook, we don't actually consider that to be more work. Rather, we view a pen-and-paper proof that doesn't specify the location as passing on the work to the reader, who must determine for herself where, and in which direction, the lemma is intended to be applied. Furthermore, when desired, strategic combinators such as `any-td` (apply the lemma anywhere it matches) can be used to avoid specifying precisely which sub-term the lemma should be applied to.

Encoding the foldr-fusion rule (Section 3.4) was a non-trivial amount of work, but once encoded, it was a reusable transformation. Furthermore, in the future we plan to extend HERMIT's representation of lemmas with logical connectives. This would allow rules such as foldr-fusion to be represented as lemmas rather than as primitive transformations, which would greatly simplify encoding them in HERMIT.

During the case study we also discovered one error in the textbook. Specifically, the inferred type of the $modify$ function [2, Page 39] does not match its usage in the program. We believe that its definition should include a $concatMap$, which would correct the type mismatch and give the program its intended semantics, so we have modified the function accordingly in our source code. However, we cannot claim this as detecting an error in a pen-and-paper proof, as this was caught by GHC's type checker, not by HERMIT.

4. Related Work

Equational reasoning is used both to prove properties of Haskell programs and to validate the correctness of program transformations. Most equational reasoning about Haskell programs is performed manually with pen-and-paper or text editors, of which there are numerous examples in the literature [e.g. 2, 7, 13, 15]. Prior to HERMIT there have been several tools for mechanical equational reasoning on Haskell programs, including the Programming Assistant for Transforming Haskell (PATH) [35], the Ulm Transformation System (Ultra) [17], and the Haskell Equational Reasoning Assistant (HERA) [14]. However, to our knowledge, none of these tools is currently being maintained. Furthermore, these tools all operate on Haskell source code (or some variant thereof), and do not attempt to support GHC-extended Haskell.

Another similar tool is the Haskell Refactorer (HaRe) [20, 34], which supports user-guided refactoring of Haskell programs. However, the objective of HaRe is slightly different, as refactoring is concerned with program transformation, whereas HERMIT supports both transformation and proof. The original version of HaRe targets Haskell 98 source code, but recently work has begun on a re-implementation of HaRe that targets GHC-extended Haskell.

Other than equational reasoning, there have been two main approaches taken to verifying properties of Haskell programs: testing and automated theorem proving. The most prominent testing tool is QuickCheck [5], which automatically generates large quantities of test cases in an attempt to find a counterexample. Other testing tools include SmallCheck [27], which exhaustively generates test values of increasing size so that it can find minimal counter examples, and Lazy SmallCheck [24, 27], which also tests partial values. Jeuring et al. [18] have recently developed infrastructure to support using QuickCheck to test type class laws, as well as to test the individual steps of user-provided equational-reasoning proofs of those laws. Of course testing does not constitute a proof, but it is lightweight and effective at finding counter-examples for false properties.

There are several tools that attempt to automatically prove properties of Haskell programs, by interfacing with an automated theorem prover and passing it (a translation of) the Haskell program and the desired properties. These include Liquid Haskell [36], Zeno [31] and the Haskell Inductive Prover (Hip) [26]. Properties in Liquid Haskell are *refinement types*, which the user may add as type annotations in the source file. Like HERMIT, Liquid Haskell and Zeno operate on GHC Core, whereas Hip translates Haskell source code directly into first-order logic. These tools can all support inductive proofs, but a limitation of Hip is that it only attempts to apply induction to user-specified conjectures, not to any intermediate lemmas that may be needed to complete the proof. HipSpec [6] is a tool built on Hip that addresses this limitation by exhaustively generating conjectures (up to a fixed term size) about the involved functions. These conjectures are first passed to Hip to prove, and any successes are then made available when attempting the main proof. Thus the user need not state the exact properties to which induction needs to be applied.

5. Future Work and Conclusions

While HERMIT has been used to successfully prototype GHC optimisations [1, 11], it is still very much an experimental tool, and development is ongoing. The next step is to add different modes to HERMIT that will limit the commands that are available. This will include a "read-only" mode, a "safe" mode, and a "super-user" mode. The read-only mode will allow navigation and alpha-renaming, but prohibit any other modifications to the code. The safe mode will only allow transformations that are known to be semantics preserving, and thus any rewrite rules or unproven lemmas will be prohibited. The super-user mode will correspond to the current capabilities of HERMIT, with no imposed limitations. We also anticipate the need for additional modes, perhaps with more fine-grained notions of safety. For example, a transformation that could potentially transform a terminating program into a diverging program should not be available in the safe mode, but a converse transformation that may introduce termination might be acceptable.

Thus far, structural induction (Section 2.2) is HERMIT's only proof technique for reasoning directly about recursive definitions, and it is limited to induction hypotheses that are one constructor deep. Future work includes generalising this to n constructors deep, and adding support for *corecursive* proof techniques [13].

We have successfully encoded some high-level transformation techniques as primitive HERMIT transformations with preconditions. The foldr-fusion rule in Section 3.4 is one example of this, and the worker/wrapper transformation [15, 28] is another. In a

prior publication [29] we described encoding worker/wrapper in HERMIT, but at the time HERMIT had no means of verifying the preconditions, so they were not mechanically enforced. Using HERMIT's new equational reasoning infrastructure described in this extended abstract, we have updated the worker/wrapper encoding such that it checks a proof that the preconditions hold before performing the transformation. All of the preconditions for the examples in that previous publication have now been verified by HERMIT, and the proofs are bundled with the HERMIT package [10]. However, encoding primitive transformations in HERMIT is a non-trivial task for the user, so our long-term goal is to build up a library of such high-level transformations, to complement HERMIT's existing library of low-level transformations.

HERMIT continues to prove useful for developing compile-time program transformation and reasoning capabilities that can be used on real Haskell programs. By mechanising proofs during compilation, HERMIT enforces the connection between the source, proof, and compiled program. GHC plugins developed using HERMIT can be deployed with Haskell's Cabal packaging system, meaning they integrate with a developer's normal work-flow. HERMIT development is on-going, and we seek to target ever-larger examples.

Acknowledgments

We would like the thank the Haskell Symposium reviewers for their useful comments on an earlier version of this work. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

References

[1] M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB is easy! In *Workshop on Partial Evaluation and Program Manipulation*, pages 71–82. ACM, 2014.

[2] R. Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

[3] T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In *International Conference on Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2011.

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[5] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[6] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec: Automating inductive proofs of program properties. In *Workshop on Automated Theory eXploration 2012*, volume 17 of *Proceedings in Computing*, pages 16–25. EasyChair, 2013.

[7] N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In *International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer, 2004.

[8] N. Dershowitz, J. Hsiang, N. A. Josephson, and D. A. Plaisted. Associative-commutative rewriting. In *International Joint Conference on Artificial Intelligence*, volume 2, pages 940–944. Morgan Kaufmann, 1983.

[9] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Haskell Symposium*, pages 1–12. ACM, 2012.

[10] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. `http://hackage.haskell.org/package/hermit`, 2014.

[11] A. Farmer, C. Höner zu Siederdissen, and A. Gill. The HERMIT in the stream: Fusing Stream Fusion's concatMap. In *Workshop on Partial Evaluation and Program Manipulation*, pages 97–108. ACM, 2014.

[12] GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.2*, 2014. URL `http://www.haskell.org/ghc/docs/7.8.2/`.

[13] J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, 2005.

[14] A. Gill. Introducing the Haskell equational reasoning assistant. In *Haskell Workshop*, pages 108–109. ACM, 2006.

[15] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

[16] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris Diderot, 1972.

[17] W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *Journal of Universal Computer Science*, 9(2):173–188, 2003.

[18] J. Jeuring, P. Jansson, and C. Amaral. Testing type class laws. In *Haskell Symposium*, pages 49–60. ACM, 2012.

[19] H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[20] H. Li, S. Thompson, and C. Reinke. The Haskell refactorer, HaRe, and its API. In *Workshop on Language Descriptions, Tools, and Applications*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 29–34. Elsevier, 2005.

[21] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[22] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579, 2009.

[23] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233. ACM, 2001.

[24] J. S. Reich, M. Naylor, and C. Runciman. Advances in lazy smallcheck. In *24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2013.

[25] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[26] D. Rosén. Proving equational Haskell properties using automated theorem provers. Master's thesis, University of Gothenburg, 2012.

[27] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values. In *Haskell Symposium*, pages 37–48. ACM, 2008.

[28] N. Sculthorpe and G. Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113–127, 2014.

[29] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2013.

[30] N. Sculthorpe, N. Frisby, and A. Gill. The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. *Journal of Functional Programming*, 24(4): 434–473, 2014.

[31] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2012.

[32] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *3rd Workshop on*

Types in Language Design and Implementation, pages 53–66. ACM, 2007.

[33] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi. Program calculation in Coq. In *Algebraic Methodology and Software Technology*, volume 6486 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2011.

[34] S. Thompson and H. Li. Refactoring tools for functional languages. *Journal of Functional Programming*, 23(3):293–350, 2013.

[35] M. Tullsen. *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, 2002.

[36] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013.

Dynamic resource adaptation for coordinating runtime systems

Extended Abstract

Stuart Gordon
Heriot-Watt University
sg315@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
S.Scholz@hw.ac.uk

Abstract

In this paper we propose a new approach towards negotiating resource distributions between several parallel applications running on a single multi-core machine. Typically, this negotiation process is delegated to the operating system or a common managed runtime layer. Alternatively, we have formulated a modest extension for arbitrary runtime systems that enables dynamic resource adaptations to be triggered from the outside, i.e., through a separate coordination application.

We demonstrate the effectiveness of the approach in the context of SaC. The paper delineates the required extensions of SaC's runtime system and it discusses how the functional setting substantially eases the process. Furthermore, we demonstrate the effectiveness of our approach when it comes to maximising the overall performance of several parallel applications that share a single multi-core system.

1. Introduction & motivation

As multi-core architectures have now become the norm, an application must not only be able to perform well, it must also have the ability to scale well over parallel architectures and operate in harmony alongside other parallel applications. One might hope that such a negotiation of shared resources can be delegated to the operating system as this traditionally happens when several single-threaded applications share a machine. Although in principle this is possible, the joint performance of several parallel applications on a shared multi-core system typically is rather unsatisfactory. This effect has various technical reasons; in the end these boil down to the fact that the operating system has little if any knowledge about the side conditions that exist in the individual parallel applications: As soon as the parallel applications jointly request more resources than available this over-subscription is resolved by the operating system without taking possible interdependencies into account. To make matters worse, different runtime systems typically have different interdependencies.

An ideal solution would be to express all parallelism as a high-level abstraction open to all languages. Attractive in principle, it has so far proven to be an elusive goal. This is evident by the rapid onset of parallel language implementations and programming models, with no consensus as to which is best. Expressing parallelism may require specific domain, or application knowledge in order to be expressed in an optimal form. Alternatively, language implementations, such as Lithe (Pan et al. 2010), attempt to provide low-level abstractions, with an emphasize efficient parallelism implemented using a standard interface, to implement runtime systems. But however affective this maybe, it requires codes to be heavily modified and reimplemented. A technique also offered by Callisto (Harris et al. 2014), a resource management layer for parallel runtimes, that relies on heavy adaptation in order to be implemented and used as a basis for all runtime systems.

In this paper we propose a radically different approach. Instead of coordinating low-level threads bottom up we propose to coordinate them top-down. The whole approach builds on the idea that it suffices to enable runtime systems to dynamically adapt the number of resources used in order to avoid an over-subscription of resources. That way, the parallel applications can be executed almost entirely independently guaranteeing efficient performance of each individual application.

The contributions of this paper are as follows:

- We propose a generic programming interface to facilitate the negotiation of shared resources.
- A web based application that serves as the user interface.
- We provide a detailed analysis of:
  - SaC's extended runtime system, providing details of the implementation, outlining the relevant benefits of a functional setting.

- we demonstrate the effectiveness of our approach detailing the the overall performance variations of several parallel applications that share a single multi-core system.

Acknowledgments

This work was supported in part by grant EP/L00058X/1 from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 24:1–24:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. . URL `http://doi.acm.org/10.1145/2592798.2592807`.

H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 376–387, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. . URL `http://doi.acm.org/10.1145/1806596.1806639`.

Editing Functional Programs Without Breaking Them

Edward Amsden Ryan Newton Jeremy Siek
Indiana University
{eamsden,rrnewton,jsiek}@indiana.edu

Abstract

We present a set of editing actions on terms in the simply-typed lambda calculus. These actions preserve the well-typedness of terms, and allow the derivation of any well-typed term beginning with any other well-typed term, without resorting to metavariables or other forms of placeholders. We are in the process of proving these properties, and we discuss how general-purpose programming might proceed given this set of editing actions.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Modern typed functional programming languages such as Haskell [11], OCaml [7], SML [13], Idris [2], and Agda [14] offer programmers extremely powerful and flexible type systems to ensure the correctness of their code. However, type systems are useful for far more than checking programs as programmers have already written them. In particular, a type system can be used to guide a term editor such that ill-typed terms are never produced in the first place.

Text editing and submission of the program to a compiler or interpreter for error checking and evaluation is the venerable and proven means of creating and maintaining programs. However, this paradigm has several disadvantages.

- It does not directly integrate semantic knowledge of program components (such as scope and types) into the editing system, requiring local parsing or linking with the language implementation to derive this information from the program text.
- It requires the programmer to reason about the well-typedness of programs either manually, or by trial-and-error by submitting them to a typechecker.
- It is unsuitable for more restrictive human-computer interaction platforms, such as touch-based mobile platforms, game consoles, and accessible interfaces.
- Most importantly, text editing actions do not relate directly to meaningful operations on programs. Insertion or deletion of text will often make a program syntactically invalid. It is likely to introduce scoping errors. It will almost certainly make the program ill-typed.

Languages such as Scratch [10], Kodu [9], and YinYang [12] are first steps toward addressing these issues, but do not offer a clear path toward leveraging the vast body of accomplished and ongoing research in programming languages. We propose instead to take a well-understood language, the lambda calculus, and show how to edit its terms directly.

In particular, we intend to use types to guide the editing operations on terms. Rather than entering programs as text, programmers will have at their disposal a set of actions for modifying, combining, and uncombining complete and well-typed lambda calculus terms. These actions are sufficient to arrive at any well-typed term from any other well-typed term, guarantee the well-typedness of derived terms, and do not require placeholders such as metavariables [14].

For this presentation, we consider the simply-typed lambda calculus. We expect this approach to generalize to polymorphic and dependently-typed calculi. We point out particular and interesting ways in which the application of this approach to more powerful type systems will differ from that presented here.

We make the following contributions:

- We describe a set of actions which allow programmers to construct and deconstruct terms in the simply-typed lambda calculus (Sections 2 and 3).
  - These actions operate only on lambda calculus terms. There are no holes or other placeholders.
- We sketch a proof that this set of actions is *sound*. That is, beginning with well-typed terms, only well-typed terms may be derived using these actions (Section 4.1).
- We sketch a proof that this set of actions is *complete*. That is, any well-typed term may be constructed via these actions, starting with any other well-typed term (Section 4.2).
- We describe how these actions support various general approaches to programming, specifically "top-down" programming (where the programmer begins building the top-level program, binding components for later construction) and "bottom-up" programming (where the programmer begins constructing small components and composes them into the top-level program) (Section 5).

2. Terms, Types and Paths

For this presentation, we consider terms (and associated types) in the simply-typed lambda calculus. We assume that terms and types in the simply-typed lambda calculus are familiar to our readers. However, we include them in our presentation for reference while reading our formulations of paths and actions and our proofs of "soundness" and "completeness" for our editing system. We use De Bruijn indices for this presentation, as this avoids issues of

$$
\begin{array}{rcl}
\text{Variables}(x) & ::= & \mathbb{N} \\
\text{Types}(t) & ::= & \texttt{unit} \mid t \to t \\
\text{Terms}(e) & ::= & x \mid \lambda : t.e \mid e\ e \mid \bullet \\
\text{Paths}(p) & ::= & \texttt{top} \mid p\ \texttt{lamty} \mid p\ \texttt{lambod} \\
 & \mid & p\ \texttt{apprator} \mid p\ \texttt{apprand} \\
 & \mid & p\ \texttt{typein} \mid p\ \texttt{typeout} \\
\text{Environments}(\Gamma) & ::= & \epsilon \mid \Gamma; t
\end{array}
$$

Figure 1. Grammars for terms, types, and paths.

$\boxed{\Gamma \vdash e : t}$

$$\textsc{UnitTy}\quad \Gamma \vdash \bullet : \texttt{unit}$$

$$\textsc{VariableSuccTy}\quad \frac{\Gamma \vdash x : t}{\Gamma; t_{\texttt{ext}} \vdash x + 1 : t}$$

$$\textsc{VariableZeroTy}\quad \Gamma; t \vdash 0 : t$$

$$\textsc{LamTy}\quad \frac{\Gamma; t_{\texttt{in}} \vdash e : t_{\texttt{out}}}{\Gamma \vdash \lambda : t_{\texttt{in}}.e : t_{\texttt{in}} \to t_{\texttt{out}}}$$

$$\textsc{AppTy}\quad \frac{\Gamma \vdash e_{\texttt{rator}} : t_{\texttt{in}} \to t_{\texttt{out}} \qquad \Gamma \vdash e_{\texttt{rand}} : t_{\texttt{in}}}{\Gamma \vdash e_{\texttt{rator}}\ e_{\texttt{rand}} : t_{\texttt{out}}}$$

Figure 2. Typing rules for the Simply-Typed Lambda Calculus

naming in the consideration of actions on terms. The typing rules are given in Figure 2 as we refer to them for proofs of soundness and completeness of the editing actions later in the paper.

In order to designate which part, or subterm, of a term we wish to operate on, we define a notion of paths into terms. *Paths* are simply sequences which recursively describe which subterm of a particular term to pick out. The grammars for terms, types, and paths appear in Figure 1.

2.1 Variables

We require a few operations on variables and variables in terms in support of the definitions of our editing actions. In particular, we will need to compare variables to see if the scope of one variable is within the scope another. We will also need to adjust variables in order to maintain their binding structure as we add and remove bindings. This is done with the $\uparrow_c\ (e)$ and $\downarrow_c\ (e)$ operations. These operations are given in Figure 3. The presentation is due to Pierce [16, p. 79] with a few modifications.

The shift operation $\uparrow_c^d (e)$ given by Pierce is parameterized over the offset d as well as the cutoff c. We will only want to shift by an offset of 1, so d is fixed at 1 and we write $\uparrow_c (e)$, or simply $\uparrow (e)$ for the case where $c = 0$.

We introduce an unshift operation $\downarrow_c\ (e)$. This is a partial function, which is undefined exactly on variables matching the cutoff. Judgements with this function in their premises do not hold in cases where it is undefined. We write $\downarrow (e)$ for the case where $c = 1$. $\downarrow (e)$ is thus undefined on variables which would be bound to the nearest binder surrounding e. Since unshifting is used when replacing a binding which has no bound occurrences with the body of the binding, this is the expected behavior.

2.2 Paths

Paths are intended to mark the part of a term which is under consideration for a particular action. For the purpose of our presentation, we consider paths as separate entities from terms. Paths are described as sequences of atoms, each of which describes a choice of subterm. There are several relations on paths and terms employed

$$
\begin{array}{ll}
\uparrow_c (x) & = \begin{cases} x & x < c \\ x + 1 & x \geq c \end{cases} \\
\uparrow_c (\lambda : t.e) & = \lambda : t.\ \uparrow_{c+1} (e) \\
\uparrow_c (e_1\ e_2) & = \uparrow_c (e_1)\ \uparrow_c (e_2) \\
\uparrow (e) & = \uparrow_0 (e) \\
\downarrow_c (x) & = \begin{cases} x & x < c - 1 \\ x - 1 & x \geq c \end{cases} \\
\downarrow_c (\lambda : t.e) & = \lambda : t.\ \downarrow_{c+1} (e) \\
\downarrow_c (e_1\ e_2) & = \downarrow_c (e_1)\ \downarrow_c (e_2) \\
\downarrow (e) & = \downarrow_1 (e)
\end{array}
$$

Figure 3. Shifting ($\uparrow$) and unshifting ($\downarrow$) of De Bruijn indices. Adapted from the presentation by Pierce [16, p. 79].

in the definitions of the editing actions. These relations are defined in Figure 4.

The relation $p(e)$ extracts the subterm (expression or type) of e to which the path p points. Extraction of subterms is used to determine their type, as well to use them when constructing new terms by λ-abstraction or application.

The relation $\gamma(p, e)$ gives the typing environment at the subterm of e to which the path p points. This relation is used to determine what bindings are in scope at a particular point in support of the variable replacement operation. It is also used by the $\texttt{c}(p, e)$ operation when determining what types are valid at a path. Finally, it is employed in the definitions of editing actions to check that new terms do in fact meet their typing constraints.

The relation $\texttt{c}(p, e)$ gives the set of types which the subterm of e to which the path p points may match. This relation allows the definitions of editing actions to ensure that they do not make the term surrounding the subterm on which they operate ill-typed by changing the type of that subterm. For instance, if e_1 is applied to e_2, then λ-abstracting e_2 will yield a well-typed term derived from e_2, but the application will no longer be well-typed in the STLC.

The relation $e_{\texttt{full}}[e_{\texttt{sub}}/p]$ or $e_{\texttt{full}}[t/p]$ gives a new term in which $e_{\texttt{sub}}$ or t is substituted for the subterm of $e_{\texttt{full}}$ to which p points. This relation is used to define the operation of actions on subterms. In general, the $p(e)$ relation is used to extract a subterm, the subterm is suitably modified, and the modified term put back in its place by the $e_{\texttt{full}}[e_{\texttt{sub}}/p]$ relation.

The relation $\texttt{appable}(p)$ asserts that a path is suitable for constructing an application with. This is used to ensure that the application operation is not employed in subterms of applications. Were this allowed, it is not clear which of several possible outcomes of this operation would be the correct one. Further, disallowing this does not affect the soundness or completeness of the editing operations. However, allowing application under applications is almost certain to be a desirable feature in the implementation, so future work will describe a resolution of this ambiguity and lift the restriction on application operations under applications.

3. Editing Actions

The core of our contributions is a set of editing actions, which describe how to combine and manipulate lambda calculus terms in a way that maintains well-typedness. These actions are shown in Figure 5.

Not all actions are available at all paths into a subterm. Actions will usually change the type of a subterm, and may not do so in a way that would make the containing term ill-typed. This may seem an onerous restriction. However, we are able to show that any well-typed term may be reached from any other well-typed term using our restriction. Further, the mechanism of constraints which we use to judge whether a type-change will make the containing term ill-

$\boxed{p(e) = e_{\text{sub}}}$

TopPath
$\texttt{top}(e) = e$

LamTyPath
$$\frac{p(e) = \lambda : t_{\text{ty}}.e_{\text{bod}}}{p\ \texttt{lamty}(e) = t_{\text{ty}}}$$

LamBodPath
$$\frac{p(e) = \lambda : t_{\text{ty}}.e_{\text{bod}}}{p\ \texttt{lambod}(e) = e_{\text{bod}}}$$

AppRatorPath
$$\frac{p(e) = e_{\text{rator}}\ e_{\text{rand}}}{p\ \texttt{apprator}(e) = e_{\text{rator}}}$$

AppRandPath
$$\frac{p(e) = e_{\text{rator}}\ e_{\text{rand}}}{p\ \texttt{apprand}(e) = e_{\text{rand}}}$$

TypeInPath
$$\frac{p(e) = t_{\text{in}} \to t_{\text{out}}}{p\ \texttt{typein}(e) = t_{\text{in}}}$$

TypeOutPath
$$\frac{p(e) = t_{\text{in}} \to t_{\text{out}}}{p\ \texttt{typeout}(e) = t_{\text{out}}}$$

$\boxed{\gamma(p, e) = \Gamma}$

TopPathCtx
$\gamma(\texttt{top}, e) = \epsilon$

LamBodPathCtx
$$\frac{\gamma(p, e) = \Gamma \qquad p(e) = \lambda : t.e_{\text{bod}}}{\gamma(p\ \texttt{lambod}, e) = \Gamma; t}$$

AppRatorPathCtx
$$\frac{\gamma(p, e) = \Gamma \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}}}{\gamma(p\ \texttt{apprator}, e) = \Gamma}$$

AppRandPathCtx
$$\frac{\gamma(p, e) = \Gamma \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}}}{\gamma(p\ \texttt{apprand}, e) = \Gamma}$$

$\boxed{\texttt{c}(p, e) = C}$

TopPathTypes
$\texttt{c}(\texttt{top}, e) = \mathcal{L}(t)$

LamTyPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad \gamma(p, e) = \Gamma p(e) = \lambda : t.e_{\text{bod}}}{\texttt{c}(p\ \texttt{lamty}, e) = \{t_{\text{in}} | \Gamma; t_{\text{in}} \vdash e_{\text{bod}} : t_{\text{out}} \wedge t_{\text{in}} \to t_{\text{out}} \in C\}}$$

LamBodPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = \lambda : t_{\text{in}}.e_{\text{bod}}}{\texttt{c}(p\ \texttt{lambod}, e) = \{t_{\text{out}} | t_{\text{in}} \to t_{\text{out}} \in C\}}$$

AppRatorPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}} \qquad \gamma(p, e) = \Gamma \qquad \Gamma \vdash e_{\text{rand}} : t_{\text{in}}}{\texttt{c}(p\ \texttt{apprator}, e) = \{t_{\text{in}} \to t_{\text{out}} | t_{\text{out}} \in C\}}$$

AppRandPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = e_{\text{rator}}\ e_{\text{rand}} \qquad \gamma(p, e) = \Gamma \qquad \Gamma \vdash e_{\text{rator}} : t_{\text{in}} \to t_{\text{out}}}{\texttt{c}(p\ \texttt{apprand}, e) = \{t_{\text{in}}\}}$$

TypeInPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = t_{\text{in}} \to t_{\text{out}}}{\texttt{c}(p\ \texttt{typein}, e) = \{t | t \to t_{\text{out}} \in C\}}$$

TypeOutPathTypes
$$\frac{\texttt{c}(p, e) = C \qquad p(e) = t_{\text{in}} \to t_{\text{out}}}{\texttt{c}(p\ \texttt{typeout}, e) = \{t | t_{\text{in}} \to t \in C\}}$$

$\boxed{e_{\texttt{full}}[e_{\text{sub}}/p] = e_{\text{new}}}$

TopPathSub
$e_{\texttt{full}}[e_{\text{sub}}/\texttt{top}] = e_{\text{sub}}$

LamTyPathSub
$$\frac{p(e_{\texttt{full}}) = \lambda : t.e \qquad e_{\texttt{full}}[\lambda : t_{\text{sub}}.e/p] = e_{\text{new}}}{e_{\texttt{full}}[t_{\text{sub}}/p\ \texttt{lamty}] = e_{\text{new}}}$$

LamBodPathSub
$$\frac{p(e_{\texttt{full}}) = \lambda : t.e \qquad e_{\texttt{full}}[\lambda : t.e_{\text{sub}}/p] = e_{\text{new}}}{e_{\texttt{full}}[e_{\text{sub}}/p\ \texttt{lambod}] = e_{\text{new}}}$$

AppRatorPathSub
$$\frac{p(e_{\texttt{full}}) = e_{\text{rator}}\ e_{\text{rand}} \qquad e_{\texttt{full}}[e_{\text{sub}}\ e_{\text{rand}}/p] = e_{\text{new}}}{e_{\texttt{full}}[e_{\text{sub}}/p\ \texttt{apprator}] = e_{\text{new}}}$$

AppRandPathSub
$$\frac{p(e_{\texttt{full}}) = e_{\text{rator}}\ e_{\text{rand}} \qquad e_{\texttt{full}}[e_{\text{rator}}\ e_{\text{sub}}/p] = e_{\text{new}}}{e_{\texttt{full}}[e_{\text{sub}}/p\ \texttt{apprand}] = e_{\text{new}}}$$

TypeInPathSub
$$\frac{p(e_{\texttt{full}}) = t_{\text{in}} \to t_{\text{out}} \qquad e_{\texttt{full}}[t_{\text{sub}} \to t_{\text{out}}/p] = e_{\text{new}}}{e_{\texttt{full}}[t_{\text{sub}}/p\ \texttt{typein}] = e_{\text{new}}}$$

TypeOutPathSub
$$\frac{p(e_{\texttt{full}}) = t_{\text{in}} \to t_{\text{out}} \qquad e_{\texttt{full}}[t_{\text{in}} \to t_{\text{sub}}/p] = e_{\text{new}}}{e_{\texttt{full}}[t_{\text{sub}}/p\ \texttt{typeout}] = e_{\text{new}}}$$

$\boxed{\texttt{appable}(p)}$

TopPathAppable
$\texttt{appable}(\texttt{top})$

LamBodAppable
$$\frac{\texttt{appable}(p)}{\texttt{appable}(p\ \texttt{lambod})}$$

Figure 4. Definitions of path relations. $\mathcal{L}(t)$ denotes the language of the nonterminal t.

Actions(a) ::=

Usage	Action	Denoted By
Construction	λ-abstract	λ
	$\rightarrow$-abstract	$\rightarrow$
	Replace $p(e)$ with x	$\texttt{replace}_x$
	Replace $p(e)$ with $\bullet$	$\texttt{replace}_\bullet$
	Apply	`apply`
Destruction	Delete binding	`unbind`
	Unapply	`unapply`
Movement	Type of a lambda	`lamty`
	Body of a lambda	`lambod`
	Operator of an app	`apprator`
	Operand of an app	`apprand`
	Input of a type arrow	`tyin`
	Output of a type arrow	`tyout`
	Up	`up`

Action Sequences(s) ::= $\epsilon \mid s\ a(p,e) \mid s\ a(p,e,e)$

Figure 5. Editing Actions

typed will also allow us to describe exactly how circumscribed the set of actions is for any term, and see how these boundaries will be extended when our editing theory is extended to polymorphic calculi.

The language of actions is given by the non-terminal a. Editor states $\mathcal{E}$ are subsets of $\mathcal{L}(p) \times \mathcal{L}(e)$[1]. To define an action, we give a rule for one of three relations: $a(p,e) \rightsquigarrow \mathcal{E}$, $a(p,e_1,e_2) \rightsquigarrow \mathcal{E}$, or $a(p_1,e_1) \overset{!}{\rightsquigarrow} (p_2,e_2)$. The non-terminal s describes sequences of actions. The relation $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$ defines how a sequence of actions takes one editor state to another, in terms of the relations on individual actions. The definitions of these relations are given in Figure 6.

The λ-abstract action wraps a λ binding around the subterm of e at path p. This does not affect which binders the variables of the subterm reference, since the action shifts the variable indices of variables which are free in the subterm. Thus, in the new λ-abstracted subterm, there are no variable occurrences bound by the new binding.

The $\rightarrow$-abstract action replaces the type t at path p by the type $\texttt{unit} \rightarrow t$. This is how types for bindings are built up.

The replacement action replaces the subterm at path p with a variable x which is in scope at that path, or with the unit term $\bullet$. This is how elements of base types and references to bindings are introduced after the bindings are introduced by λ-abstraction.

The apply action takes the subterms of e_1 and e_2 pointed to by the path p, and constructs their application under the same sequence of binders. It cannot be applied when p goes into an application, as discussed in the description of the $\texttt{appable}(p)$ relation in Section 2.

The delete binding action replaces a λ-bound subterm with the body of the binding, assuming that their are no occurences of the bound varialbe. The variable occurences in the body are adjusted to continue to point to the same bindings. This is how terms with λ bindings may be deconstructed.

The unapply action splits a term at an application site, producing a term with the operator substituted for the application, and another term with the operand substituted for the application. This is how terms with applications may be deconstructed into their component terms.

The movement operations are straightforward. Movement operations extend paths with the atoms corresponding to their names, in the case that the extended path is valid on the corresponding term. The exception is the up action, which removes the last atom from a path, corresponding to selecting the parent subterm of a subterm.

[1] $\mathcal{L}$ denotes the set of trees matched by a nonterminal.

4. Properties

We define the properties of "soundness" and "completeness" for editing semantics with respect to typing semantics, and prove that they hold for the set of actions described in Section 3. These concepts are analogous to soundness and completeness for type systems, with the crucial difference that a sound and complete editing system (with respect to a particular type system) is in fact possible.

4.1 Soundness

The soundness theorem (Theorem 1) states that if all of the terms input to an action are well-typed, then all terms in its output are well-typed as well.

In support of the statement of this theorem, we define a predicate which is true if and only if all terms in an editing state are well-typed:

Definition 1.

$$\texttt{welltyped}(\mathcal{E}) \equiv \forall (p,e) \in \mathcal{E}. \exists t \in \mathcal{L}(t). \vdash e : t$$

The formal statement of the soundness theorem is:

Theorem 1.

$$\texttt{welltyped}(\mathcal{E}_1) \wedge s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \Rightarrow \texttt{welltyped}(\mathcal{E}_2)$$

Proof. (See Section A.2) □

Informally, this theorem states that we do not "break" the well-typedness of programs. Together with the absence of holes in the terms, this theorem means that only complete and well-typed programs can occur in an editing derivation which started with a set of well-typed terms.

The proof strategy is induction over sequences of actions. In the base case (the empty sequence), no terms are added to or removed from the set, and so the preservation of well-typedness holds trivially. In the inductive step, we show that for each action, either the new terms are well-typed, or the single step relations $\rightsquigarrow$ and $\overset{!}{\rightsquigarrow}$ do not hold, and thus the action is impossible at that step. This is the case because the judgements for the $\rightsquigarrow$ relation restrict substituted terms to those whose types are in the set $c(p,e) = C$, and we can show that for any type in C, the context will typecheck given a term of that type.

4.2 Completeness

The completeness theorem (Theorem 2) says that any well-typed term can be reached from the unit term.

Theorem 2.

$$\vdash e : t \Rightarrow \exists s, \mathcal{E}. s(\{(\texttt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E} \wedge (\texttt{top}, e) \in \mathcal{E}$$

Proof. (See Section A.3) □

Informally, this theorem states that we do not give up the ability to derive any well-typed program. This property is of course important for a general software development tool, so it is encouraging to demonstrate that our editing system maintains it.

The proof strategy for the construction lemma (Lemma **??**) is to first prove, by induction on sizes of sets and induction over STLC terms without applications, that all terms in the unzipping of the term targeted for construction can be constructed. Informally, the unzipping is the set of all variables and constants from the term in

$\boxed{a(p,e) \rightsquigarrow \mathcal{E}}$

LAMABST

$$\frac{\uparrow(e_{\texttt{bod}}) = e_{\texttt{new}} \qquad \gamma(p,e) = \Gamma \qquad \Gamma; \texttt{unit} \vdash e_{\texttt{new}} : t \qquad \begin{array}{c} p(e) = e_{\texttt{bod}} \\ \texttt{c}(p,e) = C \end{array} \qquad \texttt{unit} \to t \in C \qquad e[\lambda : \texttt{unit}.e_{\texttt{new}}/p] = e_{\texttt{newnew}}}{\lambda(p,e) \rightsquigarrow \{(p, e_{\texttt{newnew}})\}}$$

ARRABST

$$\frac{p(e) = t \qquad \texttt{c}(p,e) = C \qquad \texttt{unit} \to t \in C \qquad e[\texttt{unit} \to t/p] = e_{\texttt{new}}}{\to (p,e) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

REPLACE

$$\frac{\texttt{c}(p,e) = C \qquad \gamma(p,e) = \Gamma \qquad \Gamma \vdash x : t \qquad t \in C \qquad e[x/p] = e_{\texttt{new}}}{\texttt{replace}_x(p,e) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

REPLACEUNIT

$$\frac{\texttt{c}(p,e) = C \qquad \texttt{unit} \in C \qquad e[\bullet/p] = e_{\texttt{new}}}{\texttt{replace}_\bullet(p,e) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

UNBIND

$$\frac{p(e) = \lambda : t.e_{\texttt{bod}} \qquad \downarrow(e_{\texttt{bod}}) = e_{\texttt{new}} \qquad \texttt{c}(p,e) = C \qquad \gamma(p,e) = \Gamma \qquad \Gamma \vdash e_{\texttt{new}} : t_{\texttt{new}} \qquad t_{\texttt{new}} \in C \qquad e[e_{\texttt{new}}/p] = e_{\texttt{newnew}}}{\texttt{unbind}(p,e) \rightsquigarrow \{(p, e_{\texttt{newnew}})\}}$$

UNAPPLY

$$\frac{p(e) = e_{\texttt{rator}}\ e_{\texttt{rand}} \qquad \Gamma \vdash e_{\texttt{rator}} : t_{\texttt{rator}} \qquad \Gamma \vdash e_{\texttt{rand}} : t_{\texttt{rand}} \qquad \begin{array}{c} \texttt{c}(p,e) = C \\ t_{\texttt{rator}} \in C \end{array} \qquad \begin{array}{c} \gamma(p,e) = \Gamma \\ t_{\texttt{rand}} \in C \end{array} \qquad e[e_{\texttt{rator}}/p] = e_1 \qquad e[e_{\texttt{rand}}/p] = e_2}{\texttt{unapply}(p,e) \rightsquigarrow \{(p,e_1),(p,e_2)\}}$$

$\boxed{a(p,e_1,e_2) \rightsquigarrow \mathcal{E}}$

APPLY

$$\frac{\begin{array}{c} p(e_1) = e_{\texttt{rator}} \qquad p(e_2) = e_{\texttt{rand}} \qquad \texttt{appable}(p) \qquad \texttt{c}(p,e_1) = C \\ \gamma(p,e_1) = \Gamma \qquad \gamma(p,e_2) = \Gamma \qquad \Gamma \vdash e_{\texttt{rator}} : t_{\texttt{in}} \to t_{\texttt{out}} \qquad \Gamma \vdash e_{\texttt{rand}} : t_{\texttt{in}} \qquad t_{\texttt{out}} \in C \end{array} \qquad e_1[e_{\texttt{rator}}\ e_{\texttt{rand}}/p] = e_{\texttt{new}}}{\texttt{app}(p,e_1,e_2) \rightsquigarrow \{(p, e_{\texttt{new}})\}}$$

$\boxed{a(p_1,e_1) \overset{!}{\rightsquigarrow} (p_2,e_2)}$

LAMTYMOVE

$$\frac{p(e) = \lambda t : e_{\texttt{bod}}}{\texttt{lamty}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{lamty}, e)}$$

LAMBODMOVE

$$\frac{p(e) = \lambda t : e_{\texttt{bod}}}{\texttt{lambod}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{lambod}, e)}$$

APPRATORMOVE

$$\frac{p(e) = e_{\texttt{rator}}\ e_{\texttt{rand}}}{\texttt{apprator}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{apprator}, e)}$$

APPRANDMOVE

$$\frac{p(e) = e_{\texttt{rator}}\ e_{\texttt{rand}}}{\texttt{apprand}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{apprand}, e)}$$

TYINMOVE

$$\frac{p(e) = t_{\texttt{in}} \to t_{\texttt{out}}}{\texttt{tyin}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{typein}, e)}$$

TYOUTMOVE

$$\frac{p(e) = t_{\texttt{in}} \to t_{\texttt{out}}}{\texttt{tyout}(p,e) \overset{!}{\rightsquigarrow} (p\ \texttt{typeout}, e)}$$

UPMOVE

$$\frac{q \in \{\texttt{lambod}, \texttt{lamty}, \texttt{apprator}, \texttt{apprand}, \texttt{tyin}, \texttt{tyout}\}}{\texttt{up}(p\ q, e) \overset{!}{\rightsquigarrow} (p,e)}$$

$\boxed{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2}$

ACTION

$$\frac{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \qquad (p,e) \in \mathcal{E}_2 \qquad a(p,e) \rightsquigarrow \mathcal{E}_3}{s\ a(p,e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \cup \mathcal{E}_3}$$

DOUBLEACTION

$$\frac{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \qquad (p,e_1) \in \mathcal{E}_2 \qquad (p,e_2) \in \mathcal{E}_2 \qquad a(p,e_1,e_2) \rightsquigarrow \mathcal{E}_3}{s\ a(p,e_1,e_2)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \cup \mathcal{E}_3}$$

ACTIONMUTATE

$$\frac{s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \qquad (p,e) \in \mathcal{E}_2 \qquad a(p,e) \overset{!}{\rightsquigarrow} (p_{\texttt{new}}, e_{\texttt{new}})}{s\ a(p,e)(\mathcal{E}) \rightsquigarrow (\mathcal{E}_2 - \{(p,e)\}) \cup \{(p_{\texttt{new}}, e_{\texttt{new}})\}}$$

ACTIONREFLEXIVE

$$(\mathcal{E}) \overset{*}{\rightsquigarrow} \mathcal{E}$$

Figure 6. Definitions of editing actions.

the binding context which they appear. With this proof in hand, we show by induction on the number of applications in a term how the terms from the unzipping may be combined to form the target term.

The destruction lemma is straightforward, as the action to replace a term with `unit` is sufficient to accomplish all it requires.

The proof strategy appears readily generalizable to more powerful type systems. In fact, the construction of terms will likely be less constrained, as the polymorphism of type systems such as System F [6, 17] and various dependent calculi will liberalize the constraints imposed on editing actions (Section 6.1).

5. Programming With Editing Actions

Functional languages lend themselves to two general programming strategems. In *top-down* programming, a programmer begins writing the top-level structure of a program, referencing not-yet-implemented functionality by means of identifiers which will later be bound to an implementation. In *bottom-up* programming, a programmer begins by writing small pieces of functionality, and composes them into larger pieces until the top-level program emerges. Our system supports both of these approaches, without locking the programmer into one or the other.

Top-down programming is supported primarily by the λ action. Upon encountering the need for a new piece of functionality, the programmer λ-abstracts over the structure he has written so far, and alters the type of the λ binding to be the type of the component required. The programmer can later implement this component and apply the top-level structure to it. Of course, an action to β-reduce or inline would be of great utility here (see Section 8).

Bottom-up programming is supported primarily by the apply action. Once a programmer has implemented some components, some combinator (often function composition) must be applied in order to compose them. Alternately, once an intermediate value is obtained, a function is applied to obtain the final or next intermediate value.

6. Discussion

6.1 Generalization to Other Typed λ-Calculi

The simply-typed lambda calculus is quite restrictive and does not admit many interesting programs. It is instructive to consider the application of this technique to polymorphic calculi. In particular, the operation of the type-constraint operation $\mathtt{c}(p, e)$ to the operand of an application will change significantly. In the simply-typed lambda calculus, the operand of an application is constrained to a single type, namely, the input type of the operator.

In a polymorphic calculus, the set of acceptable types expands to any type which is compatible with the input type of the operator. Further, the acceptable types of operators expands to any type with whose input type the type of the operand is compatible. This supports the intuitive expectation that a more powerful and flexible calculi will be more flexible to edit under this system as well.

6.2 Additional Actions

The set of actions described here is theoretically complete, but several more desirable actions immediately spring to mind. For instance, the top-down programming approach would benefit greatly from an operation to inline or β-reduce an application. Refactoring of programs would benefit from operations to re-order bindings and applications.

There are two ways of introducing additional actions. Actions can be added to the initial set of actions, which provides more definitional power but requires re-proving the soundness theorem. Alternately, actions can be composed to form new actions. For instance, it is plausible to imagine a composite action which λ-abstracts a term, gives the binding the appropriate type, and applies the term to the bound variable.

6.3 Implementation and User Interface

One advantage of defining editing actions directly on terms is that the the set of actions does not constrain the user interface. Text input and editing is awkward at best on touch-centric and mobile devices, game systems, and accessible interfaces. We expect the approach described here to work well on these platforms, as both movement through the program and alterations to the program are done at the granularity of subterms, rather than characters in program strings.

We are in the process of implementing these editing actions for the simply-typed lambda calculus. Our initial implementation will target Javascript for local in-browser editing of STLC terms. As we extend this work to more polymorphic calculi, we intend to implement the extensions as well. Further, we intend to perform human-computer interaction studies to ascertain the best possible user interface for this approach to term editing on multiple platforms.

One interesting aspect of the user interface is the attachment and presentation of term metadata. Such metadata might be names for bindings (which are semantically formulated as De Bruijn indices), comments, library documentation, and version history. We expect the kind and presentation of this metadata to be of great import in the experience of programmers using our proposed system.

7. Related Work

Graphical programming languages by their nature must eschew text-editing actions as the primary means of creating programs, in favor of actions on graphical elements and structures. The Scratch [10] programming language is a graphical and imperative programming language in which programs control sprites in a virtual arena. The Kodu programming language [9] is a small graphical language, running on the XBox game console and intended for children. Kodu allows users to create games by composing tiles, which are graphical representations of concurrent actors. YinYang [12] is another tile-based concurrent graphical language, which adds the ability to define new tiles and is targeted at touch devices, and intended for more general software development.

Structural editors have a long history, going back at least as far as MENTOR [5] and the Cornell Program Synthesizer [18]. A structural editor is an editor which provides actions on the syntax of a language, instead of or in addition to text-editing actions. Many modern structural editors (such as ParEdit [3] and Structured Haskell Mode [4]) take the second route, extending existing text editors with structural editing actions (for s-expressions and Haskell, respectively.) The Lamdu [8] programming environment uses a structured editor and abstract representations of programs as the basis of an integrated development environment for a Haskell-like language.

Theorem proving systems such as Isabelle [15], Coq [19], and ACL2 [1] provide actions called *tactics* for constructing proof terms. However, these actions generally do not support deconstructing or refactoring the proof terms, and the particular term produced is generally considered irrelevant so long as its existence is demonstrated.

The ability to construct terms by automated theorem proving has proven useful for programming in the dependently-typed language Idris [20]. In particular, automated theorem proving is often used to construct a term of a desired type from a similar term of a different type. Since our soundness theorem asserts that for any starting set of well-typed terms, we can only further reach well-typed terms, it is plausible to consider introducing well-typed terms from other sources, such as automated theorem provers, so long as these terms are typechecked beforehand.

8. Conclusions and Further Work

We have described a set of editing actions on terms in the simply-typed lambda calculus. Further, we have sketched proofs that this set of actions is *sound* (beginning with well-typed terms, only well-typed terms may be derived) and *complete* (any term may be reached from any other term. We have described how this set of actions may be used to produce programs in both top-down and bottom-up style. We have shown that this approach has promise for extension to more powerful typed λ-calculi.

There are several lines of further work open from this point. The simply-typed lambda calculus is, of course, not the most powerful or flexible language. Thus, we intend to generalize this approach to editing to more polymorphic (and eventually, dependent) calculi.

We are working to implement the actions described here, with the intent to eventually re-implement this system in itself. The implementation of the rules is straightfoward, but the question of the appropriate user interface opens up a cross-disciplinary line of inquiry between programming languages and human-computer interaction. Further, while the set of rules described here is certainly sufficient to derive any term from any other term, it is not at all clear that it is convenient or efficient to program with. We believe that a cross-disciplinary inquiry between the fields of human-computer interaction and programming languages will provide insight into the additional actions necessary for a pleasant and productive programming experience.

Acknowledgments

Michael Vitousek assisted with the proof strategies for this paper. Tim Zakian provided many helpful comments and assisted with LaTeX formatting.

References

[1] R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM Monograph. Academic Press, New York, 1979. ISBN 0-12-122950-5. URL `http://www.cs.utexas.edu/users/boyer/acl.pdf`.

[2] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. . URL `http://journals.cambridge.org/article_S095679681300018X`.

[3] T. Campbell. ParEdit. URL `http://mumble.net/~campbell/emacs/paredit.el`.

[4] C. Done. Structured Haskell mode. https://github.com/chrisdone/structured-haskell-mode, 2014.

[5] V. Donzeau-Gouge, G. Huet, B. Lang, G. Kahn, et al. Programming environments based on structured editors: The MENTOR experience. 1980.

[6] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supŕieur*. PhD thesis, Université Paris 7, 1972.

[7] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.01. `http://caml.inria.fr/pub/docs/manual-ocaml-4.01/`, September 2013.

[8] E. Lotem and Y. Chuchem. Lambdu. URL `https://peaker.github.io/lamdu`.

[9] M. B. MacLaurin. The design of kodu: A tiny visual programming language for children on the xbox 360. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 241–246, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. . URL `http://doi.acm.org/10.1145/1926385.1926413`.

[10] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010. ISSN 1946-6226. . URL `http://doi.acm.org/10.1145/1868358.1868363`.

[11] S. Marlow, editor. *Haskell 2010 Language Report*. 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`.

[12] S. McDirmid. Coding at the speed of touch. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 61–76, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0941-7. . URL `http://doi.acm.org/10.1145/2048237.2048246`.

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised*. MIT Press, 1997. ISBN 9780262631815. URL `http://books.google.com/books?id=e0PhKfbj-p8C`.

[14] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[15] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989. ISSN 0168-7433. . URL `http://dx.doi.org/10.1007/BF00248324`.

[16] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachussetts, 2002. ISBN 978-0-262-16209-8. URL `http://www.cis.upenn.edu/~bcpierce/tapl/`.

[17] J. Reynolds. Towards a theory of type structure. *Colloque sur la Programmation*, pages 408–425, April 1974.

[18] T. Teitelbaum and T. Reps. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sept. 1981. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/358746.358755`.

[19] The Coq Development Team. *Coq Reference Manual*. 2012. URL `http://coq.inria.fr/distrib/current/refman/index.html`.

[20] The Idris Community. *Programming in Idris, A Tutorial*. 2014. URL `http://eb.host.cs.st-andrews.ac.uk/writings/idris-tutorial.pdf`.

A. Proofs

A.1 Definitions

Very often we care that a particular term is in a state, without caring what path is associated with it. The $\texttt{has}(\mathcal{E}, e)$ predicate captures this notion:

Definition 2.

$$\texttt{has}(\mathcal{E}, e) \equiv \exists p.(p, e) \in \mathcal{E}$$

The $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$ predicate says that for all terms in a state $\mathcal{E}_1$, $\mathcal{E}_2$ has that term.

Definition 3.

$$\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2) \equiv \forall (p, e) \in \mathcal{E}_1.has(\mathcal{E}_2, e)$$

A.2 Proof of Theorem 1

Proof. (Proof in progress) □

A.3 Proof of Theorem 2

In order to prove Theorem 2, we shall require another function on terms, and two lemmas. The function $\mathtt{u}(e)$ (defined in Figure 7) splits the term tree into sequences of lambda bindings, essentially breaking it apart at application sites. Lemma 1 states that for any goal term e, the set of terms $\mathtt{u}(e)$ (with associated paths) can be derived from the unit term. Lemma 2 states that a goal term e can be derived from the set of terms $\mathtt{u}(e)$.

Lemma 1.

$$\vdash e : t \Rightarrow \exists s, \mathcal{E}.s(\{(\texttt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E} \wedge \{(\texttt{top}, e_1) | e_1 \in \mathtt{u}(e)\} \subseteq \mathcal{E}$$

Proof. By induction on $|\mathtt{u}(e)|$

Case 1: Base case: $|\mathtt{u}(e)| = 0$

(a) $|\mathtt{u}(e)|$ is never 0, so this case holds vacuously.

$$\mathtt{u}(\bullet) = \{\bullet\}$$
$$\mathtt{u}(v) = \{v\}$$
$$\mathtt{u}(\lambda : t.e) = \{\lambda : t.e_u | e_u \in \mathtt{u}(e)\}$$
$$\mathtt{u}(e_1\ e_2) = \mathtt{u}(e_1) \cup \mathtt{u}(e_2)$$

Figure 7. Definition of the term splitting function.

Case 2: : $|u(e)| > 0$.

(a) WLOG, pick some $e' \in u(e)$. Then $\vdash e : t \Rightarrow \exists s', \mathcal{E}'.s'(\{(\mathtt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E}' \wedge \{(\mathtt{top}, e_u) | e_u \in u(e) - \{e'\}\}$

(b) Assume $\vdash e : t$.

(c) By (2b), Lemma 4, and modus ponens: $\mathtt{keeps}(\{(\mathtt{top}, \bullet)\}, \mathcal{E}')$. By the definition of `keeps` (Definition 3) and of `has` (Definition 2), $\exists p'.(p', \bullet) \in \mathcal{E}'$.

(d) (Proof in progress.)

□

Lemma 2.

$\vdash e : t \Rightarrow \forall \mathcal{E}_1 \supseteq \{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\}.\exists s, \mathcal{E}_2.(s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge (\mathtt{top}, e) \in \mathcal{E}_2)$

Proof. (Proof in progress) □

With these lemmas, the proof of Theorem 2 is simple:

Proof. Assume $\vdash e : t$. Then by Lemma 1 and modus ponens, $\exists s, \mathcal{E}.s(\{(\mathtt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E} \wedge \{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\} \subseteq \mathcal{E}$. By Lemma 2 and modus ponens, $\forall \mathcal{E}_1 \supseteq \{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\}.\exists s', \mathcal{E}_2.(s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge (\mathtt{top}, e) \in \mathcal{E}_2)$. Since $\{(\mathtt{top}, e_1) | e_1 \in \mathtt{u}(e)\} \subseteq \mathcal{E}$, let $\mathcal{E}_1 = \mathcal{E}$. Then $\exists s', \mathcal{E}_2.(s'(\mathcal{E}) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge (\mathtt{top}, e) \in \mathcal{E}_2)$. By Lemma 6 and modus ponens, $s\ s'(\{(\mathtt{top}, \bullet)\}) \overset{*}{\rightsquigarrow} \mathcal{E}_2$. Then $s\ s'$ and $\mathcal{E}_2$ are the witnesses for the existential in the theorem.

□

A.4 Utility Lemmas

Lemma 3.

$$s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge \mathcal{E}_1 \subseteq \mathcal{E}_3 \Rightarrow \exists \mathcal{E}_4.(\mathcal{E}_2 \subseteq \mathcal{E}_4 \wedge s(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_4)$$

Proof. By induction on s:

- $s = \epsilon$:
  1. The only rule for the empty sequence is ACTIONREFLEXIVE.
  2. So $\mathcal{E}_1 = \mathcal{E}_2$.
  3. $\mathcal{E}_1 \subseteq \mathcal{E}_3$ by the assumption of the lemma.
  4. Then by substitution using (2) in (3), $\mathcal{E}_2 \subseteq \mathcal{E}_3$.
  5. $\mathcal{E}_3 \overset{*}{\rightsquigarrow} \mathcal{E}_3$ by ACTIONREFLEXIVE.
  6. So if we let $\mathcal{E}_4 = \mathcal{E}_3$, then we have a witness to the existential $\exists \mathcal{E}_4.(\mathcal{E}_2 \subseteq \mathcal{E}_4 \wedge s(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_4)$.
- $s = s'\ a(p, e)$:

  By cases on the derivation of $s'\ a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$:
  - ACTION:
    1. By the use of the ACTION rule in the derivation, we know that:
       (a) $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_{2s'}$.
       (b) $(p, e) \in \mathcal{E}_{2s'}$.
       (c) $a(p, e) \rightsquigarrow \mathcal{E}_a$.
       (d) $\mathcal{E}_2 = \mathcal{E}_{2s'} \cup \mathcal{E}_a$.
    2. By the inductive hypothesis, we know that $\forall \mathcal{E}_3 | \mathcal{E}_1 \subseteq \mathcal{E}_3.\exists \mathcal{E}_{4s'}.(\mathcal{E}_{2s'} \subseteq \mathcal{E}_{4s'} \wedge s'(\mathcal{E}_3) \rightsquigarrow \mathcal{E}_{4s'})$.
    3. By the properties of sets, (1b), and (2), we have $(p, e) \in \mathcal{E}_{4s'}$.
    4. By the ACTION rule, (1c), (2), and (3), we have $s'\ a(p, e)(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    5. By (2) and equational reasoning, we have that $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a$.
    6. By (1d), (5). and substitution, we have that $\mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    7. By the conjunction of (5) and (6), we have $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a \wedge \mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    8. By (7), we see that $\mathcal{E}_{4s'} \cup \mathcal{E}_a$ is a witness to the existential in our conclusion.
  - ACTIONMUTATE:
    1. By the use of the ACTIONMUTATE rule in the derivation, we know that:
       (a) $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_{2s'}$.
       (b) $(p_1, e_1) \in \mathcal{E}_{2s'}$.
       (c) $a(p_1, e_1) \overset{!}{\rightsquigarrow} (p_2, e_2)$.
       (d) $\mathcal{E}_2 = (\mathcal{E}_{2s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}$.
    2. By the inductive hypothesis, we know that $\forall \mathcal{E}_3 | \mathcal{E}_1 \subseteq \mathcal{E}_3.\exists \mathcal{E}_{4s'}.(\mathcal{E}_{2s'} \subseteq \mathcal{E}_{4s'} \wedge s'(\mathcal{E}_3) \rightsquigarrow \mathcal{E}_{4s'})$.
    3. By the properties of sets, (1b), and (2), we have $(p_1, e_1) \in \mathcal{E}_{4s'}$.
    4. By the ACTIONMUTATE rule, (1c), (2), and (3), we have $s'\ a(p, e)(\mathcal{E}_3) \overset{*}{\rightsquigarrow} ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    5. By (2) and equational reasoning, we have that $((\mathcal{E}_{2s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}) \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    6. By (1d), (5). and substitution, we have that $\mathcal{E}_2 \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    7. By the conjunction of (5) and (6), we have $((\mathcal{E}_{2s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}) \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\}) \wedge \mathcal{E}_2 \subseteq ((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$.
    8. By (7), we see that $((\mathcal{E}_{4s'} - \{(p_1, e_1)\}) \cup \{(p_2, e_2)\})$ is a witness to the existential in our conclusion.
- $s = s\ a(p, e_1, e_2)$:

  By cases on the derivation of $s'\ a(p, e_1, e_2)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$:
  - DOUBLEACTION:
    1. By the use of the DOUBLEACTION rule in the derivation, we know that:
       (a) $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_{2s'}$.
       (b) $(p, e_1) \in \mathcal{E}_{2s'}$.
       (c) $(p, e_2) \in \mathcal{E}_{2s'}$.
       (d) $a(p, e_1, e_2) \rightsquigarrow \mathcal{E}_a$.
       (e) $\mathcal{E}_2 = \mathcal{E}_{2s'} \cup \mathcal{E}_a$.
    2. By the inductive hypothesis, we know that $\forall \mathcal{E}_3 | \mathcal{E}_1 \subseteq \mathcal{E}_3.\exists \mathcal{E}_{4s'}.(\mathcal{E}_{2s'} \subseteq \mathcal{E}_{4s'} \wedge s'(\mathcal{E}_3) \rightsquigarrow \mathcal{E}_{4s'})$.
    3. By the properties of sets, (1b), (1c), and (2), we have $(p, e_1) \in \mathcal{E}_{4s'} \wedge (p, e_2) \in \mathcal{E}_4 s'$.
    4. By the DOUBLEACTION rule, (1d), (2), and (3), we have $s'\ a(p, e)(\mathcal{E}_3) \overset{*}{\rightsquigarrow} \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
    5. By (2) and equational reasoning, we have that $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a$.
    6. By (1e), (5). and substitution, we have that $\mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.

7. By the conjunction of (5) and (6), we have $\mathcal{E}_{2s'} \cup \mathcal{E}_a \subseteq \mathcal{E}_{4s'}\mathcal{E}_a \wedge \mathcal{E}_2 \subseteq \mathcal{E}_{4s'} \cup \mathcal{E}_a$.
8. By (7), we see that $\mathcal{E}_{4s'} \cup \mathcal{E}_a$ is a witness to the existential in our conclusion.

□

Lemma 4.

$$s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \rightarrow \texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$$

Proof. By induction on s.

Case 1: Base case: $s = \epsilon$.
- (a) Assume $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$.
- (b) By ACTIONREFLEXIVE rule in the derivation of $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$ (1a), $\mathcal{E}_1 = \mathcal{E}_2$.
- (c) By (1b), $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$ holds trivially.

Case 2: $s = s'a(p, e)$.
- (a) Inductive hypothesis: $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2' \Rightarrow \texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2')$
- (b) Assume:
  - i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
- (c) By (2a) and the definition of `keeps` (Definition 3), and the definition of `has` (Definition 2): $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2' \Rightarrow \forall (p, e) \in \mathcal{E}_1 \exists p'.(p', e) \in \mathcal{E}_2'$
- (d) The derivation of $s(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$ (2(b)i) is either ACTION or ACTIONMUTATE. By cases:
  - i. ACTION:
    - A. By the premises of the ACTION rule in the derivation (2(d)i): $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2'$.
    - B. By (2(d)iA), (2a), and modus ponens: $\forall (p, e) \in \mathcal{E}_1 . \exists p'.(p', e) \in \mathcal{E}_2'$
    - C. By the conclusion of the ACTION rule: $\mathcal{E}_2 = \mathcal{E}_2' \cup \mathcal{E}_3$
    - D. By (2(d)iC), $s_2' \subseteq s_2$.
    - E. By (2(d)iD), $\forall (p, e) \in \mathcal{E'}_2 . (p, e) \in \mathcal{E}_2$.
    - F. By (2(d)iD) and (2(d)iA), $\forall (p, e) \in \mathcal{E}_1 . \exists p'.(p', e) \in \mathcal{E}_2$.
    - G. By (2(d)iF), the definition of `keeps` (Definition 3), and the definition of `has` (Definition 2), $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$.
  - ii. ACTIONMUTATE:
    - A. By the premises of the ACTIONMUTATE rule in the derivation (2(d)ii): $s'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2'$.
    - B. By (2(d)iiA), (2a), and modus ponens: $\forall (p, e) \in \mathcal{E}_1 . \exists p'.(p', e) \in \mathcal{E}_2'$
    - C. By the premises of the ACTIONMUTATE rule in the derivation (2(d)ii): $a(p, e) \overset{!}{\rightsquigarrow} \mathcal{E}_3$.
    - D. By Lemma 5, (2(d)iiC), and modus ponens: $\texttt{has}(\mathcal{E}_3, e)$.
    - E. By the conclusion of the ACTIONMUTATE rule in the derivation (2(d)ii), $\mathcal{E}_2 = (\mathcal{E}_2' - \{(p, e)\}) \cup \mathcal{E}_3$
    - F. Observing that $\texttt{has}(\mathcal{E}, e) \Rightarrow \texttt{has}(\mathcal{E}' \cup \mathcal{E}, e)$, and by (2(d)iiB), (2(d)iiD) and (2(d)iiE): $\forall (p, e) \in \mathcal{E}_1 . \texttt{has}(\mathcal{E}_2, e)$ and thus, by the definition of `keeps` (Definition 3), $\texttt{keeps}(\mathcal{E}_1, \mathcal{E}_2)$.

□

Lemma 5.

$$a(p, e) \overset{!}{\rightsquigarrow} (p', e') \Rightarrow \texttt{has}(\mathcal{E}, e)$$

Proof. 1. Assume $a(p, e) \overset{!}{\rightsquigarrow} \mathcal{E}$.

2. By cases on the derivation of $a(p, e) \overset{!}{\rightsquigarrow} (p, e)$ (1):
- Case (a): LAMTYMOVE
  - i. By the conclusion of the derivation of LAMTYMOVE (2(a)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (b): LAMBODMOVE
  - i. By the conclusion of the derivation of LAMBODMOVE (2(b)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (c): APPRATORMOVE
  - i. By the conclusion of the derivation of APPRATORMOVE (2(c)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (d): APPRANDMOVE
  - i. By the conclusion of the derivation of APPRANDMOVE (2(d)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (e): TYINMOVE
  - i. By the conclusion of the derivation of TYINMOVE (2(e)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (f): TYOUTMOVE
  - i. By the conclusion of the derivation of TYOUTMOVE (2(f)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.
- Case (g): UPMOVE
  - i. By the conclusion of the derivation of UPMOVE (2(g)i), $e = e'$ and thus $\texttt{has}(\{(p', e')\}, e)$.

□

Lemma 6.

$$s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3 \Rightarrow s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$$

Proof. By induction on s_2.

Case 1: Base case: $s_2 = \epsilon$.
- (a) Assume
  - i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
  - ii. $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$
- (b) Since $s_2 = \epsilon$, the derivation of $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$ (1(a)ii) is ACTIONREFLEXIVE, and $\mathcal{E}_2 = \mathcal{E}_3$.
- (c) Since $s_2 = \epsilon$, $s_1 s_2 = s_1$.
- (d) Substituting (1c) and (1d) into (1(a)i), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$

Case 2: $s_2 = s_2' a(p, e)$.
- (a) Inductive hypothesis: $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \Rightarrow s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
- (b) Assume
  - i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
  - ii. $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$
- (c) The derivation of $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$ (2(b)ii) is either ACTION or ACTIONMUTATE. By cases:
  - i. ACTION:
    - A. By the ACTION rule in the derivation: $s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$

B. By the ACTION rule in the derivation: $(p, e) \in \mathcal{E}_3'$
C. By the ACTION rule in the derivation: $a(p, e) \rightsquigarrow \mathcal{E}_4$
D. By the ACTION rule in the derivation: $\mathcal{E}_3' \cup \mathcal{E}_4 = \mathcal{E}_3$
E. By (2(b)i), (2(c)iA), (2a) and modus ponens: $s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$.
F. By (2(c)iE), (2(c)iB), (2(c)iC) and the ACTION rule, $s_1 s_2' a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \cup \mathcal{E}_4$.
G. By substition of (2) and (2(c)iD) into (2(c)iF), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$.

ii. ACTIONMUTATE:
A. By the ACTIONMUTATE rule in the derivation: $s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
B. By the ACTIONMUTATE rule in the derivation: $(p, e) \in \mathcal{E}_3'$
C. By the ACTIONMUTATE rule in the derivation: $a(p, e) \rightsquigarrow \mathcal{E}_4$
D. By the ACTIONMUTATE rule in the derivation: $\mathcal{E}_3' \cup \mathcal{E}_4 = \mathcal{E}_3$
E. By (2(b)i), (2(c)iiA), (2a) and modus ponens: $s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$.
F. By (2(c)iiE), (2(c)iiB), (2(c)iiC) and the ACTION rule, $s_1 s_2' a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \cup \mathcal{E}_4$.
G. By substition of (2) and (2(c)iiD) into (2(c)iiF), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$.

Case 3: $s_2 = s_2' a(p, e_1, e_2)$.
(a) Inductive hypothesis: $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2 \wedge s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \Rightarrow s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
(b) Assume
i. $s_1(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_2$
ii. $s_2(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3$
(c) By the DOUBLEACTION rule in the derivation: $s_2'(\mathcal{E}_2) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$
(d) By the DOUBLEACTION rule in the derivation: $(p, e_1) \in \mathcal{E}_3'$
(e) By the DOUBLEACTION rule in the derivation: $(p, e_2) \in \mathcal{E}_3'$
(f) By the DOUBLEACTION rule in the derivation: $a(p, e_1, e_2) \rightsquigarrow \mathcal{E}_4$
(g) By the DOUBLEACTION rule in the derivation: $\mathcal{E}_3' \cup \mathcal{E}_4 = \mathcal{E}_3$
(h) By (3(b)i), (3c), (3a) and modus ponens: $s_1 s_2'(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3'$.
(i) By (3h), (3d), (3e), (3f) and the DOUBLEACTION rule, $s_1 s_2' a(p, e)(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3' \cup \mathcal{E}_4$.
(j) By substition of (3) and (3g) into (3i), $s_1 s_2(\mathcal{E}_1) \overset{*}{\rightsquigarrow} \mathcal{E}_3$.

□

Towards a native higher-order RPC

Olle Fredriksson Dan R. Ghica Bertram Wheen
University of Birmingham, UK

Abstract

We present a new abstract machine, called DCESH, which models the execution of higher-order programs running in distributed architectures. DCESH implements a native general remote higher-order function call across node boundaries. It is a modernised version of SECD enriched with specialised communication features required for implementing the RPC mechanism. The key correctness result is that the termination behaviour of the RPC is indistinguishable (bisimilar) to that of a local call. The correctness proofs and the requisite definitions for DCESH and other related abstract machines are formalised using Agda. We also formalise a generic transactional mechanism for transparently handling failure in DCESHs.

We use the DCESH as a target architecture for compiling a conventional call-by-value functional language ("FLOSKEL") which can be annotated with node information. Conventional benchmarks show that the single-node performance of FLOSKEL is comparable to that of OCAML, a semantically similar language, and that distribution overheads are not excessive.

1. Native RPC and transparent distribution

Remote Procedure Call (RPC) [7] is a widely used mechanism for higher-level inter-process communication. However, the RPC mechanism tends to be bolted on top of a pre-existing language, as a library for example, rather than be seamlessly integrated into it. This leads to significant syntactic differences between calling a local library function and a remote function. Even if these syntactic differences can be smoothed using *stubs* that wrap remote calls into local calls [6] important differences still persist, of which the most important is that arguments must be of ground types.

Generalising RPC to all types and incorporating it seamlessly in the language has been considered but dismissed on grounds of potential inefficiencies [41]. However, considering that a number of technologies that trade efficiency for convenience have been rejected on similar grounds (from machine independent languages to functional programming, garbage collection or automated program verification – to name only a few), we decided it is time to revisit this issue. Because the execution of applications is increasingly and rapidly moving from single devices to networks of (often heterogeneous) devices a case can be made that such revisiting is timely. A native RPC system, offering as an immediate benefit transparent and automated distribution, can be useful in domains where programmer effectiveness is more important than machine efficiency.

We attack this problem in a principled manner. We want the seamless integration of the RPC in the language to be not merely syntactic but semantic as well: the RPC is a native call of our language, on the same level as a conventional local call. This is realised by introducing a new kind of abstract machine which extends the conventional SECD machine [26], or rather a modernised version of it, with communication primitives. These are not general-purpose low-level communication primitives but are especially designed to support the implementation of RPCs. Even though we aim for simplicity first, some of the technicalities, especially the proofs of correctness, are quite intricate. For this reason the abstract machine net framework and its correctness properties are fully formalised using the Agda language [34]. The technical challenge is not just one of handling complicated formalisms but also mathematical, the key correctness proof requiring the adaptation of the *step-index relations* technique [2] to bisimulation

Note that we are language rather than system oriented. We assume the existence of a run-time infrastructure to handle system-level aspects associated with distribution such as failure detection, load balancing, global reset and initialisation, and so on. From a systems perspective we only deal with failure handling, which is particularly important in a distributed setting, using a general transactional approach.

The abstract machine nets can serve as a target for the compilation of a conventional call-by-value language which we call FLOSKEL. The interesting new thing about it is that there is almost nothing new about it. It uses a HASKELL-like syntax but it has the semantics of the pure fragment of OCAML. RPC calls are indistinguishable from native calls except for an annotation, which can be applied to any sub-term, indicating that it is to be executed on a different node. From the point of view of the language, and of the programmer, this annotation has no syntactic, semantic or typing implication. It is a mere pragma-like directive.

Prior work on native RPCs and seamless distribution are specialised to web programming [11], thus domain-specific. We aim to be as generic as possible in this context. Some existing work uses as a starting point interaction-based semantic paradigms which lend themselves naturally to a communication-centric implementation: Geometry of Interaction [17] and Game Semantics [18]. Such approaches have two significant disadvantages. The exotic operational behaviour makes it impossible to apply known optimisation techniques, and to interact with code compiled conventionally. This can be seen in the low performance of single-node execution of programs compiled using such techniques. On the other hand, the single-node compilation of FLOSKEL is very similar to the conventional compilation of a language such as OCAML, and the benchmarks indicate that the overhead required by the RPC run-time is not excessive.

In a nutshell, we believe that this paper can help make the case that functional languages with native RPCs could be a lot more useful than presumed.

1.1 Technical outline and contributions

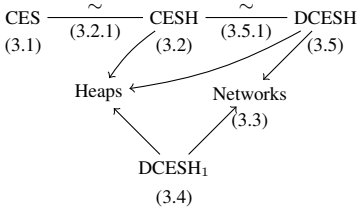
Our main contributions are in the following areas:

Compiler and run-time We describe the syntax (Sec. 2.1) and implementation (Sec. 2.2) of FLOSKEL, a general-purpose functional language with native RPCs. Our basis is a conventional compiler for such a language, and we show how it is modified to support RPCs and, additionally, *ubiquitous* functions, i.e. functions available on all nodes. Our benchmarks suggest that FLOSKEL's performance is comparable to the state of the art OCAML compiler (Sec. 2.4) for single-node execution.

Abstract machines The semantics of a core of FLOSKEL has been formalised in Agda (Sec. 3) in the form of an abstract machine that can be used to guide an implementation. To achieve this we make gradual refinements to a machine, based on Landin's SECD machine [26], that we call the *CES machine* (Sec. 3.1). First we add heaps for dynamically allocating closures, forming the *CESH machine* (Sec. 3.2); we show that the CES and CESH execution is bisimilar. We then add communication primitives (synchronous and asynchronous) by defining a general form of networks of nodes that run an instance of an underlying abstract machine (Sec. 3.3). Using these networks, we illustrate the idea of subsuming function calls by communication protocols by constructing a degenerate distributed machine, DCESH_1 (Sec. 3.4), that decomposes some machine instructions into message passing, but only runs on one node. Execution on the fully distributed CESH machine called DCESH (Sec. 3.5), is shown to be bisimilar to the CESH machine — our main theoretical result. Finally, we model a general-purpose fault-tolerant environment for DCESH-like machines (Sec. 4) by layering a transactional abstract machine that provides a simple commit-and-rollback mechanism for an abstract machine that may unexpectedly fail.

Formalisation in Agda The theorems that we present in this paper have been proved correct in Agda [34], an interactive proof assistant and programming language based on intuitionistic type theory. The definitions and proofs in this paper are intricate, so carrying them out manually would be error-prone, arduous and perhaps unconvincing. Agda has been a helpful tool in producing these proofs, providing a pleasant interactive environment in which to play with alternative definitions. To eliminate another source of error, we do not present our results in informal mathematics; the code blocks in Sec. 3 come directly from the formalisation.

The formalisation is organised as follows, where the arrows denote dependence, the lines with $\sim$ symbols bisimulations, and the parenthesised numerals section numbers:



2. FLOSKEL: a location-aware language

2.1 Syntax

At the core of the FLOSKEL language is a call-by-value functional language with user-definable algebraic data types and pattern matching. FLOSKEL is semantically similar to languages in the ML family, and syntactically similar to HASKELL. The main syntactic difference between FLOSKEL and HASKELL is that pattern matching clauses are given without the leading function name and that type annotations are given after a single colon, as in the following example:

```
map : (a → b) → [a] → [b]
  f []     = []
  f (x::xs) = f x :: map f xs
```

Node annotations An ordinary function definition, like *map*, is a *ubiquitous* function by default. This means that it is made available on all nodes in the system, and a call to such a function is always done locally – a plain old function call.

On the other hand, a function or sub-term defined with a *node annotation*, such as

```
query@Database : Query → Response
  x = ...,
```

is *located* and compiled only to the specified node (here *Database*). In the rest of the program *query* can be used like any other function, but the compiler and run-time system treat it differently. A call to *query* from a node other than *Database* is a *remote* call.

Since the programmer can use located functions like any other functions, and this is a functional language, it means that the language has, by necessity, support for higher-order functions across node boundaries. For instance the function

```
f@A : (Query → Response) → X
  q = ... use q ...
```

can be applied to *query* yielding *f query* : *X*.

Node annotations can also be applied to sub-expressions, as in the following example:

```
sum []     = 0
    (x::xs) = x + sum xs
xs@A = ...
ys@B = ...
result@C = (sum xs) @A + (sum ys) @B
```

Here we want to calculate the sum, on node *C*, of the elements of two lists located on nodes *A* and *B*. If the lists are lengthy, it is better to calculate the sums on *A* and *B*, and to then send the final sum to *C*, since this saves us having to send the full lists over the network.

2.2 Compilation

The FLOSKEL compiler [1] currently targets C using the MPI library for communication, though other targets are possible. Most of the compiler's pipeline is standard for a functional language implementation. It works by applying a series of transformations to the source program until reaching a level low enough to be straightforwardly translated to C. Since the source language has pattern matching, it first compiles the pattern matching to simple case trees [4]. Local definitions are then lifted to the top-level using lambda lifting [25], and lastly the program is closure converted [31] to support partially applied functions.

Up until the lambda lifting, a node annotation is a constructor in the abstract syntax tree of the language's expressions. The lambda lifter lifts such sub-expressions to the top-level such that annotations are afterwards associated with definitions (and not expressions).

The main work specific to FLOSKEL is done in the closure conversion and the run-time system that the compiled programs make use of.

Closures For function applications, the closure converter distinguishes between known functions – those that are on the top-level

and have a known arity, and unknown functions – those that are provided e.g. as function arguments.

A known function f that is either ubiquitous or available on the same node as the definition that is being compiled is compiled to an ordinary function call if there are enough arguments. If there are not, and the function is ubiquitous we have to construct a *partial application* closure, which contains a pointer to the function and the arguments so far. The compiler maintains the invariant that unknown functions are always in the form of a closure, whose general layout is:

g_{ptr}	g_{id}	*arity*	*payload...*

Since the function may require access to the payload of the closure, g_{ptr} is a function of arity *arity* + *1*: when applying a closure *cl* as above to arguments $x_1, ..., x_{arity}$, the call becomes $g_{ptr}\ (cl, x_1, ..., x_{arity})$ meaning that the function has access to the payload through *cl*. To construct the initial closure for a partial application of a function f of arity *arity* with *nargs* arguments, we have to conform to this rule, so we construct the closure $(f'_{ptr}, f'_{id}, n, y_1, ..., y_{nargs})$ where $n = arity - nargs$ and f' is a new ubiquitous top-level function defined as follows:

$$f'\ cl\ x_1\ ...\ x_n\ =\ \mathsf{case}\ cl\ \mathsf{of}\ (\_, \_, \_, y_1, ..., y_{nargs})\ \rightarrow\ f\ (y_1, ..., y_{nargs}, x_1, ..., x_n)$$

A family of $apply_i$ functions handle, in a standard way, applications (of i arguments) of unknown functions by inspecting the arity stored in the closure to decide whether to construct a new partial application closure with the additional arguments or to apply the function.

The field f_{id} is an integer identifier assigned to every function at compile-time used as a system-wide identifier if the function is ubiquitous, or a node-specific identifier if not. If there are k ubiquitous functions they are assigned the first k identifiers, and the nodes of the system may use identifiers greater than k for their respective located functions. Determining if a function is ubiquitous is thus a simple comparison: $f_{id} < k$. Additionally, every node has a table of functions that maps ubiquitous or local located function identifiers to local function pointers, which is used by the deserialiser.

If we have a saturated call to a known remote function, we make a call to the function $rApply_{arity}$, defined in the run-time system (to be described). If we have a non-saturated call to a known remote or located function, we construct the closure $(f'_{ptr}, f'_{id}, arity, y_1, ..., y_{nargs})$ where f' is a new ubiquitous top-level function defined as follows:

$$\begin{array}{l} f'\ cl\ x_1\ ...\ x_n\ =\ \mathsf{case}\ cl\ \mathsf{of} \\ \quad (\_, \_, \_, y_1, ..., y_{nargs})\ \rightarrow \\ \quad\quad \mathsf{if}\ myNode\ \equiv\ f_{node}\ \mathsf{then} \\ \quad\quad\quad lookup\ (f_{id})\ (y_1, ..., y_{nargs}, x_1, ..., x_n,) \\ \quad\quad \mathsf{else} \\ \quad\quad\quad rApply_{arity}\ (f_{node}, f_{id}, y_1, ..., y_{nargs}, x_1, ..., x_n) \end{array}$$

Here *myNode* is the identifier of the node the code is currently being run at. If it is the same node as the node of f, we can make an ordinary function call by looking up the function corresponding to f_{id} in the function table. Otherwise we call the run-time system function $rApply_{arity}$.

In this way, we construct a closure for located functions that looks just like the closure of an ubiquitous function.

2.3 Run-time

The run-time system defines a family of ubiquitous functions $rApply_{arity}$, that, as we saw above, are used for remote procedure calls and to construct closures for located functions. The function takes a function identifier, a node identifier, and *arity* arguments. It serialises the arguments and sends them together with the function identifier to the given node:

$$\begin{array}{l} rApply\ f_{node}\ f_{id}\ x_1\ ...\ x_{arity}\ = \\ \quad \mathsf{send}\ (f_{id}, serialise\ (x_1), ..., serialise\ (x_{arity}))\ \mathsf{to}\ f_{node}; \\ \quad \mathsf{receive}\ answer\ \mathsf{from}\ f_{node}\ \rightarrow \\ \quad answer \end{array}$$

When the node f_{node} receives this message, it looks the function up in its function table, calls it with the deserialised arguments, and sends back the result:

$$\begin{array}{l} \mathsf{receive}\ (f_{id}, y_1, ..., y_{arity})\ \mathsf{from}\ remoteNode\ \rightarrow \\ \quad \mathsf{let}\ result\ =\ lookup\ (f_{id})\ (deserialise\ (y_1), ..., deserialise\ (y_{arity})) \\ \quad \mathsf{in}\ \mathsf{send}\ result\ \mathsf{to}\ remoteNode \end{array}$$

Serialisation In a remote function call the arguments may be values from arbitrary algebraic data types (like lists and trees), in addition to primitive types and functions.

The serialisation of a primitive type is the identity function, while algebraic data-types require a traversal and flattening of the heap structure. We use tags in the lower bits of a value's field to differentiate between pointers and non-pointers, which makes this flattening straightforward. The interesting part of serialisation is how to handle closures, both in the case of ubiquitous and located functions.

For closures around ubiquitous functions, we serialise the closure almost as is, but use the function identifiers to resolve the function pointer on the receiving node, as it is not guaranteed to be the same on each node.

To handle located functions, the most straightforward implementation is to use "mobilised" closures that work by exchanging the located function with a ubiquitous function that calls *rApply* to perform the remote procedure call. This is what our implementation currently does. Our formalisation will describe an optimised variant of this scheme, which instead saves the closure on the sending node and sends a pointer to that. The optimised scheme means that we do not unnecessarily send closures containing (potentially large) arguments that are going to end up on the node they originated from anyway. The cost of this optimisation, however, is that it requires us to keep track of heap-allocated pointers across node boundaries using distributed garbage collection. The serialisation currently implemented does not require such garbage collection, but may be slow when dealing with large data.

In detail, to serialise a closure

f_{ptr}	f_{id}	*arity*	*payload...*

we put a placeholder, `CL`, in the place of f_{ptr}:

`CL`	f_{id}	*arity*	*payload'* ...

where *payload'* represents the serialised payload and `CL` is a tag that can be used to identify that this is a closure. To deserialise this on the receiving end, we look up the function pointer associated with f_{id} in the ubiquitous function table and substitute that for `CL`.

2.4 Performance benchmarks

Single-node Before we measure the performance of the implementation of the native RPC, we analyse how the single-node performance is affected by the distribution overhead even if it is not used — is it feasible for a general-purpose language to be based on the DCESH?

Fig. 1 shows absolute and relative timings of a number of small benchmarks using integers, lists, trees, recursion, and a small amount of output for printing results. We compare the performance of FLOSKEL programs compiled with our compiler, and equivalent OCAML programs compiled using `ocamlopt`, a high-performance native-code compiler. Since our compiler targets C, we further

	trees	nqueens	qsort	primes	tak	fib
FLOSKEL	91.2s	12.2s	9.45s	19.3s	16.5s	10.0s
ocamlopt	43.0s	3.10s	3.21s	6.67s	2.85s	1.68s
relative	2.12	3.94	2.94	2.9	5.77	5.95

Figure 1. Single-node performance.

	trees	nqueens	qsort	primes	tak	fib
μs/remote call	618	382	4.77	13.4	6.94	6.87
B/remote call	1490	25.8	28.1	27.0	32.0	24.0

Figure 2. Distribution overheads.

compile the generated files to native code using `gcc -O2`. We can see that the running time of programs compiled with our compiler is between two and six times greater than that of those compiled with `ocamlopt`. These results should be viewed in the light of the fact that our compiler only does a minimal amount of optimisation, whereas a considerable amount of time and effort has been put into `ocamlopt`.

Moreover, our compiler only produces C code rather than assembly. Compiling to C rather than assembly, especially in the style we use, prevents the C compiler from using whole-program optimisations and is, therefore, a serious source of inefficiencies.

Distribution overhead We measure the overhead of our implementation of native remote procedure calls by running the same programs as above, but distributed to between two and nine nodes. The distribution is done by adding node annotations in ways that generate large amounts of communication. We run the benchmarks on a single physical computer with local virtual nodes, which means that the contributions of network latencies are factored out. These measurements give the overhead of the other factors related to remote calls, like serialisation and deserialisation. The results are shown in Fig. 2. The first row, μs/remote call, is obtained by running the same benchmark with and without node annotations, taking the delta-time of those two, and then dividing by the number of remote invocations in the distributed program. The second row measures the amount of data transmitted per remote invocation, in bytes.

It is expected that this benchmark depends largely on the kinds of invocations that are done, since it is more costly to serialise and send a long list or a big closure than an integer. The benchmark hints at this; we appear to get a higher cost for remote calls that are big.

An outlier is the `nqueens` benchmark, which does not do remote invocations with large arguments, but still has a high overhead per call, because it intentionally uses many localised functions.

3. Abstract machines and nets

Having introduced the programming language, its compiler and its run-time system we now present the theoretical foundation for the correctness of the compiler. We start with the standard abstract machine model of CBV computation, which we refine, in several steps, into increasingly expressive abstract machines with heap and networking capabilities, while showing along the way that correctness is preserved, via bisimulation results. All definitions and theorems are formalised using the proof assistant Agda, the syntax of which we will follow. We give some of the key definitions and examples in Agda syntax, but most of the description of the formalisation is intended as a high-level guide to the Agda code, which is available online [1]. In order to help the reader navigate the code when a significant theorem or lemma is mentioned the fully qualified Agda name is given in a footnote. Note that we shall not formalise the whole of FLOSKEL but only a core language which coincides with Plotkin's (untyped) call-by-value PCF [37].

3.1 The CES machine

The starting point is a variation of Landin's SECD machine [26] called Modern SECD [27], which can be traced to the SECD machine variation of Henderson [22] and to the CEK machine of Felleisen [15], which we call CES (Agda module *CES*). Just like the machine of Henderson, it uses bytecode for the control component of the machine, and just like the CEK it places the continuations that originally resided in the dump directly on the stack, simplifying the machine configurations.

A CES configuration (*Config*) is a tuple consisting of a fragment of code (*Code*), an environment (*Env*), and a stack (*Stack*). Evaluation begins with an empty stack and environment, and then follows a stack discipline. Sub-terms push their result on the stack so that their super-terms can consume them. When (and if) the evaluation terminates, the program's result is the sole stack element.

Source language The source language has constructors for lambda abstractions (**λ** *t*), applications (*t* **$** *t'*), and variables (**var** *n*), represented using De Bruijn indices [13], so a variable is a natural number. Additionally, we have natural number literals (**lit** *n*), binary operations (**op** *f t t'*), and a conditional (**if0** *t* **then** t_0 **else** t_1). Because the language is untyped, we can express a fixed-point combinator without adding additional constructors.

The machine operates on bytecode and does not directly interpret the source terms, so the terms need to be compiled before they can be executed. The main work of compilation is done by the function *compile'*, which takes a term *t* and a fragment of code *c* used as a postlude. The bold upper-case names (**CLOS**, **VAR**, and so on) are the bytecode instructions, which are sequenced using _;_. Instructions can be seen to correspond to the constructs of the source language, sequentialised.

```
compile' : Term → Code → Code
compile' (λ t)       c = CLOS (compile' t RET) ; c
compile' (t $ t')    c = compile' t (compile' t' (APPL ; c))
compile' (var x)     c = VAR x ; c
compile' (lit n)     c = LIT n ; c
compile' (op f t t') c = compile' t' (compile' t (OP f ; c))
compile' (if0 b then t else f) c =
  compile' b (COND (compile' t c) (compile' f c))
```

Example 3.1 (*codeExample*)**.** To compile a term *t* we supply **END** as a postlude: *compile t* = *compile' t* **END**. The term $t = (\lambda x.\ x)\ (\lambda x\ y.\ x)$ is compiled as follows:

```
compile ((λ var 0) $ (λ (λ var 1))) = CLOS (VAR 0 ; RET) ;
  CLOS (CLOS (VAR 1 ; RET) ; RET) ; APPL ; END
```

Environments (*Env*) are lists of values (*List Value*), which are either natural numbers (**nat** *n*) or closures (**clos** *cl*). A closure (*Closure*) is a fragment of code paired with an environment (*Code* × *Env*). Stacks (*Stack*) are lists of stack elements (*List StackElem*), which are either values (**val** *v*) or continuations (**cont** *cl*), represented by closures.

Fig. 3 shows the definition of the transition relation for configurations of the CES machine. A note on Agda syntax: The instruction constructor names are overloaded as constructors for the relation; their usage is disambiguated by context. Arguments in curly braces are *implicit* and can be automatically inferred. Equality (propositional) is written _≡_.

Stack discipline is clear in the definition of the transition relation. When e.g. **VAR** is executed, the CES machine looks up the value of the variable in the environment and pushes it on the stack. A somewhat subtle part of the relation is the interplay between the

$$
\begin{array}{lll}
\textbf{VAR} & : \forall\,\{n\ c\ e\ s\ v\} \to \mathit{lookup}\ n\ e \equiv \textbf{just}\ v \to & (\textbf{VAR}\ n\ \textbf{;}\ c,e,s) \xrightarrow[CES]{} (c,e,\textbf{val}\ v :: s)\\
\textbf{CLOS} & : \forall\,\{c'\ c\ e\ s\} \to & (\textbf{CLOS}\ c'\ \textbf{;}\ c,e,s) \xrightarrow[CES]{} (c,e,\textbf{val}\ (\textbf{clos}\ (c',e)) :: s)\\
\textbf{APPL} & : \forall\,\{c\ e\ v\ c'\ e'\ s\} \to & (\textbf{APPL}\ \textbf{;}\ c,e,\textbf{val}\ v :: \textbf{val}\ (\textbf{clos}\ (c',e')) :: s) \xrightarrow[CES]{} (c',v :: e',\textbf{cont}\ (c,e) :: s)\\
\textbf{RET} & : \forall\,\{e\ v\ c\ e'\ s\} \to & (\textbf{RET},e,\textbf{val}\ v :: \textbf{cont}\ (c,e') :: s) \xrightarrow[CES]{} (c,e',\textbf{val}\ v :: s)\\
\textbf{LIT} & : \forall\,\{n\ c\ e\ s\} \to & (\textbf{LIT}\ n\ \textbf{;}\ c,e,s) \xrightarrow[CES]{} (c,e,\textbf{val}\ (\textbf{nat}\ n) :: s)\\
\textbf{OP} & : \forall\,\{f\ c\ e\ n_1\ n_2\ s\} \to & (\textbf{OP}\ f\ \textbf{;}\ c,e,\textbf{val}\ (\textbf{nat}\ n_1) :: \textbf{val}\ (\textbf{nat}\ n_2) :: s) \xrightarrow[CES]{} (c,e,\textbf{val}\ (\textbf{nat}\ (f\ n_1\ n_2)) :: s)\\
\textbf{COND-0} & : \forall\,\{c\ c'\ e\ s\} \to & (\textbf{COND}\ c\ c',e,\textbf{val}\ (\textbf{nat}\ 0) :: s) \xrightarrow[CES]{} (c,e,s)\\
\textbf{COND-1+n} & : \forall\,\{c\ c'\ e\ n\ s\} \to & (\textbf{COND}\ c\ c',e,\textbf{val}\ (\textbf{nat}\ (1+\ n)) :: s) \xrightarrow[CES]{} (c',e,s)
\end{array}
$$

Figure 3. The definition of the transition relation of the CES machine.

APPL instruction and the **RET** instruction. When performing an application, two values are required on the stack, one of which has to be a closure. The machine enters the closure, adding the value to the environment, and pushes a return continuation on the stack. The code inside a closure will be terminated by a **RET** instruction, so once the machine has finished executing the closure (and thus produced a value on the stack), that value is returned to the continuation. Note that it will be useful to establish that the CES machine is deterministic.[1]

Example 3.2. We trace the execution of Ex. 3.1 defined above, which exemplifies how returning from an application works. Here we write $a \xrightarrow[CES]{} \langle\ x\ \rangle\ b$ meaning that the machine uses rule x to transition from a to b.

$$
\begin{array}{l}
\mathsf{let}\ c_1 \;=\; \textbf{VAR}\ 0\ \textbf{;}\ \textbf{RET}\\
\quad\ c_2 \;=\; \textbf{CLOS}\ (\textbf{VAR}\ 1\ \textbf{;}\ \textbf{RET})\ \textbf{;}\ \textbf{RET}\\
\quad\ cl_1 \;=\; \textbf{val}\ (\textbf{clos}\ (c_1,[\,]));cl_2 \;=\; \textbf{val}\ (\textbf{clos}\ (c_2,[\,]))\\
\mathsf{in}\ (\textbf{CLOS}\ c_1\ \textbf{;}\ \textbf{CLOS}\ c_2\ \textbf{;}\ \textbf{APPL}\ \textbf{;}\ \textbf{END},[\,],[\,])\\
\xrightarrow[CES]{} \langle\ \textbf{CLOS}\ \rangle \quad (\textbf{CLOS}\ c_2\ \textbf{;}\ \textbf{APPL}\ \textbf{;}\ \textbf{END},[\,],[cl_1])\\
\xrightarrow[CES]{} \langle\ \textbf{CLOS}\ \rangle \quad (\textbf{APPL}\ \textbf{;}\ \textbf{END},[\,],[cl_2,cl_1])\\
\xrightarrow[CES]{} \langle\ \textbf{APPL}\ \rangle \quad (\textbf{VAR}\ 0\ \textbf{;}\ \textbf{RET},[cl_2],[\textbf{cont}\ (\textbf{END},[\,])])\\
\xrightarrow[CES]{} \langle\ \textbf{VAR}\ \mathit{refl}\ \rangle\ (\textbf{RET},[cl_2],[cl_2,\textbf{cont}\ (\textbf{END},[\,])])\\
\xrightarrow[CES]{} \langle\ \textbf{RET}\ \rangle \quad (\textbf{END},[\,],[cl_2])
\end{array}
$$

The final result is therefore the second closure, cl_2.

The CES machine *terminates with a value* v, written $cfg \downarrow_{CES} v$ if it, through the reflexive transitive closure of $\xrightarrow[CES]{}$, reaches the end of its code fragment with an empty environment, and v as its sole stack element. It *terminates*, written $cfg \downarrow_{CES}$ if there exists a value v such that it terminates with the value v. It *diverges*, written $cfg \uparrow_{CES}$ if it is possible to take another step from any configuration reachable from the reflexive transitive closure of $\xrightarrow[CES]{}$.

We do not prove formally that the compilation of CBV-PCF to the CES machine is correct, as it is a standard result [12].

3.2 CESH: A heap machine

In a compiler implementation of the CES machine targeting a low-level language, closures have to be dynamically allocated in a heap. However, the CES machine does not make this dynamic allocation explicit. Now we make it explicit in a new machine, called the CESH, which is a CES machine with an extra heap component in its configuration.

While heaps are not strictly necessary for a *presentation* of the CES machine, they are of great importance to us. The distributed machine that we will later define needs heaps for persistent storage of data, and the CESH machine forms an intermediate step between that and the CES machine. A CESH configuration is defined as

$$\mathit{Config} \;=\; \mathit{Code} \times \mathit{Env} \times \mathit{Stack} \times \mathit{Heap}\ \mathit{Closure}$$

where *Heap* is a type constructor for heaps parameterised by the type of its content. The only difference in the definition of the configuration constituents, compared to the CES machine, is that a closure value (the **clos** constructor of the *Value* type) does not contain an actual closure, but just a pointer (*Ptr*). The stack is as in the CES machine.

Fig. 4 shows those rules of the CESH machine that are significantly different than the CES: **CLOS** and **APPL**. To build a closure, the CESH allocates it in the heap, using the _►_ function, which, given a heap and an element, gives back an updated heap and a pointer to the element. When performing an application, the machine has a *pointer* to a closure, so it looks it up in the heap using the _!_ function, which, given a heap and a pointer, gives back the element that the pointer points to (if it exists).

A CESH configuration *cfg* can *terminate with a value* v, written as $cfg \downarrow_{CESH} v$, *terminate* ($cfg \downarrow_{CESH}$), or *diverge* ($cfg \uparrow_{CESH}$). These are analogous to the definitions for the CES machine, except that the CESH machine is allowed to terminate with *any* heap.

3.2.1 Correctness

To show that our definition of the machine is correct, we construct a bisimulation between the CES and CESH, which given the similarity between the two machines, is almost equality. The difference is dealing with closure values, since the CESH stores pointers rather than closures. The relation for closure values must be parameterised by the heap of the CESH configuration, where the (dereferenced) value of the closure pointer is related to the CES closure.

Formally, the relation is constructed separately for the different components of the machine configurations. For bytecode it is equality, and for closures it is defined component-wise. Values are related only if they have the same head constructor and related constituents: if the two values are number literals, they are related if they are equal; a CES closure and a pointer are related only if the pointer leads to a CESH closure that is in turn related to the CES closure. Environments are related if they have the same list spine and their values are pointwise related. The relation on stacks is defined similarly, using the relation on values and continuations. Finally, two configurations are R_{Cfg}-related if their components are related.

In the formalisation we define heaps and their properties *abstractly*, rather than using a specific heap implementation. The first key property we require is that dereferencing a pointer in a heap where that pointer was just allocated with a value gives back the same value:

$$\forall\ h\ x \to \mathsf{let}\ (h',ptr) \;=\; h \blacktriangleright x\ \mathsf{in}\ h'\ !\ ptr \equiv \textbf{just}\ x$$

We will require a preorder $\subseteq$ for *sub-heaps*. The intuitive reading for $h \subseteq h'$ is that h' can be used where h can, i.e. that h' contains at least the allocations of h. The second key property that we require

[1] *CES.Properties.determinism-CES*

$$\textbf{CLOS} \;:\; \forall\,\{c'\ c\ e\ s\ h\} \to \mathsf{let}\ (h', ptr_{cl}) \;=\; h \blacktriangleright (c', e)\ \mathsf{in}\ (\textbf{CLOS}\ c'\ ;\ c, e, s, h) \xrightarrow[CESH]{} (c, e, \textbf{val}\ (\textbf{clos}\ ptr_{cl}) :: s, h')$$

$$\textbf{APPL} \;:\; \forall\,\{c\ e\ v\ ptr_{cl}\ c'\ e'\ s\ h\} \to h\ !\ ptr_{cl} \equiv \textbf{just}\ (c', e') \to (\textbf{APPL}\ ;\ c, e, \textbf{val}\ v :: \textbf{val}\ (\textbf{clos}\ ptr_{cl}) :: s, h) \xrightarrow[CESH]{} (c', v :: e', \textbf{cont}\ (c, e) :: s, h)$$

Figure 4. The definition of the transition relation of the CESH machine (excerpt).

of a heap implementation is that allocation does not overwrite any previously allocated memory cells ($proj_1$ means first projection):

$$\forall\ h\ x \to h \subseteq proj_1\ (h \blacktriangleright x)$$

For any two heaps h and h' such that $h \subseteq h'$, if $R_{Cfg}\ cfg\ (c, e, s, h)$, then $R_{Cfg}\ cfg\ (c, e, s, h')$.[2]

Our first correctness result is that R_{Cfg} is a simulation relation.[3] The proof is by cases on the *CES* transition, and, in each case, the *CESH* machine can make analogous transitions. The property mentioned above is then used to show that R_{Cfg} is preserved.

It is helpful to introduce the notion of a *presimulation* relation, defined as:

$$\begin{aligned}&\mathit{Presimulation}\ \_\longrightarrow\_\ \_\longrightarrow'\_\ \_R\_\ =\\ &\quad \forall\ a\ a'\ b \to (a \longrightarrow a') \to a\ R\ b \to \exists\ \lambda\ b' \to (b \longrightarrow' b')\end{aligned}$$

Then, the inverse of R_{Cfg} is a presimulation.[4] In general, if R is a simulation between relations $\longrightarrow$ and $\longrightarrow'$, R^{-1} is a presimulation, and $\longrightarrow'$ is deterministic at states b related to some a, then R^{-1} is a simulation,[5] from which it follows that R_{Cfg} is a bisimulation, because we have already established that the CESH is deterministic. In particular, if $R_{Cfg}\ cfg_1\ cfg_2$ then $cfg_1\ \downarrow_{CES}\ \textbf{nat}\ n \leftrightarrow cfg_2\ \downarrow_{CESH}\ \textbf{nat}\ n$ and $cfg_1\ \uparrow_{CES}\ \leftrightarrow cfg_2\ \uparrow_{CESH}$.[6]

To finalise the proof we note that there are configurations in R_{Cfg}. One such example is the initial configuration for a fragment of code: For any c, we have $R_{Cfg}\ (c, [\,], [\,])\ (c, [\,], [\,], \emptyset)$ (where $\emptyset$ is the empty heap).

3.3 Network models

In this section we define models for networks with synchronous and asynchronous communication, that are parameterised by an underlying labelled transition system. Both kinds of networks are modelled by two-level transition systems, which is common in operational semantics for concurrent and parallel languages. A *global level* describes the transitions of the system as a whole, and a *local level* the local transitions of the nodes in the system. Synchronous communication is modelled by *rendezvous*, i.e. two nodes have to be ready to send and receive a message at a single point in time. Asynchronous communication is modelled using a "message soup", representing messages currently in transit, that nodes can add and remove messages from, reminiscent of the Chemical Abstract Machine [5].

The model (Agda module *Network*) is parameterised by the underlying transition relation of the machines $\_ \vdash \_ \xrightarrow[Machine]{\_} \_$. The sets *Node*, *Machine*, and *Msg* are additional parameters. Elements of *Node* will act as node identifiers, and we assume that these enjoy decidable equality.[7] The type *Machine* is the type of the nodes' configurations, and *Msg* the type of messages that the machines can send. The presence of the *Node* argument means that the configuration of a node may know about and can depend on its own identifier. The type constructor *Tagged* is used to separate different kinds of local transitions: A *Tagged Msg* can be τ (i.e. a silent transition), **send** *msg*, or **receive** *msg* (for $msg : Msg$).

A synchronous network (*SyncNetwork*) is an indexed family of machines, $Node \to Machine$, representing the nodes of the system. An asynchronous network (*AsyncNetwork*) is an indexed family of machines together with a list of pending messages $(Node \to Machine) \times List\ Msg$.

Fig. 5 shows the definition of the transition relation for synchronous and asynchronous networks. It is uses *update*, a function that corresponds to the usual function update (often written $(f \mid x \mapsto y)$) which updates an element of an indexed family (here relying on the decidable equality of node identifiers).

There are two ways for a synchronous network to make a transition. The first, **silent-step**, occurs when a machine in the network makes a transition tagged with τ, and is allowed at any time. The second, **comm-step**, is the aforementioned rendezvous. A node s first takes a step sending a message, and afterwards a node r (which can be s) takes a step receiving the same message. Asynchronous networks only have one rule, **step**, which can be used if a node steps with a tagged message that "agrees" with the pending messages. If the node *receives* a message, the message has to be in the list *before* the transition. If the node *sends* a message, it has to be there *after*. If the node takes a *silent* step, the list stays the same before and after.[8]

Asynchronous networks actually subsume synchronous networks.[9] Going in the other direction is not possible in general, but for some specific instances of the underlying transition relation it is, as we will see later.

3.4 DCESH$_1$: A trivially distributed machine

In higher-order distributed programs containing location specifiers, we will sometimes encounter situations where a function is not available locally. For example, when evaluating the function f in the term $(f\ @\ A)\ (g\ @\ B)$, we may need to apply the remotely available function g. Our general idea is to do this by decomposing some instructions into communication. In the example, the function f may send a message requesting the evaluation of g, meaning that the **APPL** instruction is split into a pair of instructions: **APPL-send** and **APPL-receive**.

This section outlines an abstract machine, called DCESH$_1$, which decomposes all application and return instructions into communication. The machine is trivially distributed, because it runs as the sole node in a network, sending messages only to itself. Although it is not used as an intermediate step for the proofs, it is included because it illustrates this decomposition.

A configuration of the DCESH$_1$ machine (*Machine*) is a tuple consisting of a possibly running thread (*Maybe Thread*), a closure heap (*Heap Closure*), and a "continuation heap" (*Heap* (*Closure* $\times$ *Stack*)). Since the language is sequential we have at most one thread running at once. The thread resembles a CES configuration, $Thread = Code \times Env \times Stack$, but stacks are defined differently. A stack is now a list of values paired with an optional pointer (into the continuation heap), $Stack = List\ Val \times Maybe\ ContPtr$ (*ContPtr* is a synonym for *Ptr*). When performing an application, when CES would push a continuation on the stack, the DCESH$_1$

[2] *CESH.Simulation.HeapUpdate.config*

[3] *CESH.Simulation.simulation*

[4] *CESH.Presimulation.presimulation*

[5] *Relation.presimulation-to-simulation*

[6] *CESH.Bisimulation.termination-agrees*, *CESH.Bisimulation.divergence-agrees*

[7] In MPI, they would correspond to the so called integer "node ranks".

[8] This is formalised using a function called *detag*, which creates lists of input and output messages from a tagged message.

[9] *Network.*$\longrightarrow$*Sync-to-*$\longrightarrow$*Async*$^+$

$$\textbf{silent-step} \;:\; \forall\,\{i\ m'\} \to i \vdash nodes\ i \xrightarrow[Machine]{\tau} m' \to nodes \xrightarrow[Sync]{} update\ nodes\ i\ m'$$

$$\textbf{comm-step} \;:\; \forall\,\{s\ r\ msg\ sender'\ receiver'\} \to \mathsf{let}\ nodes' = update\ nodes\ s\ sender'\ \mathsf{in}$$
$$s \vdash nodes\ s \xrightarrow[Machine]{\textbf{send}\ msg} sender' \to r \vdash nodes'\ r \xrightarrow[Machine]{\textbf{receive}\ msg} receiver' \to nodes \xrightarrow[Sync]{} update\ nodes'\ r\ receiver'$$

$$\textbf{step} \;:\; \forall\,\{nodes\}\ msgs_l\ msgs_r\ \{tmsg\ m'\ i\} \to \mathsf{let}\ (msgs_{in}, msgs_{out}) = detag\ tmsg\ \mathsf{in}$$
$$i \vdash nodes\ i \xrightarrow[Machine]{tmsg} m' \to (nodes, msgs_l +\!\!+ msgs_{in} +\!\!+ msgs_r) \xrightarrow[Async]{} (update\ nodes\ i\ m', msgs_l +\!\!+ msgs_{out} +\!\!+ msgs_r)$$

Figure 5. The definition of the transition relations for synchronous and asynchronous networks.

machine is going to stop the current thread and send a message, which means that it has to save the continuation and the remainder of the stack in the heap for them to persist the thread's lifetime.

The optional pointer in *Stack* is an element at the *bottom* of the list of values. Comparing it to the definition of the CES machine, where stacks are lists of either values or continuations (just closures), we can picture their relation: Whereas the CES machine stores the values and continuations in a single, contiguous stack, the DCESH_1 machine stores first a contiguous block of values until reaching a continuation, at which point it stores (**just**) a pointer to the continuation closure and the rest of the stack.

The definition of closures, values, and environments are otherwise just like in the CESH machine. The machine communicates with itself using two kinds of messages, **APPL** and **RET**, corresponding to the instructions that we are replacing with communication.

Fig. 6 defines the transition relation for the DCESH_1 machine, written $m \xrightarrow{tmsg} m'$ for a tagged message *tmsg* and machine configurations m and m'. Most transitions are the same as in the CESH machine, just framed with the additional heaps and the **just** meaning that the thread is running. We elide them for brevity.

The interesting rules are the decomposed application and return rules. When an application is performed, an **APPL** message containing a pointer to the closure to apply, the argument value and a pointer to a return continuation (which is first allocated) is sent, and the thread is stopped (**nothing**). We call such a machine *inactive*. The machine can receive an application message if the thread is not running. When that happens, the closure pointer is dereferenced and entered, adding the received argument to the environment. The stack is left empty apart from the continuation pointer of the received message. When returning from a function application, the machine sends a return message containing the continuation pointer and the value to return.

On the receiving end of that communication, it dereferences the continuation pointer and enters it, putting the result value on top of the stack.

Example 3.3. We trace the execution of Ex. 3.1 in a synchronous network of nodes indexed by the unit type. Heaps with pointer mappings are written $\{ptr \mapsto element\}$. The last list shown in each step is the message list of the asynchronous network.

$\mathsf{let}\ h_{cl} = \{ptr_1 \mapsto (c_1, [])\}$
$\quad h'_{cl} = \{ptr_1 \mapsto (c_1, []), ptr_2 \mapsto (c_2, [])\}$
$\quad h_{cnt} = \{ptr_{cnt} \mapsto ((\textbf{END}, []), [], \textbf{nothing})\}$
$\mathsf{in}\ (\textbf{just}\ (\textbf{CLOS}\ c_1\ \textbf{;}\ \textbf{CLOS}\ c_2\ \textbf{;}\ \textbf{APPL}\ \textbf{;}\ \textbf{END}, [], [], \textbf{nothing}), \emptyset, \emptyset), []$
$\longrightarrow\langle\ \textbf{step CLOS}\ \rangle$
$(\textbf{just}\ (\textbf{CLOS}\ c_2\ \textbf{;}\ \textbf{APPL}\ \textbf{;}\ \textbf{END}, [], [\textbf{clos}\ ptr_1], \textbf{nothing}), h_{cl}, \emptyset), []$
$\longrightarrow\langle\ \textbf{step CLOS}\ \rangle$
$(\textbf{just}\ (\textbf{APPL}\ \textbf{;}\ \textbf{END}, [], [\textbf{clos}\ ptr_2, \textbf{clos}\ ptr_1], \textbf{nothing}), h'_{cl}, \emptyset), []$
$\longrightarrow\langle\ \textbf{step APPL-send}\ \rangle$
$(\textbf{nothing}, h'_{cl}, h_{cnt}), [\textbf{APPL}\ ptr_1\ (\textbf{clos}\ ptr_2)\ ptr_{cnt}]$
$\longrightarrow\langle\ \textbf{step APPL-receive}\ \rangle$
$(\textbf{just}\ (\textbf{VAR}\ 0\ \textbf{;}\ \textbf{RET}, [\textbf{clos}\ ptr_2], [], \textbf{just}\ ptr_{cnt}), h'_{cl}, h_{cnt}), []$
$\longrightarrow\langle\ \textbf{step}\ (\textbf{VAR}\ refl)\ \rangle$
$(\textbf{just}\ (\textbf{RET}, [\textbf{clos}\ ptr_2], [\textbf{clos}\ ptr_2], \textbf{just}\ ptr_{cnt}), h'_{cl}, h_{cnt}), []$
$\longrightarrow\langle\ \textbf{step RET-send}\ \rangle$
$(\textbf{nothing}, h'_{cl}, h_{cnt}), [\textbf{RET}\ ptr_{cnt}\ (\textbf{clos}\ ptr_2)]$
$\longrightarrow\langle\ \textbf{step RET-receive}\ \rangle$
$(\textbf{just}\ (\textbf{END}, [], [\textbf{clos}\ ptr_2], \textbf{nothing}), h'_{cl}, h_{cnt}), []$

Comparing this to Example 3.2 we can see that an **APPL-send** followed by an **APPL-receive** amounts to the same thing as the **APPL** rule in the CES machine, and similarly for the **RET** instruction.

3.5 DCESH: The distributed CESH machine

We have so far seen two refinements of the CES machine. We have seen CESH, that adds heaps, and DCESH_1, that decomposes instructions into communication in a degenerate network of only one node. Our final refinement is a distributed machine, DCESH, that supports multiple nodes. The main problem that we now face is that there is no centralised heap, but each node has its own local heap. This means that, for supporting higher-order functions across node boundaries, we have to somehow keep references to closures in the heaps of *other* nodes. Another problem is efficiency; we would like a system where we do not pay the higher price of communication for locally running code. The main idea for solving these two problems is to use *remote pointers*, $RPtr = Ptr \times Node$, pointers paired with node identifiers signifying on what node's heap the pointer is located. This solves the heap problem because we always know where a pointer comes from. It can also be used to solve the efficiency problem since we can choose what instructions to run based on whether a pointer is local or remote. If it is local, we run the rules of the CESH machine. If it is remote, we run the decomposed rules of the DCESH_1 machine.

The final extension to the term language and bytecode will add support for location specifiers. We add a term construct t **@** i, and an instruction **REMOTE** $c\ i$ for its compilation. The location specifiers, t **@** i, are taken to mean that the term t should be evaluated on node i. For compilation, we require that the terms t in all location specification sub-terms t **@** i are *closed*. Terms where this does not hold are transformed automatically using lambda lifting [25] (transform every sub-term t **@** i to $t' = ((\lambda\ fv\ t.\ t)$ **@** $i)\ (fv\ t)$). The **REMOTE** $c\ i$ instruction will be used to start running a code fragment c on node i in the network. We also extend the *compile'* function to handle the new term construct:

$$compile'\ (t\ \textbf{@}\ i)\ c = \textbf{REMOTE}\ (compile'\ t\ \textbf{RET})\ i\ \textbf{;}\ c$$

Note that we reuse the **RET** instruction to return from a remote computation.

The definition of closures, values, environments and closure heaps are the same as in the CESH machine, but using *RPtr* instead of *Ptr* for closure pointers.

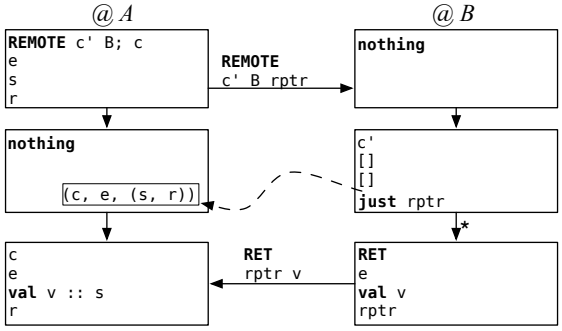
The stack combines the functionality of the CES(H) machine, permitting local continuations, with that of the DCESH_1 machine, making it possible for a stack to end with a continuation on another node. A stack element is a value or a (local) continuation signified by the **val** and **cont** constructors. A stack (*Stack*) is a list of stack elements, possibly ending with a (remote) pointer to a continuation, *List StackElem* $\times$ *Maybe ContPtr* (where $ContPtr = RPtr$). Threads

$$\textbf{APPL-send} \quad : \forall \{c\ e\ v\ ptr_{cl}\ s\ r\ h_{cl}\ h_{cnt}\} \to \mathsf{let}\ (h'_{cnt}, ptr_{cnt}) = h_{cnt} \blacktriangleright ((c,e),s,r)\ \mathsf{in}$$
$$(\textbf{just}\ (\textbf{APPL ;}\ c,e,v :: \textbf{clos}\ ptr_{cl} :: s, r), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{APPL}\ ptr_{cl}\ v\ ptr_{cnt})} (\textbf{nothing}, h_{cl}, h'_{cnt})$$
$$\textbf{APPL-receive} \ : \forall \{h_{cl}\ h_{cnt}\ ptr_{cl}\ v\ ptr_{cnt}\ c\ e\} \to h_{cl}\ !\ ptr_{cl} \equiv \textbf{just}\ (c,e) \to$$
$$(\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{APPL}\ ptr_{cl}\ v\ ptr_{cnt})} (\textbf{just}\ (c, v :: e, [\,], \textbf{just}\ ptr_{cnt}), h_{cl}, h_{cnt})$$
$$\textbf{RET-send} \quad : \forall \{e\ v\ ptr_{cnt}\ h_{cl}\ h_{cnt}\} \to$$
$$(\textbf{just}\ (\textbf{RET}, e, v :: [\,], \textbf{just}\ ptr_{cnt}), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{RET}\ ptr_{cnt}\ v)} (\textbf{nothing}, h_{cl}, h_{cnt})$$
$$\textbf{RET-receive} \quad : \forall \{h_{cl}\ h_{cnt}\ ptr_{cnt}\ v\ c\ e\ s\ r\} \to h_{cnt}\ !\ ptr_{cnt} \equiv \textbf{just}\ ((c,e),s,r) \to$$
$$(\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{RET}\ ptr_{cnt}\ v)} (\textbf{just}\ (c,e,v :: s, r), h_{cl}, h_{cnt})$$

Figure 6. The definition of the transition relation of the DCESH_1 machine (excerpt).

and machines are defined like in the DCESH_1 machine. The messages that DCESH can send are those of the DCESH_1 machine but using remote pointers instead of plain pointers, plus a message for starting a remote computation, **REMOTE** *c i* $rptr_{cnt}$. Note that sending a **REMOTE** message amounts to sending code in our formalisation, which is something that we would not like to do. However, because no code is generated at run-time, every machine can be "pre-loaded" with all the bytecode it needs, and the message only needs to contain a *reference* to a fragment of code.

Fig. 7 defines the transition relation of the DCESH machine, written $i \vdash m \xrightarrow{tmsg} m'$ for a node identifier i, a tagged message *tmsg* and machine configurations m and m'. The parameter i is taken to be the identifier of the node on which the transition is taking place. For local computations, we have rules analogous to those of the CESH machine, so we omit them and show only those for remote computations. The rules use the function $i \vdash h \blacktriangleright x$ for allocating a pointer to x in a heap h and then constructing a remote pointer tagged with node identifier i from it. When starting a remote computation, the machine allocates a continuation in the heap and sends a message containing the code and continuation pointer to the remote node in question. Afterwards the current thread is stopped.



On the receiving end of such a communication, a new thread is started, placing the continuation pointer at the bottom of the stack for the later return to the caller node. To run the apply instruction when the function closure is remote, i.e. its location is *not* equal to the current node, the machine sends a message containing the closure pointer, argument value, and continuation, like in the DCESH_1 machine. On the other end of such a communication, the machine dereferences the pointer and enters the closure with the received value. The bottom remote continuation pointer is set to the received continuation pointer. After either a remote invocation or a remote application, the machine can return if it has produced a value on the stack and has a remote continuation at the bottom of the stack. To do this, a message containing the continuation pointer and the return value is sent to the location of the continuation pointer. When receiving a return message, the continuation pointer is dereferenced and entered with the received value.

A network of abstract machines is obtained by instantiating the *Network* module with the $\longrightarrow$*Machine* relation. From here on *SyncNetwork* and *AsyncNetwork* and their transition relations refer to the instantiated versions.

Unsurprisingly, if all nodes in a synchronous network except one are inactive, then the next step is deterministic.[10] Another key ancillary property of DCESH nets is that synchronous or asynchronous networks for single threaded computations behave essentially the same,[11] which means it is enough to deal with the simpler synchronous networks.

DCESH nets *nodes* can *terminate with a value* v ($nodes \downarrow_{Sync} v$), *terminate* ($nodes \downarrow_{Sync}$), or *diverge* ($nodes \uparrow_{Sync}$). A network terminates with a value v if it can step to a network where only one node is active, and that node has reached the **END** instruction with the value v on top of its stack. The other definitions are analogous to those of the CES(H) machine.

3.5.1 Correctness

To prove the correctness of the machine, we will now establish a bisimulation between the CESH and the DCESH machines.

To simplify this development, we extend the CESH machine with a dummy rule for the **REMOTE** *c i* instruction so that both machines run the same bytecode. This rule is almost a no-op, but since we are assuming that the code we run remotely is closed, the environment is emptied, and since the compiled code c will end in a **RET** instruction a return continuation is pushed on the stack.

$$(\textbf{REMOTE}\ c'\ i\ ;\ c,e,s,h) \xrightarrow[CESH]{} (c', [\,], \textbf{cont}\ (c,e) :: s, h)$$

The relation that we are about to define is, as before, *almost* equality. But since values may be pointers to closures, it must be parameterised by heaps. A technical problem is that *both* machines use pointers, and the DCESH machine also uses *remote* pointers and has two heaps for each node: so the relation must be parameterised by all the heaps in the system. The extra parameter is a synonym for an indexed family of the closure and continuation heaps, $Heaps = Node \to DCESH.ClosHeap \times DCESH.ContHeap$. The complexity of this relation justifies our use of mechanised reasoning.

The correctness proof itself is not routine. Simply following the recipe that we used before does not work. In the old proof, there can be no circularity, since that bisimulation was constructed inductively on the structure of the CES configuration. But now both systems, CESH and DCESH, have heaps where there is a potential for circular references (e.g. a closure, residing in a heap, whose environment contains a pointer to itself), preventing a direct proof via structural induction. This is perhaps the most mathematically (and formally) challenging point of the paper. The solution lies in using the technique of *step-indexed relations*, adapted to the context

[10] *DCESH.Properties.determinism-Sync*

[11] *DCESH.Properties.*$\longrightarrow Async^+$*-to-*$\longrightarrow Sync^+$

$$\textbf{REMOTE-send} \;:\; \forall\,\{c'\ i'\ c\ e\ s\ r\ h_{cl}\ h_{cnt}\} \to \mathsf{let}\ (h'_{cnt}, rptr) \;=\; i \vdash h_{cnt} \blacktriangleright ((c,e),s,r)\ \mathsf{in}$$
$$i \vdash (\textbf{just}\ (\textbf{REMOTE}\ c'\ i'\ ;\ c,e,s,r),h_{cl},h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{REMOTE}\ c'\ i'\ rptr)} (\textbf{nothing},h_{cl},h'_{cnt})$$
$$\textbf{REMOTE-receive} \;:\; \forall\,\{h_{cl}\ h_{cnt}\ c\ rptr_{cnt}\} \to$$
$$i \vdash (\textbf{nothing},h_{cl},h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{REMOTE}\ c\ i\ rptr_{cnt})} (\textbf{just}\ (c,[\,],[\,],\textbf{just}\ rptr_{cnt}),h_{cl},h_{cnt})$$
$$\textbf{APPL-send} \;:\; \forall\,\{c\ e\ v\ ptr_{cl}\ j\ s\ r\ h_{cl}\ h_{cnt}\} \to i \not\equiv j \to \mathsf{let}\ (h'_{cnt}, rptr_{cnt}) \;=\; i \vdash h_{cnt} \blacktriangleright ((c,e),s,r)\ \mathsf{in}$$
$$i \vdash (\textbf{just}\ (\textbf{APPL}\ ;\ c,e,\textbf{val}\ v :: \textbf{val}\ (\textbf{clos}\ (ptr_{cl},j)) :: s,r),h_{cl},h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{APPL}\ (ptr_{cl},j)\ v\ rptr_{cnt})} (\textbf{nothing},h_{cl},h'_{cnt})$$
$$\textbf{APPL-receive} \;:\; \forall\,\{h_{cl}\ h_{cnt}\ ptr_{cl}\ v\ rptr_{cnt}\ c\ e\} \to h_{cl}\ !\ ptr_{cl} \equiv \textbf{just}\ (c,e) \to$$
$$i \vdash (\textbf{nothing},h_{cl},h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{APPL}\ (ptr_{cl},i)\ v\ rptr_{cnt})} (\textbf{just}\ (c,v :: e,[\,],\textbf{just}\ rptr_{cnt}),h_{cl},h_{cnt})$$
$$\textbf{RET-send} \;:\; \forall\,\{e\ v\ rptr_{cnt}\ h_{cl}\ h_{cnt}\} \to$$
$$i \vdash (\textbf{just}\ (\textbf{RET},e,\textbf{val}\ v :: [\,],\textbf{just}\ rptr_{cnt}),h_{cl},h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{RET}\ rptr_{cnt}\ v)} (\textbf{nothing},h_{cl},h_{cnt})$$
$$\textbf{RET-receive} \;:\; \forall\,\{h_{cl}\ h_{cnt}\ ptr_{cnt}\ v\ c\ e\ s\ r\} \to h_{cnt}\ !\ ptr_{cnt} \equiv \textbf{just}\ ((c,e),s,r) \to$$
$$i \vdash (\textbf{nothing},h_{cl},h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{RET}\ (ptr_{cnt},i)\ v)} (\textbf{just}\ (c,e,\textbf{val}\ v :: s,r),h_{cl},h_{cnt})$$

Figure 7. The definition of the transition relation of the DCESH machine (excerpt).

of bisimulation relations [2]. The additional *rank* parameter records how many times pointers are allowed to be dereferenced.

The rank is used in defining the relation for closure pointers $R_{rptr_{cl}}$. If the rank is zero, the relation is trivially fulfilled. If the rank is non-zero then three conditions must hold. First, the CESH pointer must point to a closure in the CESH heap; second, the remote pointer of the DCESH network must point to a closure in the heap of the location that the pointer refers to; third, the two closures must be related. The relation for stack elements $R_{StackElem}$ is almost as before, but now requires that the relation is true for *any* natural number *rank*, i.e. for any finite number of pointer dereferencings. The relation for stacks R_{Stack} now takes into account that the DCESH stacks may end in a pointer representing a remote continuation, requiring that the pointer points to something in the continuation heap of the location of the pointer, which is related to the CESH stack element. Finally, a CESH configuration and a DCESH thread are R_{Thread}-related if the thread is running and the constituents are pointwise related. Then a CESH configuration is related to a synchronous network R_{Sync} if the network has exactly one running machine that is related to the configuration.

DCESH net heaps are ordered pointwise (called $\subseteq s$ since it is the "plural" of $\subseteq$). For any CESH closure heaps h and h' such that $h \subseteq h'$ and families of DCESH heaps hs and hs' such that $hs \subseteq s\ hs'$, if $R_{Env}\ n\ h\ hs\ e_1\ e_2$ then $R_{Env}\ n\ h'\ hs'\ e_1\ e_2$ and if $R_{Stack}\ h\ hs\ s_1\ s_2$ then $R_{Stack}\ h'\ hs'\ s_1\ s_2$.[12]

Showing that R_{Sync} is a simulation relation[13] proceeds by cases on the CESH transition. In each case, the DCESH network can make analogous transitions. The property above is then used to show that R_{Sync} is preserved. It is quite immediate that the inverse of R_{Sync} is a presimulation[14] which leads to the main result that R_{Sync} is a bisimulation.[15]

In particular, if $R_{Sync}\ cfg\ nodes$ then $cfg\ \downarrow_{CESH}\ \textbf{nat}\ n \leftrightarrow nodes\ \downarrow_{Sync}\ \textbf{nat}\ n$ and $cfg\ \uparrow_{CESH}\ \leftrightarrow nodes\ \uparrow_{Sync}$,[16] and we also have that initial configurations are in R_{Sync}.[17] These final results complete the picture for the DCESH machine. We have established that we get the same final result regardless of whether we choose to run a fragment of code using the CES, the CESH, or the DCESH machine.

[12] *DCESH.Simulation-CESH.HeapUpdate.env*, *DCESH.Simulation-CESH.HeapUpdate.stack*

[13] *DCESH.Simulation-CESH.simulation-sync*

[14] *DCESH.Simulation-CESH.presimulation-sync*

[15] *DCESH.Simulation-CESH.bisimulation-sync*

[16] *DCESH.Simulation-CESH.termination-agrees-sync*, *DCESH.Simulation-CESH.divergence-agrees-sync*

[17] *DCESH.Simulation-CESH.initial-related-sync*

4. Fault-tolerance via transactions

In this section we present a generic transaction-based method for handling failure which is suitable for the DCESH. Node state is "backed up" (*commit*) at certain points in the execution, and if an exceptional condition arises, the backup is restored (*roll-back*).

This development is independent of the underlying transition relation, but the proofs rely on sequentiality. We assume that we have two arbitrary types *Machine* and *Msg*, as well as a transition relation over them:

$$\_ \xrightarrow[Machine]{\_} \_ \;:\; Machine \to Tagged\ Msg \to Machine \to \star$$

Since we have no knowledge of exceptional states in *Machine*, since it is a parameter, we define another relation, $\_ \xrightarrow[Crash]{\_} \_$, as a thin layer on top of $\_ \xrightarrow[Machine]{\_} \_$. The new definition is shown in Fig. 8 and adds the exceptional state **nothing** by extending the set of states of the relation to *Maybe Machine*. The fallible machine can make a **normal-step** transition from and to **just** ordinary *Machine* states, or it can **crash** which leaves it in the exceptional state. This means that we tolerate fail-stop faults as opposed to e.g. the more general Byzantine failures.

The additional assumptions for sequentiality are that we have a decidable predicate, $active\ :\ Machine \to \star$ on machines, and the following functions:

$$inactive\text{-}receive\text{-}active \;:\; \forall\,\{m\ m'\ msg\} \to$$
$$(m \xrightarrow[Machine]{\textbf{receive}\ msg} m') \to \neg\,(active\ m) \times active\ m'$$
$$active\text{-}silent\text{-}active \;:\; \forall\,\{m\ m'\} \to$$
$$(m \xrightarrow[Machine]{\tau} m') \to active\ m \times active\ m'$$
$$active\text{-}send\text{-}inactive \;:\; \forall\,\{m\ m'\ msg\} \to$$
$$(m \xrightarrow[Machine]{\textbf{send}\ msg} m') \to active\ m \times \neg\,(active\ m')$$

These functions express the property that if a machine is invoked, i.e. it receives a message, then it must go from an inactive to an active state. If the machine then takes a silent step, it must remain active, and when it sends a message it must go back to be-

$$\textbf{normal-step} \;:\; \forall\,\{tmsg\ m\ m'\} \to$$
$$(m \xrightarrow[Machine]{tmsg} m') \to (\textbf{just}\ m \xrightarrow[Crash]{tmsg} \textbf{just}\ m')$$
$$\textbf{crash} \;:\; \forall\,\{m\} \to$$
$$(\textbf{just}\ m \xrightarrow[Crash]{\tau} \textbf{nothing})$$

Figure 8. The definition of the transition relation of a machine that may crash.

$$
\begin{array}{l}
\textbf{silent-step} \;:\; \forall\,\{m\ n\ m'\} \to \\
\quad (\textbf{just}\ m \xrightarrow[Crash]{\tau} \textbf{just}\ m') \quad \to ((m,n) \xrightarrow[Backup]{\tau} (m',n)) \\
\textbf{receive-step} \;:\; \forall\,\{m\ n\ m'\ msg\} \to \\
\quad (\textbf{just}\ m \xrightarrow[Crash]{\textbf{receive}\ msg} \textbf{just}\ m') \to ((m,n) \xrightarrow[Backup]{\textbf{receive}\ msg} (m',m')) \\
\textbf{send-step} \;:\; \forall\,\{m\ n\ m'\ msg\} \to \\
\quad (\textbf{just}\ m \xrightarrow[Crash]{\textbf{send}\ msg} \textbf{just}\ m') \quad \to ((m,n) \xrightarrow[Backup]{\textbf{send}\ msg} (m',m')) \\
\textbf{recover} \;:\; \forall\,\{m\ n\} \to \\
\quad (\textbf{just}\ m \xrightarrow[Crash]{\tau} \textbf{nothing}) \quad \to ((m,n) \xrightarrow[Backup]{\tau} (n,n))
\end{array}
$$

Figure 9. The definition of the transition relation for a crashing machine with backup.

ing inactive. This gives us sequentiality; a machine cannot fork new threads, and cannot be invoked several times in parallel.

As the focus here is on obvious correctness and simplicity, we abstract from the method of detecting faults in a separate node, and assume that it can be done (using e.g. a heartbeat network). Similarly, we assume that we have a means of creating and restoring a backup of a node in the system; how this is done depends largely on the underlying system. We so define a machine with a backup as $\mathit{Backup} = \mathit{Machine} \times \mathit{Machine}$, where the second *Machine* denotes the backup. Using this definition, we define a backup strategy, given in Fig. 9. This strategy makes a backup just after sending and receiving messages. In the case of the underlying machine crashing, it restores the backup. Note that this is only one of many possible backup strategies. This one is particularly nice from a correctness point-of-view, because it makes a backup after every observable event, although it may not be the most performant.

We define binary relations for making transition with *some* tagged message, as follows:

$$
\begin{array}{l}
\_ \xrightarrow[Machine]{} \_ \;:\; \mathit{Machine} \to \mathit{Machine} \to \star \\
m_1 \xrightarrow[Machine]{} m_2 \;=\; \exists\,\lambda\, tmsg \to (m_1 \xrightarrow[Machine]{tmsg} m_2) \\
\_ \xrightarrow[Backup]{} \_ \;:\; \mathit{Backup} \to \mathit{Backup} \to \star \\
b_1 \xrightarrow[Backup]{} b_2 \;=\; \exists\,\lambda\, tmsg \to (b_1 \xrightarrow[Backup]{tmsg} b_2)
\end{array}
$$

Using these relations we can define the observable trace of run of a *Machine* (*Backup*), i.e. an element of the reflexive transitive closure of the above relations. First we define *IO*, the subset of tagged messages that we can observe, namely **send** and **receive**:

$$
\begin{array}{l}
\mathsf{data}\ \mathit{IO}\ (A \;:\; \star) \;:\; \star\ \mathsf{where} \\
\quad \textbf{send}\ \textbf{receive} \;:\; A \to \mathit{IO}\ A
\end{array}
$$

The following function now gives us the observable trace, given an element of $\xrightarrow[Machine]{}{}^*$ (which is defined using list-like notation) by ignoring any silent steps.

$$
\begin{array}{l}
[\![\_]\!]_M \;:\; \forall\,\{m_1\ m_2\} \to m_1 \xrightarrow[Machine]{}{}^* m_2 \to \mathit{List}\ (\mathit{IO}\ \mathit{Msg}) \\
[\![\ []\ ]\!]_M \;=\; [\,] \\
[\![\ ((\tau \qquad\quad , \_) :: \mathit{steps})\ ]\!]_M \;=\; [\![\ \mathit{steps}\ ]\!]_M \\
[\![\ ((\textbf{send}\ \ msg, \_) :: \mathit{steps})\ ]\!]_M \;=\; \textbf{send}\ \ msg :: [\![\ \mathit{steps}\ ]\!]_M \\
[\![\ ((\textbf{receive}\ msg, \_) :: \mathit{steps})\ ]\!]_M \;=\; \textbf{receive}\ msg :: [\![\ \mathit{steps}\ ]\!]_M
\end{array}
$$

$[\![\_]\!]_B$ is defined analogously. Given this definition, we can trivially prove that if we have a run $m_1 \xrightarrow[Machine]{}{}^* m_2$ then there exists a run of the *Backup* machine that starts and ends in the same state and has the same observational behaviour.[18] This is proved by constructing a crash-free *Backup* run given the *Machine* run. Obviously, the interesting question is whether we can take any *crashing* run and get a corresponding *Machine* run.

[18] *Backup.soundness*

The key to proving the result that we want, which more formally is that, given $(bs \;:\; (b_1,b_1) \xrightarrow[Backup]{}{}^* (m_2,b_2))$, there is a run $(ms \;:\; b_1 \xrightarrow[Machine]{}{}^* m_2)$ with the same observational behaviour as *bs*[19], is the following lemma[20]:

$$
\begin{array}{l}
\mathit{fast\text{-}forward\text{-}to\text{-}crash} \;:\; \forall\,\{m_1\ m_2\ b_1\ b_2\ n\} \to \\
\quad (s \;:\; (m_1,b_1) \xrightarrow[Backup]{}{}^* (m_2,b_2)) \to \\
\quad \mathit{thread\text{-}crashes}\ s \to \mathit{length}\ s \leqslant n \to \\
\quad \exists\,\lambda\,(s' \;:\; ((b_1,b_1) \xrightarrow[Backup]{}{}^* (m_2,b_2))) \to \\
\qquad (\neg\ \mathit{thread\text{-}crashes}\ s') \times ([\![\ s\ ]\!]_B \equiv [\![\ s'\ ]\!]_B) \times (\mathit{length}\ s' \leqslant n)
\end{array}
$$

Here *thread-crashes* is a decidable property on backup runs, that ensures that, if m_1 is active, then it crashes and does a recovery step at some point before it performs an observable action. The proof of *fast-forward-to-crash* is done by induction on the natural number *n*.

The above result can be enhanced further by observing that if the probability of a machine crash is not 1 then the probability of the machine *eventually* having a successful execution is 1. This means that the probability for the number *n* above to exist is also 1.[21]

5. Related work

There is a multitude of programming languages and libraries for distributed or client server computing. We focus mostly on those with a functional flavour. For surveys, see [28, 42]. Broadly speaking, we can divide them into those that use some form of explicit message passing, and those that have more implicit mechanisms for distribution and communication.

Explicit A prime example of a language for distributed computing that uses explicit message passing is Erlang [3]. Erlang is a very successful language used prominently in the telecommunication industry. Conceptually similar solutions include MPI [21] and Cloud Haskell [14]. The theoretically advanced projects Nomadic Pict [44] and the distributed join calculus [16] both support a notion of mobility for distributed agents, which enables more expressivity for the distribution of a program than the fairly static networks (with only ubiquitous *functions* being mobile) that our work uses. Session types have been used to extend a variety of languages, including functional languages, with better communication primitives [43]. Work on session types has been an inspiration also for us: the way that we compile a single program to multiple nodes can be likened to the projection operator in multiparty session types [24]. But in general, explicit languages are well-proven, but far away in the language design-space from the seamless distributed computing that we envision could be done using native RPC, because they place the significant burden of explicit communication on the programmer.

Implicit Our work generalises Remote Procedure Call (RPC) [7] with full support for higher-order functions. In *loc. cit.* it is argued that emulating a shared address space is infeasible since it requires each pointer to also contain location information, and that it is questionable whether acceptable efficiency can be achieved. These arguments certainly apply to our work, where we do just this. With the goal of expressivity in mind, however, we believe that we should *enable* the programmer to write the potentially inefficient programs that (internally) use remote pointers, because not all programs are performance critical. Furthermore, using a tagged pointer representation [30] for closure pointers means that we can tag pointers that are remote, and pay a very low, if any, performance penalty for local pointers.

[19] *Backup.completeness*

[20] *Backup.fast-forward-to-crash*

[21] This argument has not been formalised in Agda.

Remote Evaluation (REV) [40] is another generalisation of RPC, siding with us on enabling the use of higher-order functions across node boundaries. The main differences between REV and our work is that REV relies on sending code, whereas we can do both, and that it has a more elaborate distribution mechanism.

The well-researched project Eden [29] and the associated abstract machine DREAM [8], which builds on HASKELL, is a semi-implicit language. Eden allows expressing distributed algorithms at a high level of abstraction, and is mostly implicit about communication, but explicit about process creation. Eden is specified operationally using a two-level semantics similar to ours, but in the context of the call-by-need evaluation strategy. Kanor [23] is a project that similarly aims to simplify the development of distributed programs by providing a declarative language for specifying communication patterns inside an imperative host language.

Hop [38], Links [11], and ML5 [32] are examples of so called *tierless* languages that allow writing (for instance) the client and server code of web applications in unified languages with more or less seamless interoperability between them. The Links language shares our goal of unifying distributed programs into a single language with seamless interoperability between the nodes, but its focus is on web programming with client, server, and database. For that purpose it includes sub-languages for elegantly constructing and manipulating XML documents and for doing database queries. The behaviour of Links is specified using the client/server calculus [10], which is an operational semantics similar in purpose to our abstract machines, but, as its name suggests, limited to two nodes. The two nodes of the system are constructed with web-programming in mind and are not equal peers. The server is stateless to be able to handle a large number of clients and unexpected disconnections. The semantics operates on a first-order language and uses explicit substitutions. The client/server calculus also inspired work on the LSAM [33], that lifts the two-node limitation. A similar, but earlier, machine is that of dML [35]. The dML and LSAM machines are conceptually close to our machine, but described on a level that is not as readily implementable as our work, using explicit substitutions and synchronous message passing.

On the language side, our work draws inspiration from abstract machines for game semantics [17, 18] where an exotic compilation technique based on game semantics is used to implement a language like ours, but for call-by-name and without algebraic data types. Recent work shows a formalisation similar to ours, but based on the Krivine machine and the significantly less popular call-by-name evaluation strategy [19].

6. Conclusion

The main conceptual contribution of this work is a new abstract machine, or abstract machine net rather, called the DCESH. Its main feature is that function calls behave, from the point of view of the programmer, in the same way whether they are local or remote. Moreover, on a single node the behaviour of DCESH is that of a conventional SECD-like abstract machine. All correctness proofs have been formalised using Agda.

On this theoretical foundation it is natural to build a compiler for a (very conventional) CBV language where terms are location-aware. Compared to most of the literature on the topic, the thrust of this work is not *a general-purpose functional language for location-aware computing* but rather *a location-aware compilation technique for general-purpose functional languages.* The performance of the compiler, as suggested by the benchmarks, improves dramatically on previous work in terms of single-node behaviour, and the distribution overheads are not onerous.

RPC as a paradigm has been criticised for several reasons: its execution model does not match the native call model, there is no good way of dealing with failure, and it is inherently inefficient [41]. By taking an abstract machine model in which RPCs behave exactly the same as local calls, by showing how a generic transaction mechanism can handle failure, and by implementing a reasonably performant compiler we address all these problem head-on. We believe that we provide enough evidence for general native RPCs to be reconsidered in a more positive light.

6.1 Further work

The DCESH has the internal machinery required for parallel execution, but we restrict ourselves to sequential execution. In moving towards parallelism there are several language design (how to add parallelism?) and theoretical (is compilation still correct?) challenges. The language design aspects are too broad to discuss here, beyond emphasising that the thread-based mechanism of DCESH is indeed quite flexible. The only ingredient lacking is a synchronisation primitive, but that is not a serious difficulty. The theoretical challenges are mainly stemming out of the failure of the equivalence of synchronous and asynchronous networks in the presence of multiple pending messages[22]. Whether we choose to stick to a synchronous network model, which can however be rather unrealistic, or we try to work directly in the more challenging environment of asynchronous networks, remains to be seen.

The current implementation does not need distributed garbage collection. The values which are the result of CBV evaluation are always sent, along with any required closures, to the node where the function using them as arguments is executed. With this approach local garbage collection suffices. Note that this is similar to the approach that Links takes. If a large data structure needs to be held on a particular node the programmer needs to be aware of this requirement and indirect the access to it using functions. However, if we wanted to automate this process as well, and prevent some data from migrating when it's too large, the current approach to garbage collection could not cope, and distributed garbage collection would be required. Mutable references or lazy evaluation would also require it. Whether this can be done efficiently is a separate topic of research [36].

Whether code or data can or should be migrated to different nodes is a question that can be answered from a safety or from an efficiency point of view. The safety angle is very well covered by type systems such as ML5's, which prevent the unwanted exportation of local resources. The efficiency point of view can also be dealt with in a type-theoretic way, as witnessed by recent work in resource-sensitive type systems [9, 20]. The flexibility of the DCESH in terms of localising or remoting the calls (statically or even dynamically) together with a resource oriented system can pave the way towards a highly-convenient automatic orchestration system in which a program is automatically distributed among several nodes to achieve certain performance objectives.

Finally, an Agda formalisation is given only for the abstract machine and its property, which are the new theoretical contributions of the paper. However, a full formalisation of the compiler stack, remains a long-term ambition.

References

[1] Floskel, proofs and compiler implementation. `http://www.cs.bham.ac.uk/~ohf162/floskel.tar.gz`. Last accessed: 3 June 2014.

[2] A. J. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *ICFP*, pages 78–91, 2005.

[3] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN 978-0-13-285792-5.

[4] L. Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.

[22] *DCESH.Properties.* $\longrightarrow Async^{+}$*-to-* $\longrightarrow Sync^{+}$

[5] G. Berry and G. Boudol. The Chemical Abstract Machine. In *POPL*, pages 81–94, 1990.

[6] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1): 37–55, 1990.

[7] A. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[8] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Pena. Dream: The distributed eden abstract machine. In *IFL*, pages 250–269, 1997.

[9] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In Shao [39], pages 351–370. ISBN 978-3-642-54832-1.

[10] E. Cooper and P. Wadler. The RPC calculus. In *PPDP*, pages 231–242, 2009.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO*, pages 266–296, 2006.

[12] O. Danvy and K. Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *LMCS*, 4(4), 2008.

[13] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, pages 381–392, 1972. .

[14] J. Epstein, A. P. Black, and S. L. P. Jones. Towards Haskell in the cloud. In *Symposium on Haskell 2011*, pages 118–129.

[15] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *IFIP TC 2/WG 2.2*, Aug. 1986.

[16] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR*, pages 406–421, 1996.

[17] O. Fredriksson and D. R. Ghica. Seamless Distributed Computing from the Geometry of Interaction. In *TGC*, pages 34–48, 2012.

[18] O. Fredriksson and D. R. Ghica. Abstract Machines for Game Semantics, Revisited. In *LICS*, pages 560–569, 2013.

[19] O. Fredriksson and D. R. Ghica. Krivine Nets: A semantic foundation for distributed execution. In *ICFP*, 2014 (to appear).

[20] D. R. Ghica and A. I. Smith. Bounded linear types in a resource semiring. In Shao [39], pages 331–350. ISBN 978-3-642-54832-1.

[21] W. D. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT Press, 1999.

[22] P. Henderson. *Functional programming - application and implementation*. Prentice Hall International Series in Computer Science. 1980. ISBN 978-0-13-331579-0.

[23] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine. Kanor - a declarative language for explicit communication. In *PADL*, pages 190–204, 2011.

[24] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.

[25] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *FPCA*, pages 190–203, 1985.

[26] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, Jan. 1964.

[27] X. Leroy. MPRI course 2-4-2, part II: abstract machines. 2013-2014. URL `http://gallium.inria.fr/~xleroy/mpri/progfunc/`.

[28] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Pena, S. Priebe, Á. J. R. Portillo, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *HOSC*, 16(3):203–251, 2003.

[29] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *JFP*, 15(3):431–475, 2005.

[30] S. Marlow, A. R. Yakushev, and S. L. P. Jones. Faster laziness using dynamic pointer tagging. In *ICFP*, pages 277–288, 2007.

[31] Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *POPL*, pages 271–283, 1996.

[32] T. Murphy VII, K. Crary, and R. Harper. Type-Safe Distributed Programming with ML5. In *TGC 2007*, pages 108–123.

[33] K. Narita and S.-y. Nishizaki. A parallel abstract machine for the RPC calculus. In *Informatics Engineering and Information Science*, pages 320–332. Springer, 2011.

[34] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Uni. of Tech., 2007.

[35] A. Ohori and K. Kato. Semantics for communication primitives in an polymorphic language. In *POPL*, pages 99–112, 1993.

[36] D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In *IWMM*, pages 211–249, 1995.

[37] G. D. Plotkin. LCF Considered as a Programming Language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[38] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA*, pages 975–985, 2006.

[39] Z. Shao, editor. *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, 2014. Springer. ISBN 978-3-642-54832-1.

[40] J. W. Stamos and D. K. Gifford. Remote evaluation. *TOPLAS*, 12(4): 537–565, 1990.

[41] A. S. Tanenbaum and R. van Renesse. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[42] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *JFP*, 12(4&5):469–510, 2002.

[43] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[44] P. T. Wojciechowski and P. Sewell. Nomadic pict: language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.

Towards Tool Support for History Annotations in Similarity Management

Extended Abstract

Thomas Schmorleiz and Ralf Lämmel

Software Languages Team, Department of Computer Science, University of Koblenz-Landau, Germany

Abstract

When a system is needed in different variants to meet different requirements, then some form of product line engineering may need to be used. In practice, it is often preferred to develop the variants in a loosely coupled fashion as opposed to the regime of a proper ('explicit') product line from which to derive variants by some generative mechanism. For instance, the 101haskell chrestomathy (a subchrestomathy of 101) contains many similar, small, Haskell-based systems that are indeed maintained in loosely coupled fashion. In previous work, we and collaborators have proposed an approach to manage such loosely coupled variants by using a *virtual platform* and cloning-related operators. In this extended abstract, we sketch a concrete method with a supporting tool, Ann, for exploring the similarity of variants and annotating them with metadata accordingly. As a direct result, a *propagate* operator is enabled to automatically propagate changes across variants and to synthesize a to-do list for remaining manual actions. We sketch the method and the tool's application in an ongoing case study for capturing and improving the similarity of the Haskell-based variants of the 101haskell chrestomathy.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features; D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement; F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages

Keywords Haskell. Software Product Line Engineering. Variability Management. Virtual Platform. Ann.

Acknowledgement

The presented work continues previous joint work [1] with Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. The work is also inspired by Julia Rubin's framework for clone management [6].

1. Motivation and background

The corpus of the 101companies project [2] (or just '101') holds a set of variants ('contributions'), all implementing a common feature model. Many of these variants share implementations of some features because their conceptual contribution focuses on the implementation of other features. Thus, cloning of feature implementations is often performed to start the implementation of new variants. While this practice is reasonable in itself, it makes it too hard to understand the similarities of the variants; it also makes it too easy for variants to diverge from each other over time unintentionally. Thus, a form of similarity management is needed. This problem was set up as a challenge for software chrestomathies in [3] and it is, in fact, a challenge in software product line engineering [1].

We have developed a method with a supporting tool, Ann, for exploring the similarity of variants and annotating them with metadata accordingly. This work is directly based on the idea of virtual platform for software product line engineering, as presented in [1]. Our objective is not just to provide the user with information about similarities in a corpus of variants, but also to enable the automatic propagation of changes across variants, and, finally also to reduce overall complexity and unintentional divergence within the corpus. We sketch the method and tool's application in an ongoing case study for capturing and improving the similarity of 101's Haskell segment, i.e., 101haskell [4].

2. A method for variability management

We describe the method by a series of key notions.

2.1 Fragment

We examine similarity of variants and their source-code units (files) at a fragment level. Here is one possible (informal) definition of the fragment notion. That is, a fragment is a range of consecutive lines of source code that correspond to a 'major' node in the associated abstract syntax tree (AST). We assume that syntactic categories for forms of (named) abstractions are favored. Each fragment is identified by a classifier and a name. Classifiers correspond to the syntactic category at hand. For instance, in the case of Haskell, classifiers are 'data', 'type', or 'function'; names are those of data types, type synonyms, or functions.

2.2 Similarity

We compare fragments by adopting an existing approach for detecting near-miss intentional clones [5]. In particular, we pretty-print the source code in a regular manner to lay out compound constructs over several lines so that a simple text-based measure, the diff ratio, can be used for comparison. A similarity measure of '1' means equality. We lift similarity from the fragment level to the levels of

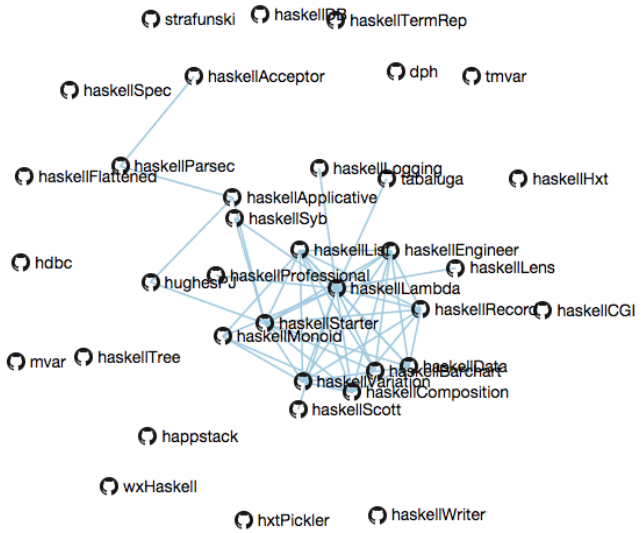


Figure 1. Visualization of similarity across the variants of 101haskell

files, folders, and variants by averaging similarities in a straightforward manner. Similarities are stored as triples of two fragment identifiers with the computed diff ratio.

For illustration, consider Figure 1, which shows the Haskell-based variants of the chrestomathy 101 (i.e., 101haskell [4]). The edges indicate similarity of variants above a certain threshold. The view is computed by the Ann tool.

2.3 Similarity evolution

We track the evolution of similarities between fragments throughout the commit history. To this end, we also need to track the fragments themselves—with regard to variant, file, and fragment renaming. We are specifically interested in two points on the timeline: the point at which a similarity was detected and the current state. Each similarity evolution can be classified according to these *types of evolution*:

Always equal The similarity was always 1.

Eventually equal The similarity was initially below 1, but it is 1 eventually.

Eventually unequal The similarity was initially 1, but it is below 1 eventually.

Always unequal The similarity was always below 1.

2.4 Annotation

Each similarity can be annotated to express the intended treatment of the similarity along evolution. That is, an annotation states how a similarity should be maintained through automated or manual actions. The annotations themselves are to be attached manually, even though defaults may be reasonably assigned in certain cases. The idea is that the user sees the similarities in the order of decreasing similarity (diff ratio), thereby prioritizing annotation of the most similar fragments. There are these *types of annotations*:

Maintain equality An equality at hand should be maintained. This can be done by merging any changes from one fragment to the other, by automated three-way-merge, or possibly by manual conflict resolution.

Maintain similarity A similarity, which is not an equality, should be maintained. A manual action is required, when the similarity (the diff ratio) decreases.

Restore equality A similarity, which is not an equality, should be turned into an equality. In a simple case, either of two fragments may be selected to override the other. It may also be necessary though to change both fragments towards an equal fragment through a manual action.

Increase similarity A similarity, which is not an equality, should be increased, based on the insight that equality is not feasible, while increased similarity is feasible. A manual action is required here.

Ignore similarity The similarity is to be ignored in that it is not reported anymore, when following the diff ratio order.

Whether or not a certain annotation is applicable to a similarity also depends on the type of evolution. For instance, the evolution type 'eventually unequal' cannot be combined with the annotation type 'maintain equality', but it can only be combined with 'restore equality' or all the other types.

2.5 Propagation

Given a repository in a new state with some previously attached similarity annotations, we can determine any sort of similarities that have arrived, increased, decreased, or vanished (assuming some threshold) and whether they have to be restored or increased. Some of these case of similarity evolution may be addressed automatically; others give rise to a to-do list to be addressed by the developer. This semantics is captured as a 'propagate' operator of our method, as adopted from the general approach described in [1].

We are currently working on the propagate operator and its integration into the workflow of the underlying version control system, which is *git* in the case of 101haskell. Overall, the idea is that *git* commands such as *commit*, *pull*, and *push* are enriched by the propagation semantics.

3. The Ann tool

Ann is an interactive tool supporting the exploration of the version history and the set of variants down to the level of folders, files, and fragments. The tool assumes that the variants are maintained by a version control system. In the case of 101, we use *git*.

Figure 2 shows the annotation view of Ann: we are in the context of a specific similarity, namely a fragment shared by two variants. We have decided to annotate the similarity with 'maintain equality' so that automatic propagation of changes would be enabled.

Figure 3 shows all variants of 101haskell on the timeline of the version history. One can study the commits to get quick access to affected variants and files as well as the associated similarities.

Figure 4 gives an impression of the variant-centric dimension of exploration. A variant is picked in the beginning (*haskellEngineer* in the figure). The most similar variants are shown. One can dive into the picked variant to select specific folders or files. One can then further dive into files to eventually annotate similarities at the fragment level.

4. State of research

We have developed all extraction tools that Ann depends on. We have further implemented all essential views, as also illustrated in this extended abstract including two dimensions of annotation: variant-centric versus commit-centric. We have refined the user interface in several iterations since the amount of data to be processed

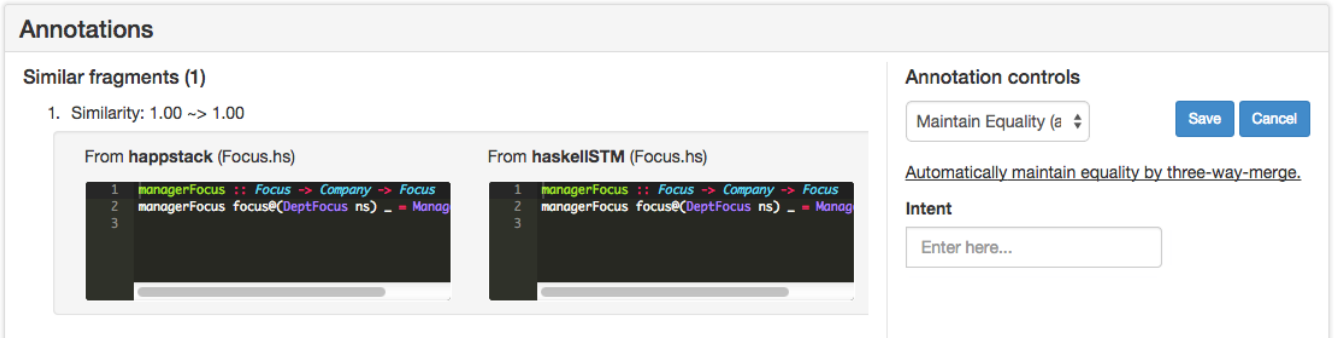


Figure 2. Annotation of a similarity

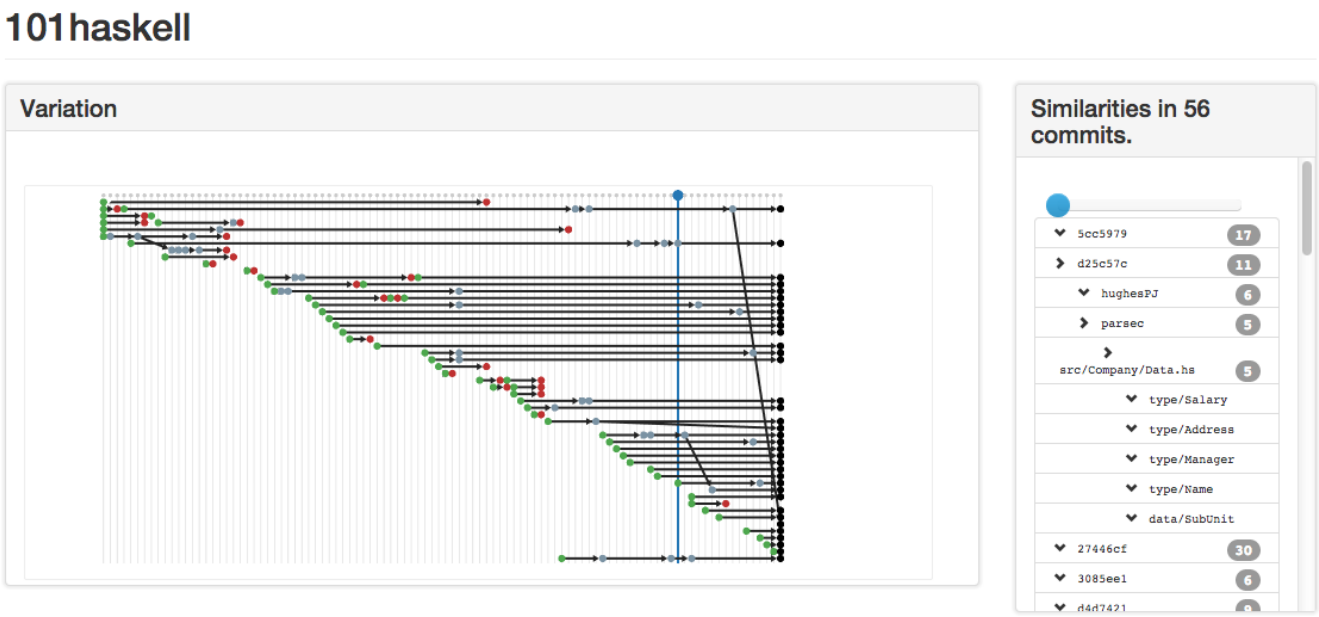


Figure 3. Variations of 101haskell

by the user was initially too large. The iterations applied good user experience (UX) principles to the design of Ann.

The tool already helped us to identify a set of variants that are conceptually or technologically outdated or unnecessarily disconnected (in terms of similarity) to other variants. The next step is to integrate the propagation operator into a git workflow by extending existing git commands and implementing new commands. After that we will aim at improving the similarity across 101haskell by bringing up, for example, the degree of equal fragments in response to the erosion (divergence) that has happened over time, when we had no tool support like Ann.

References

[1] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, and I. Schaefer. Flexible product line engineering with a virtual platform. In *Proc. of ICSE 2014*, pages 532–535. ACM, 2014.

[2] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: A Community Project on Software Technologies and Software Languages. In *Proc. of TOOLS 2012*, volume 7304 of *LNCS*, pages 58–74. Springer, 2012.

[3] R. Lämmel. Software chrestomathies. *Sci. Comput. Program.*, 2013. In press.

[4] R. Lämmel, T. Schmorleiz, and A. Varanovich. The 101haskell Chrestomathy—A Whole Bunch of Learnable Lambdas. In *Postproceedings of IFL 2013*, 2014. 12 pages. To appear in ACM DL. Available online `http://softlang.uni-koblenz.de/101haskell/`.

[5] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. of ICPC 2008*, pages 172–181. IEEE, 2008.

[6] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *Proc. ICSE 2013*, pages 1233–1236. IEEE / ACM, 2013.

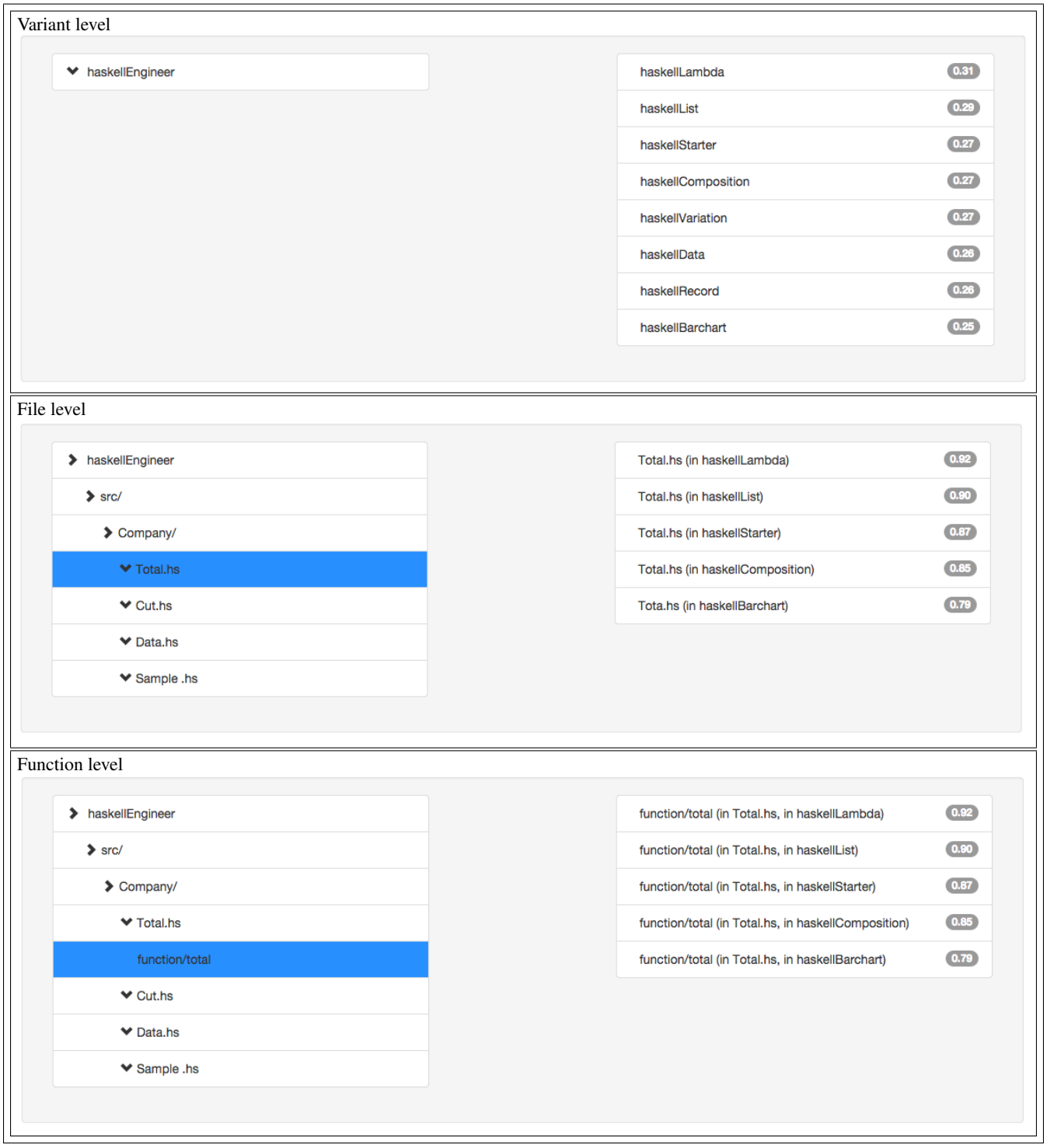


Figure 4. Levels of similarity exploration

Monoids model extensibility

or, Moxy: a language with extensibly extensible syntax

Michael Arntzenius
Carnegie Mellon University
daekharel@gmail.com

Abstract

Many languages, libraries, or systems advertise themselves as extensible, but these claims are usually informal. We observe that *monoids* formally characterize much of the essence of extensibility. We present extensibility monoids for several pre-existing systems, including grammars, macros, and monad transformers. To show the utility of this approach, we present Moxy, a syntactically extensible programming language. Moxy's implementation is surprisingly simple for the power it offers. We hope the use of monoids as an organizing principle of extensibility will offer similar utility in other domains.

Keywords Extensibility, extensible languages, macros, monoids, parser-combinators, parsing, syntax.

1. Introduction

The concept of extensibility recurs frequently in programming languages research, but lacks a concrete definition. The most one can say is "you know it when you see it". This paper observes that *monoids* capture some of the essential properties of extensibility. In Section 2, several common examples of extensibility are fruitfully analysed in terms of their monoids.

To show this approach's utility on a larger scale, in Section 3 we present Moxy, a language designed to be syntactically extensible — to permit the programmar to add syntax for new data types, operations, embedded DSLs, and so on — by using monoids to abstract extensibility. Moxy's design has several interesting qualities:

1. Moxy is not a Lisp or Scheme; extensibility is not achieved by the sacrifice of syntactic conveniences such as infix operators.[1]
2. Moxy is *self-extensible*: extensions are written in Moxy itself.
3. Moxy extensions are *modular*, that is to say, they are imported just like ordinary library definitions.
4. Moxy extensions are *scoped*; an extension can be imported within a whole file, within a module, or just within a single **let**-expression.
5. Moxy extensions are not limited to adding new forms of *expression*. They can also add new forms of *pattern* for use in pattern-matching, for example.
6. Moxy is *meta-extensible*: extensions can *themselves* be extended. For example, a Moxy programmer could create an embedded DSL as an extension library, and leave that DSL open to further extension by future programmers.
7. Moxy is *homogenously extensible*: All forms of extension — new expressions, new patterns, extensions to some library-defined extensible extension, and so forth — are accomplished by the same mechanism; no special privilege is given to built-in forms of extension, or to expressions over patterns.
8. Finally, Moxy itself is simple: about 2,000 lines of Racket code.

[1] Opinions on the desirability of this property may differ.

2. Monoids as a framework for extensibility

A monoid $\langle A, \cdot, \varepsilon \rangle$ consists of a set A, an operation $(\cdot) : A \times A \to A$, and an element $\varepsilon : A$ satisfying:

$$\begin{array}{rcll} a \cdot (b \cdot c) & = & (a \cdot b) \cdot c & \text{associativity} \\ a \cdot \varepsilon & = & a & \text{right-identity} \\ \varepsilon \cdot a & = & a & \text{left-identity} \end{array}$$

A notion of extensibility may be defined by giving a monoid where A is the set off all possible extensions, $(\cdot)$ is an operator that *combines* or *composes* two extensions, and ε is the "null extension", representing the lack of any additional behavior.

We show that monoids are a good model for extensibility by analysing five distinct forms of extensibility into their respective monoids: composition of context-free grammars, lisp-style macros, open recursion, middleware in a web framework, and monad transformers.

2.1 Composition of context-free grammars

Consider the following grammar for a λ-calculus:

$$\begin{array}{lrcl} \textbf{expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ \textbf{variables} & x & ::= & \text{... (omitted) ...} \end{array}$$

It is easy enough to add infix arithmetic to this language:

$$\begin{array}{lrcl} \textbf{expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ & & \mid & n \mid e \otimes e \\ \textbf{numerals} & n & ::= & d \mid dn \\ \textbf{digits} & d & ::= & 0 \mid 1 \mid ... \mid 9 \\ \textbf{operators} & \otimes & ::= & + \mid - \mid \times \mid / \end{array}$$

It is similarly easy to add lists:

$$\begin{array}{lrcl} \textbf{expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ & & \mid & [\,] \mid [es] \\ \textbf{expression lists} & es & ::= & e \mid e, es \end{array}$$

Formally, each of these three grammars is a separate, complete object. But there is a clear sense in which the two latter languages are composed out of parts: "λ-calculus *plus* infix arithmetic", "λ-calculus *plus* lists". And, without even writing its grammar, it is obvious there exists a fourth language: "λ-calculus *plus* infix arithmetic *plus* lists". Can we formalize this colloquial "*plus*" operator? We can; and it is even a monoid!

Context-free grammars. A context-free grammar (CFG) is a tuple $\langle \Sigma, N, P, e \rangle$, where Σ is a finite set of terminal symbols, N is a set (disjoint from Σ) of nonterminal symbols, P is a finite set of *production rules*, and $e \in N$ is an initial non-terminal. Production rules $p \in P$ have the form $N \hookrightarrow (\Sigma \cup N)^*$; that is, $P \subseteq N \times (\Sigma \cup N)^*$.

Fix a terminal set Σ, a (possibly infinite) nonterminal set N, and an initial nonterminal e. That is, consider grammars over a known alphabet (e.g. Unicode characters). The non-terminal set is also fixed, but this is no great limitation, since it is permitted to be infinite; we will consider nonterminals to come from an inexhaustible set of abstract names. And we fix an initial nonterminal e, intended to represent well-formed expressions in some language.

Having fixed all this, a grammar is fully specified by a finite set of production rules P. Our original λ-calculus is represented by:

$$\begin{array}{rcl} P_\lambda & = & \{e \hookrightarrow x, \\ & & \ e \hookrightarrow \lambda x.e, \\ & & \ e \hookrightarrow e\ e, \\ & & \ \text{... (omitted rules for } x\text{) ...}\} \end{array}$$

Composing CFGs. To compose two grammars represented as production-rule-sets, simply take their union. For example, consider the sets of rules *added* when extending our language with infix arithmetic and lists respectively:

$$\begin{array}{rcl} P_\otimes & = & \{e \hookrightarrow n,\ \ e \hookrightarrow e \otimes e, \\ & & \ n \hookrightarrow d,\ \ n \hookrightarrow dn, \\ & & \ d \hookrightarrow 0,\ \ ...,\ \ d \hookrightarrow 9, \\ & & \ \otimes \hookrightarrow +,\ \ \otimes \hookrightarrow -,\ \ \otimes \hookrightarrow \times,\ \ \otimes \hookrightarrow /\} \\ P_{[]} & = & \{e \hookrightarrow [\,],\ \ e \hookrightarrow [es], \\ & & \ es \hookrightarrow e,\ \ es \hookrightarrow e, es\} \end{array}$$

If we take $P_\lambda \cup P_\otimes$, it gives us precisely the grammar of our second language, "λ-calculus *plus* infix arithmetic". $P_\lambda \cup P_{[]}$ is our third language, "λ-calculus *plus* lists". And $P_\lambda \cup P_\otimes \cup P_{[]}$ is the hypothesized fourth language, "λ-calculus *plus* infix arithmetic *plus* lists".

We have thus successfully separated our languages into parts — one which represents λ-calculus, one which represents infix arithmetic, one which represents lists — and found an operator that can combine them again. Finally, note that this operator (union of production-rule-sets) is a (commutative, idempotent) monoid, with $\emptyset$ as identity.

Limitations. While a CFG specifies a syntax, it does not supply an algorithm for parsing. Moreover, it specifies *only* syntax; a programming language also needs *semantics*. Section 3.3 covers one approach to parsing a monoidally-extensible syntax. Section 3.4 describes how Moxy permits extensible semantics.

2.2 Macro-expansion

Consider a simple lisp-like syntax:

s-expressions	e	::=	$a \mid (es)$
s-expression lists	es	::=	$\mid e\ es$
atoms	a	::=	$s \mid n$
numerals	n	::=	$0 \mid 1 \mid ...$
symbols	s	::=	...

We formalize a simplistic version of macro-expansion by defining $\mathsf{expand}(env, e)$, which takes an environment env mapping symbols to their definitions, an s-expression e to expand, and returns ep with all macro invocations recursively replaced by their expansions. In pseudocode:

$\mathsf{expand}(env, (s\ \ es))$ **if** s **in** env = $\mathsf{expand}(env[s](es))$
$\mathsf{expand}(env, (es))$ = $\mathsf{map}(\mathsf{expand}(env, \_), (es))$
$\mathsf{expand}(env, a) = a$

(We write $env[s]$ for the macro-definition of s in env, which we take to be a function from an argument-list es to its immediate expansion under that macro.)

Where's the monoid? If macros yield a form of extensibility, how do they fit into our monoidal framework? First, we must find the set of possible extensions. Intuitively, *macros* are the extensions we are concerned with. Perhaps the set of all macro-definitions? However, there is no obvious operator to "merge" two macro-definitions.

This suggests that our notion of extension is not powerful enough. So instead of single macros, we take our extensions to be *environments* mapping macro-names to their definitions, like the env argument to expand. Our binary operator $(\cdot)$ merges two environments, so that:

$$s \textbf{ in } (env_1 \cdot env_2) \quad \textbf{iff} \quad (s \textbf{ in } env_1) \vee (s \textbf{ in } env_2)$$

$$(env_1 \cdot env_2)[s] \quad = \quad \begin{cases} env_1[s] & \textbf{if } s \textbf{ in } env_1 \\ env_2[s] & \textbf{otherwise} \end{cases}$$

By convention, $(\cdot)$ is left-biased: if the same symbol is defined in both arguments, it uses the left one.

Finally, note that $(\cdot)$ is associative, and forms an identity with the empty environment; that is, $(\cdot)$ is a monoid.

Monoid-parameterized functions. Observe that expand is a function parameterized by a value of the monoid representing extensions. It demonstrates one way of giving *semantics* to a notion of extensibility: interpret extensions into functions. This is a pattern we will see again in Moxy's parse and compile phases.

2.3 Open recursion and mixins

Open recursion is the problem of, first, defining a set of mutually-recursive functions *by parts*; that is, permitting the programmer to define only some of the functions, and complete the set by defining the rest later; and, second, of allowing the behavior of these functions to be *overridden or extended*.

For example, consider the functions $\mathsf{even}(n)$ and $\mathsf{odd}(n)$, of type $\mathbb{N} \to \mathsf{Bool}$, defined by:

$\mathsf{even}(0) = \mathsf{true}$
$\mathsf{even}(n) = \mathsf{odd}(n-1)$
$\mathsf{odd}(0) = \mathsf{false}$
$\mathsf{odd}(n) = \mathsf{even}(n-1)$

Definition *by parts* means that we could define even and odd separately, as *mixins*, and later combine them into an *instance* that implements both:

mixin EvenMixin **where**
  inherit odd
  $\mathsf{even}(0) = \mathsf{true}$
  $\mathsf{even}(n) =$ **self**.$\mathsf{odd}(n-1)$
end
mixin OddMixin **where**
  inherit even
  $\mathsf{odd}(0) = \mathsf{false}$
  $\mathsf{odd}(n) =$ **self**.$\mathsf{even}(n-1)$
end

```
instance EvenOdd where
  use EvenMixin
  use OddMixin
end
EvenOdd.even(10)   — returns true
```

Overriding means that a mixin can extend the behavior of a function in a fashion similar to traditional object-oriented subclassing:

```
mixin EvenSpyMixin where
  inherit odd
  even(n) =
    print(n);
    super.even(n)
end
instance EvenOddDebug where
  use EvenMixin
  use OddMixin
  use EvenSpyMixin
end
— prints 2, then prints 0, then returns true
EvenOddDebug.even(2)
```

2.3.1 Semantics of mutual recursion.

Setting aside for a moment the problem of open recursion, consider ordinary, "closed", mutual recursion. Suppose we wish to implement some signature $\Sigma = \langle N, \tau \rangle$ of mutually recursive functions, where N is a set of names and the function $\tau : N \to \mathsf{type}$ assigns each name a type. An *implementation* of Σ is a function giving to each name n a value of type $\tau(n)$. We write Impl_Σ for the type of implementations of the signature Σ:

$$\mathsf{Impl}_{\langle N,\tau\rangle} = \Pi(n : N).\ \tau(n)$$

We represent a group of mutually-recursive function *definitions* as a function from implementations to implementations: we take a self object, and to recursively call the function n, we call $\mathsf{self}(n)$. In this manner we make self-reference explicit. We write Defn_Σ for the type of mutually-recursive function definitions represented this way:

$$\mathsf{Defn}_\Sigma = \mathsf{Impl}_\Sigma \to \mathsf{Impl}_\Sigma$$

For example, our original mutually-recursive definition of even and odd is represented by:

$$\begin{array}{lcl}
\Sigma & = & \langle N, \tau \rangle \\
N & = & \{\mathtt{even}, \mathtt{odd}\} \\
\tau(\mathtt{even}) & = & \mathbb{N} \to \mathsf{Bool} \\
\tau(\mathtt{odd}) & = & \mathbb{N} \to \mathsf{Bool} \\
\mathsf{EvenOdd} & : & \mathsf{Defn}_\Sigma \\
\mathsf{EvenOdd}(self)(\mathtt{even})(0) & = & \mathsf{true} \\
\mathsf{EvenOdd}(self)(\mathtt{even})(n) & = & self(\mathtt{odd})(n-1) \\
\mathsf{EvenOdd}(self)(\mathtt{odd})(0) & = & \mathsf{false} \\
\mathsf{EvenOdd}(self)(\mathtt{odd})(n) & = & self(\mathtt{even})(n-1)
\end{array}$$

To obtain the desired implementations of even and odd, we take the least-fixed-point of the EvenOdd function:

$$\begin{array}{lcl}
\mathsf{even} & = & fix(\mathsf{EvenOdd})(\mathtt{even}) \\
\mathsf{odd} & = & fix(\mathsf{EvenOdd})(\mathtt{odd})
\end{array}$$

For some function $fix : (\alpha \to \alpha) \to \alpha$ satisfying $fix(f) = f(fix(f))$ for appropriate α (here, $\alpha = \mathsf{Impl}_\Sigma$).

2.3.2 Mixins as a monoid

Let's apply our monoid methodology to the problem of semantics for open recursion. We'll tackle definition by parts first, and later expand our semantics to cover overriding as well.

First, we must determine our set of extensions. The extensions we are concerned with are "mixins": partial definitions of a set of mutually recursive functions. The definitions in a mixin have access to a self object, allowing mutual- or self-recursion.

At first glance, this seems very similar to the way we formalized closed sets of mutually-recursive definitions. The only difference is that mixins are permitted to be *partial*, and omit a definition for a particular function in the signature.

However, we can use the same type Defn_Σ to represent mixins if for names n which the mixin M does not define, we simply let $M(self)(n) = self(n)$, passing the undefined method through unchanged. Thus a mixin is represented by a function from implementations to implementations: it takes a self object, and returns that object updated with each of the functions the mixin implements.

For example, EvenMixin and OddMixin are represented by the following functions:

$$\begin{array}{lcl}
\mathsf{EvenMixin}(self)(\mathtt{even})(0) & = & \mathsf{true} \\
\mathsf{EvenMixin}(self)(\mathtt{even})(n) & = & self(\mathtt{odd})(n-1) \\
\mathsf{EvenMixin}(self)(\mathtt{odd}) & = & self(\mathtt{odd}) \\
\\
\mathsf{OddMixin}(self)(\mathtt{odd})(0) & = & \mathsf{false} \\
\mathsf{OddMixin}(self)(\mathtt{odd})(n) & = & self(\mathtt{even})(n-1) \\
\mathsf{OddMixin}(self)(\mathtt{even}) & = & self(\mathtt{even})
\end{array}$$

At first, this representation seems pointless, like trying to fit a square peg (mixins) into a round hole (Defn_Σ). After all, if we take $fix(\mathsf{OddMixin})(\mathtt{odd})$, the resulting function diverges for all non-zero inputs!

However, observe that

$$(\mathsf{EvenMixin} \circ \mathsf{OddMixin}) = \mathsf{EvenOdd}$$

(as can be verified by some tedious calculations)!

That is, that we can "combine" mixins by composing them as functions; our monoidal operator is $\circ$. Our identity is just the identity function at type Impl_Σ. Function composition is associative, so we are done.

2.3.3 Mixins with overriding

$$\begin{array}{rcl}
\mathsf{Mixin}_\Sigma & = & \mathsf{Impl}_\Sigma \to \mathsf{Impl}_\Sigma \to \mathsf{Impl}_\Sigma \\
\varepsilon(self)(super) & = & super \\
(f \cdot g)(self)(super) & = & f(self)(g(self)(super))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{EvenMixin}(self)(super)(\mathtt{even})(0) & = & \mathsf{true} \\
\mathsf{EvenMixin}(self)(super)(\mathtt{even})(n) & = & self(\mathtt{odd})(n-1) \\
\mathsf{EvenMixin}(self)(super)(\mathtt{odd}) & = & super(\mathtt{odd}) \\
\mathsf{OddMixin}(self)(super)(\mathtt{odd})(0) & = & \mathsf{false} \\
\mathsf{OddMixin}(self)(super)(\mathtt{odd})(n) & = & self(\mathtt{even})(n-1) \\
\mathsf{OddMixin}(self)(super)(\mathtt{even}) & = & super(\mathtt{even})
\end{array}$$

$$\begin{array}{lcl}
\mathsf{EvenSpyMixin}(self)(super)(\mathtt{odd}) & = & super(\mathtt{odd}) \\
\mathsf{EvenSpyMixin}(self)(super)(\mathtt{even})(n) & = & \mathbf{print}(n); \\
 & & super(\mathtt{even}(n))
\end{array}$$

To actually produce an implementation:

$$\begin{array}{rcl}
impl & : & \mathsf{Mixin}_\Sigma \to \mathsf{Impl}_\Sigma \\
impl(f) & = & fix(\lambda s.f(s)(\bot))
\end{array}$$

2.4 Web framework middleware

We observe that middleware in many web frameworks (for example, Django) is essentially a stack of pairs of functions, $(\mathsf{Request} \to \mathsf{Request}) \times (\mathsf{Response} \to \mathsf{Response})$, which are composed together via the monoid:

$$\begin{array}{rcl}
\varepsilon & = & \langle \mathsf{id}_{\mathsf{Request}}, \mathsf{id}_{\mathsf{Response}} \rangle \\
\langle f_{\mathsf{req}}, f_{\mathsf{resp}} \rangle \cdot \langle g_{\mathsf{req}}, g_{\mathsf{resp}} \rangle & = & \langle f_{\mathsf{req}} \circ g_{\mathsf{req}}, g_{\mathsf{resp}} \circ f_{\mathsf{resp}} \rangle
\end{array}$$

top-level	t	::=	$d \mid$ **module** N $\{t^*\}$
expressions	e	::=	$Pn \mid PN \mid l \mid$ `(`e`)` $\mid e \oplus e$
		$\mid$	`\(`$p, ...$`)` $e \mid e$`(`$e, ...$`)`
		$\mid$	**let** d^* **in** e
		$\mid$	**case** e [$\mid$ p `->` e]*
declarations	d	::=	**val** p `=` e
		$\mid$	**fun** n`(`$p, ...$`)` `=` e [$\mid$ n`(`$p, ...$`)` `=` e]*
		$\mid$	**tag** N[`(`$n, ...$`)`]
		$\mid$	**rec** d [**and** d]*
		$\mid$	**open** PX
patterns	p	::=	$x \mid l \mid PX$[`(`$p, ...$`)`]
operators	$\oplus$	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `==` $\mid$ `<=` $\mid$ `>=` $\mid$ `<` $\mid$ `>` $\mid$ `;`
		$\mid$	...
module paths	P	::=	[N`.`]*
capital names	N		
names	n		
literals	l		

Figure 1. Grammar of Moxy, sans extensions

And then composed onto a base handler function h : Request $\rightarrow$ Response:

$$\mathsf{withMiddleware}(\langle m_{\mathsf{in}}, m_{\mathsf{out}}\rangle, h) = m_{\mathsf{out}} \circ h \circ m_{\mathsf{in}}$$

Middleware "in the wild" is more complicated, having to deal with details such as error-handling, the possibility of early exit, and so forth, but still forms a monoid. The fact that it forms a monoid can even be observed by the way one specifies what middleware to use: via a list, i.e. an element of the free monoid.

2.5 Monad transformers

Broadly speaking, monads as they appear in Haskell represent *effects* which a computation has access to: for example, Maybe represents the possibility of failure; State s represents access to a single mutable location of type s; List represents backtracking nondeterminism.

It is often useful to have multiple kinds of effect; for example, failure *and* state. For this Haskell uses *monad transformers*, type-level functions from monads to monads. Letting **Hask** be the category of Haskell types, a monad such as State s has kind **Hask** $\rightarrow$ **Hask**; a monad transformer such as StateT s has kind (**Hask** $\rightarrow$ **Hask**) $\rightarrow$ (**Hask** $\rightarrow$ **Hask**).

To combine multiple effects, one creates a stack of monad transformers, terminating it with the identity monad Identity. To combine failure (Maybe) with binary state (State Bool) and logging (Writer [String]), we write:

```
WriterT [String] (StateT Bool (MaybeT Identity))
```

Clearly monad transformers represent a notion of extensibility, namely "adding effects" to a pure base language. Unsurprisingly, they form a monoid; the operator is simply composition, and the identity is the monad transformer IdentityT. The above can be rewritten (assuming a type-level composition operator $\circ$) as:

```
(WriterT [String] ∘ StateT Bool ∘ MaybeT) Identity
```

3. Moxy

3.1 A brief introduction to Moxy, sans extensions

Moxy exists to test the hypothesis that monoids are a good way of structuring extensibility. In other respects it is a banal, humdrum language.

The basic grammar of Moxy is given in figure 1. "$e, ...$" indicates a comma-separated list of expressions e; similarly for patterns, "$p, ...$".

Syntactically, Moxy is moderately MLish; semantically, Moxy is somewhat Schemelike. Evaluation is eager; functions take multiple arguments and return one value; recursion is the only looping construct; pattern-matching is the only conditional construct (**if** is implemented as a syntax extension). Moxy has a simple module system that only handles namespacing (no ML-style functors). Strings and numbers are built-in; booleans are defined in the implicitly-imported prelude.

As an example, here is a naïve recursive implementation of the fibonacci function in Moxy:

```
fun fib(0) = 1
  | fib(1) = 1
  | fib(n) = fib(n-1) + fib(n-2)
```

Case is syntactically significant in Moxy: capitalized names are used for modules, tags, and extension points, uncapitalized for everything else.

3.2 Moxy's approach to extensibility: Extension points

Moxy aims to be *meta-extensible*: to permit extensions (for example, a DSL for parsers) that are themselves extensible (for example, defining $a+$ as an abbreviation for $aa*$). It therefore seems difficult to specify in advance what the monoid representing Moxy extensions should be.

Moxy's solution is to let the programmer define their own extension monoids, which Moxy calls *extension points*. To define an extension point, the programmer gives it a name (a Moxy identifier), a value representing the identity element, and a function representing the monoid operator. For example:

```
extension ExtCounts(0, \(x,y) x+y)
```

This defines an extension point named ExtCounts with the monoid $\langle \mathbb{N}, +, 0\rangle$. As Moxy is dynamically typed, the fact that the intended domain of ExtCounts is $\mathbb{N}$ exists only in the programmer's mind.[2]

Having defined an extension point, we can extend it with an **extend** declaration:

```
extend ExtCounts with 2 + 2
```

Within the scope of this declaration, ExtCounts will have a value 4 higher than it otherwise would. Of course, this is quite useless without some way to observe the value of an extension point and use it in parsing.

For this purpose, Moxy comes with built-in extension points which allow extending Moxy's built-in syntax classes: expressions, declarations, and patterns. For example, the InfixExprs extension point permits adding new infix operators to the expression language:

```
extend InfixExprs with
  { TSYM("."): { precedence = 10,
                 parse = [...omitted...] } }
```

3.3 Parsing Moxy

Moxy uses monadic parser-combinators after the style of Parsec to implement a Pratt-style recursive descent parser. In practice what the latter means is that infix operators are handled by means of a table mapping the token for a given operator (say, + for addition) to its precedence, paired with a parser for its right-hand-side. When parsing an expression we keep track of the precedence we are currently parsing at, and if we encounter an operator of looser

[2] Indeed, there is nothing to say that the intended domain is not $\mathbb{Z}$ instead.

precedence we stop parsing and return. Otherwise we recursively invoke the parser for the operator we found, with the expression we've parsed so far as an additional argument.

This permits arbitrary infix, suffix, and mixfix operators. However, it requires that each infix operator have a unique token identifying it; no overloading is possible (at least, not in the parser).

Moxy currently has a separate tokenization phase, which unfortunately limits the expressiveness of parse extensions. We believe this can be eliminated in future by means of a scannerless-parsing technique, such as that used by Parsec's Text.Parsec.Token module.

The monad which Moxy uses for parsing, in addition to behaving like Parsec, also acts as a Reader ParseEnv, where ParseEnv is a datastructure representing all extensions to all extension points currently in scope. A ParseEnv is effectively a mapping from extension points to their values. It is the parameterization of the parser by this value that lets extensions affect parsing (just as the parameterization of expand by a macro-environment lets macro-definitions affect macro-expansion).

For example, when the parser tries to parse the start of an expression, it first examines the current value of the Exprs extension point. Exprs' domain is dictionaries mapping tokens to parsers; its monoid operation is right-biased merge and its identity is the empty dictionary. If the next token is present in the value of Exprs, then we invoke the parser it is bound to. Otherwise we try built-in parse productions such as literals, variables, and function application.

Similar techniques are used for extending declarations and patterns.

3.4 Compiling Moxy

Nodes in Moxy's AST are instances of abstract interfaces. For example, an expression is merely something which *knows how to compile itself*. That is, expressions are "records"[3] with a functional compile field. This compile function, when supplied with a *resolve environment*, returns the intermediate-representation[4] compilation of the expression in question.

The *resolve environment* is effectively merely the lexical environment of the expression, a dictionary which tells the compiler which variable names are in scope and how references to them are to be compiled.

4. Prior work

Lisp and Scheme are obvious predecessors in the search for extensible forms of programming. Moxy is in large part an attempt to replicate the ease and power of Lisp-family macros in a non-Lisp setting.

SugarJ does almost everything Moxy does and more [1]. However, it weighs in at 26k LOC, while Moxy is a mere 2k LOC. Moxy can use extensions per-scope, not just per-file. Moxy has a REPL, but SugarJ's approach is probably compatible with a REPL as well. Moxy allows for non-syntactic notions of scoped extensibility (but I have no motivating examples). The Moxy approach currently has no story for integrating with existing languages (but it probably could be done).

Moxy's use of a Pratt-style parser to aid extensibility is similar to that of Magpie. [4]

Other previous syntactically extensible systems include OMeta [3], Sweet.js, Seed7, and Omar et al's Type-Specific Languages [2].

[3] In this case, represented as hash-tables.

[4] In this case, s-expressions representing Racket code.

References

[1] **SugarJ: Library-based Syntactic Language Extensibility**. Sebastian Erdweg, Tillman Rendel, Christian Kästner and Klaus Ostermann. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391-406. ACM, 2011.

[2] **Safely Composable Type-Specific Languages**. C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin and J. Aldrich. *European Conference on Object-Oriented Programming (ECOOP 2014) Uppsala, Sweden, July 28 – August 1, 2014*.

[3] **Experimenting with Programming Languages**. Alessandro Warth. Technical Report TR-2008-003, Viewpoints Research Institute, 2009.

[4] **Extending Syntax from Within a Language**. Bob Nystrom. `http://journal.stuffwithstuff.com/2011/02/13/extending-syntax-from-within-a-language/`, 2011-02-13, accessed 2014-09-25.

[5] **Parsec: Direct Style Monadic Parser Combinators For The Real World**. Daan Leijen and Erik Meijer. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Towards efficient implementations of effect handlers

– Extended Abstract –

Steven Keuchel
Universiteit Gent
steven.keuchel@ugent.be

Tom Schrijvers
Universiteit Gent
tom.schrijvers@ugent.be

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Languages

Keywords effect handlers, modularity, delimited control, monads

1. Introduction

In recent years algebraic effects and effect handlers emerged as a compelling alternative to monads as a basis for effectful programming in functional programming languages. This approach provides language primitives for defining new abstract effectful operations, which the programmer uses to write his own programs, and effect handlers that instantiate the abstract operations with concrete implementations.

Abstract operations are composable and each effect handler instantiates a specific subset of the operations of a computation. Thus this approach yields modular abstraction and modular instantiation of effects similar to monad transformers and monad type classes. Transporting monads and monad transformers to different languages is possible but difficult or awkward. Some monads, like for example monadic parser combinators, crucially rely on infinite recursion and lazy evaluation. Modular abstraction with monads is achieved by means of type classes for each kind of effect and modular instantiation is achieved by type class instances that lift monad type classes over monad transformers. This crucially relies on automatic type class resolution.

Effect handlers are one possible alternative to monads and monad transformers for getting modularity in the handling of effects into a variety of functional programming languages, especially those that use strict evaluation and do not provide type classes.

2. Effect handlers

We present the general use of effect handlers using pseudo-syntax that is similar to the Frank language by Lindley and McBride [9]. The first step is to declare the signature of the abstract operations of an effect. The state effect for instance has two abstract operations: 1. get that retrieves the current state and 2. put that updates the state.

$$
\begin{array}{l}
sig\ State\ S \\
\quad = get : [\,]\ S \\
\quad \mid\ put : S \to [\,]\ Unit
\end{array}
$$

This declares get to be an effectful operation that results in a value of type S. put is an effectful operation that takes an a and returns a unit value. We use the notation $[\,]\ X$ to denote *computations* that upon execution return a value of type X.

We can write computations using these abstract operations. The $next$ computation performs state effects to increment a natural number. The first line is the type signature of $next$ which states that $next$ results in a value of type Nat and can perform $State\ Nat$ effects. The second line is the implementation of $next$ in terms of the abstract operations.

$$
\begin{array}{l}
next : [State\ Nat]\ Nat \\
next = get \to n; put\ (suc\ n); n
\end{array}
$$

The second step is to define a handler $state$ that implements the operations of the $State\ S$ effect. The handler $state$ takes as parameters an initial state s and a suspended computation on which it 'pattern matches'. The first two lines of the implementation handle the cases of the abstract operations get and put. The third line handles the case of a finished computation that resulted in a value v where we allow the handler to transform the result value.

$$
\begin{array}{l}
state : S \to [State\ S\ ?\ X] \to [\,]\ X \\
state\ s\ [get\ \ \ ?\ k] = state\ s\ ?\ k\ s \\
state\ \_\ [put\ s\ ?\ k] = state\ s\ ?\ k\ () \\
state\ \_\ [v] \qquad\quad = v
\end{array}
$$

3. Problem statement

If effect handlers are to be used as the principal paradigm of a language to model side-effects, their efficient implementation becomes an important point. The focus of existing work on languages [1, 9] and embedded domain-specific languages [3, 8] for programming with effect handlers is still to explore the expressivity of this alternative approach to effects. It should come as no surprise that benchmarking existing implementations shows that the performance of these systems is not yet competitive to monad transformers.

One of the main reasons for this is that these systems implement effect handlers indirectly, either by reducing them to a free-monad implementation in lazy languages or to (delimited) continuations in strict languages. This leaves a lot of room for improvement.

It is easier for a direct implementation to avoid technical pitfalls that impact performance and to leverage the structure of effect handlers to perform optimizations. More specifically the following concerns can be addressed in a better way.

1. The shift/reset operators are the most popular way to describe delimited continuations and are also the most commonly provided primitives by languages that implement delimited continuations. However, there is an impedance mismatch between effect handlers and delimited continuations using the shift/reset operators. In exception and effect handlers the delimiter describes the handling of the effectful operation in contrast to shift/reset where shift determines the *handler*. This also means for each reset delimiter there can be different handlers provided to different invocations of shift. As a consequence an indirect implementation might lose the opportunity to optimize using the fact that there is a constant handler for each effect handler. A conceptually better fit are delimited continuations using run/fcontrol [11].

2. State is an important concept that arises very often in the implementation of handlers of a variety of effects. Each handler invocation can potentially alter the state of the handler and the state needs to be threaded properly between invocations.

   Brady [3] treats the state of handlers explicitly by keeping track of a list of resources for a given handler stack. Kammar et al. [6] allow handlers to be parameterized and alter the parameters for handling the operations of the continuation.

   In the reduction of effect handlers to delimited control operators, the state of handlers is captured inside a closure that implements the handlers. A direct implementation of effect handlers should keep track of state explicitly to make state passing more efficient. Ideally, passing the state should be as efficient as passing an argument to a function.

3. Direct implementations of delimited continuations for call-by-value languages [7, 10] capture the continuation by copying the part of the stack up to the delimiter to the heap. When the continuation is invoked, this copy is pushed back onto the control stack. A direct implementation of effect handlers will necessarily perform comparable operations.

   However, if the implementation of a handler uses the continuation in a restricted way several optimizations are possible. If a continuation is only used in tail positions, the copying of the stack segment is unnecessary[7]; if the handler does not use the continuation at all, e.g. traditional exceptions, we can unwind the stack immediately before passing control to the handler function and thus free resources early.

   As it turns out, in practice a lot of effects fall into this these restricted categories [2]. It is therefore important to perform an analysis and optimize accordingly.

4. Towards efficient handlers

We want to address the performance concerns of effect handlers to help their adoption. Specifically we want to address the implementation and optimization of common cases that do not use the full power of effect handlers.

To this end, we are developing a definitional machine for effect handlers based on a definitional machine for delimited continuations [4, 7] that provides generic low-level primitives for the implementation of effect-handlers and a small call-by-value λ-calculus with effect handlers. In our development we address the following performance concerns:

4.1 Handler resolution

At the invocation point of an abstraction operation control is passed to handler for the effect. For this the concrete handler needs to be looked up. One of the problems appearing is how to do this efficiently.

In the implementation of exception handlers, the delimiter try is either pushing exception-handler marks explicitly on the control stack or is creating a table that maps code return addresses to exception handling code [5]. The throwing of an exception will unwind the control stack – executing cleanup code along the way – and use runtime-type information about the thrown value and the type of exception handlers to find the matching handler.

Obviously this dynamic lookup of effect handlers and the use of type-information is utterly slow. Using effect typing we can explicitly resolve the possible handlers statically and keep track of stack marks, handler functions and handler state explicitly and efficiently.

4.2 Fast linear code

We perform an analyses to detect handlers that invoke the continuations only tail position and handlers that discard the continuations. In these two cases we can avoid unnecessary work for capturing the continuation and also avoid duplicating state which would be necessary for potentially different continuations.

In this restricted setting the specialization of code for a specific set of handlers also becomes easier. We envision the inlining of known handlers to remove any overhead introduced by abstractions as much as possible.

References

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 2014.

[2] J. Berdine, P. O'hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15 (2-3):181–208, 2002.

[3] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 133–144. ACM, 2013.

[4] R. Dyvbig, S. P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(06): 687–730, 2007.

[5] L. Goldthwaite. Technical report on c++ performance. *ISO/IEC PDTR*, 18015, 2006.

[6] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 145–158. ACM, 2013.

[7] O. Kiselyov. Delimited control in Ocaml, abstractly and concretely. *Theoretical Computer Science*, 2012.

[8] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70. ACM, 2013.

[9] S. Lindley and M. Conor. Do Be Do Be Do. draft, 2014.

[10] M. Masuko and K. Asai. Direct implementation of shift and reset in the mincaml compiler. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 49–60. ACM, 2009.

[11] D. Sitaram. Handling control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 147–155, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. . URL `http://doi.acm.org/10.1145/155090.155104`.