The 31st Symposium on Implementation and Application of Functional Languages

National University of Singapore, 25th – 27th September 2019

Editors: Jurriën Stutterheim and Wei Ngan Chin

Contents

A space-efficient call-by-value virtual machine for gradual set-theoretic types

Giuseppe Castagna
CNRS

Guillaume Duboc
ENS Lyon

Victor Lanvin
Université de Paris

Jeremy G. Siek
Indiana University

ACM Reference Format:
Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. 2020. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Abstract. We describe the design and implementation of a virtual machine for programming languages that use gradual typing with set-theoretic types focusing on practical issues such as runtime space efficiency, and efficient implementation of tail recursive calls.

1 INTRODUCTION

Gradual typing is an approach proposed by Siek and Taha [23] to combine the safety guarantees of static typing with the programming flexibility of dynamic typing. In gradually typed programs some parts of a program may be given types and their correctness is checked at compile time (which is static typing), while some other parts are left untyped and any eventual type errors are reported at run-time (which is dynamic typing). Programmers may specify which parts are which by using suitable type annotations. A gradually-typed program is then rejected at compile time only if the statically typed parts do not type-check and/or if there exists some dynamically typed parts for which there are always failures at runtime (e.g., the application of a number to an argument).

Recently, gradual typing has received a lot of attention both from academia and industry. It is becoming the standard approach for adding static typing to dynamic languages such as, among others, JavaScript [2, 9, 20, 21], PHP [10], Python [19, 30], Clojure [3], and Scheme [28, 29]. These designs often include the use of union, intersection and, to a lesser extent, negation types (the *set-theoretic types* of the title), because such types are needed to capture many programming patterns common in dynamic languages. For instance, union and intersection types are present (in more or less limited forms) and heavily used in TypeScript [2], Flow [9] (two gradually-typed versions of JavaScript by Microsoft and Facebook, respectively) and Typed Racket [29] (a gradually-typed version of Scheme), while support for negation types are in a pull request for TypeScript. The compilers of these latter extensions do not implement gradual typing to its full extent. As we explain later, for a gradually typed language to be "sound", the compiler must insert dynamic type checks (called *type casts*) in the compiled code at the boundaries between the untyped and typed parts of the program. These checks provide strong type-based guarantees that can be used by the programmer and the compiler to reason about and optimize statically typed code. Instead, TypeScript, Flow, and Typed Racket forgo these runtime checks, and compile both the typed and untyped parts to an underlying untyped language, which, as usual, performs pervasive runtime checking. While this approach still guarantees that programs "do not get stuck", it does not provide the strong type-based guarantees of sound gradual typing. For example, it allows a number to mascarade as a string inside of statically typed code!

There are two main reasons for not implementing gradual typing thoroughly. One is somehow philosophical: you may not want to change the semantics of the dynamically typed language on which gradual types are grafted (especially when this is a very popular language such as JavaScript); type annotations are then just debugging and documentation glosses that must not alter the program they are added to. The other is eminently practical: the addition of dynamic type checks in the run-time code may hugely penalize performance [27]; to limit this impact special care must be taken in defining the compilation of untyped code and the execution of the compiled code [1, 18].

This degradation of performances for sound gradual typing has a main culprit: the use of higher order functions in untyped code which yields a compiled code where dynamic type tests are applied to unknown functions. As we explain in Section 2.3, the only reasonable way to cast a function to a particular type is to delay the runtime check to the moment of its application and check that its argument and result are both of the expected types. This delayed checking is accomplished by proxying the function, and when a function passes through many casts, it would naively be wrapped in a long chain of proxies, making performance utterly degrade. Resolving this problem requires specific implementation techniques [18].

The degradation of performances just described is dramatically amplified by the presence of union and intersection types: while in their absence determining the type expected for the input and the output of a function is just the matter of reading a couple of fields, the presence of union and intersection type makes this operation more complicated and computationally demanding; and since this computation must be performed at runtime, it adds a further critical overhead to the performance bottleneck of sound gradual typing. No implementation techniques have been developed to handle this further overhead, yet.

To summarize, gradual typing is the approach chosen by major IT actors to inject a dose of static typing into dynamic languages, but this injection also requires the use of union and intersection types. To provide stronger soundness guarantees the compiler must insert

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

dynamic checks into the compiled code, but this may dramatically degrade performances. While compilation and implementation techniques exist for standard type structures, there is no equivalent for union and intersection types. This lack constitutes an objective obstacle to the adoption of sound gradual typing for dynamic languages. This work aims at filling this gap, by defining a virtual machine that copes with the overhead of using set-theoretic types. Before presenting the formal development we discuss some examples to give a more detailed account of the context.

Gradual Typing. The main idea of gradual typing is to introduce the type `?` to represent the *unknown type*. For example, the type `Int → ?` is the type of a function which will output an element of unknown type when given an input of integer type. Types with unknown components stem from either explicit annotations or because they are deduced. For instance in

```
let foo (x : ?) = x + 1
let bar (x : ?) = x
let baz (x : Int) = bar(x)
```

it is natural to assign the type `? → Int` to `foo`, `? → ?` to `bar`, and `Int → ?` to `baz`. This is so because in gradual typing the types `Int` and `?` are *consistent*, that is, they are two types that at runtime *may* turn out to be the same (in particular because the `?` may turn out to be any type and, thus, `Int`).[1] A gradual type-checker checks that types are consistent rather than equal, which is why it allows to use expressions of type unknown `?` where an expression of a static type such as `Int` is expected (e.g., the `x` in the body of `foo`) and use an expression of a static type such as `Int` where an expression of unknown type is expected (e.g., the `x` in the body of `baz`). The absence of type annotation can be either treated as an implicit `?` annotation (dynamic languages are a special case of it) or by using classic type reconstruction techniques (as in ML).

Cast Calculus. In practice, if an expression e of type `?` is used in a context that requires an integer, such as $e + 1$, the gradual type checker will consider that `?` is consistent with `Int`, and the expression $e + 1$ will type-check. But this might backfire, at runtime, if e evaluates to anything else than an integer. In order to ensure that this is not the case, the compiler will insert in the program safeguards that dynamically enforce type constraints on gradually-typed expressions. For instance, $e{+}1$ will be compiled into $e\langle\texttt{Int}\rangle{+}1$ where $e\langle\texttt{Int}\rangle$ is a type-cast expression that dynamically checks whether the result of e has type `Int`. In particular, `foo` will be compiled as `foo(x : ?) = (x⟨Int⟩ + 1)`. The target language of this compilation, with explicit type-casts, is called a *cast calculus*.
Let us recap the process with the following example:

```
let f (condition) (x : ?) =
  if condition then x + 1 else ¬x
```

This is a function that takes two arguments, `condition` and `x`, and returns either `x+1` if `condition` is `true`, or the negation of `x`, `¬x`, if `condition` is `false`. This code is typed with `Bool → ? → ?` by gradual type system of Siek and Vachharajani [24]. This knowledge is then used to insert dynamic type checks to ensure that the value bound to `x` will be, according to the case, an integer or a Boolean:

```
let f (condition) (x : ?) =
    if condition then x<Int> + 1 else ¬(x<Bool>)
```

But looking again at this example, we see that its derived type `Bool → ? → ?` is quite imprecise. For example, if we pass a value that is neither an integer nor a Boolean (e.g., a list) as the last argument `x` to `f`, then this application is well-typed due to the type-casts which enforces the type of `x`, even though the execution will always fail, independently of the value of `condition`. Likewise, the type gives no useful information about the result of `f`, even though it will clearly be either an integer or a Boolean, and nothing else. This problem can be solved by using more precise types and, in this case, set-theoretic types.

Set-theoretic types. Set-theoretic types include intersection types $t_1 \wedge t_2$, union types $t_1 \vee t_2$, and negation types $\neg t$. These constructors respect naive set intuitions, so that an expression of type t_1 and t_2 can be given the type $t_1 \wedge t_2$, etc. In our previous example, they allow the programmer to annotate the argument `x` more precisely:

```
let f (condition) (x :  (Int | Bool) & ?) =
    if condition then x<Int> + 1 else ¬(x<Bool>)
```

where "`|`" denotes union and "`&`" denotes intersection. The union `(Int | Bool)` indicates that a value of this type is *either* an integer *or* a Boolean, and the intersection indicates that `x` has *both* type `(Int | Bool)` *and* type `?`. Intuitively, this type annotation means that the function `f` accepts for `x` a value of any type (which is indicated by `?`), as long as this value is also either an integer *or* a Boolean. The use of the intersection of a union type with "`?`" to type a parameter corresponds to a programming style in which the programmer asks the system to *statically* enforce that the function will be applied only to arguments in the union type and delegates to the system any *dynamic* check regarding the use of the parameter in the body of the function. The system by Castagna et al. [7] is able to deduce for this code example the type:

```
Bool → ((Int | Bool) & ?) → (Int | Bool)
```

The return type is no longer gradual, thanks to set-theoretic types. This is a useful feature as more precision in types may be a source of optimizations. In this case, it might be crucial for the run-time type checker or the compiler to have the information that the result will be of type `(Int | Bool)`. However, the use of set-theoretic types brings new complexity issues when dealing with casts. In this example, it was easy to reconstruct the return type of `f` because the operators + and negation ¬ had simple static types: respectively, `Int×Int→Int` and `Bool→Bool`. But set-theoretic types encode complex function types whose return types are not obvious and need to be computed. For example, intersection types can be used to encode *ad-hoc polymorphism* (a.k.a., function overloading), where a piece of code acts on more than one type with different behavior in each case. See the type $\tau = (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$, which denotes functions that will return an integer if applied to an integer, and will return a Boolean if applied to a Boolean. It is possible to compute the domain of such a type (i.e., of all functions of this type), denoted by $\texttt{dom}\,\tau = \texttt{Int} \vee \texttt{Bool}$, that is the union of all the possible input types. But what is the precise return type of such a function? It depends on what the argument of such a function is: either an integer or a Boolean—or both, denoted by the union type `Int ∨ Bool`. Therefore, an operator on types needs to be defined

[1] For example the three types of `foo`, `bar`, and `baz` are pairwise consistent, but the type `Bool → ?` is consistent only with the first two.

which we denote by $\circ$. More precisely, we denote by $\tau_1 \circ \tau_2$ the type of an application of a function of type τ_1 to an argument of type τ_2. In the example with $\tau = (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$, it gives $\tau \circ \texttt{Int} = \texttt{Int}$, $\tau \circ \texttt{Bool} = \texttt{Bool}$, and $\tau \circ (\texttt{Int} \vee \texttt{Bool}) = \texttt{Int} \vee \texttt{Bool}$. The execution of a cast calculus with set-theoretic types requires the operations of domain and result to be computed at run-time (e.g., when an unknown function is cast to the type τ above), likewise for the subtyping relation. In this paper we show how to limit the impact of these computations on performance.

Overview. In Section 2, we start by presenting the intuitions behind the gradual set-theoretic types, we describe the main challenges to obtain a space efficient implementation, and we formally describe the cast language implemented by our virtual machine: its syntax, dynamic, and static semantics. Section 3 describes our virtual machine and how to compile the cast language into its bytecode. Section 4 studies the space efficiency of our machine and Section 5 its time efficiency via a chosen set of benchmarks. Discussion of related work of future extension and a conclusion end the presentation. For space reasons, several definition and all the proofs are relegated to an appedix available online. The source code of our virtual machine and of the benchmarks is available at https://github.com/gliboc/cast-machine.

2 CAST LANGUAGE

In this section we present the cast language implemented by our virtual machine. The syntax is that of the cast language by Castagna et al. [7] tweaked for efficient implementation. In particular, we modify the syntax of casts to remove unessential parts and to memoize the information about the domain of function types (for the reasons we explain in Section 2.2). So our casts will be of the form $\langle \tau_1 \rangle^{\tilde{\tau}_2}$, where $\tilde{\tau}_2$ is used to store information about the domain of τ_1 when this is a function type and it is $\bot$ otherwise. We do not explain how to compile a gradually-typed program into a program of the cast language since this is essentially[2] the same as in [7]. To define the cast language we have to introduce the gradual types and their subtyping relation (the *consistency* relation is factored out by the compilation), the syntax of the calculus and its dynamic and static semantics, as we do next.

2.1 Gradual set-theoretic types

Let b range over a set $\mathcal{B}$ of *basic types* (e.g., $\mathcal{B} = \{\texttt{Int}, \texttt{Bool}\}$). Following the approach of Castagna et al. [7], we define both gradual and static (a.k.a. non-gradual) set-theoretic types as the terms produced coinductively by the following grammars

$$\begin{array}{rl} \text{static types} & t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} \\ \text{gradual types} & \tau ::= ? \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \end{array}$$

and that satisfy the following conditions:

- *(regularity)* the term has a finite number of different sub-terms;
- *(contractivity)* every infinite branch of a type contains an infinite number of occurrences of products or arrows.

We refer the reader to [13] and [7] for more explanation about these restrictions. We introduce the following abbreviations for gradual types: $\tau_1 \wedge \tau_2 \stackrel{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2)$, $\tau_1 \setminus \tau_2 \stackrel{\text{def}}{=} \tau_1 \wedge \neg\tau_2$, $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$, and likewise for static types. We refer to b, $\times$, and $\to$ as *type constructors* and to $\vee$, $\wedge$, $\neg$, and $\setminus$ as *type connectives*.

Type connectives are only truly meaningful in presence of a subtyping relation: union and intersection can respectively be defined as the least upper bound and the greatest lower bound for this relation, while $\mathbb{0}$ and $\mathbb{1}$ can be respectively defined as the extrema of the lattice. In this work, we choose to reuse the semantic subtyping relation defined on both static and gradual set-theoretic types by Castagna et al. [7]. This definition being complex, we omit the details in this paper, and instead give the following sound but not complete set of rules for intuition:

$$\frac{}{\tau \le \tau} \qquad \frac{}{\tau \le \mathbb{1}} \qquad \frac{\tau_1 \le \tau_2}{\neg\tau_2 \le \neg\tau_1} \qquad \frac{\tau_1 \le \tau_3 \quad \tau_2 \le \tau_4}{\tau_1 \times \tau_2 \le \tau_3 \times \tau_4}$$

$$\frac{\tau_3 \le \tau_1 \quad \tau_2 \le \tau_4}{\tau_1 \to \tau_2 \le \tau_3 \to \tau_4} \qquad \frac{\tau \le \tau_1 \quad \tau \le \tau_2}{\tau \le \tau_1 \wedge \tau_2} \qquad \frac{\tau_1 \le \tau \quad \tau_2 \le \tau}{\tau_1 \vee \tau_2 \le \tau}$$

In particular, ? is only a subtype of itself and of $\mathbb{1}$, and the same holds for $\neg$?.

2.2 Space Efficiency

There are two major problems one must solve when defining a space-efficient cast language with set-theoretic types.

Function Application. The first problem concerns the application of cast functions (i.e., of functions that have a cast applied to them). This issue comes from the fact that checking the type of a function against a cast is highly impractical. Consider for example:

```
let f (x : ?) = if x = 42 then 42 else true
```

This function can be given the dynamic type ? $\to$?. Since, intuitively, ? stands for any type (or the absence of type), this function can be cast to type Int $\to$ Int. Moreover, it is clear that, for a certain value of its parameter ($x = 42$), this function returns an integer. Therefore, there are some execution contexts where casting this function to Int $\to$ Int is safe and will not produce a runtime error. However, it is also clear that this function *cannot be given* type Int $\to$ Int since it can also return a Boolean, and in some execution contexts, such a cast would fail.

The solution to decide whether a cast function such as $\texttt{f}\langle \texttt{Int} \to \texttt{Int} \rangle$ should produce an error is to delay the evaluation of casts after the application of the function, as is usually done in the gradual typing literature [26]. The basic idea is to define a reduction rule for cast functions that resembles the following one, where $e\langle\tau\rangle$ denotes a cast of the expression e to the type τ and v denote values:

$$[\beta_{\langle\rangle}] \qquad v_1\langle\tau_1 \to \tau_2\rangle\ v_2 \quad \longrightarrow \quad (v_1\ v_2\langle\tau_1\rangle)\langle\tau_2\rangle$$

The presence of set-theoretic types make this problem much more difficult however, because casts applied to functions can contain arbitrarily complex types (intersections of unions of arrows). Hence, we need to use operators on types to compute the domain and the result type of an application. Using the operators outlined in the introduction and formally defined in [5], the rule becomes:

$$[\beta_{\langle\rangle}] \qquad v_1\langle\tau\rangle\ v_2 \quad \longrightarrow \quad (v_1\ v_2\langle\mathsf{dom}(\tau)\rangle)\langle\tau \circ \mathsf{type}(v_2)\rangle$$

where $\mathsf{dom}(\tau)$ computes the domain of τ, and $\tau \circ \tau'$ computes the result type of the application of a function of type τ to an argument of type τ', and $\mathsf{type}(e)$ returns the type of the expression e.

[2] The only difference is that whenever the compilation in [7] produces a cast of the form $\langle \tau' \stackrel{\ell}{\Rightarrow} \tau \rangle$ here we produce the cast $\langle \tau \rangle^{\mathsf{dom}\,\tau}$

Cast Accumulation. The second problem concerns casts that accumulate and create sequences of growing length. When the cast is in tail position, it can blow up the return stack during execution. For example, consider the mutually recursive functions:

```
let rec odd : Int -> ? = fun n ->
    if n = 0 then false<?>
    else (even (n-1))<?>
and even : Int -> Bool = fun n ->
    if n = 0 then true
    else (odd (n-1))<Bool>
```

Originally, these functions are written by the programmer in a gradually-typed language without casts, which corresponds to the code above without the green parts (i.e., the casts). During type-checking, the type annotation on function `odd`, which is `Int→?`, drives the type-checker to add explicit type-casts that enforce `?` to be the return type of `odd`. In the same way, in the body of `even`, the call to `odd` is considered to return an element of type `?` and has to be cast to `Bool`. Now, a naive evaluation of `odd 5` would yield:

```
odd 5  →  (even 4)<?>
       →  (odd 3)<Bool><?>
       →  (even 2)<?><Bool><?>
       →  (odd 1)<Bool><?><Bool><?>
       →  (even 0)<?><Bool><?><Bool><?>
```

This short evaluation sequence highlights the problem many cast semantics have, which is space-inefficiency due to the accumulation of casts, possibly breaking tail-recursion. A solution that is used by the Grift compiler [18] is to implement a calculus on these casts, called the coercion calculus, in order to allow the composition of casts. Space efficiency is then achieved with the reduction that takes a chain-cast expression $E\langle c_1\rangle\langle c_2\rangle$ and computes $E\langle c_1 \fatsemi c_2\rangle$ where the $\fatsemi$ operation normalizes the sequencing of coercions c_1 and c_2 into a bounded-size representation. This approach has been carried out in the context of a simply-typed language (with functions, products), but it does not easily transfer to set-theoretic types.

We argue that, in the absence of blame-tracking, set-theoretic types can easily solve this problem. Remark that, intuitively, if an expression can be successfully cast to τ and to τ', it means that it has both of these types or, equivalently, that it has type $\tau \wedge \tau'$. Therefore, an expression $E\langle\tau\rangle\langle\tau'\rangle$ can be "compressed" to $E\langle\tau \wedge \tau'\rangle$. This idea is at the core of our space-efficient cast language we present next.

2.3 Language syntax

The expressions of the cast language are defined as follows (where $x \in \text{Var}$ ranges over variables and c over constants):

$$\begin{array}{lll} \text{Expr} & E ::= & x \mid c \mid \mu^{\tau\to\tau}\, f\, x\,.\,E \mid E\,E \mid E\langle\tau\rangle^{\tilde\tau} \\ & & \mid \texttt{let}\ x = E\ \texttt{in}\ E \mid \texttt{if}\ E\,E\,E \mid (E,E) \mid \pi_i\, E \\ \text{Types}^{\perp} & \tilde\tau ::= & \tau \mid \perp \end{array}$$

For the most part, this language is a standard λ-calculus with constants, pairs (E,E), projections $\pi_i\, E$ (with $i \in \{1,2\}$), and a let construct. $(\mu^{\tau_1\to\tau_2}\, f\, x\,.\,E)$ stands for an abstraction explicitly annotated with type $\tau_1 \to \tau_2$, which uses f as a recursive binder. Intuitively, the application of such a function to a value v (values are defined in the next section) reduces to $E[f := \mu^{\tau_1\to\tau_2}\, f\, x\,.\,E][x := v]$. More importantly, the construct $E\langle\tau\rangle^{\tilde\tau}$ stands for an expression that is cast to type τ, and that can safely be applied to elements of type $\tilde\tau$, provided it is a function (otherwise, $\tilde\tau = \perp$). Leaving the exponent part aside, this is a fairly standard cast construct [23, 26] where only information about the target type is kept (we do not need the source type, nor do we handle blame). If E reduces to some constant c then $E\langle\tau\rangle^{\tilde\tau}$ reduces to c if c has type τ, and fails otherwise.

The need for the exponent part comes from the fact that, as we previously said, we merge successive casts together into a single one (in our implementation space depends more on the number of casts than on the form of their types). This operation is straightforward thanks to set-theoretic types: an expression $E\langle\tau\rangle\langle\tau'\rangle$ is "compressed" to $E\langle\tau \wedge \tau'\rangle$. However, for function casts, a different operation must be done on the domain of the cast. Using the semantics for cast applications we described before, an application $v\langle \texttt{Int}\to\texttt{Int}\rangle\langle ?\to ?\rangle\langle\texttt{Bool}\to\texttt{Bool}\rangle v'$ must never succeed, since this would result in the argument v' being cast from `Bool` to `?` to `Int` before being passed to the function v, and no value can be of both types `Bool` and `Int`. However, if we were to merge these casts naively, we would obtain the application $v\langle(\texttt{Int}\to\texttt{Int})\wedge(?\to ?)\wedge(\texttt{Bool}\to\texttt{Bool})\rangle v'$. Since $(\texttt{Int}\to\texttt{Int})\wedge(?\to ?)\wedge(\texttt{Bool}\to\texttt{Bool})$ is a subtype of $(\texttt{Int}\vee ?\vee\texttt{Bool})\to(\texttt{Int}\vee ?\vee\texttt{Bool})$, then in particular this application will succeed whenever v' is of type either `Int` or `Bool`, which would be unsound. More formally, the issue is that taking the intersection of two function types increases their domains (the domain of an intersection being the union of the domains), while casting a function to two different types makes it less likely to succeed when applied. To compress casts by intersections, thus, we record casts on the domain of a function independently from its type, hence the exponent. In particular we now compress $E\langle\tau\rangle^{\tau_1}\langle\tau'\rangle^{\tau_2}$ into $E\langle\tau\wedge\tau'\rangle^{\tau_1\wedge\tau_2}$. Adding this information to casts, the previous application becomes $v\langle\texttt{Int}\to\texttt{Int}\rangle^{\texttt{Int}}\langle ?\to ?\rangle^{?}\langle\texttt{Bool}\to\texttt{Bool}\rangle^{\texttt{Bool}} v'$, which reduces to $v\langle(\texttt{Int}\to\texttt{Int})\wedge(?\to ?)\wedge(\texttt{Bool}\to\texttt{Bool})\rangle^{\texttt{Int}\wedge ?\wedge\texttt{Bool}} v'$. This expression will then always fail, since the type of the argument is now checked against the exponent, and $\texttt{Int}\wedge ?\wedge\texttt{Bool}$ is empty (since $\texttt{Int}\wedge\texttt{Bool}$ is so) and thus no value can have this type.

2.4 Big step semantics

We present the big step semantics of the cast calculus since it is closer to our our virtual machine (cf. Theorem 2). The (equivalent) small step semantics is used just to state the type safety property and omitted for space reasons (it can be found in Appendix B.1). First we define the syntax of values, for which we partition the set of constants (ranged over by c) in functional constants (i.e., constants with a function type) ranged over by $o \in \text{Functionals}$ and non functional ones ranged over by $k \in \text{Constants}$:

$$\begin{array}{lll} \text{Func} & u := & o \mid \mu^{\tau\to\tau}\, f\, x\,.\,E \\ \text{Val} & v := & k \mid (v,v) \mid \left[u, \mathcal{E}, \langle\tau\rangle^{\tau}\right]^m \\ \text{Mark} & m := & \star \mid \square \end{array}$$

The operator `type`[3] is defined on values $v \in \text{Val}$ as:

$$\begin{array}{rl} \texttt{type}(\left[u, \mathcal{E}, \langle\tau_1\rangle^{\tau_2}\right]^m) & = \tau_1 \\ \texttt{type}\ k & = \mathbb{B}(k) \wedge ? \wedge \neg ? \\ \texttt{type}\,(v_1, v_2) & = \texttt{type}\ v_1 \times \texttt{type}\ v_2 \end{array}$$

[3]This operator is used to get the type of values that are used as arguments of an application in order to compute the result type of the application. For more details on the definition, especially in the case of constants, see the Section 2.5 (cf., rule [Const]).

where $\mathbb{B}$ maps non functional constants into their basic type. Our values include closures $\left[u, \mathcal{E}, \langle\tau\rangle^{\tau}\right]^{m}$, which are formed by a functional expression u, an environment $\mathcal{E}$ which is a finite function $\texttt{Var} \to \texttt{Val}$, a pair of types and, at the index, a mark m. This mark distinguishes between two cases: when $m = \square$, it means that the cast $\langle\tau\rangle^{\tau}$ is waiting to be applied to the function inside the closure; when $m = \star$, $\langle\tau\rangle^{\tau}$ is kept as a static type information regarding this function, but the closure will be applied without adding or enforcing type-casts. In other terms, $\square$-closures represent functions to which a cast is applied (they will be reduced by using $\beta_{\langle\rangle}$) while $\star$-closures are usual functions without any cast (they will be reduced by the usual β). We distinguished between functional constants o and constants k, because we want to have a support for pre-defined functions in our language (e.g., +, *incr*, …), and because casts behave differently on these two kinds of constants.

Closures are created by capturing the current environment and the type in the abstraction. At first, such a closure should not enforce any typing constraints, hence the $\star$ mark. Functional constants are wrapped into similar closure only with empty environments since they do not have free variables needing to be bound.

$$\mathcal{E} \vdash \mu^{\tau_1 \to \tau_2}\, f\, x\, .\, E \Rightarrow \left[\mu^{\tau_1 \to \tau_2}\, f\, x\, .\, E,\, \mathcal{E},\, \langle \tau_1 \to \tau_2 \rangle^{\tau_1}\right]^{\star}$$

Then, the following rules handle function application. The first, $[\beta_\star]$, is a standard untyped function application that corresponds to the standard β-reduction. The second, $[\beta_\square]$, enforces cast constraints on the arguments and on the result of the application, which this time corresponds to the rule $[\beta_{\langle\rangle}]$ outlined earlier. We denote by $\mathcal{E}\{x := v\}$ a copy of $\mathcal{E}$ which outputs v when given x.

$$[\beta_\star]\ \frac{\mathcal{E} \vdash E_1 \Rightarrow v_1 = [\mu^{\tau}\, f\, x\, .\, E,\, \mathcal{E}',\, \kappa]^{\star} \qquad \mathcal{E} \vdash E_2 \Rightarrow v_2 \qquad \mathcal{E}'\{x := v_2\ f := v_1\} \vdash E \Rightarrow v}{\mathcal{E} \vdash E_1\, E_2 \Rightarrow v}$$

$$[\beta_\square]\ \frac{\mathcal{E} \vdash E_1 \Rightarrow [u,\, \mathcal{E}',\, \langle\tau_1\rangle^{\tau_2}]^{\square} \qquad v_1 = [u,\, \mathcal{E}',\, \langle\tau_1\rangle^{\tau_2}]^{\star} \qquad \mathcal{E} \vdash E_2 \Rightarrow v_2 \qquad \mathcal{E} \vdash v_2\langle\tau_2\rangle^{\mathsf{dom}\,\tau_2} \Rightarrow v_3 \qquad \mathcal{E} \vdash (v_1\, v_3)\langle\tau_1 \circ \mathsf{type}(v_2)\rangle^{\mathsf{dom}(\tau_1 \circ \mathsf{type}(v_2))} \Rightarrow v}{\mathcal{E} \vdash E_1\, E_2 \Rightarrow v}$$

Let us see how $[\beta_\square]$ works. When applying a closure $[u,\, \mathcal{E}',\, \langle\tau_1\rangle^{\tau_2}]^{\square}$, the index type τ_2 should already capture the type of the argument. Therefore, the argument E_2 is evaluated to a value v_2 and then cast to v_3 using τ_2. During that time, the type $\tau = \tau_1 \circ \mathsf{type}(v_2)$ is computed in order to best approximate the result type of the function call. Finally, $(v_1\, v_3)\langle\tau\rangle^{\mathsf{dom}\,\tau}$ can be evaluated, where v_1 is the closure of the beginning whose mark has been set to $\star$, meaning that the next reduction used will be the untyped one, $[\beta_\star]$.

The problem of accumulating casts is solved by systematically reducing such proxies both on constants and on closures. Reduction of casts on closures is achieved by the following rule:

$$\frac{\mathcal{E} \vdash E \Rightarrow [u,\, \mathcal{E}',\, \langle\tau_3\rangle^{\tau_4}]^{m}}{\mathcal{E} \vdash E\,\langle\tau_1\rangle^{\tau_2} \Rightarrow \left[u,\, \mathcal{E}',\, \langle\tau_1 \wedge \tau_3\rangle^{\tau_2 \wedge \tau_4}\right]^{\square}}$$

This rule uses intersection types to compress type-casts on closures. As we show in Section 4, this representation is bounded in space according to the number of type annotations in the source program. After computing the new type-cast, the closure is tagged with $\square$ in order to signify that it should be applied using rule $[\beta_\square]$, which enforces type constraints.

In order to reduce casts on constants, we consider the fact that the gradual part of a cast does not influence the result of the cast. Indeed, if we cast an integer to a gradual type, as in `42<Int∧?>` or `42<Bool∧?>`, the only parts of theses casts that have a consequence are `Int` or `Bool`, since `?` could represent anything. Therefore, we can erase `?` in both these casts, and re-apply the newly-obtained cast.[4] To this purpose we define the extrema of gradual types in order to be able to cast constants to gradual types.

Definition 1. **(Gradual Extrema)** *For every gradual type τ,*

- *the* gradual maximum $\tau^{\Uparrow}$ *is obtained by replacing every* covariant *occurrence of* ? *by* $\mathbb{1}$*, and every* contravariant *occurrence by* $\mathbb{0}$
- *the* gradual minimum $\tau^{\Downarrow}$ *is obtained by replacing every* contravariant *occurrence of* ? *by* $\mathbb{1}$*, and every* covariant *occurrence by* $\mathbb{0}$

This definition is made so that the type-cast from $\mathbb{B}(k)$ to τ_1 will succeed whenever $\mathbb{B}(k) \leq \tau_1^{\Uparrow}$—i.e., that it is not a problem to ignore the gradual part of the cast that is erased by taking the gradual extrema, since all constants are implicitly gradually typed (see Footnote 1). The condition resulting from, written $\mathbb{B}(k) \leq \tau_1^{\Uparrow}$, is implemented by the two following rules, which imply the full reduction of casts on constants:

$$\frac{\mathcal{E} \vdash E \Rightarrow k \quad \mathbb{B}(k) \leq \tau_1^{\Uparrow}}{\mathcal{E} \vdash E\langle\tau_1\rangle^{\tau_2} \Rightarrow k} \qquad \frac{\mathcal{E} \vdash E \Rightarrow k \quad \mathbb{B}(k) \not\leq \tau_1^{\Uparrow}}{\mathcal{E} \vdash E\langle\tau_1\rangle^{\tau_2} \Rightarrow \texttt{Fail}}$$

The full set of rules for the big step semantics can be found in Figure 10 in the annexes.

2.5 Type System

Since a cast language is not meant to be used by a programmer, but rather by the compiler as an intermediate language, defining its type system is only necessary to prove the soundness of its semantics, which is the point of this section. The type systems uses most of the standard rules of a simply-typed lambda-calculus with pairs plus classic subsumption for subtyping:

$$[\textsc{Subsume}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E : \tau}\ \tau' \leq \tau$$

However, there are two major differences coming from the typing rules for non-functional constants and for casts, stated as follows:

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash k : \mathbb{B}(k) \wedge\ ? \wedge \neg?} \qquad [\textsc{Cast}]\ \frac{\Gamma \vdash E : \tau}{\Gamma \vdash E\langle\tau_1\rangle^{\tilde{\tau}_2} : \tau_1}$$

The rule for casts states that, since τ_1 is the most precise type that can be given to $E\langle\tau_1\rangle^{\tau_2}$ if the cast succeeds, we simply give this type to the expression, provided E is well-typed. Notice that both τ and $\tilde{\tau}_2$ are disregarded by the rule

The rule for constants is certainly the most bizarre one. Intuitively, this rule should be understood in this way: a constant k can be given type $\mathbb{B}(k)$ (which is its expected type), but can also be implicitely downcast to type ? and to $\neg$?. This comes from the fact that, in order to further optimize space consumption, we chose to remove all casts on non-functional constants. Consider for example the cast constant `42<?>`: this cast can never be the cause of an error, as a value can always be cast to the dynamic type, independently of its type. Therefore, such a cast can be removed without altering the

[4] This reduction erases the information given by having type-casts such as ? on constants. Our type system will take that into account by adding a typing rule which allows the type-checker to implicitly type constants by "?" (and by "¬?" as well).

semantics of the program. However, this means in particular that 42<?>, which is of type ?, reduces to 42, which would intuitively be of type Int. Such a reduction would violate type preservation, hence the need for the rule [Const] which allows constant to be implicitely consider of type ?.
The part ¬? can be understood with the same reasoning, as ? and ¬? are actually semantically equivalent (but incomparable for subtyping). Indeed, since ? can intuitively represent any type τ, it can also represent the negation of any type $\neg\tau$. Therefore, ¬? can represent any double negation $\neg\neg\tau$, thus any type τ.
Using this type system, we can then prove for small step semantics the traditional lemmas from which the type safety theorem follows:

Lemma 1 (Preservation). *If* $\Gamma \vdash E : \tau$ *and* $E \longrightarrow E'$, *then* $\Gamma \vdash E' : \tau$.

Lemma 2 (Progress). *If* $\Gamma \vdash E : \tau$ *and* E *is closed, then either* E *is a value or* $E \longrightarrow E'$ *for some* E'.

Theorem 1 (Type safety). *If* $\Gamma \vdash E : \tau$ *and* E *is closed, then either* E *diverges, or* $E \longrightarrow^\star v$ *for some value* v, *or* $E \longrightarrow^\star \texttt{Fail}$.

In order to prove that using $[\beta_\square]$ is practical, we now build a virtual machine that computes function calls space-efficiently.

3 VIRTUAL MACHINE

In this section we define our virtual machine and the compilation of the cast language into it. The main interest of our machine is that it provides a space-efficient implementation in presence of set-theoretic types. The two main problems of space-efficiency which we tackle are the efficient representation of casts and the compression of suite of casts (a sensitive problem in the presence of tail recursive calls of functions with cast).

3.1 Structure

A state of our machine is a 4-tuple (c, e, s, d) composed of a code pointer c, the current environment e, an operand stack s, and a control stack d (also called "dump") which handles return frames. The last three components of the machine are defined by the following grammar (the code c is defined below).

$$
\begin{array}{rrll}
 & \mathsf{s} ::= & \varnothing \mid v \,.\, \mathsf{s} \mid \tilde{\kappa} \,.\, \mathsf{s} \mid \Omega \,.\, \mathsf{s} & \textit{Operand Stack} \\
 & \mathsf{d} ::= & \varnothing \mid \tilde{\kappa} \,.\, \mathsf{d} \mid (\mathsf{c}, \mathsf{e}).\, \mathsf{d} & \textit{Control Stack} \\
 & \mathsf{e} ::= & \varnothing \mid v \,.\, \mathsf{e} & \textit{Environment} \\
\mathsf{MVal} & v ::= & c \mid (v, v) \mid [\mathsf{c}, \mathsf{e}, \kappa]^m & \textit{Machine Values} \\
 & \kappa ::= & \langle \tau \rangle^\tau & \textit{Type-casts} \\
 & \tilde{\kappa} ::= & \tau^{\tilde{\tau}} & \textit{Type Pairs} \\
 & m ::= & \star \mid \square & \textit{Reduction Marks}
\end{array}
$$

The stack stores machine values v which encode the values of the cast language), type-casts κ, used to cast machine values that were pushed on the stack, and a failure mark Ω which denotes the failure of a cast. Machine values MVal include constants, pairs, and closures $[\mathsf{c}, \mathsf{e}, \kappa]^m$ containing a piece of code c, an environment e, a type-cast κ and a mark which indicates how to apply the closure, similarly as in Section 2.4 in the big-step semantics. Each machine value MVal is associated to its minimal (w.r.t. subtyping) type by the type operator defined as:

$$
\begin{array}{lcl}
\mathsf{type}([\mathsf{c}, \mathsf{e}, \langle \tau_1 \rangle^{\tau_2}]^m) & = & \tau_1 \\
\mathsf{type}(c) & = & \mathbb{B}(c) \wedge ? \wedge \neg ? \\
\mathsf{type}((v_1, v_2)) & = & \mathsf{type}(v_1) \times \mathsf{type}(v_2)
\end{array}
$$

As for the operational semantics in Section 2.4, the mark □ in a closures indicates that this closure has a cast pending to be applied, while the ⋆ mark indicates that a closure does not require any casts to be applied. No closure can contain a cast of the form $\langle \tau \rangle^\perp$, which is a pattern that indicates a cast that may only succeed on constants, and which fails on functions. However this pattern, in the form of $\tilde{\kappa}$, can appear on both the operand and the control stack.
The code of the virtual machine is a suite of instructions defined as follows:

$$
\begin{array}{rll}
\mathsf{c} ::= & \varnothing \mid \mathtt{instr}\,;\mathsf{c} & \\
\mathtt{instr} ::= & \mid \mathsf{push}\ obj & \textit{Stack push} \\
 & \mid \mathsf{app} \mid \mathsf{ret} & \textit{Function call} \\
 & \mid \mathsf{tap} & \textit{Tail}\ \mathsf{app} \\
 & \mid \mathsf{tca}\ \tilde{\kappa} & \textit{Cast tail}\ \mathsf{app} \\
 & \mid \mathsf{cast} & \textit{Type-cast} \\
 & \mid \mathsf{ifz}\,(\mathsf{c}, \mathsf{c}) & \textit{Conditional} \\
 & \mid \mathsf{let} \mid \mathsf{end} & \textit{Let binders} \\
 & \mid \mathsf{pair} & \textit{Pairs} \\
obj ::= & n \mid k \mid o \mid (\mathsf{c}, \kappa) \mid \tilde{\kappa} & n \in \mathsf{Integers} \\
o ::= & \times \mid + \mid - \mid = \mid \mathit{fst} \mid \mathit{snd} &
\end{array}
$$

The push instruction injects identifiers, constants, closures, and types into the execution by pushing them onto the stack. Pair of types $\tilde{\kappa}$ are passed around in the stack and used by the tca instructions to create, compress, and apply casts at the execution.

3.2 Compilation

Our machine uses De Bruijn indices. To replace variables with indices, compilation uses a list of variables ρ and calls $\mathtt{lookup}\,\rho\,x$ which returns the index of the first occurrence of x in ρ.
The compilation process is decomposed into two functions. The function $C[\![]\!]_\rho : \mathsf{Expr} \to \mathsf{Bytes}$ is the general one. It calls $\mathcal{T}[\![]\!]_\rho$ to compile expressions of that cast language that are in tail position, that is, the bodies of functions and the bodies of let expressions that are in tail position:

$$
\begin{array}{lcl}
C[\![c]\!]_\rho & = & \mathsf{push}\ \ c \\
C[\![x]\!]_\rho & = & \mathsf{push}\ \ (\mathtt{lookup}\ \rho\ x) \\
C[\![\mu^{\tau_1 \to \tau_2} f\, x\, .\, E]\!]_\rho & = & \mathsf{push}\ \ \left(\mathcal{T}[\![E]\!]_{f.x.\rho}\,,\ \langle \tau_1 \to \tau_2 \rangle^{\tau_1}\right) \\
C[\![E\,\langle \tau_1 \rangle^{\tau_2}]\!]_\rho & = & \mathsf{push}\ \ \langle \tau_1 \rangle^{\tau_2}\,;\, C[\![E]\!]_\rho\,;\, \mathsf{cast} \\
C[\![E_1\, E_2]\!]_\rho & = & C[\![E_1]\!]_\rho\,;\, C[\![E_2]\!]_\rho\,;\, \mathsf{app} \\
C[\![\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2]\!]_\rho & = & C[\![E_1]\!]_\rho\,;\, \mathsf{let}\,;\, C[\![E_2]\!]_{x.\rho}\,;\, \mathsf{end} \\
C[\![(E_1, E_2)]\!]_\rho & = & C[\![E_1]\!]_\rho\,;\, C[\![E_2]\!]_\rho\,;\, \mathsf{pair}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}[\![E_1\, E_2]\!]_\rho & = & C[\![E_1]\!]_\rho\,;\, C[\![E_2]\!]_\rho\,;\, \mathsf{tap} \\
\mathcal{T}[\![(E_1\, E_2)\,\langle \tau_1 \rangle^{\tau_2}]\!]_\rho & = & C[\![E_1]\!]_\rho\,;\, C[\![E_2]\!]_\rho\,;\, \mathsf{tca}\ \langle \tau_1 \rangle^{\tau_2} \\
\mathcal{T}[\![\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2]\!]_\rho & = & C[\![E_1]\!]_\rho\,;\, \mathsf{let}\,;\, \mathcal{T}[\![E_2]\!]_{x.\rho} \\
\mathcal{T}[\![E]\!]_\rho & = & C[\![E]\!]_\rho\,;\, \mathsf{ret}
\end{array}
$$

Our machine follows a classic eval-apply pattern with some specificities. In particular, the evaluation of a μ-abstraction pushes on the stack (see the definition of push further on) the closure which contains the body of the function (compiled for tail position), an environment containing both the parameter and the recursion variable, and a cast formed by the type annotation of the function and indexed with its domain. Applications are evaluated from left to right by the instruction app. When they are in tail position applications are compiled using the tap instruction that—contrary to

app—does not save the calling context. If a cast is applied to a tail call, then the special tca instruction is used instead: acccording to the case, this instruction composes the cast on the tail call with the one on the top of the dump or the one in the callee, thus avoiding the problem of accumulation of casts we described before. Finally, casts that are not applied to tail calls are simply handled as plain operators whose arguments are first pushed on the stack and then evaluated by the execution of the cast instruction.

3.3 Transitions

3.3.1 Parameter functions. Several functions allow us to abstract functionalities of the virtual machine. The $\mathsf{push}_\mathsf{e}(obj, \mathsf{s})$ function handles adding elements to the stack.

$$\begin{array}{ll} \mathsf{push}_\mathsf{e}(n, \mathsf{s}) & = \mathsf{e}(n).\,\mathsf{s} \\ \mathsf{push}_\mathsf{e}(k, \mathsf{s}) & = k.\,\mathsf{s} \\ \mathsf{push}_\mathsf{e}((\mathsf{c}, \kappa), \mathsf{s}) & = [\mathsf{c}, \mathsf{e}, \kappa]^\star.\,\mathsf{s} \\ \mathsf{push}_\mathsf{e}(\tilde{\kappa}, \mathsf{s}) & = \tilde{\kappa}.\,\mathsf{s} \end{array}$$

The composition of two pairs of types is defined as the symmetric operator that satisfies:

$$\begin{array}{lcl} {\tau^\bot}_{\tilde{\tau}_2} \fatsemi {\tau_1}^{\tilde{\tau}_2} & = & {\tau_1}^{\tilde{\tau}_2} \fatsemi {\tau^\bot} \;=\; (\tau \wedge \tau_1)^\bot \\ {\tau_1}^{\tau_2} \fatsemi {\tau_3}^{\tau_4} & = & (\tau_1 \wedge \tau_3)^{\tau_2 \wedge \tau_4} \end{array}$$

The cast function implements the application of casts on constants or closures using intersections to compress successive applications of function casts.

$$\begin{array}{lcl} \mathsf{cast}\left(k,\ {\tau_1}^{\tilde{\tau}_2}\right) & = & k \quad \text{if } \mathbb{B}(k) \leq \tau_1{}^{\Uparrow} \\ \mathsf{cast}\left([\mathsf{c}, \mathsf{e}, \langle\tau_1\rangle^{\tau_2}]^m,\ {\tau_3}^{\tilde{\tau}}\right) & = & [\mathsf{c}, \mathsf{e}, \langle\tau_1 \wedge \tau_3\rangle^{\tau_2 \wedge \tilde{\tau}}]^\square \quad \text{if } \tilde{\tau} \neq \bot \\ \mathsf{cast}\,(v,\ \tilde{\kappa}) & = & \Omega \quad \text{in all other cases} \end{array}$$

The dump d can contain either call frames or type pairs, which is what allows the elimination of tail calls even with casts on them. During a tail call, this function accumulates casts on top of the dump stack, in order to apply the resulting cast when the tail calls are over and the result of the computation is returned.

$$\begin{array}{ll} \mathsf{dump}\,(\tilde{\kappa}_1,\ \tilde{\kappa}_2\,.\,\mathsf{d}) & = (\tilde{\kappa}_1 \fatsemi \tilde{\kappa}_2).\,\mathsf{d} \\ \mathsf{dump}\,(\tilde{\kappa},\ (\mathsf{c}, \mathsf{e})\,.\,\mathsf{d}) & = \tilde{\kappa}\,.\,(\mathsf{c}, \mathsf{e})\,.\,\mathsf{d} \\ \mathsf{dump}\,(\tilde{\kappa},\ \varnothing) & = \tilde{\kappa} \end{array}$$

The table in Figure 1 describes the complete set of transitions of the virtual machine. There are three kinds of reduction: the usual function application $[\beta_\star]$, tail function calls $[t\beta_\star]$, and cast tail function calls $[c\beta_\star]$. While the first two are standard, the third one mixes a tail function call and a delayed cast application. It works by pushing a type-cast on the dump to be applied later, while performing a tail function call. There exists a typed version of each of these reductions: $[\beta_\square]$, $[t\beta_\square]$ and $[c\beta_\square]$ which are performed when the closure applied has a type-cast on it. Finally, the return instruction ret is also performed by two transitions: the usual one $[R_\star]$, which replaces the current frame by the top frame on the dump; and a typed one $[R_\square]$, which applies the type-cast on top of the dump to the current value on top of the stack.

3.4 Example

Let us compute odd 5 from the odd/even example of Section 2.2. By inlining the function even into the definition of odd, we can compile odd into a piece of bytecode c_odd, which is put inside the closure $[\mathsf{c}_\mathsf{odd}, \mathsf{e}', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^\star$. Now, we want to compute:

$$C[\![\mathtt{odd}\ \ 5]\!]_\rho = \mathsf{push}\ \mathit{odd}\,;\mathsf{push}\ 5\,;\mathsf{app}$$

These several key points illustrate the most important transitions used to compute odd 5 (abstracting inessential details away, such as De Bruijn indexes and recursion variables):

- First, the function call odd 5 is handled by regular application:

$$\begin{array}{l} \left(\mathsf{app},\ \mathsf{e},\ [\mathsf{c}_\mathsf{odd}, \mathsf{e}', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^\star.\,5,\ \varnothing\right) \\ \quad \to \left(\mathsf{c}_\mathsf{odd},\ \{n := 5\}.\,\mathsf{e}',\ \varnothing,\ (\varnothing, \mathsf{e})\right) \end{array}$$

- Then, the code of odd is executed and, since $n = 5$ is not zero, the conditional branch for the code (even (n-1))<?> is chosen. This expression is in tail position *and* consists of a cast on a function call. Therefore, the $[c\beta_\star]$ transition is applied and adds a type-cast "?" on the dump stack. Then, a tail call tap is executed, that is, a function call that does not push the current frame on the dump stack.

$$\begin{array}{l} \left(\mathsf{tca}\ ?,\ \{n := 5\}.\,\mathsf{e}',\ [\mathsf{c}_\mathsf{even}, \mathsf{e}'', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^\star.\,4,\ (\varnothing, \mathsf{e})\right) \\ \quad \to \left(\mathsf{tap},\ \{n := 5\}.\,\mathsf{e}',\ [\mathsf{c}_\mathsf{even}, \mathsf{e}'', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^\star.\,4,\ ?\,.\,(\varnothing, \mathsf{e})\right) \\ \quad \to \left(\mathsf{c}_\mathsf{even},\ \{n := 4\}.\,\mathsf{e}'',\ \varnothing,\ ?\,.\,(\varnothing, \mathsf{e})\right) \end{array}$$

- The execution of c_even is similar to the one of c_odd, and adds to the dump the typecast Bool which *composes* with the cast ? already on the top of the dump, resulting in the following:

$$\to \left(\mathsf{c}_\mathsf{odd},\ \{n := 3\}.\,\mathsf{e}',\ \varnothing,\ (\mathtt{Bool} \wedge ?)\,.\,(\varnothing, \mathsf{e})\right)$$

- These cast tail calls keep decreasing the value of n, building up a *single* type-cast on top of the dump, until even 0 is called, which returns true. But since a type-cast has been put on the dump stack, the result true must pass this type-cast before being returned, which yields the following final execution:

$$\begin{array}{l} \left(\mathsf{push}\ \mathsf{true}\,;\mathsf{ret},\ \{n := 0\}.\,\mathsf{e}'',\ \varnothing,\ (\mathtt{Bool} \wedge ?)\,.\,(\varnothing, \mathsf{e})\right) \\ \quad \to \left(\mathsf{ret},\ \{n := 0\}.\,\mathsf{e}'',\ \mathsf{true},\ (\mathtt{Bool} \wedge ?)\,.\,(\varnothing, \mathsf{e})\right) \\ \quad \to \left(\mathsf{cast}\,;\mathsf{ret},\ \{n := 0\}.\,\mathsf{e}'',\ (\mathtt{Bool} \wedge ?)\,.\,\mathsf{true},\ (\varnothing, \mathsf{e})\right) \\ \quad \to \left(\mathsf{ret},\ \{n := 0\}.\,\mathsf{e}'',\ \mathsf{true},\ (\varnothing, \mathsf{e})\right) \\ \quad \to \left(\varnothing,\ \mathsf{e},\ \mathsf{true},\ \varnothing\right) \end{array}$$

This example shows how the mechanism for handling casts on tail calls work in our virtual machine, by compressing type-casts on the dump stack using intersection types. Next we prove that this mechanism respects the semantics of our language.

We denote by $[\![\ ]\!]$ the function that maps the cast language values v into the corresponding machine values v, and extend it to environments $[\![\mathcal{E}]\!]$ pointwise. The link between the big step semantics and the virtual machine, is stated by the following theorem.

Theorem 2 (Soundness). *For every term E and environment $\mathcal{E}$, if $\mathcal{E} \vdash E \Rightarrow v$, then* $\forall(\mathsf{c}, \mathsf{s}, \mathsf{d})\,(C[\![E]\!]\,;\mathsf{c}, [\![\mathcal{E}]\!], \mathsf{s}, \mathsf{d}) \to^\star (\mathsf{c}, [\![\mathcal{E}]\!], [\![v]\!].\mathsf{s}, \mathsf{d})$

3.5 Symbolic casts

A drawback of this virtual machine is that it relies on high-order operations on types: dom, $\circ$, $\wedge$ (see the three $\_\beta_\square$ rules in Figure 1). Thank to the representation of types we describe in Section 4.1.1, intersection are less costly than the first two, since they consist of a simple merge of Binary Decision Diagrams (BDDs). This is why we

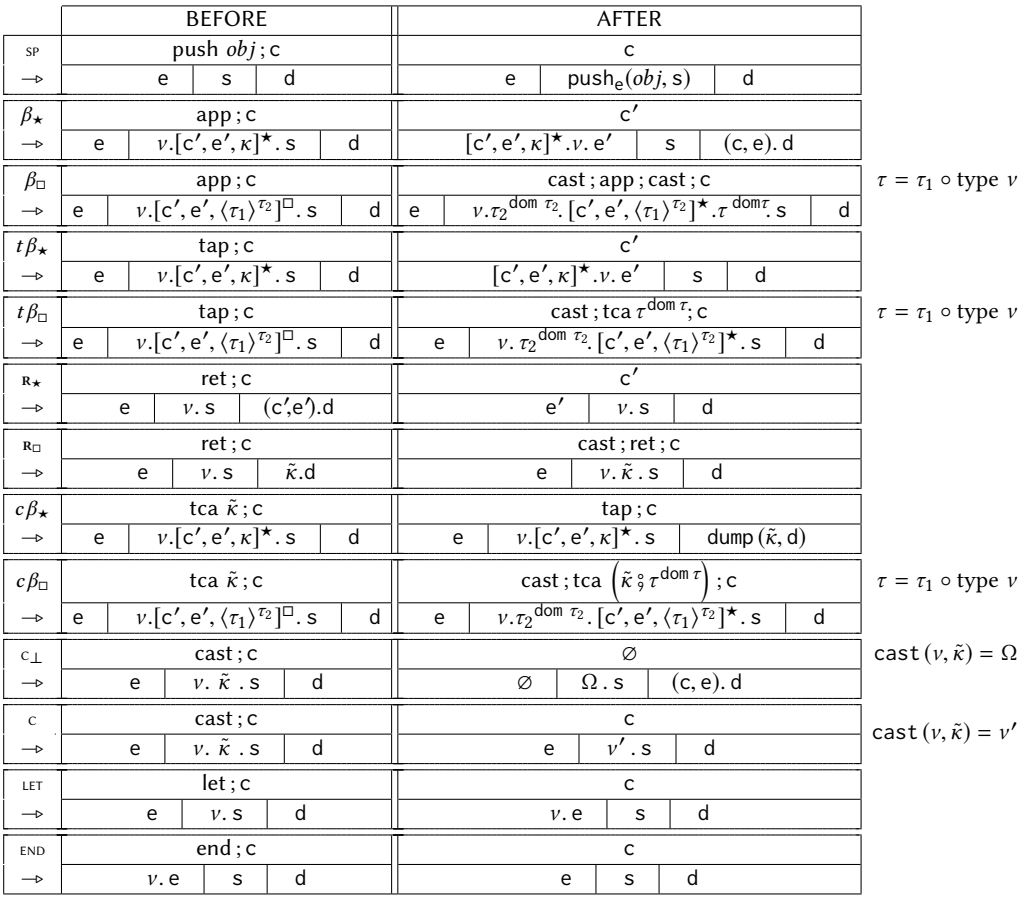


Fig. 1: Transitions of the virtual machine

concentrate on dom and ∘ and explore the possibility of delaying their application in a symbolic structure for casts.
This yields a variant of the virtual machine, which differs from the first in that it replaces pairs of types $\tilde{\kappa}$ by symbolic casts Σ:

$$\begin{array}{rrl} \Sigma\text{Bytes} & \mathsf{c} ::= & \varnothing \mid \mathtt{instr}\,;\mathsf{c} \\ & \Sigma ::= & \tilde{\kappa} \mid \mathtt{dom}\,\Sigma \mid \mathtt{app}_\tau(\Sigma) \\ & \mathtt{instr} ::= & \ldots \mid \mathtt{tca}\,\Sigma \mid \ldots \end{array}$$

Symbolic casts encode in their structure the calls to domain dom and cast composition ∘, and they are lazily evaluated by eval:

$$\begin{array}{ll} \mathtt{eval}\,(\mathtt{dom}\,\Sigma) & = \tilde{\tau}_2{}^{\mathtt{dom}\,\tilde{\tau}_2} \quad \text{with } \mathtt{eval}\,(\Sigma) = \tau_1{}^{\tilde{\tau}_2} \\ \mathtt{eval}\,(\mathtt{app}_\tau\,\Sigma) & = \tau_r{}^{\mathtt{dom}\,\tau_r} \quad \text{with } \mathtt{eval}\,(\Sigma) = \tau_1{}^{\tilde{\tau}_2} \\ & \qquad\quad \text{and } \tau_r = \tau_1 \circ \tau \end{array}$$

Structures. The structures of the machine are slightly different as well, as again we replace pairs of types $\tilde{\kappa}$ with symbolic casts Σ.

$$\begin{array}{rrll} \Sigma\text{Val} & v ::= & c \mid (v, v) \mid [\mathsf{c}, \mathsf{e}, \Sigma]^m & \textit{Machine values} \\ & \mathsf{s} ::= & \varnothing \mid v\,.\,\mathsf{s} \mid \Sigma.\,\mathsf{s} \mid \Omega\,.\,\mathsf{s} & \textit{Operand stack} \\ & \mathsf{d} ::= & \varnothing \mid \Sigma.\,\mathsf{d} \mid (\mathsf{c}, \mathsf{e}).\,\mathsf{d} & \textit{Control stack} \end{array}$$

Parameter functions. Since intersections are not symbolic operations, the composition of two symbolic pairs Σ_1, Σ_2 is defined as:

$$\begin{array}{lcl} \tau^\perp \fatsemi \tau_1{}^{\tilde{\tau}_2} & = & \tau_1{}^{\tilde{\tau}_2} \fatsemi \tau^\perp = (\tau \wedge \tau_1)^\perp \\ \tau_1{}^{\tau_2} \fatsemi \tau_3{}^{\tau_4} & = & (\tau_1 \wedge \tau_3)^{\tau_2 \wedge \tau_4} \\ \Sigma_1 \fatsemi \Sigma_2 & = & \mathtt{eval}\,(\Sigma_1) \fatsemi \mathtt{eval}\,(\Sigma_2) \end{array}$$

The transition rules of this virtual machine with symbolic casts stay the same, except for the three rules described in Figure 2.

4 SPACE EFFICIENCY

In this section we study the space efficiency of our virtual machine. There are two space-related problems to be considered: (1) the memory blueprint of casts created during the execution (2) the size of the structures (the three stacks for control, operands, and environment) during the execution.

4.1 Cast representation and compression

Our operational semantics uses intersection types to compress types, and the type operators (dom, ∘) to build new casts. To achieve space efficiency, we need to show that the representations of these type-casts are bounded in size. For this aspect of the space efficiency we can actually provide a formal proof based on the cast language itself. Let $|E|$ be the cast language expression obtained from erasing all casts from E. We have

Theorem 3. *For each program E there exists a constant factor c such that for all E', if $E \to^\star E'$, then* $\mathrm{size}(E') \leq c \cdot \mathrm{size}(|E'|)$.

where size is a function defined on the cast language which measures the size of the representation of an expression. This theorem

$\beta_\square$	app ; c			cast ; app ; cast ; c		
$\rightarrow$	e	$v.[\mathsf{c}', \mathsf{e}', \Sigma]^\square$. s	d	e	$v.\Sigma_d.\,[\mathsf{c}', \mathsf{e}', \Sigma]^\star.\Sigma_r$. s	d
$t\beta_\square$	tap ; c			cast ; tca Σ_r; c		
$\rightarrow$	e	$v.[\mathsf{c}', \mathsf{e}', \Sigma]^\square$. s	d	e	$v.\,\Sigma_d.\,[\mathsf{c}', \mathsf{e}', \Sigma]^\star$. s	d
$c\beta_\square$	tca Σ_1 ; c			cast ; tca $(\Sigma_1 \fatsemi \Sigma_r)$; c		
$\rightarrow$	e	$v.[\mathsf{c}', \mathsf{e}', \Sigma]^\square$. s	d	e	$v.\Sigma_d.\,[\mathsf{c}', \mathsf{e}', \Sigma]^\star$. s	d

Fig. 2: Modified transitions of the virtual machine with symbolic casts (with $\Sigma_r = \mathsf{app}_{\text{type}}\ _v(\Sigma)$ and $\Sigma_d = \mathsf{dom}(\Sigma)$)

states that the space required for the execution of a program which uses the type annotations of gradual typing is bounded, as it stays within a constant factor of the space required to execute the same untyped program. In other terms, the space used by casts during the execution is bounded by a factor constant at all times. To see this we need to be more precise and define the size required to represent a program:

$$\begin{aligned} \text{size}\, x \;=\; \text{size}\, c \quad &= 1 \\ \text{size}\,(E_1\, E_2) = \text{size}\,(E_1, E_2) \quad &= 1 + \text{size}\, E_1 + \text{size}\, E_2 \\ &\vdots \\ \text{size}\,(\pi_i\, E) \quad &= 1 + \text{size}\, E \\ \textbf{size}\ (\mathbf{E}\langle\tau_1\rangle^{\tilde{\tau}_2}) \quad &= \mathbf{1} + \textbf{size}\,\mathbf{E} + \textbf{size}\,\tau_1 + \textbf{size}\ \tilde{\tau}_2 \end{aligned}$$

Establishing a bound on the size of the representation of types used during execution—which is the difference in size between untyped and typed versions of a program—is key to the proof of this theorem. The existence of this bound is due to the fact that in our machine types are represented by *Binary Decision Diagrams* (BDD). So in the definition above the size of a type is the size of the BDD representing it.

4.1.1 Binary decision diagrams. The subtyping algorithm for set-theoretic types works with types in disjunctive normal forms, which are best represented by Boolean functions [4]. It follows that the classic representation structure for set-theoretic types is BDDs. We will now present the representation of set-theoretic types which is used in CDuce, and therefore in the implementation of our virtual machine. First, we introduce an equivalent definition for types based on *atoms*. Let b range over a set $\mathcal{B}$ of basic types. Gradual set-theoretic types are the possibly infinite terms produced coinductively by

$$\begin{array}{lll} \text{Atoms} & a ::= & b \mid ? \mid \tau \to \tau \\ \text{Types} & \tau ::= & a \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \end{array}$$

with the same condition and abbreviation as in Section 2.1. Frisch et al. [13] proved that every type is equivalent to (i.e., denotes the same set of values as) a type in Disjunctive Normal Form:

$$\bigvee_{i\in I}\left(\bigwedge_{p\in P_i} a_p \wedge \bigwedge_{n\in N_i} \neg a_n\right) \tag{1}$$

BDDs are defined by the grammar $B ::= \bot \mid \top \mid a?\,B : B$ and have the following interpretation:

$$\begin{aligned} [\![\top]\!] \quad &= \mathbb{1} \\ [\![\bot]\!] \quad &= \mathbb{0} \\ [\![a?\,B_1 : B_2]\!] \quad &= (a \wedge [\![B_1]\!]) \vee (\neg a \wedge [\![B_2]\!]) \end{aligned}$$

which allows to convert any BDD to a type in disjunctive normal form—see Figure 3 for an example. To ensure that the atoms occurring on a path are distinct, a total order is defined on the atoms

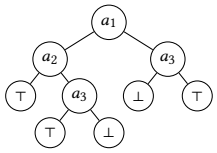


Fig. 3: BDD for $(a_1 \wedge a_2) \vee (a_1 \wedge \neg a_2 \wedge a_3) \vee (\neg a_1 \wedge \neg a_3)$

which imposes that on every path the order of the labels strictly increases. Besides, hash consing is used for atoms and, thus, several occurrence of the same atom (e.g., a_3 in Fig. 3) share the same representation. Hence the depth of a BDD is upper-bounded by the number of atoms available to build it. This yields the result:

Lemma 3. *For B a BDD, let $\mathcal{A} = \{a \in B\}$ be the set of distinct atoms in B, and $\alpha = |\mathcal{A}|$, we have*

$$\text{size}(B) \le 2^\alpha \log_2 \alpha + \sum_{a\in\mathcal{A}} \text{size}\, a \stackrel{\text{def}}{=} C_{\mathcal{A}}$$

In this formula, we consider the maximum number of nodes of a tree of depth α, which is 2^α. All the distinct atoms are stored separately using size $\sum_{a\in\mathcal{A}} \text{size}\, a$, each node of the tree being a reference to a stored atom (*i.e.* of size $\log_2 \alpha$).[5] What we now prove is that, in fact, the set of distinct atoms that can be used to build casts for a given program is fixed and does not vary during the execution. An initial cast expression E contains a bounded amount of type annotations and type-casts, from which only a bounded amount of atoms can be extracted. And because the creation of type-casts in the operational semantics is conservative in that it never creates any new atom, we can bound the size of any BDD-represented type-cast during the execution of a program by $C_{\mathcal{A}}$, with $\mathcal{A}$ the set of distinct atoms initially derived from the program. Formally:

Theorem 4. *Let E be a cast expression. There exists a finite set of atoms $\mathcal{A}$ such that for all E', if $E \longrightarrow^\star E'$, then every type-cast occurring E' is represented by a BDD using exclusively atoms from $\mathcal{A}$.*

Corollary 5. *Any type-cast $\langle\tau_1\rangle^{\tilde{\tau}_2}$ in E' is bounded in size, by* $\text{size}\,\langle\tau_1\rangle^{\tilde{\tau}_2} \le 2C_{\mathcal{A}}$

Theorem 4 comes from the fact that the operations on types used at run-time do not create new atoms. To create new casts the machine uses type intersection $\wedge$, and the domain and result operators dom, $\circ$. Let us describe how each operation handles atoms:

Intersections: consider the intersection of two BDDs. Let B_1 and B_2 denote generic BBDs, $B_1 = a_1?\,C_1 : D_1$, $B_2 = a_2?\,C_2 : D_2$. Intersections of BDDs are defined as follows:

$$\bot \wedge B = B \wedge \bot = \bot \qquad \top \wedge B = B \wedge \top = B$$

[5] This is a conservative approximation: the actual representation of types is a record of several BDDs, one for each type constructor, as described in Section 4.3 of [4].

bench	fun. calls	app (%)	tap(%)	tca (%)
sieve	$2.6 \cdot 10^6$	59.6%	20.8%	19.6%
odd-even	10^7	$\sim 0\%$	0%	$\sim 99\%$

Fig. 4: Benchmark of Space Consumption (Tail Calls)

bench	tail call depth	non-tail call depth	dump len.
sieve	7942	1000	1009
odd-even	10^7	1	2

Fig. 5: Benchmark of Space Consumption (Dump Size)

$$B_1 \wedge B_2 = \begin{cases} a_1?\, C_1 \wedge C_2 : D_1 \wedge D_2 & \text{for } a_1 = a_2 \\ a_1?\, C_1 \wedge B_2 : D_1 \wedge B_2 & \text{for } a_1 < a_2 \\ a_2?\, B_1 \wedge C_2 : B_1 \wedge D_2 & \text{for } a_1 > a_2 \end{cases}$$

This definition makes it clear that the atoms of $B_1 \wedge B_2$ are included in the union of the atoms of B_1 and B_2, so no new atom is created. The same property holds for union and negation.

Domain operator: if τ is a function type then it can be represented in disjunctive normal form as [13]:

$$\tau = \bigvee_{i \in I} \left(\bigwedge_{p \in P_i} (\sigma_p \to \tau_p) \wedge \bigwedge_{n \in N_i} \neg(\sigma_n \to \tau_n) \right)$$

and its domain is defined as $\mathsf{dom}\,\tau = \bigwedge_{i \in I} \bigvee_{p \in P_i} \sigma_p$ (see [13]).

Since unions and intersections do no create new atoms, then it is enough to include in the initial set of atoms $\mathcal{A}$ all the atoms used in the domain σ_p of every single function atom $a = \sigma_p \to \tau_p$ in E.

Result operator: for τ the function type defined above, the *codomain* is defined as

$$\mathsf{cod}\,\tau = \bigvee_{i \in I} \bigvee_{p \in P_i} \tau_p$$

We know that, for any type σ, the result types $\tau \circ \sigma$ will be formed using the atoms τ_p of $\mathsf{cod}\,\tau$ — see the annexes for a formal definition of $\tau \circ \sigma$ which makes this more explicit. Therefore, in the same way as we did for the domain, adding all the atoms of the codomain τ_p of every single function atom $a = \sigma_p \to \tau_p$ in E allows the conservation of atoms when using the result operator $\circ$.

This principle of conservation allows us to conclude with Theorem 4; and then, Corollary 5 yields a bound on the size of every type-cast during the execution of a program E. Since our semantics does not create chains of casts during execution, this allow us to conclude that the space overhead of casts during execution is indeed, bounded, as stated in Theorem 3. For a formal proof of these claims, the reader can refer to Section F in the annexes.

4.2 Tail recursion and stack space efficiency

The space-efficiency of this machine comes from three sources:

- a bounded representation for type-casts
- a semantics which reduces all chains of casts into a single one
- an instruction to handle casts on tail function calls

We formally proved in the previous section the first two points. We studied the third on examples and by comparisons with other works, but we did not provide a formal characterization of this aspect. Instead, we ran a couple of benchmarks that confirmed the efficiency in terms of function calls for the *odd-even* and *sieve* programs (explained in next section). In particular, Fig. 4 shows the distribution of functions calls over app calls (which perform usual typed or untyped function application, saving the current frame on the dump), the tap calls (which perform usual tail call, not saving the frame), and the tca calls (which perform cast tail calls). When the last percentages are high, it means that a significant portion of typed app calls were avoided. Another important metric is the size of the dump stack, which should be of the same order as the maximum depth of non-terminal recursive calls. This is the case both for *sieve* and for *odd-even* as shown in Figure 5. For a less empiric characterization we plan as future work to build up on [8] and find a class of space-efficiency to which our machine belongs.

5 PERFORMANCE

This work aims at alleviating the performance issues of a language with gradual set-theoretic types. The two key ingredients we developped to achieve this are:

(1) Cast compression using set-theoretic types to prevent the accumulation of multiple casts on an expression.
(2) Compressing casts in tail position using cast compression on the dump stack.

We chose our benchmarks to test the impact of these two features.

5.1 Benchmarks

We describe each benchmark, and why it was chosen to test our virtual machine.

Sieve This program finds prime numbers using the Sieve of Eratosthenes. It was among those which nailed in the coffin of sound gradual typing in Takikawa et al. [27], with a mean overhead of 100x compared to the untyped running times. The mean overhead we obtain with our solution is 7.6x.

Odd-even This is the program we gave as an example in Section 2.2: it computes whether an integer is odd or even using two mutually recursive functions. Usually, in sound gradual typing, the typed version of this program is not tail recursive, which incurs bad performances compared to the untyped version. In our machine, it is tail recursive and we obtain an overhead of 1.5x, which goes down to 1.15x with symbolic optimizations.

Cast-acc This ad-hoc (and highly unrealistic) program was written to illustrate a pitfall of our current optimization using symbolic computations: it does not memoize previously done cast computations. Here, we cast a function multiple times and then use it repetitively, and the cost is higher with symbolic computations because casts on the function have to be recomputed with each use. Knowing this pitfall exists makes the good performances of the **SymbolicCap** runtime in the other benchmarks even more interesting.

Polyadhoc This program, similarly to **Odd-even**, runs mutually recursive functions, one which is statically typed, the other which is partially typed. Except that this time, the cast inserted in the code is not a basic type, but an intersection of arrow types. It is essentially the same as **Odd-even**, except that domains and result types are harder to compute.

5.2 Experimental Setup

We implemented our machine in OCaml. The project is about 3000 lines of code (available at https://github.com/gliboc/cast-machine),

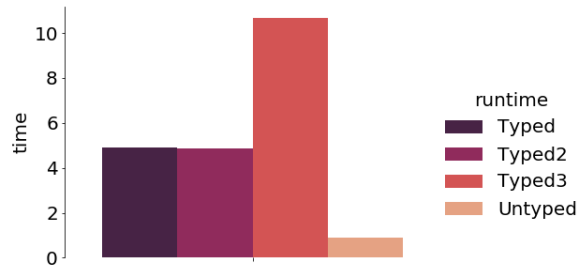


Fig. 6: Sieve Benchmark

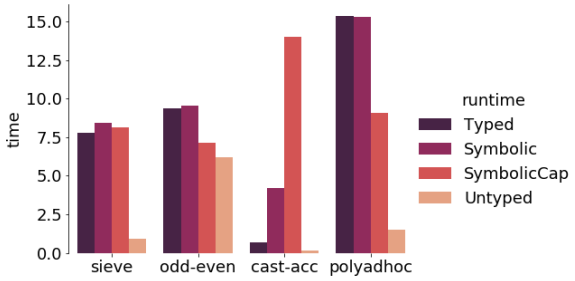


Fig. 7: Full Benchmarks

and relies on the CDuce library for its representation of types (https://gitlab.math.univ-paris-diderot.fr/cduce/cduce). Our tests were executed on a machine with a CPU i7-5600U (Clock 2,6 GHz).

Runtimes. The execution times of the four benchmarks described above are presented in Figure 7.

- The **Typed** runtime corresponds to a typed execution of the program. There can be several typed versions, as in the case of **Sieve**, or just one (the other cases).
- The **Symbolic** runtime corresponds to using casts where the operations of domain and result are symbolic.
- The **SymbolicCap** runtime corresponds to the case in which also the intersection of type-casts are symbolic.
- The **Untyped** corresponds to running the program without any compiler-inserted casts.

What Figure 7 shows is that, as expected, the typed execution is much slower than the untyped one. However, this overhead is reasonable compared to the results obtained for sound gradual typing by Takikawa et al. [27]. Moreover, the results show that there is room for improvement, as we obtain better results by using symbolic computations for casts. Finally, the benchmarks, in particular **cast-acc**, strongly suggest that addition of memoization to the computation of casts may further improve performance, yielding important speed ups in particular cases.

5.3 Evaluation Criteria

There are many ways to optimize the code running in a virtual machine. Here, we are interested in measuring the overhead that occurs when executing different versions of a program annotated with gradual types. A gradual type system allows both fully typed and untyped versions of a program, and so does the machine we built. The difference comes from using different reduction rules: the untyped one, $[\beta_\star]$, or the typed one $[\beta_\square]$. By comparing the execution time of these different versions of the same programs, which we call configurations, we intend to measure directly the overhead of gradual typing on our machine.

Takikawa et al. [27] define the notion of N-deliverable to compare different configurations: a configuration is N-deliverable if its performance is no worse than an Nx slowdown compared to the completely untyped configuration. Another interesting definition is that a configuration is N/M-usable if its performance is worse than an Nx slowdown and no worse than an Mx slowdown compared to the completely untyped configuration. Finally, for any choice of N and M, a configuration is unacceptable if it is neither N-deliverable nor N/M-usable. For $N = 3$ and $M = 10$, the *sieve* benchmark in [27] concludes that most configurations incur an overwhelming overhead (on average, more than 100 times the untyped running time). We reproduce this benchmark on our machine for comparison, and obtain the results of Figure 6.

The four bars of Figure 6 correspond to four configurations of interest of the program *sieve*. This program consists in two modules: (1) an implementation of *streams* using thunks and pairs; (2) an implementation of the Sieve of Eratosthenes using the *streams* module.

The different configurations we consider are obtained by variating the degree of annotations in each of these modules. In particular:

- **Untyped**: no type annotations, no type-casts
- **Typed**: we annotated two functions in each module
- **Typed2**: only the module *streams* is completely annotated
- **Typed3**: only the code of *sieve* is completely annotated

We obtain an overhead of 5x for the **Typed2** configuration, meaning it is a 5-deliverable according to the metrics of Takikawa et al. [27], and therefore a 3/10-usable. Similarly, **Typed3** has an overhead of around 10x compared to the untyped *sieve*. This is still significant, but it is way faster than the maximum overhead of more than 110x obtained in Takikawa et al. [27] and still acceptable by the reasonable standards of 3/10-usability.

5.4 Impact of Set-Theoretic Operators

Our implementation relies on the CDuce implementation to compute four type operators: (1) decide subtyping $\leq$; (2) the domain operator $\mathtt{dom}$; (3) the apply operator $\circ$; (4) the intersection of types. We tried to minimize the use of these operations in our machine by using a symbolic representation of casts. Our implementation of this technique yields the results seen in Figure 7. We notice in particular that the most interesting configuration is the one which uses symbolic intersections.

6 RELATED WORK

We followed an approach similar to the one from Siek and Garcia [22], who built abstract machines for the gradually-typed lambda calculus, and used parameterization to model several different semantics for gradual typing such as eager or lazy cast checking, as well as different kinds of blame tracking. This broad approach of experimenting the different features of gradual typing with abstract machine before plugging gradual type systems into real systems reflects the current evolving state of gradual typing.

The problem of cast accumulation was recognized by Herman et al. [16], who proposes a solution based on the coercions of Henglein [15]. Siek et al. [26] present an efficient algorithm for compressing coercions. Alternatively, Siek and Wadler [25] propose a solution

based on compressing a sequence of casts into a pair of casts, to and from the least upper bound of the types (with respect to type precision). One can view the present paper as generalizing this approach to languages with set-theoretic types and subtyping.

Contracts are a generalization of casts to handle arbitrary predicates [12]. Greenberg [14] proves that the space overhead for contracts can be bounded by a constant by taking care to never wrap the same contract on a value multiple times. Feltey et al. [11] implement and evaluate this approach in the Racket contract library, which underlies the implementation of sound gradual typing in Typed Racket. The approach that we present, based on the BDD representation of types, enables finer-grained sharing which we conjecture leads to better compression.

While the present paper focuses on efficiency but leaves out blame tracking, Keil and Thiemann [17] develop an operational semantics for intersection and union types that includes blame tracking, but they do not consider space efficiency.

On a more set-theoretic perspective, an interesting approach is dedicated to the inference of interfaces — the type constraints of functions — and in particular of intersections of interfaces. This type inference makes it possible to fully annotate a module more quickly, in order to faster bridge the gap between untyped and partially-typed performances. The paper [6] treats the subject, but in a context of non-gradual set-theoretic types.

7 FUTURE WORK

Blame tracking. In gradual typing, blame tracking makes it possible to find which cast in the code led to a failure. It should satisfy two properties: *blame safety*, and *type safety*. Blame safety means that an expression that could reduce to a value should never be blamed, and soundness that a well-typed expression can either reduce to a value, diverge, or be blamed. Our machine has difficulties in assigning blame, because the compression of casts using type intersections lose the information of blame labels.

However, it might be possible to compress the labels as well as the casts. We conjecture that if a function terminates, then the sequence of blame labels of its casts can be expressed as a regular expression whose size is bounded according to the number of blame labels in the original program. This regular expression would record the arrival of each casts on an expression: therefore, when a type-cast fails on a value, it would be possible to blame the earliest type-cast that was incompatible with the value. A fallback solution would be to handle sets of blame labels, of which there finitely many, instead of regular expressions, but this would yield far less precise blames.

Benchmarks and Language Extensions. We would also like to test more thoroughly our implementation by adapting the rest of the benchmarks of Takikawa et al. [27], and by finding other tests specific to set-theoretic types. There are also some language features that could be of interest once our machine is sufficiently improved, such as (*i*) using type intersections in annotations—currently it is only possible to annotate functions with a function type and (*ii*) extending the type system with polymorphism.

8 CONCLUSION

The goal of this work was to study the implementation of functional languages using a gradual type system with set-theoretic types. Our main contribution is our technique of combining intersection types and domain caching to obtain a space efficient compression of cast compositions. This, combined with various other implementation techniques we described (caching of casts in closures, use of the dump to efficiently implement cast application in tail position, the symbolic computation of type operations) yields an implementation satisfactory in space consumption and is not extremely penalizing in time consumption. The time overhead due to set-theoretic types is still too important but, as we discussed in Section 5 so is the room for improvement, that we plan to explore in future work

REFERENCES

[1] S. Bauman, C. F. Bolz-Tereick, J. G. Siek, and S. Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *PACMPL*, 1(OOPSLA):54, 2017.

[2] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[3] A. Bonnaire-Sergeant, R. Davies, and S. Tobin-Hochstadt. Practical optional types for Clojure. In *ESOP 2016*, pages 68–94. Springer, 2016.

[4] G. Castagna. Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science*, 2019. https://arxiv.org/abs/1809.01427. To appear.

[5] G. Castagna and V. Lanvin. Gradual typing with union and intersection types. *PACMPL*, 1(ICFP):41, 2017.

[6] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *POPL '15*, volume 50, pages 289–302, 2015.

[7] G. Castagna, V. Lanvin, T. Petrucciani, and J. G. Siek. Gradual typing: a new perspective. *PACMPL*, 3(POPL):16, 2019.

[8] W. Clinger. Proper tail recursion and space efficiency. In *PLDI '98*, 1998.

[9] Facebook. Flow documentation, . https://flow.org/en/docs/lang/.

[10] Facebook. Hack documentation, . https://docs.hhvm.com/hack/.

[11] D. Feltey, B. Greenman, C. Scholliers, R.B. Findler, and V. St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *PACMPL*, 2(OOPSLA): 133:1–133:27, October 2018.

[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02*, pages 48–59, October 2002.

[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 55(4):19, 2008.

[14] M. Greenberg. Space-efficient manifest contracts. In *POPL '15*, pages 181–194. ACM, 2015.

[15] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[16] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.

[17] M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. In *ICFP 2015*, pages 375–386, 2015.

[18] A. Kuhlenschmidt, D. Almahallawi, and J. G. Siek. Toward efficient gradual typing for structural types via coercions. In *PLDI '19*, 2019.

[19] J. Lehtosalo and D. J. Greaves. Language with a pluggable type system and optional runtime monitoring of type errors. In *Workshop on Scripts to Programs (STOP)*, 2011.

[20] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *POPL '15*, pages 167–180. ACM, 2015.

[21] G. Richards, F. Zappa Nardelli, and J. Vitek. Concrete types for TypeScript. In *ECOOP 2015*, 2015.

[22] J. G. Siek and R. Garcia. Interpretations of the gradually-typed lambda calculus. In *Workshop on Scheme and Functional Programming*, pages 68–80, 2012.

[23] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[24] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Symposium on Dynamic languages*, page 7. ACM, 2008.

[25] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL 2010*, pages 365–376, 2010.

[26] J. G. Siek, P. Thiemann, and P. Wadler. Blame and coercion: together again for the first time. In *PLDI '15*, pages 425–435, 2015.

[27] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *POPL '16*, pages 456–468, 2016.

[28] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *DSL '06*, pages 964–974. ACM, 2006.

[29] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL '08*, pages 395–406. ACM, 2008.

[30] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *DSL '14*, pages 45–56, 2014.

A symbolic execution semantics for TopHat

Nico Naus
Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
n.naus@uu.nl

Tim Steenvoorden
Software Science
Radboud University
Nijmegen, The Netherlands
tim@cs.ru.nl

Markus Klinik
Software Science
Radboud University
Nijmegen, The Netherlands
m.klinik@cs.ru.nl

ABSTRACT

Task-Oriented Programming (TOP) is a programming paradigm that allows declarative specification of workflows. TOP is typically used in domains where functional correctness is essential, and where failure can have financial or strategical consequences. In this paper we aim to make formal verification of software written in TOP easier. Currently, only testing is used to verify that programs behave as intended. We use symbolic execution to guarantee that no aberrant behaviour can occur. In previous work we presented TopHat, a formal language that implements the core aspects of TOP. In this paper we develop a symbolic execution semantics for TopHat. Symbolic execution allows to prove that a given property holds for all possible execution paths of TopHat programs.

We show that the symbolic execution semantics is consistent with the original TopHat semantics, by proving soundness and completeness. We present an implementation of the symbolic execution semantics in Haskell. By running example programs, we validate our approach. This work represents a step forward in the formal verification of TOP software.

ACM Reference Format:
Nico Naus, Tim Steenvoorden, and Markus Klinik. 2020. A symbolic execution semantics for TopHat. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Task-Oriented Programming paradigm (TOP) is an abstraction over workflow specifications. The idea of TOP is to describe the work that needs to be done, in which order, by which person. From this specification, an application can be generated that helps to coordinate people and machines to execute the work. The iTasks framework [Plasmeijer et al. 2012] is an implementation of the paradigm in the functional programming language Clean. In earlier work [Steenvoorden et al. 2019], we presented the programming language TopHat, written $\widehat{\text{TOP}}$, to distill the core features of TOP into a language suitable for formal treatment. The usefulness of TOP has been demonstrated in several projects that applied it to implement various applications. It has been used by the Netherlands Royal Navy [Jansen et al. 2018], the Dutch Tax Office [Stutterheim et al. 2017] and the Dutch Coast Guard [Lijnse et al. 2012]. Furthermore, it has potential for application in domains like healthcare and Internet of Things [Koopman et al. 2018].

Applications in these kinds of domains are often mission critical, where programming mistakes can have severe consequences. In order to verify that a $\widehat{\text{TOP}}$ program behaves as intended, we would like to show that it satisfies a given property. A common way to do this is to write test cases, or to generate random input, and verify that all outcomes fulfil the property. Writing tests manually is time consuming and cumbersome. Testing interactive applications needs people to operate the application, maybe making use of a way to record and replay interactions. With this kind of testing there is no guarantee that all possible execution paths are covered.

To overcome these issues, we apply symbolic execution. Instead of executing tasks with test input, or letting a user interactively test the application, we run tasks on symbolic input. Symbolic input consists of tokens that represent any value of a certain type. When a program branches, the execution engine records the conditions over the symbolic input that lead to the different branches. These conditions can then be compared to a given predicate to check if the predicate holds under all conditions. We let an SMT solver verify these statements.

In this way we can guarantee that given predicates over the outcome of a TOP program always hold. Since iTasks is not suitable for formal reasoning, we instead apply symbolic execution to $\widehat{\text{TOP}}$ [Steenvoorden et al. 2019][1], by systematically changing the semantic rules of the original language.

1.1 Contributions

This paper makes the following contributions.

- We present a symbolic execution semantics for $\widehat{\text{TOP}}$, a programming language for workflows embedded in the simply typed λ-calculus.
- We prove soundness and completeness of the symbolic semantics with respect to the original $\widehat{\text{TOP}}$ semantics.
- We present an implementation of the symbolic execution semantics in Haskell.

1.2 Structure

Section 2 gives a brief overview of $\widehat{\text{TOP}}$ and its concepts. Section 3 introduces some examples to demonstrate the goal of our symbolic execution analysis. In Section 4, the $\widehat{\text{TOP}}$ language is defined. Section 5 goes on to define the formal semantics of the symbolic execution. In Section 6, soundness and completeness are shown for the symbolic execution semantics with respect to the original $\widehat{\text{TOP}}$ semantics. In Section 7 related work is discussed, and Section 8 concludes.

2 $\widehat{\text{TOP}}$

This section briefly introduces the task-oriented programming language $\widehat{\text{TOP}}$, and discusses our vision about symbolic evaluation of this language.

IFL'19, September 2019, Singapore
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Preprint is available from http://www.cs.ru.nl/~steenvoo/papers/ppdp-tophat.pdf

The $\widehat{\text{TOP}}$ language consists of two parts, the host language and the task language. Programs in $\widehat{\text{TOP}}$ are called *tasks*. The basic elements of tasks are editors. Using combinators, tasks can be combined into larger tasks.

The task language is embedded in a simply typed lambda calculus with references, conditionals, booleans, integers, strings, pairs, lists and unary and binary operations on these types. References allow tasks to communicate with each other, sharing information across task boundaries. The simply typed λ-calculus does not have recursion. By restricting references to only hold basic types, strong normalisation of the calculus is guaranteed. The full syntax of the host language is listed in Section 4. Next, we discuss the main constructs of the task language.

2.1 Editors

Editors are the most basic tasks. They are used to communicate with the outside world. Editors are an abstraction over widgets in a GUI library or on webpage forms. Users can change the value held by an editor, in the same way they can manipulate widgets in a GUI.

When a TOP implementation generates an application from a task specification, it derives user interfaces for the editors. The appearance of an editor is influenced by its type. For example, an editor for a string can be represented by a simple input field, a date by a calendar, and a location by a pin on a map.

There are three different editors in $\widehat{\text{TOP}}$.

$\square\, v$ Valued editor.
: This editor holds a value v of a certain type. The user can replace the value by a new value of the same type.

$\boxtimes\, \tau$ Unvalued editor.
: This editor holds no value, and can receive a value of type τ. When that happens, it turns into a valued editor.

$\blacksquare\, l$ Shared editor.
: This editor refers to a store location l. Its observable value is the value stored at that location. When it receives a new value, this value will be stored at location l.

2.2 Combinators

Editors can be combined into larger tasks using combinators. Combinators describe the way people collaborate. Tasks can be performed in sequence or in parallel, or there is a choice between two tasks.

The following combinators are available in $\widehat{\text{TOP}}$. Here, t stands for tasks and e for arbitrary expressions. The concrete syntax of the language is described in Section 4.1

$t \blacktriangleright e$ Step.
: Users can work on task t. As soon as t has a value, that value is passed on to the right hand side e. The expression e is a function, taking the value as an argument, resulting in a new task.

$t \triangleright e$ User Step.
: Users can work on task t. When t has a value, the step becomes enabled. Users can then send a continue event to the combinator. When that happens, the value of t is passed to the right hand side, with which it continues.

$t_1 \bowtie t_2$ Composition.
: Users can work on tasks t_1 and t_2 in parallel.

$t_1 \blacklozenge t_2$ Choice.
: The system chooses between t_1 or t_2, based on which task first has a value. If both tasks have a value, the system chooses the left one.

$e_1 \Diamond e_2$ User choice.
: A user has to make a choice between either the left or the right hand side. The user continues to work on the chosen task.

In addition to editors and combinators, $\widehat{\text{TOP}}$ also contains the fail task ($\lightning$). Programmers can use this task to indicate that a task is not reachable or viable. When the right hand side of a step combinator evaluates to $\lightning$, the step will not proceed to that task.

2.3 Observations

Several observations can be made on tasks. Using the value function $\mathcal{V}$, the current value of a task can be determined. The value function is a partial function, since not all tasks have a value. For example empty editors and steps do not have a value.

One can also observe whether or not a task is failing, by means of the failing function $\mathcal{F}$. The task $\lightning$ is failing, as is a parallel combination of failing tasks ($\lightning \bowtie \lightning$).

The step combinator makes use of both functions in order to determine if it can step. First, it uses $\mathcal{V}$ to see if the left hand side produces a value. If that is the case, it uses the $\mathcal{F}$ function to see if it is safe to step to the right hand side. The complete definition of the value and failing function are discussed in Section 5.2.

2.4 Input

Input events drive evaluation of tasks. Because tasks are typed, input is typed as well. Editors only accept input of the correct type. Examples are replacing a value in an editor, or sending a continue event to a user step. When the system receives a valid event, it gives this event to the current task, which reduces to a new task. Everything in between interaction steps is evaluated atomically with respect to inputs.

Input events are synchronous, which means the order of execution is completely determined by the order of the events. In particular, the order of input events determine the progression of parallel branches.

3 EXAMPLES

In this section we study three examples to illustrate how the language $\widehat{\text{TOP}}$ works and what kind of properties we would like to prove.

3.1 Positive value

This example demonstrates how to prove that the first observable value of a program can only be a positive number. Consider the program in Listing 1.

Listing 1: A task that only steps on a positive input value.

$\boxtimes\text{INT} \blacktriangleright \lambda x.\ \textbf{if}\ x > 0\ \textbf{then}\ \square x\ \textbf{else}\ \lightning$

It asks the user to input a value of type INT. This value is then passed on to the right hand side. If the value is greater than zero, an editor containing the entered value is returned. At this point, the task has an observable value, and we consider it done. Otherwise the step does not proceed and the task does not have an observable value. The user can enter a different input value.

Imagine that we would like to prove that no matter which value is given as input, the first observable value has to be a value greater than zero.

Symbolic execution of this program proceeds as follows. The symbolic execution engine generates a fresh symbolic input s for the editor on the left. The engine then arrives at the conditional. In order to take the then-branch, the condition $s > 0$ needs to hold. This branch will then result in $\Box\, s$, in which case the program has an observable value. The engine records this endpoint together with its path condition $s > 0$. The else-branch applies if the condition does not hold, but this leads to a failing task. Therefore, the step is not taken and the programs stays the same. No additional program state is generated.

Symbolic execution returns a list of all possible program end states, together with the path conditions that led to them. If all end states agree with the desired property, it is guaranteed that the property holds for all possible inputs.

In this example, the only end state is the expression $\Box\, s$ with path condition $s > 0$. From that we can conclude that no matter what input is given, the only result value possible has to be greater than zero.

3.2 Tax subsidy request

Stutterheim et al. [2017] worked with the Dutch tax office to develop a demonstrator for a fictional but realistic law about solar panel subsidies. In this section we study a simplified version of this, translated to $\widehat{\text{TOP}}$, to illustrate how symbolic execution can be used to prove that the program implements the law.

This example proves that a citizen will get subsidy only under the following conditions.

- The roofing company has confirmed that they installed solar panels for the citizen.
- The tax officer has approved the request.
- The tax officer can only approve the request if the roofing company has confirmed, and the request is filed within one year of the invoice date.
- The amount of the granted subsidy is at most 600 EUR.

Listing 2: Subsidy request and approval workflow at the Dutch tax office.

```
let provideCitizenInformation = ⊠Date in                                  1
let provideDocuments = ⊠Amount ⋈ ⊠Date in                                 2
let companyConfirm = □True ◊ □False in                                    3
let officerApprove = λinvoiceDate. λtoday. λconfirmed.                    4
  □False ◊ if (today − invoiceDate < 365 ∧ confirmed)                     5
    then □True                                                            6
    else ↯ in                                                             7
provideCitizenInformation ► λtoday.                                       8
provideDocuments ⋈ companyConfirm ►                                       9
  λ⟨⟨invoiceAmount, invoiceDate⟩ , confirmed⟩ .                           10
officerApprove invoiceDate today confirmed ► λapproved.                   11
let subsidyAmount = if approved                                           12
  then min 600 (invoiceAmount / 10) else 0 in                             13
□⟨subsidyAmount, approved, confirmed, invoiceDate, today⟩                 14
```

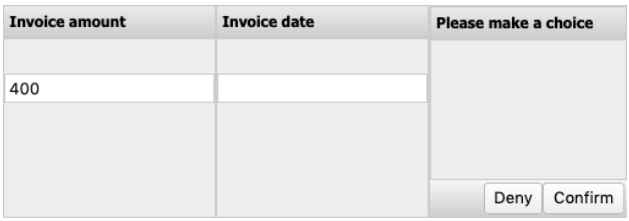


Figure 1: Graphical user interface for the task in Listing 2. In parallel, the citizen is asked to enter the invoice amount and the invoice date of the installed solar panels, and the roofing company is asked to deny or confirm they actually installed the solar panels.

Listing 2 shows the program. To enhance readability of the example, we omit type annotations and make use of pattern matching on tuples. The program works as follows. First, the citizen has to enter their personal information (Line 8). In the original demonstrator this included the citizen service number, name, and home address. Here, we simplified the example so that the citizen only has to enter the invoice date. A date is specified using an integer representing the number of days since 1 January 2000.

In the next step (Line 9), in parallel the citizen has to provide the invoice documents of the installed solar panels, while the roofing company has to confirm that they have actually installed solar panels at the citizen's address. Once the invoice and the confirmation are there, the tax officer has to approve the request (Line 11). The officer can always decline the request, but they can only approve it if the roofing company has confirmed and the application date is within one year of the invoice date (Line 5). The result of the program is the amount of the subsidy, together with all information needed to prove the required properties (Line 14). The graphical user interface belonging to two steps in this process are shown in Fig. 1.

The result of the overall task is a tuple with the subsidy amount, the officer's approval, the roofing company's confirmation, the invoice amount, the invoice date, and today's date. Returning all this information allows the following predicate to be stated, which verifies the correctness of the implementation. The predicate has 5 free variables, which correspond to the returned values.

$$\begin{aligned} \psi(s,a,c,i,t) &= s \geq 0 \supset c && (1)\\ &\wedge\ s \geq 0 \supset a && (2)\\ &\wedge\ a \supset (c \wedge t - i < 365) && (3)\\ &\wedge\ s \leq 600 && (4)\\ &\wedge\ \neg a \supset s \equiv 0 && (5) \end{aligned}$$

The predicate ψ states that (1) if subsidy s has been payed, the roofing company must have confirmed c, (2) if subsidy has been payed, the officer must have approved a, (3) the officer can approve only if the roofing company has confirmed and today's date t is within 356 days of the invoice date i, and (4) the subsidy is maximal 600 EUR. Finally, (5) if the officer has not approved, the subsidy must be 0.

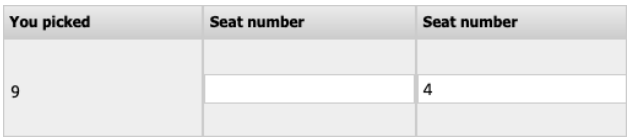


Figure 2: Graphical user interface generated from the specification in Listing 3. Three users are booking seats in parallel. The first user booked seat 9, the second did not enter a seat number yet, and the third is about to book seat 4.

3.3 Flight booking

In this section we develop a small flight booking system. The purpose of this example is to demonstrate how symbolic execution handles references and lists. We prove that when the program terminates, every passenger has exactly one seat, and that no two passengers have the same seat. This program is a simplified version of what we presented in earlier work [Steenvoorden et al. 2019].

Listing 3: Flight booking.

```
let maxSeats = 50 in                                              1
let bookedSeats = ref [] in                                       2
let bookSeat = ⊠Int ▶ λx .                                        3
  if not (x ∈ !bookedSeats) ∧ x ≤ maxSeats                        4
    then bookedSeats := x :: !bookedSeats ▶ λ_ . □x               5
    else ↯ in                                                     6
bookSeat ⋈ bookSeat ⋈ bookSeat ▶ λ_ .                             7
□(!bookedSeats)                                                   8
```

The program, shown in Listing 3, consists of three parallel seat booking tasks (Line 7). There is a shared list that stores all booked seats so far (Line 2). In order to book a seat, a passenger has to enter a seat number (Line 3). A guard expression makes sure that only free seats can be booked (Line 4). The exclamation mark denotes dereferencing. When the guard is true, the list of booked seats is updated, and the user can see his booked seat (Line 5). The main expression runs the seat booking task three times in parallel (Line 7), simulating three concurrent customers. The program returns the list of booked seats. The graphical user interface, generated from the specification in Listing 3, is shown in Fig. 2.

With the returned list, we can state the predicate to verify the correctness of the booking process.

$$\psi(l) = \mathsf{len}\, l \equiv 3 \tag{1}$$
$$\wedge\ \mathsf{uniq}\, l \tag{2}$$

The predicate specifies that all three passengers got exactly one seat (1), and that all seats are unique (2), which means that no two passenger got the same seat. The unary operators for list length (len) and uniqueness (uniq) are available in the predicate language. List length is a capability of SMT-LIB, while uniq is our own addition.

4 LANGUAGE

The language presented in this section is nearly identical to the original $\widehat{\text{TOP}}$ language presented by Steenvoorden et al. [2019]. The main difference with the original grammar is the addition of symbolic values.

Symbolic execution for functional programming languages struggles with higher order features. This topic is under active study, and is not the focus of our work. Therefore, we restrict symbols to only represent values of basic types. This restriction is of little importance in the domains we are interested in. Allowing users to enter higher order values is not useful in most workflow applications. Apart from input, all other higher order features are unrestricted.

The following subsections describe in detail how all elements of the $\widehat{\text{TOP}}$ language deal with the addition of symbols.

4.1 Expressions, values, and types

The syntax of $\widehat{\text{TOP}}$ is listed in Fig. 3. Two main changes have been made with regards to the original $\widehat{\text{TOP}}$ grammar. The differences with the original syntax are highlighted in grey boxes. First, symbols s have been added to the syntax of expressions. However, they are not intended to be used by programmers, similar to locations l. Instead, they are generated by the semantics as placeholders for symbolic inputs. Second, unary and binary operations have been made explicit.

Expressions

e	::=	$\lambda x : \tau\,.\ e \mid e_1\ e_2$	– abstraction, application
	|	$x \mid c \mid \langle\rangle$	– variable, constant, unit
	|	$u\ e_1 \mid e_1\ o\ e_2$	– unary, binary operation
	|	**if** e_1 **then** e_2 **else** e_3	– conditional
	|	$\langle e_1, e_2\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$	– pair, projections
	|	$[\,]_\beta \mid e_1 :: e_2$	– nil, cons
	|	$\mathsf{head}\ e \mid \mathsf{tail}\ e$	– first element, list tail
	|	**ref** $e \mid !e \mid e_1 := e_2 \mid l$	– references, location
	|	$p \mid s$	– pretask, symbol

Constants

c	::=	$B \mid I \mid S$	– boolean, integer, string

Unary Operations

u	::=	$\neg \mid - \mid \mathsf{len} \mid \mathsf{uniq}$	– not, negate, length, unique

Binary Operations

o	::=	$< \mid \leq \mid \equiv \mid \not\equiv \mid \geq \mid >$	– equational
	|	$+ \mid - \mid \times \mid /$	– numerical
	|	$\wedge \mid \vee$	– conjunction, disjunction
	|	$+\!\!+ \mid \in$	– append, elementhood

Pretasks

p	::=	$\square\, e \mid \boxtimes \beta \mid \blacksquare\, e$	– editors: valued, unvalued, shared
	|	$e_1 \blacktriangleright e_2 \mid e_1 \triangleright e_2$	– steps: internal, external
	|	$\lightning \mid e_1 \bowtie e_2$	– fail, composition
	|	$e_1 \blacklozenge e_2 \mid e_1 \lozenge e_2$	– choice: internal, external

Figure 3: Syntax of Symbolic $\widehat{\text{TOP}}$ expressions.

Symbols are treated as values (Fig. 4). They have therefore been added to the grammar of values. Also, every symbol has a type, and basic operations can take symbols as arguments. As a result, we must now also regard unary and binary operations as values. Therefore we make these operations explicit in this language description, where in the original they were left implicit.

The types of $\widehat{\text{TOP}}$ remain the same (Fig. 5). However, we do need an additional typing rule, T-Sym in Fig. 6, to type symbols, since

Values

$$
\begin{array}{lcll}
v & ::= & \lambda x : \tau \,.\, e \mid \langle v_1, v_2 \rangle \mid \langle\rangle & \text{– abstraction, pair, unit} \\
& \mid & [\,]_\beta \mid v_1 :: v_2 \mid c & \text{- nil, cons, constant} \\
& \mid & l \mid t \mid s & \text{– location, task, symbol} \\
& \mid & u\ v \mid v_1\ o\ v_2 & \text{– unary/binary operation}
\end{array}
$$

Tasks

$$
\begin{array}{lcll}
t & ::= & \Box\, v \mid \boxtimes \beta \mid \blacksquare\, l & \text{– editors} \\
& \mid & t_1 \blacktriangleright e_2 \mid t_1 \triangleright e_2 & \text{– steps} \\
& \mid & \lightning \mid t_1 \bowtie t_2 & \text{– fail, combination} \\
& \mid & t_1 \blacklozenge t_2 \mid e_1 \lozenge e_2 & \text{– choices}
\end{array}
$$

Figure 4: Syntax of values in Symbolic $\widehat{\text{TOP}}$.

Types

$$
\begin{array}{lcll}
\tau & ::= & \tau_1 \to \tau_2 \mid \beta & \text{– function, basic} \\
& \mid & \textsc{Ref}\ \tau \mid \textsc{Task}\ \tau & \text{– reference, task}
\end{array}
$$

Basic types

$$
\begin{array}{lcll}
\beta & ::= & \beta_1 \times \beta_2 \mid \textsc{List}\ \beta \mid \textsc{Unit} & \text{– product, list, unit} \\
& \mid & \textsc{Bool} \mid \textsc{Int} \mid \textsc{String} & \text{– boolean, integer, string}
\end{array}
$$

Figure 5: Syntax of Symbolic $\widehat{\text{TOP}}$ types.

$$
\textsc{T-Sym}\ \frac{s : \beta \in \Gamma}{\Gamma, \Sigma \vdash s : \beta}
$$

Figure 6: Additional typing rule for Symbolic $\widehat{\text{TOP}}$.

they are now part of our expression syntax. The type of symbols is kept track of in the environment Γ.

4.2 Inputs

In symbolic execution, we do not know what the input of a program will be. In our case this means that we do not know which events will be sent to editors. This is reflected in the definition of symbolic inputs and actions in Fig. 7

Symbolic inputs

$$
\begin{array}{lcll}
i & ::= & a \mid \mathsf{F}\, i \mid \mathsf{S}\, i & \text{– symbolic action, to first, to second}
\end{array}
$$

Symbolic actions

$$
\begin{array}{lcll}
a & ::= & s & \text{– symbol} \\
& \mid & \mathsf{C} \mid \mathsf{L} \mid \mathsf{R} & \text{– continue, go left, go right}
\end{array}
$$

Figure 7: Syntax of inputs and actions in Symbolic $\widehat{\text{TOP}}$.

Inputs are still the same and consist of paths and actions. Paths are tagged with one or more F (first) and S (second) tags. Actions no longer contain concrete values, but only symbols. This means that instead of concrete values, editors can only hold symbols.

4.3 Path conditions

Concrete execution of $\widehat{\text{TOP}}$ programs is driven by concrete inputs, which select one branch of conditionals, or make a choice. Since no concrete information is available during symbolic execution, the symbolic execution semantics records how each execution path depends on the symbolic input. This is done by means of path conditions. Figure 8 lists the syntax of path conditions.

Path conditions

$$
\begin{array}{lcll}
\varphi & ::= & c \mid s & \text{– constant, symbol} \\
& \mid & \langle\rangle \mid \langle \varphi_1, \varphi_2 \rangle & \text{– unit, pairs} \\
& \mid & [\,]_\beta \mid \varphi_1 :: \varphi_2 & \text{– nil, cons} \\
& \mid & u\ \varphi \mid \varphi_1\ o\ \varphi_2 & \text{– symbolic unary/binary operation}
\end{array}
$$

Figure 8: Syntax of path conditions.

Path conditions are a subset of the values of basic type β. They can contain symbols, constants, pairs, lists, and operations on them.

5 SEMANTICS

In this section we discuss the symbolic execution semantics for $\widehat{\text{TOP}}$. The structure of the symbolic semantics closely resembles that of the concrete semantics. It consists of three layers, a big step symbolic evaluation semantics for the host language, a big step symbolic normalisation semantics for the task language, and a small step driving semantics that processes user inputs. Figure 9 gives an overview of the relations between the different semantics.

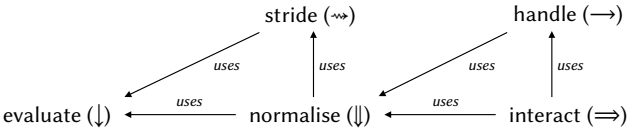


Figure 9: Semantic functions defined in this report and their relation.

They are described in the following sections. We will study their interesting aspects, and the changes made with respect to the concrete semantics.

5.1 Symbolic evaluation

The host language is a simply typed lambda calculus with references and basic operations. Most of the symbolic evaluation rules closely resemble the concrete semantics. The original evaluation relation ($\hat{\downarrow}$) had the form $e, \hat{\sigma} \,\hat{\downarrow}\, \hat{v}, \hat{\sigma}'$, where an expression e in a state $\hat{\sigma}$ evaluates to a value $\hat{v}$ in state $\hat{\sigma}'$. The hat distinguishes the old concrete and the new symbolic variants. The new relation ($\downarrow$) adds path conditions φ to the output and has the form $e, \sigma \downarrow \overline{v, \sigma', \varphi}$.

The symbolic semantics can generate multiple outcomes. This is denoted in the evaluation with a line over the result, which can be read as $\overline{v, \sigma', \varphi} = \{(v_1, \sigma_1', \varphi_1), \cdots, (v_n, \sigma_n', \varphi_n)\}$. The set that results from symbolic execution can be interpreted as follows. Each element is a possible endpoint in the execution of a task. It is guarded by a condition φ over the symbolic input. Execution only arrives at the value v and state σ' when the inputs satisfy φ.

To illustrate the difference between concrete and symbolic evaluation, Fig. 10 lists one rule from the concrete semantics and its corresponding symbolic counterpart.

$$
\textsc{E-Edit}\ \frac{e, \hat{\sigma} \,\hat{\downarrow}\, \hat{v}, \hat{\sigma}'}{\Box\, e, \hat{\sigma} \,\hat{\downarrow}\, \Box\, \hat{v}, \hat{\sigma}'} \qquad \textsc{SE-Edit}\ \frac{e, \sigma \downarrow \overline{v, \sigma', \varphi}}{\Box\, e, \sigma \downarrow \overline{\Box\, v, \sigma', \varphi}}
$$

Figure 10: The evaluation rule from the concrete and the symbolic semantics for the editor expression.

The E-Edit rule evaluates the expression held in an editor to a value. The SE-Edit does the same, but since it is concerned with symbolic execution, the expression can contain symbols. We therefore do not know beforehand which concrete value will be produced, or even which path the execution will take. If the expression contains a conditional that depends on a symbol, there can be multiple possible result values.

Most symbolic rules closely resemble their concrete counterparts, and follow directly from them. The rules are not listed here, a full overview can be found in Appendix A.1.

The only interesting rule is the one for conditionals, listed in Fig. 11. The concrete semantics has two separate rules for the **then**

$$\boxed{e, \sigma \downarrow \overline{v, \sigma', \varphi}}$$

SE-If

$$\frac{e_1, \sigma \downarrow \overline{v_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2} \qquad e_3, \sigma' \downarrow \overline{v_3, \sigma''', \varphi_3}}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \sigma \downarrow \overline{v_2, \sigma'', \varphi_1 \wedge \varphi_2 \wedge v_1} \cup \overline{v_3, \sigma''', \varphi_1 \wedge \varphi_3 \wedge \neg v_1}}$$

Figure 11: Part of the symbolic evaluation semantics.

and the **else** branch. The symbolic semantics has one combined rule SE-If. Since e_1 can contain symbols, it can evaluate to multiple values. The rule keeps track of all options. It calculates the **then**-branch, and records in the path condition that execution can only reach this branch if v_1 becomes True. The rule does the same for the **else**-branch, except it requires that v_1 becomes False. Note that both e_2 and e_3 are evaluated using the same state σ', which is the resulting state after evaluating e_1.

5.2 Observations

The symbolic normalisation and driving semantics make use of observations on tasks, just like the concrete semantics.

The partial function $\mathcal{V}$ can be used to observe the value of a task. Its definition is given in Fig. 12. It is unchanged with respect to the original.

$$\begin{aligned}
&\mathcal{V} : \text{Tasks} \times \text{States} \rightharpoonup \text{Values} \\
&\mathcal{V}(\square\, v, \sigma) = v \\
&\mathcal{V}(\boxtimes\, \tau, \sigma) = \bot \\
&\mathcal{V}(\blacksquare\, l, \sigma) = \sigma(l) \\
&\mathcal{V}(\lightning, \sigma) = \bot \\
&\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) = \bot \\
&\mathcal{V}(t_1 \triangleright e_2, \sigma) = \bot \\
&\mathcal{V}(t_1 \bowtie t_2, \sigma) = \begin{cases} \langle v_1, v_2 \rangle & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
&\mathcal{V}(t_1 \blacklozenge t_2, \sigma) = \begin{cases} v_1 & \textbf{when } \mathcal{V}(t_1, \sigma) = v_1 \\ v_2 & \textbf{when } \mathcal{V}(t_1, \sigma) = \bot \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
&\mathcal{V}(t_1 \lozenge t_2, \sigma) = \bot
\end{aligned}$$

Figure 12: Task value observation function $\mathcal{V}$.

The function $\mathcal{F}$ observes if a task is failing. Its definition is given in Fig. 13. A task is failing if it is the fail task ($\lightning$), or if it consists of only failing tasks. This function differs from its concrete counterpart in the clause for user choice. As symbolic normalisation can yield

$$\begin{aligned}
&\mathcal{F} : \text{Tasks} \times \text{States} \rightarrow \text{Booleans} \\
&\mathcal{F}(\square\, v, \sigma) = \text{False} \\
&\mathcal{F}(\boxtimes\, \tau, \sigma) = \text{False} \\
&\mathcal{F}(\blacksquare\, l, \sigma) = \text{False} \\
&\mathcal{F}(\lightning, \sigma) = \text{True} \\
&\mathcal{F}(t_1 \blacktriangleright e_2, \sigma) = \mathcal{F}(t_1, \sigma) \\
&\mathcal{F}(t_1 \triangleright e_2, \sigma) = \mathcal{F}(t_1, \sigma) \\
&\mathcal{F}(t_1 \bowtie t_2, \sigma) = \mathcal{F}(t_1, \sigma) \wedge \mathcal{F}(t_2, \sigma) \\
&\mathcal{F}(t_1 \blacklozenge t_2, \sigma) = \mathcal{F}(t_1, \sigma) \wedge \mathcal{F}(t_2, \sigma) \\
&\mathcal{F}(e_1 \lozenge e_2, \sigma) = \bigwedge \Big(\{ \mathcal{F}(t_1, \sigma_1') \mid e_1, \sigma \Downarrow \overline{t_1, \sigma_1'} \} \cup \\
&\qquad\qquad \{ \mathcal{F}(t_2, \sigma_2') \mid e_2, \sigma \Downarrow \overline{t_2, \sigma_2'} \} \Big)
\end{aligned}$$

Figure 13: Task failing observation function $\mathcal{F}$.

multiple results, all of the results must be failing to make a user choice failing.

5.3 Normalisation

Normalization ($\Downarrow$) reduces tasks until they are ready to receive input. Very little has to be changed to accommodate symbolic execution. Just like the evaluation semantics it now gathers sets of results, each element guarded by a path condition. Figure 14 lists the normalisation semantics.

$$\boxed{e, \sigma \Downarrow \overline{t, \sigma', \varphi}}$$

SN-Done

$$\frac{e, \sigma \downarrow \overline{t, \sigma', \varphi_1} \qquad t, \sigma' \rightsquigarrow \overline{t', \sigma'', \varphi_2}}{e, \sigma \Downarrow \overline{t, \sigma', \varphi_1}}\ \sigma' = \sigma'' \wedge t = t'$$

SN-Repeat

$$\frac{e, \sigma \downarrow \overline{t, \sigma', \varphi_1} \qquad t, \sigma' \rightsquigarrow \overline{t', \sigma'', \varphi_2} \qquad t', \sigma'' \Downarrow \overline{t'', \sigma''', \varphi_3}}{e, \sigma \Downarrow \overline{t'', \sigma''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}\ \sigma' \neq \sigma'' \vee t \neq t'$$

Figure 14: Symbolic normalisation semantics.

Normalisation makes use of the small step striding semantics ($\rightsquigarrow$). Its details are not important here. For more background, we refer to the appendix.

5.4 Handling

The handling semantics ($\rightarrow$) deals with user input. In the symbolic case there are symbols instead of concrete inputs. A complete overview of the rules can be found in Appendix A.4. Figure 15 lists the interesting rules of the symbolic handling semantics.

The three rules for the editors (SH-Change, SH-Fill, SH-Update) clearly show how symbols enter the symbolic execution. The first one for example generates a fresh symbol s and returns an editor containing it.

There are several task combinators where the result depends on user input. For example, the parallel combinator ($\bowtie$) receives an input for either the left or the right branch. To accommodate for all possibilities, the SH-And rule generates both cases. It tags the

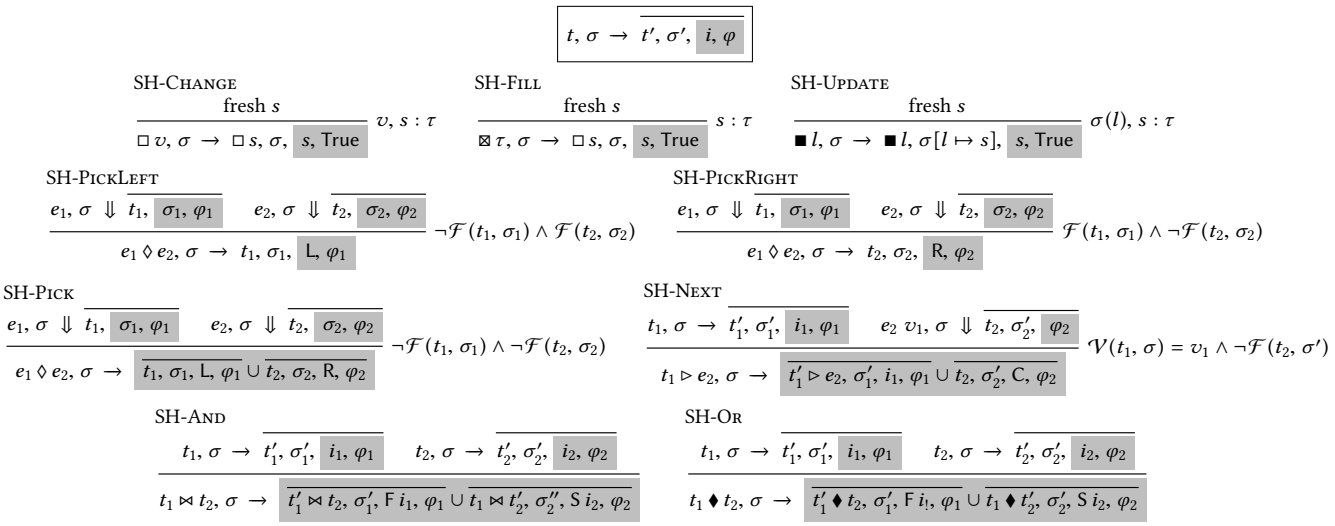


Figure 15: Symbolic handling semantics.

$$
\begin{aligned}
&\mathit{simulate} : \text{Tasks} \times \text{States} \times [\text{Inputs}] \times \text{Predicates} \to \mathcal{P}(\text{Values} \times [\text{Inputs}] \times \text{Predicates}) \\
&\mathit{simulate}\ (t, \sigma, I, \varphi) \quad = \bigcup\{\mathit{simulate}'\ (\text{True}, t, t', \sigma', I \oplus [i'], \varphi \wedge \varphi') \mid t, \sigma \Rightarrow t', \sigma', i', \varphi'\} \\
&\mathit{simulate}' : \text{Booleans} \times \text{Tasks} \times \text{Tasks} \times \text{States} \times [\text{Inputs}] \times \text{Predicates} \to \mathcal{P}(\text{Values} \times [\text{Inputs}] \times \text{Predicates}) \\
&\mathit{simulate}'\ (\mathit{again}, t, t', \sigma', I, \varphi) = \\
&\begin{cases}
\varnothing & \neg\mathcal{S}(\varphi) \\
\{(v, I, \varphi)\} & \mathcal{S}(\varphi) \wedge \mathcal{V}(t', \sigma') = v \\
\mathit{simulate}\ (t', \sigma', I, \varphi) & \mathcal{S}(\varphi) \wedge \mathcal{V}(t', \sigma') = \bot \wedge t' \neq t \\
\bigcup\{\mathit{simulate}'\ (\text{False}, t', t'', \sigma'', I \oplus [i'], \varphi \wedge \varphi') \mid t', \sigma' \Rightarrow t'', \sigma'', i', \varphi'\} & \mathcal{S}(\varphi) \wedge \mathcal{V}(t', \sigma') = \bot \wedge t' = t \wedge \mathit{again} \\
\varnothing & \mathcal{S}(\varphi) \wedge \mathcal{V}(t', \sigma') = \bot \wedge t' = t \wedge \neg\mathit{again}
\end{cases}
\end{aligned}
$$

Figure 16: Simulation function definition.

inputs for the first branch with F and inputs for the second branch with S.

The same principle applies to the external choice combinator ($\Diamond$). The three rules SH-PickLeft, SH-PickRight, and SH-Pick are needed to disallow choosing failing tasks. There is one rule for the case where only the right is failing, one rule when the left is failing, and one for when none of the options are failing.

After input has been handled, tasks are normalised. The combination of those two steps is taken care of by the driving ($\Rightarrow$) semantics, listed in Fig. 17.

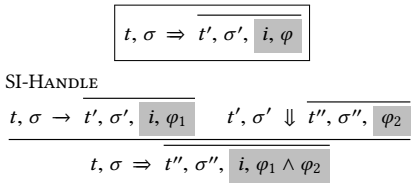


Figure 17: Symbolic driving semantics.

5.5 Simulating

The symbolic driving semantics is a small step semantics. Every step simulates one symbolic input. In order to compute every possible execution, the driving semantics needs to be applied repeatedly, until the task is done. We define a task to be done when it has an observable value: $\mathcal{V}(t', \sigma') \neq \bot$. The simulation function listed in Fig. 16 is recursively called to produce a list of end states and path conditions. It accumulates all symbolic inputs and returns for each possible execution the observable task value v, the path condition φ, and the state σ. We consider a task, state and path condition to be an end state if the task value can be observed, and the path condition is satisfiable, represented by the function $\mathcal{S}$.

The recursion terminates when one of the following conditions is met.

- $\neg\mathcal{S}(\varphi)$ When the path condition cannot be satisfied, we know that all future steps will not be satisfiable either. In fact, no future path condition will be satisfiable, and we can therefore safely remove it.
- $\mathcal{V}(t, \sigma)$ When the current task has a value it is an end state, which we can return.
- $\mathcal{V}(t', \sigma') = \bot \wedge t = t' \wedge \neg\mathit{again}$ When the current task does not produce a value, and it is equal to the previous task except from symbol names in editors, the *simulate* function performs one look-ahead step in case the task is waiting for an independent symbol. This one step look-ahead is encoded by the parameter *again*. When this parameter is set to False, one step look-ahead has been performed and *simulate* does not continue further. If the task has a value it is returned, otherwise the branch is pruned.

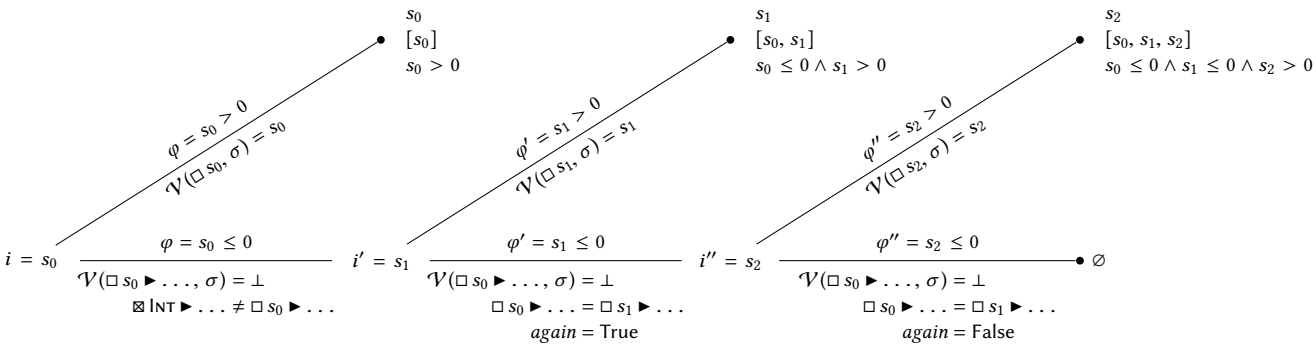


Figure 18: Application of the simulation function to Listing 1.

To better illustrate how the *simulate* function works, we study how it simulates Listing 1. Figure 18 gives a schematic overview of the application of *simulate*. First, it calls the drive semantics to calculate what input the task takes. Users can enter a fresh symbol s_0, as listed on the left. The symbolic execution then branches, since it reaches a conditional. Two cases are generated. Either $s_0 > 0$, the upper branch, or $s_0 \leq 0$, the branch to the right. In the first case, the resulting task has a value, and the symbolic execution ends returning that value and the input. In the second case, the resulting task does not have a value, and the new task is different from the previous task. Therefore, it recurses, and *simulate* is called again.

A fresh symbol s_1 is generated. Again, s_1 can either be greater than zero, or less or equal. In the first case, the resulting task has a value, and the execution ends. In the second case however, the task does not have a value, and we find that the task has not been altered (apart from the new symbol). This results in a recursive call to *simulate*′ with *again* set to False.

Once more a fresh symbol s_2 is generated, and s_2 can be greater than zero, or less or equal. In the first case, the task has a value and we are done. In the second case, it does not have a value, the task again has not changed, but *again* is False and therefore symbolic execution prunes this branch.

This example demonstrates a couple of things. From manual inspection, it is clear that only the first iteration returns an interesting result. When s_0 is greater than zero, the task results in a value that is greater than zero. When the input is less than or equal to zero, simulation continues with the task unchanged.

Why does the simulation still proceed then? Since the editor $\boxtimes$ changes to $\square$, the tasks are not the same after the first step. This causes *simulate* to run an extra iteration. It finds that the task still does not have a value, but now the task has changed. Then *simulate* performs one look-ahead step, by setting the *again*-parameter to False. When this look-ahead does not return a value, the branch is pruned.

5.6 Solving

To check the satisfiability of path conditions $\mathcal{S}(\varphi)$, as well as the properties stated about a program, we make use of an external SMT solver. In the implementation we use Z3, although any other SMT solver supporting SMT-LIB could be used.

For Listing 1, we would like to prove that after any input sequence I, the path conditions φ imply that the value v of the resulting task t' is greater than 0.

$$\varphi \supset v > 0 \qquad \textbf{where}\, v = \mathcal{V}(t', \sigma')$$

As shown in Fig. 18, there are three paths we need to verify. Therefore, we send the following three statements to the SMT solver for verification:

(1) $s_0 > 0 \supset s_0 > 0$
(2) $s_0 \leq 0 \wedge s_1 > 0 \supset s_1 > 0$
(3) $s_0 \leq 0 \wedge s_1 \leq 0 \wedge s_2 > 0 \supset s_2 > 0$

In this example all are trivially solvable.

5.7 Implementation

We implemented our language and its symbolic execution semantics in Haskell. With the help of a couple of GHC extensions, the grammar, typing rules and semantics are almost one-to-one translatable into code. Our tool generates execution trees like the one shown in Fig. 18, which keep track of intermediate normalisations, symbolic inputs, and path conditions. All path conditions are converted to SMT-LIB compatible statements and are verified using the Z3 SMT solver. As of now we do not have a parser, programs must be specified directly as abstract syntax trees.

As is usually the case with symbolic execution, the number of paths grows quickly. The examples in Listings 2 and 3 generate respectively 2112 and 1166 paths, which takes about a minute to calculate. Solving them, however, is almost instantaneous.

5.8 Outlook

Assertions. Other work on symbolic execution often uses assertions, which are included in the program itself. One could imagine an assertion statement **assert** ψ t in $\widehat{\text{TOP}}$ that roughly works as follows. First the SMT solver verifies the property ψ against the current path condition. If the assertion fails, an error message is generated. Then the program continues with task t.

Example 5.1. Consider the following small example program.

$\boxtimes \textsc{Int} \blacktriangleright \lambda x\,.\, \square(\textbf{ref}\ x) \blacktriangleright \lambda l.\ \textbf{assert}\ (!l \equiv x)\ (\square\ \text{"Done"})$

This program asks the user to enter an integer. The entered value is then stored in a reference. The assertion that follows ensures that the store has been updated correctly. Finally the string "Done" is returned.

Assertions have access to all variables in scope, unlike properties as we have currently implemented them. We can overcome this by returning all values of interest at the end of the program.

$\boxtimes\text{INT} \blacktriangleright \lambda x\,.\,\square(\textbf{ref}\ x) \blacktriangleright \lambda\text{store}\,.\,\square\text{"Done"} \blacktriangleright \lambda\_\,.\,\square(x,!\text{store})$

It is now possible to verify that the property $\psi(x,s) = x \equiv s$ holds. This demonstrates that our approach has expressive power similar to assertions. Having assertions in our language would be more convenient for programmers however, and we would like add them in the future.

Input-dependent predicates. Another feature we would like to support in the future are input-dependent predicates.

Example 5.2. Consider the following small program.

$\boxtimes\text{INT} \blacktriangleright \lambda x\,.\,\textbf{if}\ x > 0\ \textbf{then}\ \square\text{"Thank you"}\ \textbf{else}\ \square\text{"Error"}$

The user inputs an integer. If the integer is larger than zero, the program prints a thank you message. If the integer is smaller than zero, an error is returned.

If we want to prove that given a positive input, the program never returns "Error", we need to be able to talk about inputs directly in predicates. Currently our symbolic execution does not support this.

6 PROPERTIES

In this section we describe what it means for the symbolic execution semantics to be correct. We prove it sound and complete with respect to the concrete semantics of $\widehat{\text{TOP}}$.

In order to relate the two semantics, we use the concrete inputs listed in Fig. 19.

Concrete inputs

$$j \quad ::= \quad a \mid \mathsf{F}\,j \mid \mathsf{S}\,j \quad \text{– action, to first, to second}$$

Concrete actions

$$a \quad ::= \quad c \quad \text{– constant}$$
$$\mid \quad \mathsf{C} \mid \mathsf{L} \mid \mathsf{R} \quad \text{– continue, go left, go right}$$

Figure 19: Syntax of concrete inputs.

6.1 Soundness

In order to validate the symbolic execution semantics, we want to show that for every individual symbolic execution step there exists a corresponding concrete one. This soundness property is expressed by Theorem 6.1.

Theorem 6.1 (Soundness of driving). *For all tasks t, states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, such that $t, \sigma \Rightarrow \overline{t', \sigma', i, \varphi}$ we have $M\varphi$ implies $t, M\sigma \overset{Mi}{\Longrightarrow} t'', \sigma''$, $Mt' \equiv t''$ and $M\sigma' \equiv \sigma''$.*

The proof for this lemma is rather straightforward. Since the driving semantics makes use of the handling and the normalisation semantics, we require two lemmas. One showing that the handling semantics is sound, Lemma 6.2, and one showing that the normalisation semantics is sound, Lemma 6.3.

Lemma 6.2 (Soundness of handling). *For all tasks t, states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, such that $t, \sigma \overset{i}{\rightarrow} \overline{t', \sigma', \varphi}$, we have $M\varphi$ implies $t, M\sigma \xrightarrow{Mi} t'', \sigma''$, $t'M \equiv t''$ and $\sigma' M \equiv \sigma''$*

Lemma 6.2 is proven by induction over t. The full proof is listed in Appendix C.

Lemma 6.3 (Soundness of normalisation). *For all expressions e, states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, such that $e, \sigma \Downarrow \overline{t, \sigma', \varphi}$, we have $M\varphi$ implies $e, M\sigma \hat{\Downarrow} t', \sigma''$, $tM \equiv t'$ and $\sigma' M \equiv \sigma''$.*

Since Lemma 6.3 makes use of both the striding and the evaluation semantics, we must show soundness for those too.

Lemma 6.4 (Soundness of striding). *For all tasks t, states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, such that $t, \sigma \rightsquigarrow \overline{t', \sigma', \varphi}$, $M\varphi$ implies $t, M\sigma \hat{\rightsquigarrow} \bar{t'}, \bar{\sigma'}$, $Mt' \equiv \bar{t'} \wedge M\sigma' \equiv \bar{\sigma'}$.*

Lemma 6.5 (Soundness of evaluation). *For all expressions e, states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, such that $e, \sigma \downarrow \overline{v, \sigma', \varphi}$, we have that $M\varphi$ implies $t, M\sigma \hat{\downarrow} \bar{v}, \bar{\sigma'} \wedge Mv \equiv \bar{v} \wedge M\sigma' \equiv \bar{\sigma'}$.*

The full proofs of Lemmas 6.3 to 6.5 are listed in the appendix.

6.2 Completeness

We also want to show that for every concrete execution, a symbolic one exists.

In order to state this Theorem, we require a simulation relation $i \sim j$, which means that the symbolic input i follows the same direction as the concrete input j. This relation is defined below.

Definition 6.6 (Input simulation). A symbolic input i simulates a concrete input j denoted as $i \sim j$ in the following cases.

$s \sim a$, where s is a symbol and a a concrete action.

$i \sim j \supset \mathsf{F}\,i \sim \mathsf{F}\,j$

$i \sim j \supset \mathsf{S}\,i \sim \mathsf{S}\,j$

This allows us to define the completeness property as listed in Theorem 6.7.

Theorem 6.7 (Completeness of driving). *For all t, σ, j such that $t, \sigma \overset{j}{\Rightarrow} \hat{t'}, \hat{\sigma'}$ there exists an $i \sim j$ such that $t, \sigma'' \Rightarrow t'', \sigma''', i, \varphi$.*

The proof of Theorem 6.7 is rather simple. We show that handling is complete (Lemma 6.8) and that the subsequent normalisation is complete (Lemma 6.9).

Lemma 6.8 (Completeness of handling). *For all t, σ, j such that $t, \sigma \overset{j}{\rightarrow} \hat{t'}, \hat{\sigma'}$ there exists an $i \sim j$ such that $t, \sigma'' \rightarrow t'', \sigma''', i, \varphi$.*

Lemma 6.8 is proved by induction over t. We only need to show that every concrete execution is also a symbolic one. The only change needed to convert from concrete to symbolic is the adaption of the input.

Since handling makes use of normalisation, striding, and evaluation, we need to prove that they too are complete. These properties are listed in Lemmas 6.9 to 6.11

Lemma 6.9 (Completeness of normalisation). *For all e, σ such that $e, \sigma \hat{\Downarrow} \hat{t}, \hat{\sigma}$ there exists a symbolic execution $e, \sigma \Downarrow \hat{t}, \hat{\sigma}, \text{True}$.*

Lemma 6.10 (Completeness of striding). *For all t, σ such that $t, \sigma \hat{\rightsquigarrow} \hat{t}, \hat{\sigma}$ there exists a symbolic execution $t, \sigma \downarrow \hat{t}, \hat{\sigma}$,* True.

Lemma 6.11 (Completeness of evaluation). *For all e, σ such that $e, \sigma \hat{\Downarrow} \hat{v}, \hat{\sigma}$ there exists a symbolic execution $e, \sigma \downarrow \hat{v}, \hat{\sigma}$,* True.

Lemmas 6.9 to 6.11 follow trivially, since every concrete execution in these semantics is also a symbolic one.

7 RELATED WORK

Symbolic execution. Symbolic execution [Boyer et al. 1975; King 1975] is typically being applied to imperative programming languages, for example Bucur et al. [2014] prototype a symbolic execution engine for interpreted imperative languages. Cadar et al. [2008] use it to generate test cases for programs that can be compiled to LLVM bytecode. Jaffar et al. [2012] use it for verifying safety properties of C programs.

In recent years it has been used for functional programming languages as well. To name some examples, there is ongoing work by Hallahan et al. [2017] and Xue [2019] to implement a symbolic execution engine for Haskell. Giantsios et al. [2017] use symbolic execution for a mix of concrete and symbolic testing of programs written in a subset of Core Erlang. Their goal is to find executions that lead to a runtime error, either due to an assertion violation or an unhandled exception. Chang et al. [2018] present a symbolic execution engine for a typed lambda calculus with mutable state where only some language constructs recognize symbolic values. They claim that their approach is easier to implement than full symbolic execution and simplifies the burden on the solver, while still considering all execution paths.

The difficulty of symbolic execution for functional languages lies in symbolic higher-order values, that is functions as arguments to other functions. Hallahan et al solve this with a technique called *defunctionalization*, which requires all source code to be present, so that a symbolic function can only be one of the present lambda expressions or function definitions. Giantosis et al also require all source code to be present, but they only analyze first-order functions. They can execute higher-order functions, but only with concrete arguments. Our method also requires closed well-typed terms, so we never execute a higher-order function in isolation. Furthermore, we currently do not allow functions and tasks as task values. Together, this means that symbolic values can never be functions.

Contracts. Another method for guaranteeing correctness of programs are *contracts*. Contracts refine static types with additional conditions. They are enforced at runtime. Contracts were first presented by Meyer [1992] for the Eiffel programming language. Findler and Felleisen [2002] applied this technique to functional programming by implementing a contract checker for Scheme. Their contracts are assertions for higher-order programs. Contracts can be used to specify properties more fine-grained than what a static type system could check. It is possible, for example, to refine the arguments or return values of functions to numbers in a certain range, to positive numbers or non-empty lists.

Nguyen et al. [2017] combine contracts and symbolic execution to provide *soft contract checking*. The two ideas go hand in hand in that contracts aid symbolic execution with a language for specifications and properties for symbolic values, and symbolic execution provides compile-time guarantees and test case generation. They present a prototype implementation to verify Racket programs.

Axiomatic program verification. One of the classical methods of proving partial correctness of programs is Hoare's axiomatic approach [Hoare 1969], which is based on pre- and postconditions. See Nielson and Nielson [1992] for a nice introduction to the topic. The axiomatic approach is usually applied to imperative programs, requires manually stating loop invariants, and manually carrying out proofs.

Some work has been done to bring the axiomatic method to functional programming. The current state of SMT solving allows for automated extraction and solving of a large amount of proof obligations. Notable works in this field are for example the Hoare Type Theory by Nanevski et al. [2006], the Hoare and Dijkstra Monads by Nanevski et al. [2008]; Swamy et al. [2013], or the Hoare logic for the state monad by Swierstra [2009].

The difference between the work cited here and our work is that the axiomatic method focuses on stateful computations, while we try to incorporate input as well.

8 CONCLUSION

In this paper, we have demonstrated how to apply symbolic execution to $\widehat{\text{TOP}}$ in order to verify individual programs. We have developed both a formal system and an implementation of a symbolic execution semantics. Our approach has been validated by proving the formal system correct, and by running the implementation on example programs.

8.1 Future work

There are many ways in which we would like to continue this line of work.

First, we believe that more can be done with symbolic execution. Our current approach only allows proving predicates over task results and input values. We cannot, however, prove properties that depend on the order of the inputs. Since the symbolic execution currently returns a list of symbolic inputs, we think this extension is feasible.

Second, our symbolic execution only applies to $\widehat{\text{TOP}}$. We would like to see if we can fit it to iTasks. This poses several challenges. iTasks does not have a formal semantics in the sense that $\widehat{\text{TOP}}$ has. The current implementation in Clean is the closest thing available to a formal specification. There are also a few language features in iTasks that are not covered by $\widehat{\text{TOP}}$, for example loops.

Third, we would like to apply different kinds of analysis alltogether. Can a certain part of the program be reached? Does a certain property hold at every point in the program? Are two programs equal? And what does it mean for two programs to be equal? We think that these properties require a different approach.

ACKNOWLEDGMENTS

This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

REFERENCES

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 239–254. https://doi.org/10.1145/2541940.2541977

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings 8th USENIX Symposium on Operating Systems Design and Implementation (8th OSDI'08)*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, San Diego, California, USA, 209–224.

Stephen Chang, Alex Knauth, and Emina Torlak. 2018. Symbolic types for lenient symbolic execution. *PACMPL* 2, POPL (2018), 40:1–40:29.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ICFP*. 48–59.

Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2017. Concolic testing for functional languages. *Sci. Comput. Program.* 147 (2017), 109–134. https://doi.org/10.1016/j.scico.2017.04.008

William T. Hallahan, Anton Xue, and Ruzica Piskac. 2017. Building a Symbolic Execution Engine for Haskell. In *Proceedings of TAPAS 17*.

Tony Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM: Communications of the ACM* 12 (1969).

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *Computer Aided Verification (23ed CAV'12)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Lecture Notes in Computer Science (LNCS), Vol. 7358. Springer-Verlag (New York), Berkeley, CA, USA, 758–766.

JM Jansen, F Bolderheij, et al. 2018. Dynamic Resource and Task Management. In *NL ARMS Netherlands Annual Review of Military Studies 2018*. Springer, 91–105.

James C. King. 1975. A New Approach to Program Testing. *SIGPLAN Not.* 10, 6 (April 1975), 228–233. https://doi.org/10.1145/390016.808444

Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop, RWDSL@CGO 2018, Vienna, Austria, February 24-24, 2018*. ACM, 4:1–4:11. https://doi.org/10.1145/3183895.3183902

Bas Lijnse, Jan Martin Jansen, Rinus Plasmeijer, et al. 2012. Incidone: A task-oriented incident coordination tool. In *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM*, Vol. 12.

Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and separation in Hoare type theory. *ACM SIGPLAN Notices* 41, 9 (Sept. 2006), 62–73. https://doi.org/10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 229–240. https://doi.org/10.1145/1411204.1411237

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2017. Higher order symbolic execution for contract verification and refutation. *J. Funct. Program* 27 (2017).

Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics With Applications - A Formal Introduction*. Wiley.

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, Danny De Schreye, Gerda Janssens, and Andy King (Eds.). ACM, 195–206. https://doi.org/10.1145/2370776.2370801

Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A formal foundation for task-oriented programming. In *PPDP'19 (accepted for publication)*.

Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2017. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK*.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 387–398. https://doi.org/10.1145/2491956.2491978

Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 440–451.

Anton Xue. 2019. Lazy Counterfactual Symbolic Execution. *(in submission)* (2019).

A COMPLETE SYMBOLIC SEMANTICS

A.1 Symbolic evaluation rules

$$\boxed{e, \sigma \downarrow \overline{v, \sigma', \varphi}}$$

SE-Value
$$\frac{}{v, \sigma \downarrow v, \sigma, \text{True}}$$

SE-Pair
$$\frac{e_1, \sigma \downarrow \overline{v_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2}}{\langle e_1, e_2 \rangle, \sigma \downarrow \overline{\langle v_1, v_2 \rangle, \sigma'', \varphi_1 \wedge \varphi_2}}$$

SE-First
$$\frac{e_1, \sigma \downarrow \overline{v_1, \sigma', \varphi}}{\text{fst}\langle e_1, e_2 \rangle, \sigma \downarrow \overline{v_1, \sigma', \varphi}}$$

SE-Second
$$\frac{e_2, \sigma \downarrow \overline{v_2, \sigma', \varphi}}{\text{snd}\langle e_1, e_2 \rangle, \sigma \downarrow \overline{v_2, \sigma', \varphi}}$$

SE-Cons
$$\frac{e_1, \sigma \downarrow \overline{v_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2}}{e_1 :: e_2, \sigma \downarrow \overline{v_1 :: v_2, \sigma'', \varphi_1 \wedge \varphi_2}}$$

SE-Head
$$\frac{e, \sigma \downarrow \overline{v_1 :: v_2, \sigma', \varphi}}{\text{head}\, e, \sigma \downarrow \overline{v_1, \sigma', \varphi}}$$

SE-Tail
$$\frac{e, \sigma \downarrow \overline{v_1 :: v_2, \sigma', \varphi}}{\text{tail}\, e, \sigma \downarrow \overline{v_2, \sigma', \varphi}}$$

SE-App
$$\frac{e_1, \sigma \downarrow \overline{\lambda x : \tau . e_1', \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2} \qquad e_1'[x \mapsto v_2], \sigma'' \downarrow \overline{v_1, \sigma''', \varphi_3}}{e_1 e_2, \sigma \downarrow \overline{v_1, \sigma''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}$$

SE-If
$$\frac{e_1, \sigma \downarrow \overline{v_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2} \qquad e_3, \sigma' \downarrow \overline{v_3, \sigma''', \varphi_3}}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \sigma \downarrow \overline{v_2, \sigma'', \varphi_1 \wedge \varphi_2 \wedge v_1} \cup \overline{v_3, \sigma''', \varphi_1 \wedge \varphi_3 \wedge \neg v_1}}$$

SE-Ref
$$\frac{e, \sigma \downarrow \overline{v, \sigma', \varphi} \qquad l \notin Dom(\sigma')}{\textbf{ref}\, e, \sigma \downarrow \overline{l, \sigma'[l \mapsto v], \varphi}}$$

SE-Deref
$$\frac{e, \sigma \downarrow \overline{l, \sigma', \varphi}}{!e, \sigma \downarrow \overline{\sigma'(l), \sigma', \varphi}}$$

SE-Assign
$$\frac{e_1, \sigma \downarrow \overline{l, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2}}{e_1 := e_2, \sigma \downarrow \overline{\langle\rangle, \sigma''[l \mapsto v_2], \varphi_1 \wedge \varphi_2}}$$

SE-Edit
$$\frac{e, \sigma \downarrow \overline{v, \sigma', \varphi}}{\Box\, e, \sigma \downarrow \overline{\Box\, v, \sigma', \varphi}}$$

SE-Enter
$$\frac{}{\boxtimes \tau, \sigma \downarrow \boxtimes \tau, \sigma, \text{True}}$$

SE-Update
$$\frac{e, \sigma \downarrow \overline{l, \sigma', \varphi}}{\blacksquare\, e, \sigma \downarrow \overline{\blacksquare\, l, \sigma', \varphi}}$$

SE-Then
$$\frac{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi}}{e_1 \blacktriangleright e_2, \sigma \downarrow \overline{t_1 \blacktriangleright e_2, \sigma', \varphi}}$$

SE-Next
$$\frac{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi}}{e_1 \triangleright e_2, \sigma \downarrow \overline{t_1 \triangleright e_2, \sigma', \varphi}}$$

SE-And
$$\frac{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{t_2, \sigma'', \varphi_2}}{e_1 \bowtie e_2, \sigma \downarrow \overline{t_1 \bowtie t_2, \sigma'', \varphi_1 \wedge \varphi_2}}$$

SE-Or
$$\frac{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{t_2, \sigma'', \varphi_2}}{e_1 \blacklozenge e_2, \sigma \downarrow \overline{t_1 \blacklozenge t_2, \sigma'', \varphi_1 \wedge \varphi_2}}$$

SE-Xor
$$\frac{}{e_1 \lozenge e_2, \sigma \downarrow e_1 \lozenge e_2, \sigma, \text{True}}$$

SE-Fail
$$\frac{}{\lightning, \sigma \downarrow \lightning, \sigma, \text{True}}$$

A.2 Symbolic striding rules

$$\boxed{t, \sigma \rightsquigarrow \overline{t', \sigma', \varphi}}$$

SS-ThenStay
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{t_1' \blacktriangleright e_2, \sigma', \varphi}} \mathcal{V}(t_1', \sigma') = \bot$$

SS-ThenFail
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi} \quad e_2\ v_1, \sigma' \downarrow \overline{t_2, \sigma'', \_}}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{t_1' \blacktriangleright e_2, \sigma', \varphi}} \mathcal{V}(t_1', \sigma') = v_1 \wedge \mathcal{F}(t_2, \sigma'')$$

SS-ThenCont
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \quad e_2\ v_1, \sigma' \downarrow \overline{t_2, \sigma'', \varphi_2}}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{t_2, \sigma'', \varphi_1 \wedge \varphi_2}} \mathcal{V}(t_1', \sigma') = v_1 \wedge \neg\mathcal{F}(t_2, \sigma'')$$

SS-OrLeft
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}} \mathcal{V}(t_1', \sigma') = v_1$$

SS-OrRight
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \quad t_2, \sigma' \rightsquigarrow \overline{t_2', \sigma'', \varphi_2}}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{t_2', \sigma'', \varphi_1 \wedge \varphi_2}} \mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = v_2$$

SS-OrNone
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \quad t_2, \sigma' \rightsquigarrow \overline{t_2', \sigma'', \varphi_2}}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{t_1' \blacklozenge t_2', \sigma'', \varphi_1 \wedge \varphi_2}} \mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = \bot$$

SS-Edit
$$\frac{}{\square\, v, \sigma \rightsquigarrow \square\, v, \sigma, \mathsf{True}}$$

SS-Fill
$$\frac{}{\boxtimes\, \tau, \sigma \rightsquigarrow \boxtimes\, \tau, \sigma, \mathsf{True}}$$

SS-Update
$$\frac{}{\blacksquare\, l, \sigma \rightsquigarrow \blacksquare\, l, \sigma, \mathsf{True}}$$

SS-Fail
$$\frac{}{\lightning, \sigma \rightsquigarrow \lightning, \sigma, \mathsf{True}}$$

SS-Xor
$$\frac{}{e_1 \lozenge e_2, \sigma \rightsquigarrow e_1 \lozenge e_2, \sigma, \mathsf{True}}$$

SS-Next
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}{t_1 \rhd e_2, \sigma \rightsquigarrow \overline{t_1' \rhd e_2, \sigma', \varphi}}$$

SS-And
$$\frac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \quad t_2, \sigma' \rightsquigarrow \overline{t_2', \sigma'', \varphi_2}}{t_1 \bowtie t_2, \sigma \rightsquigarrow \overline{t_1' \bowtie t_2', \sigma'', \varphi_1 \wedge \varphi_2}}$$

A.3 Symbolic normalisation rules

$$\boxed{e, \sigma \Downarrow \overline{t, \sigma', \varphi}}$$

SN-Done
$$\frac{e, \sigma \downarrow \overline{t, \sigma', \varphi_1} \quad t, \sigma' \rightsquigarrow \overline{t', \sigma'', \varphi_2}}{e, \sigma \Downarrow \overline{t, \sigma', \varphi_1}} \sigma' = \sigma'' \wedge t = t'$$

SN-Repeat
$$\frac{e, \sigma \downarrow \overline{t, \sigma', \varphi_1} \quad t, \sigma' \rightsquigarrow \overline{t', \sigma'', \varphi_2}\ t', \sigma'' \Downarrow \overline{t'', \sigma''', \varphi_3}}{e, \sigma \Downarrow \overline{t'', \sigma''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}} \sigma' \neq \sigma'' \vee t \neq t'$$

A.4 Symbolic handling rules

$$\boxed{t, \sigma \;\to\; \overline{t', \sigma', \; i, \varphi}}$$

SH-Change
$$\frac{\text{fresh } s}{\Box\, v, \sigma \;\to\; \Box\, s, \sigma, \; s, \mathsf{True}} \; v, s : \tau$$

SH-Fill
$$\frac{\text{fresh } s}{\boxtimes\, \tau, \sigma \;\to\; \Box\, s, \sigma, \; s, \mathsf{True}} \; s : \tau$$

SH-Update
$$\frac{\text{fresh } s}{\blacksquare\, l, \sigma \;\to\; \blacksquare\, l, \sigma[l \mapsto s], \; s, \mathsf{True}} \; \sigma(l), s : \tau$$

SH-PassNext
$$\frac{t_1, \sigma \;\to\; \overline{t_1', \sigma', \; i, \varphi}}{t_1 \triangleright e_2, \sigma \;\to\; \overline{t_1' \triangleright e_2, \sigma', \; i, \varphi}} \; \mathcal{V}(t_1, \sigma) = \bot$$

SH-PassNextFail
$$\frac{t_1, \sigma \;\to\; \overline{t_1', \sigma_1', \; i, \varphi} \qquad e_2\; v_1, \sigma \;\Downarrow\; \overline{t_2, \sigma_2', \; \_}}{t_1 \triangleright e_2, \sigma \;\to\; \overline{t_1' \triangleright e_2, \sigma_1', \; i, \varphi}} \; \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{F}(t_2, \sigma_2')$$

SH-Next
$$\frac{t_1, \sigma \;\to\; \overline{t_1', \sigma_1', \; i_1, \varphi_1} \qquad e_2\; v_1, \sigma \;\Downarrow\; \overline{t_2, \sigma_2', \; \varphi_2}}{t_1 \triangleright e_2, \sigma \;\to\; \overline{t_1' \triangleright e_2, \sigma_1', i_1, \varphi_1} \cup \overline{t_2, \sigma_2', \mathsf{C}, \varphi_2}} \; \mathcal{V}(t_1, \sigma) = v_1 \wedge \neg\mathcal{F}(t_2, \sigma')$$

SH-PassThen
$$\frac{t_1, \sigma \;\to\; \overline{t_1', \sigma', \; i, \varphi}}{t_1 \blacktriangleright e_2, \sigma \;\to\; \overline{t_1' \blacktriangleright e_2, \sigma', \; i, \varphi}}$$

SH-Pick
$$\frac{e_1, \sigma \;\Downarrow\; \overline{t_1, \; \sigma_1, \varphi_1} \qquad e_2, \sigma \;\Downarrow\; \overline{t_2, \; \sigma_2, \varphi_2}}{e_1 \lozenge e_2, \sigma \;\to\; \overline{t_1, \sigma_1, \mathsf{L}, \varphi_1} \cup \overline{t_2, \sigma_2, \mathsf{R}, \varphi_2}} \; \neg\mathcal{F}(t_1, \sigma_1) \wedge \neg\mathcal{F}(t_2, \sigma_2)$$

SH-PickLeft
$$\frac{e_1, \sigma \;\Downarrow\; \overline{t_1, \; \sigma_1, \varphi_1} \qquad e_2, \sigma \;\Downarrow\; \overline{t_2, \; \sigma_2, \varphi_2}}{e_1 \lozenge e_2, \sigma \;\to\; t_1, \sigma_1, \; \mathsf{L}, \varphi_1} \; \neg\mathcal{F}(t_1, \sigma_1) \wedge \mathcal{F}(t_2, \sigma_2)$$

SH-PickRight
$$\frac{e_1, \sigma \;\Downarrow\; \overline{t_1, \; \sigma_1, \varphi_1} \qquad e_2, \sigma \;\Downarrow\; \overline{t_2, \; \sigma_2, \varphi_2}}{e_1 \lozenge e_2, \sigma \;\to\; t_2, \sigma_2, \; \mathsf{R}, \varphi_2} \; \mathcal{F}(t_1, \sigma_1) \wedge \neg\mathcal{F}(t_2, \sigma_2)$$

SH-And
$$\frac{t_1, \sigma \;\to\; \overline{t_1', \sigma_1', \; i_1, \varphi_1} \qquad t_2, \sigma \;\to\; \overline{t_2', \sigma_2', \; i_2, \varphi_2}}{t_1 \bowtie t_2, \sigma \;\to\; \overline{t_1' \bowtie t_2, \sigma_1', \mathsf{F}\, i_1, \varphi_1} \cup \overline{t_1 \bowtie t_2', \sigma_2'', \mathsf{S}\, i_2, \varphi_2}}$$

SH-Or
$$\frac{t_1, \sigma \;\to\; \overline{t_1', \sigma_1', \; i_1, \varphi_1} \qquad t_2, \sigma \;\to\; \overline{t_2', \sigma_2', \; i_2, \varphi_2}}{t_1 \blacklozenge t_2, \sigma \;\to\; \overline{t_1' \blacklozenge t_2, \sigma_1', \mathsf{F}\, i_!, \varphi_1} \cup \overline{t_1 \blacklozenge t_2', \sigma_2', \mathsf{S}\, i_2, \varphi_2}}$$

A.5 Symbolic driving rules

$$\boxed{t, \sigma \;\Rightarrow\; \overline{t', \sigma', \; i, \varphi}}$$

SI-Handle
$$\frac{t, \sigma \;\to\; \overline{t', \sigma', \; i, \varphi_1} \qquad t', \sigma' \;\Downarrow\; \overline{t'', \sigma'', \; \varphi_2}}{t, \sigma \;\Rightarrow\; \overline{t'', \sigma'', \; i, \varphi_1 \wedge \varphi_2}}$$

B $\widehat{\text{TOP}}$ SEMANTICS

B.1 Typing rules

$$\boxed{\Gamma, \Sigma \vdash e : \tau}$$

$$\frac{c \in B}{\Gamma, \Sigma \vdash c : \textsc{Bool}}\ \textsc{T-ConstBool} \qquad \frac{c \in I}{\Gamma, \Sigma \vdash c : \textsc{Int}}\ \textsc{T-ConstInt} \qquad \frac{c \in S}{\Gamma, \Sigma \vdash c : \textsc{String}}\ \textsc{T-ConstString}$$

$$\frac{}{\Gamma, \Sigma \vdash \langle\rangle : \textsc{Unit}}\ \textsc{T-Unit} \qquad \frac{x : \tau \in \Gamma}{\Gamma, \Sigma \vdash x : \tau}\ \textsc{T-Var} \qquad \frac{\Sigma(l) = \beta}{\Gamma, \Sigma \vdash l : \textsc{Ref}\ \beta}\ \textsc{T-Loc}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2}{\Gamma, \Sigma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ \textsc{T-Pair} \qquad \frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma, \Sigma \vdash \mathsf{fst}\, e : \tau_1}\ \textsc{T-First} \qquad \frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma, \Sigma \vdash \mathsf{snd}\, e : \tau_2}\ \textsc{T-Second}$$

$$\frac{}{\Gamma, \Sigma \vdash [\,]_\beta : \textsc{List}\ \beta}\ \textsc{T-ListEmpty} \qquad \frac{\Gamma, \Sigma \vdash e_1 : \beta \quad \Gamma, \Sigma \vdash e_2 : \textsc{List}\ \beta}{\Gamma, \Sigma \vdash e_1 :: e_2 : \textsc{List}\ \beta}\ \textsc{T-ListCons}$$

$$\frac{\Gamma, \Sigma \vdash e : \textsc{List}\ \beta}{\Gamma, \Sigma \vdash \mathsf{head}\, e : \beta}\ \textsc{T-ListHead} \qquad \frac{\Gamma, \Sigma \vdash e : \textsc{List}\ \beta}{\Gamma, \Sigma \vdash \mathsf{tail}\, e : \textsc{List}\ \beta}\ \textsc{T-ListTail} \qquad \frac{\Gamma[x : \tau_1], \Sigma \vdash e : \tau_2}{\Gamma, \Sigma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \textsc{T-Abs}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma, \Sigma \vdash e_2 : \tau_1}{\Gamma, \Sigma \vdash e_1 e_2 : \tau_2}\ \textsc{T-App}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Bool} \quad \Gamma, \Sigma \vdash e_2 : \tau \quad \Gamma, \Sigma \vdash e_3 : \tau}{\Gamma, \Sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}\ \textsc{T-If} \qquad \frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \textbf{ref}\, e : \textsc{Ref}\ \beta}\ \textsc{T-Ref} \qquad \frac{\Gamma, \Sigma \vdash e : \textsc{Ref}\ \beta}{\Gamma, \Sigma \vdash !e : \beta}\ \textsc{T-Deref}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Ref}\ \beta \quad \Gamma, \Sigma \vdash e_2 : \beta}{\Gamma, \Sigma \vdash e_1 := e_2 : \textsc{Unit}}\ \textsc{T-Assign}$$

$$\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \square\, e : \textsc{Task}\ \tau}\ \textsc{T-Edit} \qquad \frac{}{\Gamma, \Sigma \vdash \boxtimes\, \tau : \textsc{Task}\ \tau}\ \textsc{T-Enter} \qquad \frac{\Gamma, \Sigma \vdash e : \textsc{Ref}\ \beta}{\Gamma, \Sigma \vdash \blacksquare\, e : \textsc{Task}\ \beta}\ \textsc{T-Update}$$

$$\frac{}{\Gamma, \Sigma \vdash \lightning : \textsc{Task}\ \tau}\ \textsc{T-Fail} \qquad \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \textsc{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \blacktriangleright e_2 : \textsc{Task}\ \tau_2}\ \textsc{T-Then}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \textsc{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \triangleright e_2 : \textsc{Task}\ \tau_2}\ \textsc{T-Next} \qquad \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \bowtie e_2 : \textsc{Task}\ (\tau_1 \times \tau_2)}\ \textsc{T-And}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\ \tau}{\Gamma, \Sigma \vdash e_1 \blacklozenge e_2 : \textsc{Task}\ \tau}\ \textsc{T-Or} \qquad \frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\ \tau}{\Gamma, \Sigma \vdash e_1 \lozenge e_2 : \textsc{Task}\ \tau}\ \textsc{T-Xor}$$

B.2 Evaluation rules

$$\boxed{e, \hat{\sigma} \downarrow \hat{v}, \hat{\sigma}'}$$

$$\frac{e_1, \hat{\sigma} \downarrow \lambda x : \tau . \hat{e_1}', \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{v_2}, \hat{\sigma}'' \quad e_1'[x \mapsto v_2], \hat{\sigma}'' \downarrow \hat{v_1}, \hat{\sigma}'''}{e_1 e_2, \hat{\sigma} \downarrow \hat{v_1}, \hat{\sigma}'''}\ \textsc{E-App}$$

$$\frac{e_1, \hat{\sigma} \downarrow \text{True}, \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{v_2}, \hat{\sigma}''}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \hat{\sigma} \downarrow \hat{v_2}, \hat{\sigma}''}\ \textsc{E-IfTrue} \qquad \frac{e, \hat{\sigma} \downarrow \hat{v}, \hat{\sigma}' \quad l \notin Dom(\hat{\sigma}')}{\textbf{ref}\, e, \hat{\sigma} \downarrow l, \hat{\sigma}'[l \mapsto \hat{v}]}\ \textsc{E-Ref}$$

$$\frac{e_1, \hat{\sigma} \downarrow \text{False}, \hat{\sigma}' \quad e_3, \hat{\sigma}' \downarrow \hat{v_3}, \hat{\sigma}''}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \hat{\sigma} \downarrow \hat{v_3}, \hat{\sigma}''}\ \textsc{E-IfFalse} \qquad \frac{e, \hat{\sigma} \downarrow l, \hat{\sigma}'}{!e, \hat{\sigma} \downarrow \hat{\sigma}'(l), \hat{\sigma}'}\ \textsc{E-Deref} \qquad \frac{}{v, \hat{\sigma} \downarrow v, \hat{\sigma}}\ \textsc{E-Value}$$

$$\frac{e_1, \hat{\sigma} \downarrow l, \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{v_2}, \hat{\sigma}''}{e_1 := e_2, \hat{\sigma} \downarrow \langle\rangle, \hat{\sigma}''[l \mapsto \hat{v_2}]}\ \textsc{E-Assign} \qquad \frac{e_1, \hat{\sigma} \downarrow \hat{v_1}, \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{v_2}, \hat{\sigma}''}{\langle e_1, e_2\rangle, \hat{\sigma} \downarrow \langle \hat{v_1}, \hat{v_2}\rangle, \hat{\sigma}''}\ \textsc{E-Pair}$$

$$\frac{e_1, \hat{\sigma} \downarrow \hat{v_1}, \hat{\sigma}'}{\mathsf{fst}\langle e_1, e_2\rangle, \hat{\sigma} \downarrow \hat{v_1}, \hat{\sigma}'}\ \textsc{E-First} \qquad \frac{e_2, \hat{\sigma} \downarrow \hat{v_2}, \hat{\sigma}'}{\mathsf{snd}\langle e_1, e_2\rangle, \hat{\sigma} \downarrow \hat{v_2}, \hat{\sigma}'}\ \textsc{E-Second} \qquad \frac{e_1, \hat{\sigma} \downarrow \hat{v_1}, \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{v_2}, \hat{\sigma}''}{e_1 :: e_2, \hat{\sigma} \downarrow \hat{v_1} :: \hat{v_2}, \hat{\sigma}''}\ \textsc{E-Cons}$$

$$\frac{e, \hat{\sigma} \downarrow \hat{v_1} :: \hat{v_2}, \hat{\sigma}'}{\mathsf{head}\, e, \hat{\sigma} \downarrow \hat{v_1}, \hat{\sigma}'}\ \textsc{E-Head} \qquad \frac{e, \hat{\sigma} \downarrow \hat{v_1} :: \hat{v_2}, \hat{\sigma}'}{\mathsf{tail}\, e, \hat{\sigma} \downarrow \hat{v_2}, \hat{\sigma}'}\ \textsc{E-Tail}$$

$$\frac{e, \hat{\sigma} \downarrow \hat{v}, \hat{\sigma}'}{\square\, e, \hat{\sigma} \downarrow \square\, \hat{v}, \hat{\sigma}'}\ \textsc{E-Edit} \qquad \frac{e, \hat{\sigma} \downarrow l, \hat{\sigma}'}{\blacksquare\, e, \hat{\sigma} \downarrow \blacksquare\, l, \hat{\sigma}'}\ \textsc{E-Update} \qquad \frac{e_1, \hat{\sigma} \downarrow \hat{t_1}, \hat{\sigma}'}{e_1 \blacktriangleright e_2, \hat{\sigma} \downarrow \hat{t_1} \blacktriangleright e_2, \hat{\sigma}'}\ \textsc{E-Then}$$

$$\frac{e_1, \hat{\sigma} \downarrow \hat{t_1}, \hat{\sigma}'}{e_1 \triangleright e_2, \hat{\sigma} \downarrow \hat{t_1} \triangleright e_2, \hat{\sigma}'}\ \textsc{E-Next} \qquad \frac{e_1, \hat{\sigma} \downarrow \hat{t_1}, \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{t_2}, \hat{\sigma}''}{e_1 \bowtie e_2, \hat{\sigma} \downarrow \hat{t_1} \bowtie \hat{t_2}, \hat{\sigma}''}\ \textsc{E-And}$$

$$\frac{e_1, \hat{\sigma} \downarrow \hat{t_1}, \hat{\sigma}' \quad e_2, \hat{\sigma}' \downarrow \hat{t_2}, \hat{\sigma}''}{e_1 \blacklozenge e_2, \hat{\sigma} \downarrow \hat{t_1} \blacklozenge \hat{t_2}, \hat{\sigma}''}\ \textsc{E-Or}$$

B.3 Striding rules

$$\boxed{t, \hat{\sigma} \hat{\rightsquigarrow} \hat{t}', \hat{\sigma}'}$$

S-ThenStay
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}'}{t_1 \blacktriangleright e_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}' \blacktriangleright e_2, \hat{\sigma}'} \mathcal{V}(\hat{t_1}', \hat{\sigma}') = \bot$$

S-ThenFail
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}' \quad e_2\ \hat{v_1}, \hat{\sigma}' \hat{\downarrow} \hat{t_2}, \hat{\sigma}''}{t_1 \blacktriangleright e_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}' \blacktriangleright e_2, \hat{\sigma}'} \mathcal{V}(\hat{t_1}', \hat{\sigma}') = \hat{v_1} \wedge \mathcal{F}(\hat{t_2}, \hat{\sigma}'')$$

S-ThenCont
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}' \quad e_2\ \hat{v_1}, \hat{\sigma}' \hat{\downarrow} \hat{t_2}, \hat{\sigma}''}{t_1 \blacktriangleright e_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_2}, \hat{\sigma}''} \mathcal{V}(\hat{t_1}', \hat{\sigma}') = \hat{v_1} \wedge \neg\mathcal{F}(\hat{t_2}, \hat{\sigma}'')$$

S-OrLeft
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}'}{t_1 \blacklozenge t_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}'} \mathcal{V}(\hat{t_1}', \hat{\sigma}') = \hat{v_1}$$

S-OrRight
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}' \quad t_2, \hat{\sigma}' \hat{\rightsquigarrow} \hat{t_2}', \hat{\sigma}''}{t_1 \blacklozenge t_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_2}', \hat{\sigma}''} \mathcal{V}(\hat{t_1}', \hat{\sigma}') = \bot \wedge \mathcal{V}(\hat{t_2}', \hat{\sigma}'') = \hat{v_2}$$

S-OrNone
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}' \quad t_2, \hat{\sigma}' \hat{\rightsquigarrow} \hat{t_2}', \hat{\sigma}''}{t_1 \blacklozenge t_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}' \blacklozenge \hat{t_2}', \hat{\sigma}''} \mathcal{V}(\hat{t_1}', \hat{\sigma}') = \bot \wedge \mathcal{V}(\hat{t_2}', \hat{\sigma}'') = \bot$$

S-Edit
$$\frac{}{\square\, v, \hat{\sigma} \hat{\rightsquigarrow} \square\, v, \hat{\sigma}}$$

S-Fill
$$\frac{}{\boxtimes\, \tau, \hat{\sigma} \hat{\rightsquigarrow} \boxtimes\, \tau, \hat{\sigma}}$$

S-Update
$$\frac{}{\blacksquare\, l, \hat{\sigma} \hat{\rightsquigarrow} \blacksquare\, l, \hat{\sigma}}$$

S-Fail
$$\frac{}{\lightning, \hat{\sigma} \hat{\rightsquigarrow} \lightning, \hat{\sigma}}$$

S-Xor
$$\frac{}{e_1 \lozenge e_2, \hat{\sigma} \hat{\rightsquigarrow} e_1 \lozenge e_2, \hat{\sigma}}$$

S-Next
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}'}{t_1 \triangleright e_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}' \triangleright e_2, \hat{\sigma}'}$$

S-And
$$\frac{t_1, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}', \hat{\sigma}' \quad t_2, \hat{\sigma}' \hat{\rightsquigarrow} \hat{t_2}', \hat{\sigma}''}{t_1 \bowtie t_2, \hat{\sigma} \hat{\rightsquigarrow} \hat{t_1}' \bowtie \hat{t_2}', \hat{\sigma}''}$$

B.4 Normalisation rules

$$\boxed{e, \hat{\sigma} \hat{\Downarrow} \hat{t}, \hat{\sigma}'}$$

N-Done
$$\frac{e, \hat{\sigma} \hat{\downarrow} \hat{t}, \hat{\sigma}' \quad \hat{t}, \hat{\sigma}' \hat{\rightsquigarrow} \hat{t}', \hat{\sigma}''}{e, \hat{\sigma} \hat{\Downarrow} \hat{t}, \hat{\sigma}'} \hat{\sigma}' = \hat{\sigma}'' \wedge \hat{t} = \hat{t}'$$

N-Repeat
$$\frac{e, \hat{\sigma} \hat{\downarrow} \hat{t}, \hat{\sigma}' \quad \hat{t}, \hat{\sigma}' \hat{\rightsquigarrow} \hat{t}', \hat{\sigma}'' \quad \hat{t}', \hat{\sigma}'' \hat{\Downarrow} \hat{t}'', \hat{\sigma}'''}{e, \hat{\sigma} \hat{\Downarrow} \hat{t}'', \hat{\sigma}'''} \hat{\sigma}' \neq \hat{\sigma}'' \vee \hat{t} \neq \hat{t}'$$

B.5 Handling rules

$$\boxed{t, \hat{\sigma} \xrightarrow{j} \hat{t}', \hat{\sigma}'}$$

H-Change
$$\frac{}{\square\, v, \hat{\sigma} \xrightarrow{v'} \square\, v', \hat{\sigma}} v, v' : \tau$$

H-Fill
$$\frac{}{\boxtimes\, \tau, \hat{\sigma} \xrightarrow{v} \square\, v, \hat{\sigma}} v : \tau$$

H-Update
$$\frac{}{\blacksquare\, l, \hat{\sigma} \xrightarrow{v} \blacksquare\, l, \hat{\sigma}[l \mapsto v]} \sigma(l), v : \tau$$

H-Next
$$\frac{e_2\ \hat{v_1}, \sigma \hat{\Downarrow} \hat{t_2}, \hat{\sigma}'}{t_1 \triangleright e_2, \sigma \xrightarrow{\mathsf{C}} \hat{t_2}, \hat{\sigma}'} \mathcal{V}(t_1, \sigma) = \hat{v_1} \wedge \neg\mathcal{F}(\hat{t_2}, \hat{\sigma}')$$

H-PickLeft
$$\frac{e_1, \sigma \Downarrow \hat{t_1}, \hat{\sigma}'}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{L}} \hat{t_1}, \hat{\sigma}'} \neg\mathcal{F}(\hat{t_1}, \hat{\sigma}')$$

H-PickRight
$$\frac{e_2, \sigma \hat{\Downarrow} \hat{t_2}, \hat{\sigma}'}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{R}} \hat{t_2}, \hat{\sigma}'} \neg\mathcal{F}(\hat{t_2}, \hat{\sigma}')$$

H-PassThen
$$\frac{t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{j} \hat{t_1'} \blacktriangleright e_2, \sigma'}$$

H-PassNext
$$\frac{t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \triangleright e_2, \sigma \xrightarrow{j} \hat{t_1'} \triangleright e_2, \sigma'}$$

H-FirstAnd
$$\frac{t_1, \sigma \xrightarrow{j} \hat{t_1}', \hat{\sigma}'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{F}\, j} \hat{t_1}' \bowtie t_2, \hat{\sigma}'}$$

H-SecondAnd
$$\frac{t_2, \sigma \xrightarrow{j} \hat{t_2}', \hat{\sigma}'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{S}\, j} t_1 \bowtie \hat{t_2}', \hat{\sigma}'}$$

H-FirstOr
$$\frac{t_1, \sigma \xrightarrow{j} \hat{t_1}', \hat{\sigma}'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{F}\, j} \hat{t_1}' \blacklozenge t_2, \hat{\sigma}'}$$

H-SecondOr
$$\frac{t_2, \sigma \xrightarrow{j} \hat{t_2}', \hat{\sigma}'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{S}\, j} t_1 \blacklozenge \hat{t_2}', \hat{\sigma}'}$$

B.6 Driving rules

$$\boxed{\hat{t}, \hat{\sigma} \overset{j}{\Rightarrow} \hat{t}', \hat{\sigma}'}$$

I-Handle
$$\frac{t, \sigma \xrightarrow{j} \hat{t}', \hat{\sigma}' \quad \hat{t}', \hat{\sigma}' \hat{\Downarrow} \hat{t}'', \hat{\sigma}''}{t, \sigma \overset{j}{\Rightarrow} \hat{t}'', \hat{\sigma}''}$$

C SOUNDNESS PROOFS

Proof of Lemma 6.5. We prove Lemma 6.5 by induction over e.

Case $e = v$

One rule applies, namely $\dfrac{\text{SE-Value}}{v, \sigma \downarrow v, \boxed{\sigma, \mathsf{True}}}$ Since this rule does not generate constraints, any M will do. Since neither the state, nor the expression is altered by the evaluation rule $\dfrac{\text{E-Value}}{v, \hat{\sigma} \hat{\downarrow} v, \hat{\sigma}}$, this case holds true trivially.

Case $e = \langle e_1, e_2 \rangle$

One rule applies, namely $\dfrac{\text{SE-Pair} \quad e_1, \sigma \downarrow \overline{v_1, \sigma', \boxed{\varphi_1}} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \boxed{\varphi_2}}}{\langle e_1, e_2 \rangle, \sigma \downarrow \overline{\langle v_1, v_2 \rangle, \sigma'', \boxed{\varphi_1 \wedge \varphi_2}}}$

Provided that $M\varphi_1 \wedge M\varphi_2$, we need to demonstrate that $\dfrac{\text{E-Pair} \quad e_1, \hat{\sigma} \hat{\downarrow} \hat{v_1}, \hat{\sigma'} \qquad e_2, \hat{\sigma'} \hat{\downarrow} \hat{v_2}, \hat{\sigma''}}{\langle e_1, e_2 \rangle, \hat{\sigma} \hat{\downarrow} \langle \hat{v_1}, \hat{v_2} \rangle, \hat{\sigma''}}$ with $\hat{\sigma} = M\sigma$, $M\langle v_1, v_2 \rangle \equiv \langle \hat{v_1}, \hat{v_2} \rangle$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \hat{\downarrow} \hat{v_1}, \hat{\sigma'} \wedge M_1 v_1 \equiv \hat{v_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and $\forall M_2.M_2\varphi_2 \supset e_2, M_2\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''} \wedge M_2 v_2 \equiv \hat{v_2} \wedge M_2\sigma' \equiv \hat{\sigma''}$

Since M satisfies both φ_1 and φ_2, and we know that $M\sigma' \equiv \hat{\sigma'}$ we obtain that $e_1, M\sigma \hat{\downarrow} \hat{v_1}, \hat{\sigma'}$, $e_2, M\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''}$, $M\sigma' \equiv \hat{\sigma'}$, $Mv_1 \equiv \hat{v_1}$ and $Mv_2 \equiv \hat{v_2}$ and therefore $M\langle v_1, v_2 \rangle \equiv \langle \hat{v_1}, \hat{v_2} \rangle$. From the IH we directly obtain that $M\sigma'' \equiv \hat{\sigma''}$.

Case $e = \mathsf{fst}\, e$

One rule applies, namely $\dfrac{\text{SE-First} \quad e_1, \sigma \downarrow \overline{v_1, \sigma', \boxed{\varphi}}}{\mathsf{fst}\langle e_1, e_2 \rangle, \sigma \downarrow \overline{v_1, \sigma', \boxed{\varphi}}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-First} \quad e_1, \hat{\sigma} \hat{\downarrow} \hat{v_1}, \hat{\sigma'}}{\mathsf{fst}\langle e_1, e_2 \rangle, \hat{\sigma} \hat{\downarrow} \hat{v_1}, \hat{\sigma'}}$ with $\hat{\sigma} = M\sigma$, $Mv_1 \equiv \hat{v_1}$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \langle \hat{v_1}, \hat{v_2} \rangle, \hat{\sigma'} \wedge M_1\langle v_1, v_2 \rangle \equiv \langle \hat{v_1}, \hat{v_2} \rangle \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we directly obtain that $\mathsf{fst}\, e, \sigma \hat{\downarrow} \hat{v_1}, Mv_1 \equiv \hat{v_1}$ and $M\sigma' \equiv \hat{\sigma'}$.

Case $e = \mathsf{snd}\, e$

One rule applies, namely $\dfrac{\text{SE-Second} \quad e_2, \sigma \downarrow \overline{v_2, \sigma', \boxed{\varphi}}}{\mathsf{snd}\langle e_1, e_2 \rangle, \sigma \downarrow \overline{v_2, \sigma', \boxed{\varphi}}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Second} \quad e_2, \hat{\sigma} \hat{\downarrow} \hat{v_2}, \hat{\sigma'}}{\mathsf{snd}\langle e_1, e_2 \rangle, \hat{\sigma} \hat{\downarrow} \hat{v_2}, \hat{\sigma'}}$ with $\hat{\sigma} = M\sigma$, $Mv_2 \equiv \hat{v_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \langle \hat{v_1}, \hat{v_2} \rangle, \hat{\sigma'} \wedge M_1\langle v_1, v_2 \rangle \equiv \langle \hat{v_1}, \hat{v_2} \rangle \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we directly obtain that $\mathsf{snd}\, e, \sigma \hat{\downarrow} \hat{v_2}, Mv_2 \equiv \hat{v_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

Case $e = e_1 :: e_2$

One rule applies, namely $\dfrac{\text{SE-Cons} \quad e_1, \sigma \downarrow \overline{v_1, \sigma', \boxed{\varphi_1}} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \boxed{\varphi_2}}}{e_1 :: e_2, \sigma \downarrow \overline{v_1 :: v_2, \sigma'', \boxed{\varphi_1 \wedge \varphi_2}}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Cons} \quad e_1, \hat{\sigma} \hat{\downarrow} \hat{v_1}, \hat{\sigma'} \qquad e_2, \hat{\sigma'} \hat{\downarrow} \hat{v_2}, \hat{\sigma''}}{e_1 :: e_2, \hat{\sigma} \hat{\downarrow} \hat{v_1} :: \hat{v_2}, \hat{\sigma''}}$ with $bar\sigma = M\sigma$, $Mv_1 :: v_2 \equiv \hat{v_1} :: \hat{v_2}$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \hat{\downarrow} \hat{v_1}, \hat{\sigma'} \wedge M_1 v_1 \equiv \hat{v_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and $\forall M_2.M_2\varphi_2 \supset e_2, M_2\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''} \wedge M_2 v_2 \equiv \hat{v_2} \wedge M_2\sigma' \equiv \hat{\sigma''}$
Since M satisfies both φ_1 and φ_2, and we know that $M\sigma' \equiv \hat{\sigma'}$ we obtain that $e_1, M\sigma \hat{\downarrow} \hat{v_1}, \hat{\sigma'}$, $e_2, M\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''}$, $M\sigma' \equiv \hat{\sigma'}$, $Mv_1 \equiv \hat{v_1}$ and $Mv_2 \equiv \hat{v_2}$ and therefore $M(v_1 :: v_2) \equiv \hat{v_1} :: \hat{v_2}$. From the IH we directly obtain that $M\sigma'' \equiv \hat{\sigma''}$.

Case $e = \mathsf{head}\, e$

One rule applies, namely $\dfrac{\text{SE-Head}\quad e, \sigma \downarrow \overline{v_1 :: v_2, \sigma', \varphi}}{\mathsf{head}\, e, \sigma \downarrow \overline{v_1, \sigma', \varphi}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Head}\quad e, \hat{\sigma} \hat{\downarrow} \hat{v_1} :: \hat{v_2}, \hat{\sigma'}}{\mathsf{head}\, e, \hat{\sigma} \hat{\downarrow} \hat{v_1}, \hat{\sigma'}}$ with $\hat{\sigma} = M\sigma$, $Mv_1 \equiv \hat{v_1}$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \hat{v_1} :: \hat{v_2}, \hat{\sigma'} \wedge M_1(v_1 :: v_2) \equiv \hat{v_1} :: \hat{v_2} \wedge M_1\sigma' \equiv \hat{\sigma'}$
Since M satisfies φ, we directly obtain that $\mathsf{head}\, e, \sigma \hat{\downarrow} \hat{v_1}$, $Mv_1 \equiv \hat{v_1}$ and $M\sigma' \equiv \hat{\sigma'}$.

Case $e = \mathsf{tail}\, e$

One rule applies, namely $\dfrac{\text{SE-Tail}\quad e, \sigma \downarrow \overline{v_1 :: v_2, \sigma', \varphi}}{\mathsf{tail}\, e, \sigma \downarrow \overline{v_2, \sigma', \varphi}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Tail}\quad e, \hat{\sigma} \hat{\downarrow} \hat{v_1} :: \hat{v_2}, \hat{\sigma'}}{\mathsf{tail}\, e, \hat{\sigma} \hat{\downarrow} \hat{v_2}, \hat{\sigma'}}$ with $\hat{\sigma} = M\sigma$, $Mv_2 \equiv \hat{v_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \hat{v_1} :: \hat{v_2}, \hat{\sigma'} \wedge M_1(v_1 :: v_2) \equiv \hat{v_1} :: \hat{v_2} \wedge M_1\sigma' \equiv \hat{\sigma'}$
Since M satisfies φ, we directly obtain that $\mathsf{tail}\, e, \sigma \hat{\downarrow} \hat{v_2}$, $Mv_2 \equiv \hat{v_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

Case $e = e_1 e_2$

One rule applies, namely $\dfrac{\text{SE-App}\quad e_1, \sigma \downarrow \overline{\lambda x : \tau.e_1', \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2} \qquad e_1'[x \mapsto v_2], \sigma'' \downarrow \overline{v_1, \sigma''', \varphi_3}}{e_1 e_2, \sigma \downarrow \overline{v_1, \sigma''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}$

Provided that $M\varphi_1 \wedge M\varphi_2 \wedge M\varphi_3$, we need to demonstrate that $\dfrac{\text{E-App}\quad e_1, \hat{\sigma} \hat{\downarrow} \lambda x : \tau.\hat{e_1}', \hat{\sigma'} \qquad e_2, \hat{\sigma'} \hat{\downarrow} \hat{v_2}, \hat{\sigma''} \qquad e_1'[x \mapsto v_2], \hat{\sigma''} \hat{\downarrow} \hat{v_1}, \hat{\sigma'''}}{e_1 e_2, \hat{\sigma} \hat{\downarrow} \hat{v_1}, \hat{\sigma'''}}$ with $\hat{\sigma} = M\sigma$, $Mv_1 \equiv \hat{v_1}$ and $M\sigma''' \equiv \hat{\sigma'''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \hat{\downarrow} \lambda x : \tau.\hat{e_1'}, \hat{\sigma'} \wedge M_1 \lambda x : \tau.e_1' \equiv \lambda x : \tau.\hat{e_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset e_2, M_2\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''} \wedge M_2 v_2 \equiv \hat{v_2} \wedge M_2\sigma'' \equiv \hat{\sigma''}$ and
$\forall M_3.M_3\varphi_3 \supset \hat{e_1'}[x \mapsto \hat{v_2}], M_3\sigma'' \hat{\downarrow} \hat{v_1}, \hat{\sigma'''} \wedge M_3 v_1 \equiv \hat{v_1} \wedge M_3\sigma''' \equiv \hat{\sigma'''}$.
Since M satisfies both φ_1, φ_2 and φ_3, and we know that $M\sigma' \equiv \hat{\sigma'}$ and $M\sigma'' \equiv \hat{\sigma''}$, we obtain that $e_1, M\sigma \hat{\downarrow} \lambda x : \tau.\hat{e_1'}, \hat{\sigma'}$, $e_2, M\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''}$ and $\hat{e_1'}[x \mapsto \hat{v_2}], M\sigma'' \hat{\downarrow} \hat{v_1}, \hat{\sigma'''}$. We can then directly conclude that $Mv_1 \equiv \hat{v_1}$ and $M\sigma''' \equiv \hat{\sigma'''}$.

Case $e = \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3$

One rule applies, namely $\dfrac{\text{SE-If}\quad e_1, \sigma \downarrow \overline{v_1, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2} \qquad e_3, \sigma' \downarrow \overline{v_3, \sigma''', \varphi_3}}{\textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3, \sigma \downarrow \overline{v_2, \sigma'', \varphi_1 \wedge \varphi_2 \wedge v_1} \cup \overline{v_3, \sigma''', \varphi_1 \wedge \varphi_3 \wedge \neg v_1}}$

In case that $M\varphi_1 \wedge M\varphi_2 \wedge Mv_1$, we need to demonstrate that $\dfrac{\text{E-IfTrue}\quad e_1, \hat{\sigma} \hat{\downarrow} \text{True}, \hat{\sigma'} \qquad e_2, \hat{\sigma'} \hat{\downarrow} \hat{v_2}, \hat{\sigma''}}{\textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3, \hat{\sigma} \hat{\downarrow} \hat{v_2}, \hat{\sigma''}}$ with $\hat{\sigma} = M\sigma$, $Mv_2 = \hat{v_2}$ and $M\sigma'' = \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \hat{\downarrow} \hat{v_1}, \hat{\sigma'} \wedge M_1 v_1 \equiv \hat{v_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset e_2, M_2\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''} \wedge M_2 v_2 \equiv \hat{v_2} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.

Since M satisfies φ_1, and $Mv_1 = \mathsf{True}$, we know from the application of the induction hypothesis above, that $\hat{v_1} = \mathsf{True}$. Furthermore, M satisfies φ_2, so we directly obtain that $Mv_2 = \hat{v_2}$ and $M\sigma'' = \hat{\sigma''}$.

In case that $M\varphi_1 \wedge M\varphi_3 \wedge M\neg v_1$, we need to demonstrate that $\dfrac{\text{E-IfFalse}}{\dfrac{e_1, \hat{\sigma} \hat{\downarrow} \mathsf{False}, \hat{\sigma'} \qquad e_3, \hat{\sigma'} \hat{\downarrow} \hat{v_3}, \hat{\sigma''}}{\textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3, \hat{\sigma} \hat{\downarrow} \hat{v_3}, \hat{\sigma''}}}$ with $\hat{\sigma} = M\sigma$, $Mv_3 = \hat{v_3}$ and $M\sigma'' = \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \hat{\downarrow} \hat{v_1}, \hat{\sigma'} \wedge M_1 v_1 \equiv \hat{v_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_3.M_3\varphi_3 \supset e_3, M_3\sigma' \hat{\downarrow} \hat{v_3}, \hat{\sigma''} \wedge M_3 v_3 \equiv \hat{v_3} \wedge M_3\sigma'' \equiv \hat{\sigma''}$.

Since M satisfies φ_1, and $Mv_1 = \mathsf{False}$, we know from the application of the induction hypothesis above, that $\hat{v_1} = \mathsf{False}$. Furthermore, M satisfies φ_3, so we directly obtain that $Mv_3 = \hat{v_3}$ and $M\sigma'' = \hat{\sigma''}$.

Case $e = \textbf{ref}\, e$

One rule applies, namely $\dfrac{\text{SE-Ref}}{\dfrac{e, \sigma \downarrow \overline{v, \sigma', \varphi} \qquad l \notin Dom(\sigma')}{\textbf{ref}\, e, \sigma \downarrow \overline{l, \sigma'[l \mapsto v], \varphi}}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Ref}}{\dfrac{e, \hat{\sigma} \hat{\downarrow} \hat{v}, \hat{\sigma'} \qquad l \notin Dom(\hat{\sigma'})}{\textbf{ref}\, e, \hat{\sigma} \hat{\downarrow} l, \hat{\sigma'}[l \mapsto \hat{v}]}}$ with $\hat{\sigma} = M\sigma$, $Ml \equiv l$ and $M\sigma'[l \mapsto v] \equiv \hat{\sigma'}[l \mapsto \hat{v}]$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \hat{v}, \hat{\sigma'} \wedge M_1 v \equiv \hat{v} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

We assume that the assignment of location references happens in a deterministic manner, and that we can therefore conclude that exactly the same l is used in both cases. Since l cannot contain any symbols, $Ml \equiv l$ holds trivially.

Since M satisfies φ, we obtain that $e, M\sigma \hat{\downarrow} \hat{v}, \hat{\sigma'}$ and $Mv \equiv \hat{v}$. This, together with $M\sigma' \equiv \hat{\sigma'}$ obtained from the induction hypothesis, we can conclude that $M\sigma'[l \mapsto v] \equiv \hat{\sigma'}[l \mapsto \hat{v}]$.

Case $e = !e$

One rule applies, namely $\dfrac{\text{SE-Deref}}{\dfrac{e, \sigma \downarrow \overline{l, \sigma', \varphi}}{!e, \sigma \downarrow \overline{\sigma'(l), \sigma', \varphi}}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Deref}}{\dfrac{e, \hat{\sigma} \hat{\downarrow} l, \hat{\sigma'}}{!e, \hat{\sigma} \hat{\downarrow} \hat{\sigma'}(l), \hat{\sigma'}}}$ with $\hat{\sigma} = M\sigma$, $M\sigma'(l) \equiv \hat{\sigma'}(l)$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} l, \hat{\sigma'} \wedge M_1 l \equiv l \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Note that since l cannot contain any symbols, $Ml \equiv l$ holds trivially.

Since M satisfies φ, we immediately obtain $e, M\sigma \hat{\downarrow} l, \hat{\sigma'}$, and $M\sigma' \equiv \hat{\sigma'}$.

Case $e = e_1 := e_2$

One rule applies, namely $\dfrac{\text{SE-Assign}}{\dfrac{e_1, \sigma \downarrow \overline{l, \sigma', \varphi_1} \qquad e_2, \sigma' \downarrow \overline{v_2, \sigma'', \varphi_2}}{e_1 := e_2, \sigma \downarrow \overline{\langle\rangle, \sigma''[l \mapsto v_2], \varphi_1 \wedge \varphi_2}}}$

Provided that $M\varphi_1 \wedge M\varphi_2$, we need to demonstrate that $\dfrac{\text{E-Assign}}{\dfrac{e_1, \hat{\sigma} \hat{\downarrow} l, \hat{\sigma'} \qquad e_2, \hat{\sigma'} \hat{\downarrow} \hat{v_2}, \hat{\sigma''}}{e_1 := e_2, \hat{\sigma} \hat{\downarrow} \langle\rangle, \hat{\sigma''}[l \mapsto \hat{v_2}]}}$ with $\hat{\sigma} = M\sigma$, $M\langle\rangle \equiv \langle\rangle$, which holds true trivially, and $M\sigma''[l \mapsto v_2] \equiv \hat{\sigma''}[l \mapsto \hat{v_2}]$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \hat{\downarrow} l, \hat{\sigma'} \wedge M_1 l \equiv l \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset e_2, M_2\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''} \wedge M_2 v_2 \equiv \hat{v_2} \wedge M_2\sigma' \equiv \hat{\sigma''}$

Since M satisfies both φ_1 and φ_2, and we know that $M\sigma' \equiv \hat{\sigma'}$, we obtain that $e_1, M\sigma \hat{\downarrow} l, \hat{\sigma'}$, $e_2, M\sigma' \hat{\downarrow} \hat{v_2}, \hat{\sigma''}$, $Ml \equiv l$, $Mv_2 \equiv \hat{v_2}$ and $M\sigma'' \equiv \hat{\sigma''}$ and therefore $M\sigma''[l \mapsto v_2] \equiv \hat{\sigma''}[l \mapsto \hat{v_2}]$.

Case $e = \square\, e$

One rule applies, namely $\dfrac{e, \sigma \downarrow \overline{v, \sigma', \varphi}}{\square\, e, \sigma \downarrow \overline{\square\, v, \sigma', \varphi}}$ SE-Edit

Provided that $M\varphi$, we need to demonstrate that $\dfrac{e, \hat{\sigma} \hat{\downarrow} \hat{v}, \hat{\sigma'}}{\square\, e, \hat{\sigma} \hat{\downarrow} \square\, \hat{v}, \hat{\sigma'}}$ E-Edit with $\hat{\sigma} = M\sigma$, $M \square\, v \equiv \square\, \hat{v}$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.

$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \hat{v}, \hat{\sigma'} \wedge M_1 v \equiv \hat{v} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we obtain that $e, M\sigma \hat{\downarrow} \hat{v}, \hat{\sigma'}$, $M \square\, v \equiv \square\, \hat{v}$. We can furthermore directly conclude that $\sigma' M \equiv \hat{\sigma'}$.

Case $e = \boxtimes\, \tau$

One rule applies, namely $\dfrac{}{\boxtimes\, \tau, \sigma \downarrow \boxtimes\, \tau, \sigma, \mathsf{True}}$ SE-Enter

Provided that $M\varphi$, we need to demonstrate that $\dfrac{}{\boxtimes\, \tau, \hat{\sigma} \hat{\downarrow} \boxtimes\, \tau, \hat{\sigma}}$ E-Enter with $\hat{\sigma} = M\sigma$, $M \boxtimes\, \tau \equiv \boxtimes\, \tau$, which holds trivially since types do not hold symbols, and $M\sigma \equiv \hat{\sigma}$, which also hols trivially from the premise.

Case $e = \blacksquare\, e$

One rule applies, namely $\dfrac{e, \sigma \downarrow \overline{l, \sigma', \varphi}}{\blacksquare\, e, \sigma \downarrow \overline{\blacksquare\, l, \sigma', \varphi}}$ SE-Update

Provided that $M\varphi$, we need to demonstrate that $\dfrac{e, \hat{\sigma} \hat{\downarrow} l, \hat{\sigma'}}{\blacksquare\, e, \hat{\sigma} \hat{\downarrow} \blacksquare\, l, \hat{\sigma'}}$ E-Update with $\hat{\sigma} = M\sigma$, $M \blacksquare\, l \equiv \blacksquare\, l$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.

$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} l, \hat{\sigma'} \wedge M_1 l \equiv l \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we obtain that $e, M\sigma \hat{\downarrow} l, \hat{\sigma'}$, and $M \blacksquare\, l \equiv \blacksquare\, l$. We can furthermore directly conclude that $M\sigma' \equiv \hat{\sigma'}$.

Case $e = e_1 \blacktriangleright e_2$

One rule applies, namely $\dfrac{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi}}{e_1 \blacktriangleright e_2, \sigma \downarrow \overline{t_1 \blacktriangleright e_2, \sigma', \varphi}}$ SE-Then

Provided that $M\varphi$, we need to demonstrate that $\dfrac{e_1, \hat{\sigma} \hat{\downarrow} \hat{t_1}, \hat{\sigma'}}{e_1 \blacktriangleright e_2, \hat{\sigma} \hat{\downarrow} \hat{t_1} \blacktriangleright e_2, \hat{\sigma'}}$ E-Then with $\hat{\sigma} = M\sigma$, $Mt_1 \blacktriangleright e_2 \equiv \hat{t_1} \blacktriangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.

$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \hat{t_1}, \hat{\sigma'} \quad M_1 t_1 \equiv \hat{t_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we obtain that $e, M\sigma \hat{\downarrow} \hat{t_1}, \hat{\sigma'}$ and $Mt_1 \blacktriangleright e_2 \equiv \hat{t_1} \blacktriangleright e_2$. We can furthermore directly conclude that $M\sigma' \equiv \hat{\sigma'}$.

Case $e = e_1 \triangleright e_2$

One rule applies, namely $\dfrac{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi}}{e_1 \triangleright e_2, \sigma \downarrow \overline{t_1 \triangleright e_2, \sigma', \varphi}}$ SE-Next

Provided that $M\varphi$, we need to demonstrate that $\dfrac{e_1, \hat{\sigma} \hat{\downarrow} \hat{t_1}, \hat{\sigma'}}{e_1 \triangleright e_2, \hat{\sigma} \hat{\downarrow} \hat{t_1} \triangleright e_2, \hat{\sigma'}}$ E-Next with $\hat{\sigma} = M\sigma$, $Mt_1 \triangleright e_2 \equiv \hat{t_1} \blacktriangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.

$\forall M_1.M_1\varphi \supset e, M_1\sigma \hat{\downarrow} \hat{t_1}, \hat{\sigma'} \wedge M_1 t_1 \equiv \hat{t_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we obtain that $e, M\sigma \mathbin{\hat{\downarrow}} \hat{t_1}, \hat{\sigma'}$ and $Mt_1 \rhd e_2 \equiv \hat{t_1} \rhd e_2$. We can furthermore directly conclude that $M\sigma' \equiv \hat{\sigma'}$.

Case $e = e_1 \blacklozenge e_2$

One rule applies, namely $\dfrac{\text{SE-Or}}{e_1, \sigma \downarrow \overline{t_1, \sigma', \varphi_1} \quad e_2, \sigma' \downarrow \overline{t_2, \sigma'', \varphi_2}}{e_1 \blacklozenge e_2, \sigma \downarrow \overline{t_1 \blacklozenge t_2, \sigma'', \varphi_1 \wedge \varphi_2}}$

Provided that $M\varphi_1 \wedge M\varphi_2$, we need to demonstrate that $\dfrac{\text{E-Or}}{e_1, \hat{\sigma} \mathbin{\hat{\downarrow}} \hat{t_1}, \hat{\sigma'} \quad e_2, \hat{\sigma'} \mathbin{\hat{\downarrow}} \hat{t_2}, \hat{\sigma''}}{e_1 \blacklozenge e_2, \hat{\sigma} \mathbin{\hat{\downarrow}} \hat{t_1} \blacklozenge \hat{t_2}, \hat{\sigma''}}$ with $\hat{\sigma} = M\sigma$, $Mt_1 \blacklozenge t_2 \equiv \hat{t_1} \blacklozenge \hat{t_2}$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset e_1, M_1\sigma \mathbin{\hat{\downarrow}} \hat{t_1}, \hat{\sigma'} \wedge M_1 t_1 \equiv \hat{t_1} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset e_2, M_2\sigma \mathbin{\hat{\downarrow}} \hat{t_1}, \hat{\sigma'} \wedge M_2 t_2 \equiv \hat{t_2} \wedge M_2\sigma' \equiv \hat{\sigma'}$

Since M satisfies both φ_1 and φ_2, and we know that $M\sigma' \equiv \hat{\sigma'}$, we obtain that $e_1, M\sigma \mathbin{\hat{\downarrow}} \hat{t_1}, \hat{\sigma'}$, $e_2, M\sigma' \mathbin{\hat{\downarrow}} \hat{t_2}, \hat{\sigma}$, $Mt_1 \equiv \hat{t_1}$ and $Mt_2 \equiv \hat{t_2}$ and therefore $Mt_1 \blacklozenge t_2 \equiv \hat{t_1} \blacklozenge \hat{t_2}$. From the IH we directly obtain that $M\sigma'' \equiv \hat{\sigma''}$.

Case $e = e_1 \lozenge e_2$

One rule applies, namely $\dfrac{\text{SE-Xor}}{e_1 \lozenge e_2, \sigma \downarrow e_1 \lozenge e_2, \sigma, \mathsf{True}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Xor}}{e_1 \lozenge e_2, \hat{\sigma} \mathbin{\hat{\downarrow}} e_1 \lozenge e_2, \hat{\sigma}}$ with $\hat{\sigma} = M\sigma$, $Me_1 \lozenge e_2 \equiv \hat{e_1} \lozenge \hat{e_2}$, which holds trivially, and $M\sigma \equiv \hat{\sigma}$, which also holds trivially from the premise.

Case $e = \lightning$

One rule applies, namely $\dfrac{\text{SE-Fail}}{\lightning, \sigma \downarrow \lightning, \sigma, \mathsf{True}}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{\text{E-Fail}}{\lightning, \hat{\sigma} \mathbin{\hat{\downarrow}} \lightning, \hat{\sigma}}$ with $\hat{\sigma} = M\sigma$, $M \lightning \equiv \lightning$, which holds trivially since fail do not hold symbols, and $M\sigma \equiv \hat{\sigma}$, which also hols trivially from the premise.

□

Proof of Lemma 6.4. We prove Lemma 6.4 by induction over t.

Case $t = t_1 \blacktriangleright e_2$

Three rules apply.

Case $\dfrac{\text{SS-ThenStay}}{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{t_1' \blacktriangleright e_2, \sigma', \varphi}}\ \mathcal{V}(t_1', \sigma') = \bot$

Provided that $M\varphi \equiv \mathsf{True}$ we need to demonstrate that $\dfrac{\text{S-ThenStay}}{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma'}}{t_1 \blacktriangleright e_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}' \blacktriangleright e_2, \hat{\sigma'}}\ \mathcal{V}(\hat{t_1}', \hat{\sigma'}) = \bot$ with $\hat{\sigma} = M\sigma$, $Mt_1' \blacktriangleright e_2 \equiv \hat{t_1'} \blacktriangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$ and $Mt_1' \equiv \hat{t_1'}$, and therefore also $Mt_1' \blacktriangleright e_2 \equiv \hat{t_1'} \blacktriangleright e_2$, and from the induction hypothesis, we directly obtain $M\sigma' \equiv \hat{\sigma'}$.

Case $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi} \qquad e_2\ v_1, \sigma' \downarrow \overline{t_2, \sigma'', \_}}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{t_1' \blacktriangleright e_2, \sigma', \varphi}}$ SS-ThenFail $\mathcal{V}(t_1', \sigma') = v_1 \wedge \mathcal{F}(t_2, \sigma'')$

Provided that $M\varphi \equiv$ True we need to demonstrate that $\dfrac{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma}' \qquad e_2\ \hat{v_1}, \hat{\sigma}' \mathrel{\hat{\downarrow}} \hat{t_2}, \hat{\sigma}''}{t_1 \blacktriangleright e_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}' \blacktriangleright e_2, \hat{\sigma}'}$ S-ThenFail $\mathcal{V}(\hat{t_1}', \hat{\sigma}') = \hat{v_1} \wedge \mathcal{F}(\hat{t_2}, \hat{\sigma}'')$ with $\hat{\sigma} = M\sigma$, $Mt_1' \blacktriangleright e_2 \equiv \hat{t_1'} \blacktriangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$ and $Mt_1' \equiv \hat{t_1'}$, and therefore also $Mt_1' \blacktriangleright e_2 \equiv \hat{t_1'} \blacktriangleright e_2$, and from the induction hypothesis, we directly obtain $M\sigma' \equiv \hat{\sigma'}$.

Case $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \qquad e_2\ v_1, \sigma' \downarrow \overline{t_2, \sigma'', \varphi_2}}{t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{t_2, \sigma'', \varphi_1 \wedge \varphi_2}}$ SS-ThenCont $\mathcal{V}(t_1', \sigma') = v_1 \wedge \neg\mathcal{F}(t_2, \sigma'')$

Provided that $M\varphi_1 \wedge M\varphi_2$ we need to demonstrate that $\dfrac{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma}' \qquad e_2\ \hat{v_1}, \hat{\sigma}' \mathrel{\hat{\downarrow}} \hat{t_2}, \hat{\sigma}''}{t_1 \blacktriangleright e_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_2}, \hat{\sigma}''}$ S-ThenCont $\mathcal{V}(\hat{t_1}', \hat{\sigma}') = \hat{v_1} \wedge \neg\mathcal{F}(\hat{t_2}, \hat{\sigma}'')$ with $\hat{\sigma} = M\sigma$, $Mt_2 \equiv \hat{t_2} \blacktriangleright e_2$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \supset M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$.
From Lemma 6.5 we know that
$\forall M_2.M_2\varphi_2 \supset e_2\hat{v_1}M_2\sigma' \mathrel{\hat{\downarrow}} \hat{t_2}, \hat{\sigma''} \quad M_2 t_2 \equiv \hat{t_2} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.

Since M satisfies both φ_1 and φ_2, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$ and $e_2\hat{v_1}M\sigma' \mathrel{\hat{\downarrow}} \hat{t_2}, \hat{\sigma''}$, $Mt_2 \equiv \hat{t_2}$, and from the induction hypothesis, we directly obtain $M\sigma'' \equiv \hat{\sigma''}$.

Case $t = t_1 \blacklozenge t_2$

Three rules apply.

Case $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}$ SS-OrLeft $\mathcal{V}(t_1', \sigma') = v_1$

Provided that $M\varphi \equiv$ True we need to demonstrate that $\dfrac{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma}'}{t_1 \blacklozenge t_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma}'}$ S-OrLeft $\mathcal{V}(\hat{t_1}', \hat{\sigma}') = \hat{v_1}$ with $\hat{\sigma} = M\sigma$, $Mt_1' \equiv \hat{t_1'}$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \quad M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$, $Mt_1' \equiv \hat{t_1'}$ and $M\sigma' \equiv \hat{\sigma'}$.

Case $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \qquad t_2, \sigma' \rightsquigarrow \overline{t_2', \sigma'', \varphi_2}}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{t_2', \sigma'', \varphi_1 \wedge \varphi_2}}$ SS-OrRight $\mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = v_2$

Provided that $M\varphi_1 \wedge M\varphi_2$ we need to deomstrate that $\dfrac{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma}' \qquad t_2, \hat{\sigma}' \mathrel{\hat{\rightsquigarrow}} \hat{t_2}', \hat{\sigma}''}{t_1 \blacklozenge t_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_2}', \hat{\sigma}''}$ S-OrRight $\mathcal{V}(\hat{t_1}', \hat{\sigma}') = \bot \wedge \mathcal{V}(\hat{t_2}', \hat{\sigma}'') = \hat{v_2}$ with $\hat{\sigma} = M\sigma$, $Mt_2' \equiv \hat{t_2'}$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset t_2, M_2\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t_2'}, \hat{\sigma''} \wedge M_2 t_2' \equiv \hat{t_2'} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.

Since M satisfies both φ_1 and φ_2, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$ and $t_2, M\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t_2'}, \hat{\sigma''}$, $Mt_2' \equiv \hat{t_2'}$ and $M\sigma'' \equiv \hat{\sigma''}$.

Case $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \qquad t_2, \sigma' \rightsquigarrow \overline{t_2', \sigma'', \varphi_2}}{t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{t_1' \blacklozenge t_2', \sigma'', \varphi_1 \wedge \varphi_2}}$ SS-OrNone $\mathcal{V}(t_1', \sigma') = \bot \wedge \mathcal{V}(t_2', \sigma'') = \bot$

Provided that $M\varphi_1 \wedge M\varphi_2$ we need to demonstrate that $\dfrac{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma}' \qquad t_2, \hat{\sigma'} \mathrel{\hat{\rightsquigarrow}} \hat{t_2}', \hat{\sigma}''}{t_1 \blacklozenge t_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}' \blacklozenge \hat{t_2}', \hat{\sigma}''}$ S-OrNone $\mathcal{V}(\hat{t_1}', \hat{\sigma}') = \bot \wedge \mathcal{V}(\hat{t_2}', \hat{\sigma}'') = \bot$ with $\hat{\sigma} = M\sigma$, $Mt_1' \blacklozenge t_2' \equiv \hat{t_1'} \blacklozenge \hat{t_2'}$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset t_2, M_2\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t_2'}, \hat{\sigma''} \wedge M_2 t_2' \equiv \hat{t_2'} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.

Since M satisfies both φ_1 and φ_2, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$ and $t_2, M\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t_2'}, \hat{\sigma''}$, $Mt_1' \blacklozenge t_2' \equiv \hat{t_1'} \blacklozenge \hat{t_2'}$ and $M\sigma'' \equiv \hat{\sigma''}$.

Case $t = \square\, v$

One rule applies, namely $\dfrac{}{\square\, v, \sigma \rightsquigarrow \square\, v, \sigma, \mathsf{True}}$ SS-Edit

Provided that M True, we need to demonstrate that $\dfrac{}{\square\, v, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \square\, v, \hat{\sigma}}$ S-Edit with $\hat{\sigma} = M\sigma$, $M \square\, v \equiv \square\, \hat{v}$ and $M\sigma \equiv \hat{\sigma}$. This holds trivially.

Case $t = \boxtimes\, \tau$

One rule applies, namely $\dfrac{}{\boxtimes\, \tau, \sigma \rightsquigarrow \boxtimes\, \tau, \sigma, \mathsf{True}}$ SS-Fill

Provided that M True, we need to demonstrate $\dfrac{}{\boxtimes\, \tau, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \boxtimes\, \tau, \hat{\sigma}}$ S-Fill with $\hat{\sigma} = M\sigma$, $M \boxtimes\, \tau \equiv \boxtimes\, \tau$ and $M\sigma \equiv \hat{\sigma}$. This holds trivially.

Case $t = \blacksquare\, l$

One rule applies, namely $\dfrac{}{\blacksquare\, l, \sigma \rightsquigarrow \blacksquare\, l, \sigma, \mathsf{True}}$ SS-Update

Provided that M True, we need to demonstrate $\dfrac{}{\blacksquare\, l, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \blacksquare\, l, \hat{\sigma}}$ S-Update with $\hat{\sigma} = M\sigma$, $M \blacksquare\, l \equiv \blacksquare\, l$ and $M\sigma \equiv \hat{\sigma}$. This holds trivially.

Case $t = \lightning$

One rule applies, namely $\dfrac{}{\lightning, \sigma \rightsquigarrow \lightning, \sigma, \mathsf{True}}$ SS-Fail

Provided that M True, we need to demonstrate that $\dfrac{}{\lightning, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \lightning, \hat{\sigma}}$ S-Fail with $\hat{\sigma} = M\sigma$, $M \lightning \equiv \lightning$ and $M\sigma \equiv \hat{\sigma}$. This holds trivially.

Case $t = e_1 \lozenge e_2$

One rule applies, namely $\dfrac{}{e_1 \lozenge e_2, \sigma \rightsquigarrow e_1 \lozenge e_2, \sigma, \mathsf{True}}$ SS-Xor

Provided that M True, we need to demonstrate that $\dfrac{}{e_1 \lozenge e_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} e_1 \lozenge e_2, \hat{\sigma}}$ S-Xor with $\hat{\sigma} = M\sigma$, $Me_1 \lozenge e_2 \equiv e_1 \lozenge e_2$ and $M\sigma \equiv \hat{\sigma}$. This holds trivially.

Case $t = t_1 \rhd e_2$

One rule applies, namely $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi}}{t_1 \rhd e_2, \sigma \rightsquigarrow \overline{t_1' \rhd e_2, \sigma', \varphi}}$ SS-Next

Provided that $M\varphi$, we need to demonstrate that $\dfrac{}{e_1 \lozenge e_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} e_1 \lozenge e_2, \hat{\sigma}}$ S-Xor with $\hat{\sigma} = M\sigma$, $Mt_1' \rhd e_2 \equiv \hat{t_1'} \rhd e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$.
Since M satisfies φ, we directly obtain $t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}, Mt_1' \triangleright e_2 \equiv \hat{t_1'} \triangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

Case $t = t_1 \bowtie t_2$

One rule applies, namely $\dfrac{t_1, \sigma \rightsquigarrow \overline{t_1', \sigma', \varphi_1} \qquad t_2, \sigma' \rightsquigarrow \overline{t_2', \sigma'', \varphi_2}}{t_1 \bowtie t_2, \sigma \rightsquigarrow \overline{t_1' \bowtie t_2', \sigma'', \varphi_1 \wedge \varphi_2}}$ SS-And

Provided that $M\varphi_1 \wedge M\varphi_2$ we need to demonstrate $\dfrac{t_1, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}', \hat{\sigma'} \qquad t_2, \hat{\sigma'} \mathrel{\hat{\rightsquigarrow}} \hat{t_2}', \hat{\sigma''}}{t_1 \bowtie t_2, \hat{\sigma} \mathrel{\hat{\rightsquigarrow}} \hat{t_1}' \bowtie \hat{t_2}', \hat{\sigma''}}$ S-And with $\hat{\sigma} = M\sigma, Mt_1' \bowtie t_2' \equiv \hat{t_1'} \bowtie \hat{t_2'}$ and $M\sigma'' \equiv \hat{\sigma''}$.

From the induction hypothesis, we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'} \quad M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset t_2, M_2\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t_2'}, \hat{\sigma''} \quad M_2 t_2' \equiv \hat{t_2'} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.
Since M satisfies both φ_1 and φ_2, we know that $t_1, M\sigma \mathrel{\hat{\rightsquigarrow}} \hat{t_1'}, \hat{\sigma'}$ and $t_2, M\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t_2'}, \hat{\sigma''}$, $Mt_1' \blacklozenge t_2' \equiv \hat{t_1'} \blacklozenge \hat{t_2'}$ and $M\sigma'' \equiv \hat{\sigma''}$.

□

Proof of Lemma 6.3. We prove Lemma 6.3 by induction over e.
The base case is when the SN-Done rule applies.

$$\frac{e, \sigma \downarrow \overline{t, \sigma', \varphi_1} \qquad t, \sigma' \rightsquigarrow \overline{t', \sigma'', \varphi_2}}{e, \sigma \Downarrow \overline{t, \sigma', \varphi_1}}\ \sigma' = \sigma'' \wedge t = t' \quad \text{SN-Done}$$

Provided that $M\varphi_1 \wedge M\varphi_2$
we need to demonstrate that $\dfrac{e, \hat{\sigma} \hat{\downarrow} \hat{t}, \hat{\sigma'} \qquad \hat{t}, \hat{\sigma'} \mathrel{\hat{\rightsquigarrow}} \hat{t'}, \hat{\sigma''}}{e, \hat{\sigma} \hat{\Downarrow} \hat{t}, \hat{\sigma'}}\ \hat{\sigma'} = \hat{\sigma''} \wedge \hat{t} = \hat{t'}$ N-Done with $\hat{\sigma} = M\sigma, Mt \equiv \hat{t}$ and $M\sigma' \equiv \hat{\sigma'}$.
By Lemma 6.5 and Lemma 6.4, we know that
$\forall M_1.M_1\varphi_1 \supset e, M_1\sigma \hat{\downarrow} \hat{t}, \hat{\sigma'} \wedge M_1 t \equiv \hat{t} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset t, M_2\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t'}, \hat{\sigma''} \wedge M_2 t' \equiv \hat{t'} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.
We assume M to satisfy both φ_1 and φ_2, we have $e, M\sigma \hat{\downarrow} \hat{t}, \hat{\sigma'}$ since $M\sigma \equiv \hat{\sigma}$.
The only induction step is when

$$\frac{e, \sigma \downarrow \overline{t, \sigma', \varphi_1} \qquad t, \sigma' \rightsquigarrow \overline{t', \sigma'', \varphi_2} \quad t', \sigma'' \Downarrow \overline{t'', \sigma''', \varphi_3}}{e, \sigma \Downarrow \overline{t'', \sigma''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}\ \sigma' \neq \sigma'' \vee t \neq t' \quad \text{SN-Repeat}$$

applies. In this case, where we have that $M\varphi_1 \wedge M\varphi_2 \wedge M\varphi_3$, we need to demonstrate that $\dfrac{e, \hat{\sigma} \hat{\downarrow} \hat{t}, \hat{\sigma'} \qquad \hat{t}, \hat{\sigma'} \mathrel{\hat{\rightsquigarrow}} \hat{t'}, \hat{\sigma''} \qquad \hat{t'}, \hat{\sigma''} \hat{\Downarrow} \hat{t''}, \hat{\sigma'''}}{e, \hat{\sigma} \hat{\Downarrow} \hat{t''}, \hat{\sigma'''}}\ \hat{\sigma'} \neq \hat{\sigma''} \vee \hat{t} \neq \hat{t'}$ N-Repeat with $\hat{\sigma} = M\sigma, Mt'' \equiv \hat{t''}$ and $M\sigma''' \equiv \hat{\sigma'''}$.
Again by Lemma 6.5 and Lemma 6.4, we know that
$\forall M_1.M_1\varphi_1 \supset e, M_1\sigma \hat{\downarrow} \hat{t}, \hat{\sigma'} \wedge M_1 t \equiv \hat{t} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset t, M_2\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t'}, \hat{\sigma''} \wedge M_2 t' \equiv \hat{t'} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.
Furthermore, we know by applying the induction hypothesis that $\forall M_3.M_3\varphi_3 \supset t', M_3\sigma'' \hat{\Downarrow} \hat{t''}, \hat{\sigma'''} \wedge M_3 t'' \equiv \hat{t''} \wedge M_3\sigma''' \equiv \hat{\sigma'''}$.
Since M satisfies φ_1, φ_2 and φ_3, we can conclude that
$e, M\sigma \hat{\downarrow} \hat{t}, \hat{\sigma'}, Mt \equiv \hat{t} \wedge M\sigma' \equiv \hat{\sigma'}$ and $t, M\sigma' \mathrel{\hat{\rightsquigarrow}} \hat{t'}, \hat{\sigma''}$ and $Mt' \equiv \hat{t'} \wedge M\sigma'' \equiv \hat{\sigma''}$. This finally gives us $t', M\sigma'' \hat{\Downarrow} \hat{t''}, \hat{\sigma'''}$ from which we can conclude that which we needed to prove, namely $Mt'' \equiv \hat{t''} \wedge M\sigma''' \equiv \hat{\sigma'''}$. □

Proof of Lemma 6.2. We prove Lemma 6.2 by induction over t.

Case $t = \square\, v$

One rule applies, namely $\dfrac{\text{SH-Change}\quad \text{fresh } s}{\square\, v, \sigma \;\to\; \square\, s, \sigma, \boxed{s, \mathsf{True}}}\; v, s : \tau$

Provided that M True we need to demonstrate that $\dfrac{\text{H-Change}}{\square\, v, \hat{\sigma} \xrightarrow{v}{}' \square\, v', \hat{\sigma}}\; v, v' : \tau$ with $\hat{\sigma} = M\sigma$ and $Ms = v'$, $M \square\, s \equiv \square\, v'$ and $M\sigma \equiv \hat{\sigma}$.

This follows trivially from the premise.

Case $t = \boxtimes\, \tau$

One rule applies, namely $\dfrac{\text{SH-Fill}\quad \text{fresh } s}{\boxtimes\, \tau, \sigma \;\to\; \square\, s, \sigma, \boxed{s, \mathsf{True}}}\; s : \tau$

Provided that M True we need to demonstrate that $\dfrac{\text{H-Fill}}{\boxtimes\, \tau, \hat{\sigma} \xrightarrow{v} \square\, v, \hat{\sigma}}\; v : \tau$ with $\hat{\sigma} = M\sigma$ and $Ms = v$, $M \square\, s \equiv \square\, v$ and $M\sigma \equiv \hat{\sigma}$.

This follows trivially from the premise.

Case $t = \blacksquare\, l$

One rule applies, namely $\dfrac{\text{SH-Update}\quad \text{fresh } s}{\blacksquare\, l, \sigma \;\to\; \blacksquare\, l, \sigma[l \mapsto s], \boxed{s, \mathsf{True}}}\; \sigma(l), s : \tau$

Provided that M True we need to demonstrate that $\dfrac{\text{H-Update}}{\blacksquare\, l, \hat{\sigma} \xrightarrow{v} \blacksquare\, l, \hat{\sigma}[l \mapsto v]}\; \sigma(l), v : \tau$ with $\hat{\sigma} = M\sigma$ and $Ms = v$, $M \blacksquare\, l \equiv \blacksquare\, l$ and $M\sigma[l \mapsto s] \equiv \hat{\sigma}[l \mapsto v]$.

$\dfrac{\text{H-Update}}{\blacksquare\, l, \hat{\sigma} \xrightarrow{v} \blacksquare\, l, \hat{\sigma}[l \mapsto v]}\; \sigma(l), v : \tau$ with $\hat{\sigma} = M\sigma$ follows trivially. $M \blacksquare\, l \equiv \blacksquare\, l$ follows trivially, since locations cannot contain symbols. $M\sigma[l \mapsto s] \equiv \hat{\sigma}[l \mapsto v]$ can be concluded from the fact that $\hat{\sigma} = M\sigma$ and $Ms = v$.

Case $t = t_1 \triangleright e_2$

In this case, two rules apply.

Case $\dfrac{\text{SH-Next}\quad t_1, \sigma \;\to\; \overline{t_1', \sigma_1', \boxed{i_1, \varphi_1}} \qquad e_2\, v_1, \sigma \;\Downarrow\; \overline{t_2, \sigma_2', \boxed{\varphi_2}}}{t_1 \triangleright e_2, \sigma \;\to\; \boxed{\overline{t_1' \triangleright e_2, \sigma_1', i_1, \varphi_1} \cup \overline{t_2, \sigma_2', \mathsf{C}, \varphi_2}}}\; \mathcal{V}(t_1, \sigma) = v_1 \wedge \neg \mathcal{F}(t_2, \sigma')$

In the case $M\varphi_1$, we need to demonstrate that $\dfrac{\text{H-PassNext}\quad t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \triangleright e_2, \sigma \xrightarrow{j} \hat{t_1'} \triangleright e_2, \sigma'}$ with $\hat{\sigma} = M\sigma$ and $j = Mi$, $Mt_1' \triangleright e_2 \equiv \hat{t_1'} \triangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.

$\forall M_1 . M_1\varphi_1 \supset t_1, M_1\sigma \xrightarrow{M_1 i} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we have $t_1, M\sigma \xrightarrow{Mi} \hat{t_1'}, \hat{\sigma'}$, $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1' \triangleright e_2 \equiv \hat{t_1'} \triangleright e_2$ since this can be concluded from $Mt_1' \equiv \hat{t_1'}$.

In the case $M\varphi_2$, we need to demonstrate that $\dfrac{\text{H-Next}\quad e_2\, \hat{v_1}, \sigma \Downarrow \hat{t_2}, \hat{\sigma'}}{t_1 \triangleright e_2, \sigma \xrightarrow{\mathsf{C}} \hat{t_2}, \hat{\sigma'}}\; \mathcal{V}(t_1, \sigma) = \hat{v_1} \wedge \neg \mathcal{F}(\hat{t_2}, \hat{\sigma'})$ with $\hat{\sigma} = M\sigma$, $Mt_2 \equiv \hat{t_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

From Lemma 6.3 we obtain that $\forall M_1 . M_1\varphi \supset e_2 v_1, M\sigma \Downarrow \hat{t_2}, \hat{\sigma'} \wedge Mt_2 \equiv \hat{t_2} \wedge M\sigma' \equiv \hat{\sigma'}$.

This gives us exactly what we needed to prove this case.

Case $\dfrac{t_1, \sigma \;\to\; \overline{t_1', \sigma', \boxed{i, \varphi}}}{\overline{t_1 \rhd e_2, \sigma \;\to\; t_1' \rhd e_2, \sigma', \boxed{i, \varphi}}}\;\text{SH-PassNext}\;\mathcal{V}(t_1, \sigma) = \bot$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \rhd e_2, \sigma \xrightarrow{j} \hat{t_1'} \rhd e_2, \sigma'}\;\text{H-PassNext}$ with $\hat{\sigma} = M\sigma$ and $j = Mi$, $Mt_1' \rhd e_2 \equiv \hat{t_1'} \rhd e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.

$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \xrightarrow{M_1 i} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we have $t_1, M\sigma \xrightarrow{Mi} \hat{t_1'}, \hat{\sigma'}$, $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1' \rhd e_2 \equiv \hat{t_1'} \rhd e_2$ since this can be concluded from $Mt_1' \equiv \hat{t_1'}$.

Case $t = t_1 \blacktriangleright e_2$

One rule applies, namely $\dfrac{t_1, \sigma \;\to\; \overline{t_1', \sigma', \boxed{i, \varphi}}}{\overline{t_1 \blacktriangleright e_2, \sigma \;\to\; t_1' \blacktriangleright e_2, \sigma', \boxed{i, \varphi}}}\;\text{SH-PassThen}$

Provided that $M\varphi$, we need to demonstrate that $\dfrac{t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{j} \hat{t_1'} \blacktriangleright e_2, \sigma'}\;\text{H-PassThen}$ with $\hat{\sigma} = M\sigma$ and $j = Mi$, $Mt_1' \blacktriangleright e_2 \equiv \hat{t_1'} \blacktriangleright e_2$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.

$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \xrightarrow{M_1 i} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we have $t_1, M\sigma \xrightarrow{Mi} \hat{t_1'}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1' \blacktriangleright e_2 \equiv \hat{t_1'} \blacktriangleright e_2$ since this can be concluded from $Mt_1' \equiv \hat{t_1'}$.

Case $t = e_1 \Diamond e_2$

In this case, three rules apply.

Case $\dfrac{e_1, \sigma \Downarrow \overline{t_1, \boxed{\sigma_1, \varphi_1}} \qquad e_2, \sigma \Downarrow \overline{t_2, \boxed{\sigma_2, \varphi_2}}}{e_1 \Diamond e_2, \sigma \;\to\; \boxed{\overline{t_1, \sigma_1, \mathsf{L}, \varphi_1} \cup \overline{t_2, \sigma_2, \mathsf{R}, \varphi_2}}}\;\text{SH-Pick}\;\neg\mathcal{F}(t_1, \sigma_1) \wedge \neg\mathcal{F}(t_2, \sigma_2)$

Either we have that $M(\varphi_1 \wedge s = \mathsf{L})$ or $M(\varphi_2 \wedge s = \mathsf{R})$. In the first case, the proof is identical to the SH-PickLeft rule. In the second cse, the proof is identical to the SH-PickRight rule.

Case $\dfrac{e_1, \sigma \Downarrow \overline{t_1, \boxed{\sigma_1, \varphi_1}} \qquad e_2, \sigma \Downarrow \overline{t_2, \boxed{\sigma_2, \varphi_2}}}{e_1 \Diamond e_2, \sigma \;\to\; t_1, \sigma_1, \boxed{\mathsf{L}, \varphi_1}}\;\text{SH-PickLeft}\;\neg\mathcal{F}(t_1, \sigma_1) \wedge \mathcal{F}(t_2, \sigma_2)$

Provided that $M(\varphi_2 \wedge s = \mathsf{L})$, we need to demonstrate that $\dfrac{e_1, \sigma \Downarrow \hat{t_1}, \hat{\sigma'}}{e_1 \Diamond e_2, \sigma \xrightarrow{\mathsf{L}} \hat{t_1}, \hat{\sigma'}}\;\text{H-PickLeft}\;\neg\mathcal{F}(\hat{t_1}, \hat{\sigma'})$ with $\hat{\sigma} = M\sigma$, $Mt_1 \equiv \hat{t_1}$ and $M\sigma' \equiv \hat{\sigma'}$.

From Lemma 6.3 we obtain that $\forall M_1.M_1\varphi \supset e_1, M\sigma \Downarrow \hat{t_1}, \hat{\sigma'} \wedge Mt_1 \equiv \hat{t_1} \wedge M\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we have $e_1, M\sigma \Downarrow [\hat{t_1}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1 \equiv \hat{t_1}$.

Case $\dfrac{e_1, \sigma \Downarrow \overline{t_1, \boxed{\sigma_1, \varphi_1}} \qquad e_2, \sigma \Downarrow \overline{t_2, \boxed{\sigma_2, \varphi_2}}}{e_1 \Diamond e_2, \sigma \;\to\; t_2, \sigma_2, \boxed{\mathsf{R}, \varphi_2}}\;\text{SH-PickRight}\;\mathcal{F}(t_1, \sigma_1) \wedge \neg\mathcal{F}(t_2, \sigma_2)$

Provided that $M(\varphi_2 \wedge s = \mathsf{R})$ we need to demonstrate that $\dfrac{e_2, \sigma \Downarrow \hat{t_2}, \hat{\sigma'}}{e_1 \Diamond e_2, \sigma \xrightarrow{\mathsf{R}} \hat{t_2}, \hat{\sigma'}}\;\text{H-PickRight}\;\neg\mathcal{F}(\hat{t_2}, \hat{\sigma'})$ with $\hat{\sigma} = M\sigma$, $Mt_2 \equiv \hat{t_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

From Lemma 6.3 we obtain that $\forall M_1.M_1\varphi \supset e_2, M\sigma \Downarrow \hat{t_2}, \hat{\sigma'} \wedge Mt_2 \equiv \hat{t_2} \wedge M\sigma' \equiv \hat{\sigma'}$.
Since M satisfies φ, we have $e_2, M\sigma \Downarrow []\hat{t_2}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_2 \equiv \hat{t_2}$.

Case $t = t_1 \bowtie t_2$

In this case, two rules apply.
Case

Provided that $M\varphi$, we need to demonstrate that $\dfrac{t_1, \sigma \xrightarrow{j} \hat{t_1}', \hat{\sigma'}}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{F}\, j} \hat{t_1}' \bowtie t_2, \hat{\sigma'}}$ H-FirstAnd with $\hat{\sigma} = M\sigma, Mt_1' \bowtie t_2 \equiv \hat{t_1'} \bowtie t_2$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \xrightarrow{M_1 i} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we have $t_1, M\sigma \xrightarrow{Mi} \hat{t_1'}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1' \bowtie t_2 \equiv \hat{t_1'} \bowtie t_2$, which follows from $Mt_1' \equiv \hat{t_1'}$.

Case

Provided that $M\varphi$, we need to demonstrate that $\dfrac{t_2, \sigma \xrightarrow{j} \hat{t_2}', \hat{\sigma'}}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{S}\, j} t_1 \bowtie \hat{t_2}', \hat{\sigma'}}$ H-SecondAnd with $\hat{\sigma} = M\sigma, Mt_1 \bowtie t_2' \equiv t_1 \bowtie \hat{t_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_2, M_1\sigma \xrightarrow{M_1 i} \hat{t_2'}, \hat{\sigma'} \wedge M_1 t_2' \equiv \hat{t_2'} \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we have $t_2, M\sigma \xrightarrow{Mi} \hat{t_2'}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1 \bowtie t_2' \equiv t_1 \bowtie \hat{t_2'}$, which follows from $Mt_2' \equiv \hat{t_2'}$.

Case $t = e_1 \blacklozenge e_2$

One rule applies, namely $\dfrac{t_1, \sigma \to \overline{t_1', \sigma_1', i_1, \varphi_1} \qquad t_2, \sigma \to \overline{t_2', \sigma_2', i_2, \varphi_2}}{t_1 \blacklozenge t_2, \sigma \to \overline{t_1' \blacklozenge t_2, \sigma_1', \mathsf{F}\, i_!, \varphi_1} \cup \overline{t_1 \blacklozenge t_2', \sigma_2', \mathsf{S}\, i_2, \varphi_2}}$ SH-Or

In the case that $M\varphi_1$, we need to demonstrate that $\dfrac{t_1, \sigma \xrightarrow{j} \hat{t_1}', \hat{\sigma'}}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{F}\, j} \hat{t_1}' \blacklozenge t_2, \hat{\sigma'}}$ H-FirstOr with $\hat{\sigma} = M\sigma$ and $M\,\mathsf{F}\, i = \mathsf{F}\, j$, $Mt_1' \blacklozenge t_2 \equiv \hat{t_1'} \bowtie t_2$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \xrightarrow{M_1 i} \hat{t_1'}, \hat{\sigma'} \wedge M_1 t_1' \equiv \hat{t_1'} \wedge M_1\sigma' \equiv \hat{\sigma'}$.

Since M satisfies φ, we have $t_1, M\sigma \xrightarrow{Mi} \hat{t_1'}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1' \blacklozenge t_2 \equiv \hat{t_1'} \bowtie t_2$, which follows from $Mt_1' \equiv \hat{t_1'}$.

In the case that $M\varphi_2$, we need to demonstrate that $\dfrac{t_2, \sigma \xrightarrow{j} \hat{t_2}', \hat{\sigma'}}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{S}\, j} t_1 \blacklozenge \hat{t_2}', \hat{\sigma'}}$ H-SecondOr with $\hat{\sigma} = M\sigma$ and $M\,\mathsf{S}\, i = \mathsf{S}\, j$, $Mt_1 \blacklozenge t_2' \equiv t_1 \bowtie \hat{t_2}$ and $M\sigma' \equiv \hat{\sigma'}$.

By the induction hypothesis we obtain the following.
$\forall M_1.M_1\varphi_1 \supset t_2, M_1\sigma \xrightarrow{M_1 i} \hat{t_2'}, \hat{\sigma'} \wedge M_1 t_2' \equiv \hat{t_2'} \wedge M_1\sigma' \equiv \hat{\sigma'}$

Since M satisfies φ, we have $t_2, M\sigma \xrightarrow{Mi} \hat{t_2'}, \hat{\sigma'}$ and $M\sigma' \equiv \hat{\sigma'}$, which we needed to show, as well as $Mt_1 \blacklozenge t_2' \equiv t_1 \bowtie \hat{t_2'}$, which follows from $Mt_2' \equiv \hat{t_2'}$.

□

Proof of Lemma 6.1. We prove Lemma 6.1 as follows. There is only one rule that applies, namely $\dfrac{\text{SI-Handle}}{t, \sigma \;\to\; \overline{t', \sigma', \boxed{i, \varphi_1}} \qquad t', \sigma' \;\Downarrow\; \overline{t'', \sigma'', \boxed{\varphi_2}}}{t, \sigma \;\Rightarrow\; \overline{t'', \sigma'', \boxed{i, \varphi_1 \wedge \varphi_2}}}$.

Provided that $M\varphi_1 \wedge \varphi_2$, we need to demonstrate that $\dfrac{\text{I-Handle}}{t, \sigma \xrightarrow{j} \hat{t'}, \hat{\sigma'} \qquad \hat{t'}, \hat{\sigma'} \Downarrow \hat{t''}, \hat{\sigma''}}{t, \sigma \overset{j}{\Rightarrow} \hat{t''}, \hat{\sigma''}}$ with $\hat{\sigma} = M\sigma$ and $j = Mi, Mt'' \equiv \hat{t''}$ and $M\sigma'' \equiv \hat{\sigma''}$.

Lemma 6.2 and Lemma 6.3 respectively give us that
$\forall M_1.M_1\varphi_1 \supset t_1, M_1\sigma \xrightarrow{M_1 i} \hat{t'}, \hat{\sigma'} \wedge M_1 t' \equiv \hat{t'} \wedge M_1\sigma' \equiv \hat{\sigma'}$ and
$\forall M_2.M_2\varphi_2 \supset t', M_2\sigma' \Downarrow \hat{t''}, \hat{\sigma''} \wedge M_2 t'' \equiv \hat{t''} \wedge M_2\sigma'' \equiv \hat{\sigma''}$.

Since M satisfies both φ_1 and φ_2, we obtain exactly what we needed to prove, namely $t_1, M\sigma \xrightarrow{Mi} \hat{t'}, \hat{\sigma'}, t', M\sigma' \Downarrow \hat{t''}, \hat{\sigma''}, Mt'' \equiv \hat{t''}$ and $M\sigma'' \equiv \hat{\sigma''}$.

□

D COMPLETENESS PROOFS

Proof of Lemma 6.8. We prove Lemma 6.8 by induction over t.

Case $t = \square\, v$

One rule applies in this case, namely $\dfrac{\text{H-Change}}{\square\, v, \hat{\sigma} \xrightarrow{v'} \square\, v', \hat{\sigma}}\; v, v' : \tau$

Take $i = s$ and assume $\sigma'' = \sigma.\, s \sim v'$ holds by definition. Then by the SH-Change rule, we know that a symbolic execution exists.

Case $t = \boxtimes\, \tau$

One rule applies in this case, namely $\dfrac{\text{H-Fill}}{\boxtimes\, \tau, \hat{\sigma} \xrightarrow{v} \square\, v, \hat{\sigma}}\; v : \tau$

Take $i = s$ and assume $\sigma'' = \sigma.\, s \sim v$ holds by definition. Then by the SH-Fill rule, we know that a symbolic execution exists.

Case $t = \blacksquare\, l$

One rule applies in this case, namely $\dfrac{\text{H-Update}}{\blacksquare\, l, \hat{\sigma} \xrightarrow{v} \blacksquare\, l, \hat{\sigma}[l \mapsto v]}\; \sigma(l), v : \tau$

Take $i = s$ and assume $\sigma'' = \sigma.\, s \sim v'$ holds by definition. Then by the SH-Update rule, we know that a symbolic execution exists.

Case $t = t_1 \rhd e_2$

Two rules apply in this case

Case $\dfrac{\text{H-Next}}{e_2\; \hat{v_1}, \sigma \Downarrow \hat{t_2}, \hat{\sigma'}}{t_1 \rhd e_2, \sigma \xrightarrow{\mathsf{C}} \hat{t_2}, \hat{\sigma'}}\; \mathcal{V}(t_1, \sigma) = \hat{v_1} \wedge \neg \mathcal{F}(\hat{t_2}, \hat{\sigma'})$

Take $i = s$ and assume $\sigma'' = \sigma.\, s \sim \mathsf{C}$ holds by definition. Then by the SH-Next rule, we know that a symbolic execution exists.

Case $\dfrac{\text{H-PassNext}}{t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \rhd e_2, \sigma \xrightarrow{j} \hat{t_1'} \rhd e_2, \sigma'}$

By application of the induction hypothesis, we obtain the following.
For all t_1, σ, j such that $t_1, \sigma \underset{j}{\to} t_1', \sigma'$ there exists an $i \sim j$ such that $t_1'', \sigma'' \;\to\; t_1''', \sigma''', i, \varphi$.
From this we can conclude that there exists a symbolic execution $t_1 \rhd e_2, \sigma \;\to\; t_1''' \rhd e_2, \sigma''', i, \varphi$, and that $i \sim j$.

Case $t = t_1 \blacktriangleright e_2$

One rule applies in this case, namely $\dfrac{t_1, \sigma \xrightarrow{j} \hat{t_1'}, \sigma'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{j} \hat{t_1'} \blacktriangleright e_2, \sigma'}$ H-PassThen

By application of the induction hypothesis, we obtain the following.
For all t_1, σ, j such that $t_1, \sigma \underset{j}{\rightarrow} t_1', \sigma'$ there exists an $i \sim j$ such that $t_1'', \sigma'' \rightarrow t_1''', \sigma''', i, \varphi$.
From this we can conclude that there exists a symbolic execution $t_1 \blacktriangleright e_2, \sigma \rightarrow t_1''' \blacktriangleright e_2, \sigma''', i, \varphi$, and $i \sim j$.

Case $t = e_1 \lozenge e_2$

Two rules apply in this case.

Case $\dfrac{e_1, \sigma \Downarrow \hat{t_1}, \hat{\sigma'}}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{L}} \hat{t_1}, \hat{\sigma'}} \neg\mathcal{F}(\hat{t_1}, \hat{\sigma'})$ H-PickLeft

Take $i = s.\ s \sim \mathsf{L}$ holds by definition.
Lemma 6.9 gives us the following.
There exists a symbolic execution $e_1, \sigma \Downarrow t_1, \sigma_1, \varphi$.
There exists a symbolic execution $e_2, \sigma_1 \Downarrow t_2, \sigma_2, \varphi$.

We can now conclude that a symbolic execution exists. Either by the SH-PickLeft rule, in case $\mathcal{F}(t_2, \sigma_2)$, or by the SH-Pick rule in case $\neg\mathcal{F}(t_2, \sigma_2)$.

Case $\dfrac{e_2, \sigma \hat{\Downarrow} \hat{t_2}, \hat{\sigma'}}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{R}} \hat{t_2}, \hat{\sigma'}} \neg\mathcal{F}(\hat{t_2}, \hat{\sigma'})$ H-PickRight

Take $i = s.\ s \sim \mathsf{L}$ holds by definition.
Lemma 6.9 gives us the following.
There exists a symbolic execution $e_1, \sigma \Downarrow t_1, \sigma_1, \varphi$.
There exists a symbolic execution $e_2, \sigma_1 \Downarrow t_2, \sigma_2, \varphi$.

We can now conclude that a symbolic execution exists. Either by the SH-PickRight rule, in case $\mathcal{F}(t_1, \sigma_1)$, or by the SH-Pick rule in case $\neg\mathcal{F}(t_1, \sigma_1)$.

Case $t = t_1 \blacklozenge t_2$

Two rules applies in this case.

Case $\dfrac{t_1, \sigma \xrightarrow{j} \hat{t_1}', \hat{\sigma'}}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{F}\,j} \hat{t_1}' \blacklozenge t_2, \hat{\sigma'}}$ H-FirstOr

Take $i = \mathsf{F}\,i$.
By application of the induction hypothesis, we obtain the following.
For all t_1, σ, j such that $t_1, \sigma \xrightarrow{j} t_1', \sigma'$ there exists an $i \sim j$ such that $t_1'', \sigma'' \rightarrow t_1''', \sigma''', i, \varphi$.

From this, we can conclude that $\mathsf{F}\,i \sim \mathsf{F}\,j$. From SH-Or, and the conclusion of the induction hypothesis, we can conclude that there exists an i such that $t_1 \blacklozenge t_2, \sigma \rightarrow t_1' \blacklozenge t_2, \sigma', i, \varphi$.

Case $\dfrac{t_2, \sigma \xrightarrow{j} \hat{t_2}', \hat{\sigma'}}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{S}\,j} t_1 \blacklozenge \hat{t_2}', \hat{\sigma'}}$ H-SecondOr

Take $i = \mathsf{S}\,i$.
By application of the induction hypothesis, we obtain the following.
For all t_2, σ, j such that $t_2, \sigma \xrightarrow{j} t_2', \sigma'$ there exists an $i \sim j$ such that $t_2'', \sigma'' \rightarrow t_2''', \sigma''', i, \varphi$.

From this, we can conclude that $\mathsf{S}\,i \sim \mathsf{S}\,j$. From SH-Or, and the conclusion of the induction hypothesis, we can conclude that there exists an i such that $t_1 \blacklozenge t_2, \sigma \;\to\; t_1 \blacklozenge t_2', \sigma', i, \varphi$.

Case $t = t_1 \bowtie t_2$

Two rules applies in this case.

Case H-FirstAnd
$$\frac{t_1, \sigma \xrightarrow{j} \hat{t_1}', \hat{\sigma}'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{F}\,j} \hat{t_1}' \bowtie t_2, \hat{\sigma}'}$$

Take $i = \mathsf{F}\,i$.

By application of the induction hypothesis, we obtain the following.

For all t_1, σ, j such that $t_1, \sigma \xrightarrow{j} t_1', \sigma'$ there exists an $i \sim j$ such that $t_1'', \sigma'' \;\to\; t_1''', \sigma''', i, \varphi$.

From this, we can conclude that $\mathsf{F}\,i \sim \mathsf{F}\,j$. From SH-And, and the conclusion of the induction hypothesis, we can conclude that there exists an i such that $t_1 \bowtie t_2, \sigma \;\to\; t_1' \blacklozenge t_2, \sigma', i, \varphi$.

Case H-SecondAnd
$$\frac{t_2, \sigma \xrightarrow{j} \hat{t_2}', \hat{\sigma}'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{S}\,j} t_1 \bowtie \hat{t_2}', \hat{\sigma}'}$$

Take $i = \mathsf{S}\,i$.

By application of the induction hypothesis, we obtain the following.

For all t_2, σ, j such that $t_2, \sigma \xrightarrow{j} t_2', \sigma'$ there exists an $i \sim j$ such that $t_2'', \sigma'' \;\to\; t_2''', \sigma''', i, \varphi$.

From this, we can conclude that $\mathsf{F}\,i \sim \mathsf{S}\,j$. From SH-And, and the conclusion of the induction hypothesis, we can conclude that there exists an i such that $t_1 \bowtie t_2, \sigma \;\to\; t_1 \bowtie t_2', \sigma', i, \varphi$.

□

Proof of Theorem 6.7. The driving semantics only consists of one rule, namely $\dfrac{t, \sigma \xrightarrow{j} \hat{t}', \hat{\sigma}' \qquad \hat{t}', \hat{\sigma}' \Downarrow \hat{t}'', \hat{\sigma}''}{t, \sigma \overset{j}{\Rightarrow} \hat{t}'', \hat{\sigma}''}$ I-Handle.

By Lemma 6.8 we obtain the following.

$t, \sigma \underset{j}{\to} \hat{t}', \hat{\sigma}' \supset \exists i.t, \sigma \;\to\; t', \sigma', i, \varphi \wedge i \sim j$

Then by Lemma 6.9 we obtain the following.

$\hat{t}', \hat{\sigma}' \Downarrow \hat{t}'', \hat{\sigma}'' \supset \hat{t}', \hat{\sigma}' \;\Downarrow\; t'', \sigma'', \varphi'$

From the above, together with the SI-Handle rule, we can conclude that there exists a symbolic execution $t, \sigma \;\Rightarrow\; t', \sigma', i, \varphi \wedge i \sim j$.

□

Mystery Functions

Making specifications, unit tests, and implementations coexist in the mind of undergraduate students

Olivier Danvy
Yale-NUS College & School of Computing
National University of Singapore
danvy@acm.org

ABSTRACT

This article documents how to make the concepts of specification, properties of specifications, unit tests, soundness of unit tests, implementation, and satisfaction coexist correctly, harmoniously, and effectively in the mind of undergraduate students, using an off-the-shelf proof assistant. The concepts are instilled through a family of puzzles: given the specification of a mystery function (i.e., conditions this function should satisfy), which functions—if any—satisfy this specification? Each puzzle is solved using a combination of informal induction (as in machine learning), structural recursion (to implement the function), and formal induction (to prove satisfaction), and in a consolidating way such that each concept has a place that makes sense, instead of being perceived as vague, alien, arbitrary, and irreproducible. The "Eureka!" moment that concludes each informal induction and the subsequent formal induction that confirms that the student was right make mystery functions rewarding, and repetition makes them addictive. Mystery functions can also be used as a vector for expansion as well as for reflection.

ACM Reference Format:
Olivier Danvy. 2020. Mystery Functions: Making specifications, unit tests, and implementations coexist in the mind of undergraduate students. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION AND MOTIVATION

> *We should encourage the young people who are now learning about computers to feel some of the spirit of intellectual adventure which, I think, [computing] can engender.* – Christopher Strachey (1963)

The present article describes an early transition from the Coq proof assistant [Bertot & Castéran 2004] being a topic of study to it being a tool to study other topics. At the end of this transition, the students appear secure in their tiny subset of Coq where each proof step is explicit, and eager to expand it. Over the last 8 years, 250 students have been exposed to this transition, first at Aarhus University and now at Yale-NUS College, and their knowledge was tested with a term project (including a report) and an oral exam. Typical term projects ranged from proving that an interpret-compile-run diagram commutes to formalizing a static program analysis, an algorithm, or the chapter of a textbook.

The transition is carried out through a family of puzzles involving concepts that computer scientists are expected to be familiar with: specification, unit tests, implementation, and how an implementation satisfies a specification. The family of puzzles is unthreatening in the sense that it is enough that the students recognize a specification, a unit test, etc. when they see one, and to this end, each definition explicitly has "specification", "unit test", etc. in its name. Each puzzle is solved as follows. Given the specification of a function (i.e., conditions this function should satisfy), we first need to decide whether this specification is vacuous (no functions satisfy it), ambiguous (at least two distinct functions satisfy it), or whether exactly one function satisfies it. If the specification determines a unique function, we need to find out which one—a non-trivial question because the specification cannot be transliterated into a program in general, hence the name "mystery function". To find this function, we incrementally tabulate it (making the tabulation take the form of a unit-test function) until we figure out what it does ("Eureka!"). We then turn to how the function should do what it is supposed to do, resulting in an implementation. We can then test this implementation and verify that it satisfies the specification.

The rest of this article is structured as follows. The overall approach is presented as a template (Section 2) and then illustrated (Sections 3, 4, and 5). Three aftermaths are outlined (Sections 6, 7, and 8). The design of effective mystery functions is described (Section 9), as well as their use as a vector towards other concepts such as characteristic functions, abstract data types, compositionality (or lack of), structural induction, and structural recursion (Section 10). Caveats are also listed (Section 11).

Prerequisites and notations: This article assumes a basic knowledge of pure functional programming (structurally recursive functions) and an elementary acquaintance with Coq: Peano numbers, nothing but the standard libraries (e.g., `Arith`, `Bool`), and tactics that have one explainable effect in the goal window. This subset of Coq can be presented as providing a domain-specific language for writing mathematical proofs at the level of high school.

2 THE OVERALL APPROACH

Each mystery function is defined with a specification:

```
Definition specification_of_mystery_function_XX (mf : ...) :=
  ...
```

This specification states conditions that the mystery function should satisfy.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

One then should prove whether the specification uniquely determines a function, if any:

```
Proposition there_is_at_most_one_mystery_function_XX :
  forall f g : ...,
    specification_of_mystery_function_XX f ->
    specification_of_mystery_function_XX g ->
    forall ...,
      f ... = g ....
```

Assuming that this proof is successful, one then proceeds to tabulate the mystery function, incrementally constructing its graph and representing it as a Boolean-valued unit-test function:

```
Definition unit_test_for_mystery_function_XX mf :=
  (mf ... =x= ...) && (mf ... =x= ...) && ... (* etc. *).
```

where `=x=` is an infix notation for a Boolean comparison function at a suitable type and `&&` is an infix notation for Boolean conjunction.

Typically, the tabulation follows the inductive structure of the domain or of the codomain of the mystery function, until the aha! moment typical of this informal inductive reasoning, resulting in an implementation:

```
Definition mystery_function_XX (... : ...) ... : ... :=
  ...
```

To be on the safe side, one then verifies that the implementation passes the unit test, i.e., that the following computation yields `true`:

```
Compute (unit_test_for_mystery_function_XX mystery_function_XX).
```

And then one proves that the implementation also satisfies the specification:

```
Theorem there_is_at_least_one_mystery_function_XX :
  specification_of_mystery_function_XX mystery_function_XX.
```

Typically, this proof is carried out by induction—a formal counterpart of the earlier informal induction.

So all in all, having proved

- that there is at most one mystery function that satisfies the specification (i.e., if there are two such functions, these two functions are the same) and
- that there is at least one such mystery function (i.e., having exhibited one),

one has proved that there is exactly one function that satisfies the specification, and one has found this mystery function.

However, in the course of proving whether there is at most one mystery function that satisfies the specification, or in the course of tabulating it, one may realize that several such functions exist. One then

- conjures up two functions that satisfy the specification and that are distinct (i.e., that map at least one given input value to two distinct output values), and
- formally proves that the specification does not uniquely determine a function, i.e., is ambiguous:

```
Theorem there_are_at_least_two_mystery_functions_XX :
  exists f g : ...,
    specification_of_mystery_function_XX f /\
    specification_of_mystery_function_XX g /\
    exists x : ...,
      f x <> g x.
```

Or again one realizes that no functions exist that satisfy the specification:

```
Theorem there_are_zero_mystery_functions_XX :
  forall mf : ...,
    specification_of_mystery_function_XX mf ->
    exists x : ...,
      mf x <> mf x.
```

Either way, the puzzle is solved: either the specification determines a unique function, or the specification is ambiguous, or it is vacuous. The following sections illustrate each of these three cases.

3 A MYSTERY FUNCTION THAT IS UNIQUE

Here is the specification of a mystery function from natural numbers to natural numbers [Tao 2006, Section 3.1]:

```
Definition specification_of_mystery_function_00 (mf : nat -> nat) :=
  mf 0 = 1 /\ forall i j : nat, mf (S (i + j)) = mf i + mf j.
```

This specification does not follow the inductive structure of natural numbers and so it cannot "just" be transliterated into a structurally recursive definition.

That there is at most one such mystery function is proved by induction:

```
Proposition there_is_at_most_one_mystery_function_00 :
  forall f g : nat -> nat,
    specification_of_mystery_function_00 f ->
    specification_of_mystery_function_00 g ->
    forall n : nat,
      f n = g n.
```

Let us start enumerating the graph of this mystery function by tabulating it, representing this table as a unit-test function:

- The first clause of the specification says that `0` is mapped to `1`:

  ```
  Definition unit_test_for_mystery_function_00a (mf : nat -> nat) :=
    (mf 0 =n= 1) (* etc. *).
  ```

  (`=n=` is an infix notation for a Boolean comparison function over two natural numbers)
- `mf 1`
  = {because $1 = 1 + (0 + 0)$}
  `mf (S (0 + 0))`
  = {second clause of the specification}
  `mf 0 + mf 0`
  = {first clause of the specification, twice}
  `1 + 1`
  = {calculemus}
  `2`

  We are now in position to expand the definition of the unit-test function with one more test:

  ```
  Definition unit_test_for_mystery_function_00b (mf : nat -> nat) :=
    (mf 0 =n= 1) && (mf 1 =n= 2) (* etc. *).
  ```

- `mf 2`
  = {because $2 = 1 + (1 + 0)$}
  `mf (S (1 + 0))`
  = {second clause of the specification}
  `mf 1 + mf 0`
  = {second test in the unit-test function and
  first clause of the specification}
  `2 + 1`
  =
  `3`

Out of vigilance, one should verify that the alternative starting point, $2 = 1 + (0 + 1)$, leads one to the same result. (If it didn't, that would provide a ground for proving that the specification is vacuous.)

We are now in position to expand the definition of the unit-test function with one more test:

```
Definition unit_test_for_mystery_function_00c (mf : nat -> nat) :=
  (mf 0 =n= 1) && (mf 1 =n= 2) && (mf 2 =n= 3) (* etc. *).
```

At that point (observation), the pattern suggests that the mystery function is the successor function (hypothesis), which we can test by adding one more clause to the unit-test function (experiment).

- `mf 3`
  = {because $3 = 1 + (1 + 1)$}
  `mf (S (1 + 1))`
  = {second clause of the specification}
  `mf 1 + mf 1`
  = {second test in the unit-test function, twice}
  `2 + 2`
  =
  `4`

Again, out of vigilance, one should also verify that the alternative starting points, $3 = 1 + (2 + 0)$ and $3 = 1 + (0 + 2)$, lead one to the same result. (If either of them didn't, they would provide a ground for proving that the specification is vacuous.)

We are now in position to expand the definition of the unit-test function with one more test:

```
Definition unit_test_for_mystery_function_00d (mf : nat -> nat) :=
  (mf 0 =n= 1) && (mf 1 =n= 2) && (mf 2 =n= 3) && (mf 3 =n= 4)
  (* etc. *).
```

So, "Eureka!"—the mystery function is the successor function:

```
Definition mystery_function_00 := S.
```

Or then again, less succinctly:

```
Definition mystery_function_00 (n : nat) : nat :=
  S n.
```

Let us check that this implementation passes the unit test:

```
Compute (unit_test_for_mystery_function_00d mystery_function_00).
```

This computation yields `true`, and therefore the implementation does pass the unit test.

Finally let us prove that this implementation satisfies the specification:

```
Theorem there_is_at_least_one_mystery_function_00 :
  specification_of_mystery_function_00 mystery_function_00.
```

The proof is by cases.

All in all, we have proved that the specification determines a unique function.

4 A MYSTERY FUNCTION THAT IS NOT UNIQUE

After the successor function, the predecessor function. Here is the specification of a "mystery" function over natural numbers:

```
Definition specification_of_mystery_function_01 (mf : nat -> nat) :=
  forall n : nat,
    mf (S n) = n.
```

Attempting to prove that there is at most one such mystery function, one gets stuck in the `O` case.

So let us define a maker of functions that will satisfy the specification, parameterizing it with the `O` case:

```
Definition make_mystery_function_01 (z n : nat) :=
  match n with
  | 0    => z
  | S n' => n'
  end.

Lemma about_make_mystery_function_01 :
  forall z : nat,
    specification_of_mystery_function_01 (make_mystery_function_01 z).
```

The proof is by cases.

We are now in position to formally prove that the specification does not uniquely determine a function:

```
Theorem there_are_at_least_two_mystery_functions_01 :
  exists f g : nat -> nat,
    specification_of_mystery_function_01 f /\
    specification_of_mystery_function_01 g /\
    exists n : nat,
      f n <> g n.
```

Lemma `about_make_mystery_function_01` provides two distinct witnesses for the two first conjuncts, and `0` is the witness for the last conjunct.

All in all, we have proved that the specification is ambiguous.

5 A MYSTERY FUNCTION THAT DOES NOT EXIST

After the successor function and the predecessor function, a bit of both. Here is the specification of a "mystery" function over natural numbers:

```
Definition specification_of_mystery_function_02 (mf : nat -> nat) :=
  (forall n : nat, mf n = S n) /\ (forall n : nat, mf (S n) = n).
```

That there is at most one such mystery function is proved by induction:

```
Proposition there_is_at_most_one_mystery_function_02 :
  forall f g : nat -> nat,
    specification_of_mystery_function_02 f ->
    specification_of_mystery_function_02 g ->
    forall n : nat,
      f n = g n.
```

However, this "function" does not exist, since it can map the same input value to two distinct output values, a contradictory aspect that is captured by the following lemma, which capitalizes on the first positive natural number (any other would do just as well):

```
Lemma about_mystery_function_02 :
  forall mf : nat -> nat,
    specification_of_mystery_function_02 mf ->
    mf 1 = 0 /\ mf 1 = 2.
```

Therefore no functions exist that satisfy this specification:

```
Theorem there_are_zero_mystery_functions_02 :
  forall mf : nat -> nat,
    specification_of_mystery_function_02 mf ->
    exists n : nat,
      mf n <> mf n.
```

The lemma provides a witness (here, `1`) for the inequality.

All in all, we have proved that the specification is vacuous.

6 AFTERMATH OF SECTION 3: WHAT IF MULTIPLE IMPLEMENTATIONS EXIST?

As a spinoff of Section 3, it makes sense to point out that while a specification can determine a unique function, several implementations of this function can exist. For example, and based on the identity `forall x : nat, S x = x + 1`:

```
Definition mystery_function_00_alt := fun (n : nat) => n + 1.

Theorem there_is_at_least_one_mystery_function_00_alt :
  specification_of_mystery_function_00 mystery_function_00_alt.
```

In practice, some implementations are more efficient than others.

7 AFTERMATH OF SECTION 3: SOUNDNESS OF A UNIT-TEST FUNCTION

As a spinoff of Section 3, it makes sense to point out how proving the soundness of the unit-test function with respect to the specification makes one emulate the computational steps of the successive tests using proof steps:

```
Theorem soundness_of_the_unit_test_function_for_mystery_function_00 :
  forall mf : nat -> nat,
    specification_of_mystery_function_00 mf ->
    unit_test_for_mystery_function_00c mf = true.
Proof.
  unfold specification_of_mystery_function_00.
  unfold unit_test_for_mystery_function_00c.
  intros mf [H_O H_S].
  (* Goal: (mf 0 =n= 1) && (mf 1 =n= 2) && (mf 2 =n= 3) = true *)
  rewrite -> H_O.
  (* Goal: (1 =n= 1) && (mf 1 =n= 2) && (mf 2 =n= 3) = true *)
  rewrite -> (Nat.eqb_refl 1).
  (* Goal: true && (mf 1 =n= 2) && (mf 2 =n= 3) = true *)
  rewrite -> (andb_true_l (mf 1 =n= 2)).
  (* Goal: (mf 1 =n= 2) && (mf 2 =n= 3) = true *)
  (* etc. *)
```

Each proof step achieves a computational step, and that gives everyone pause, because how does a computation happen exactly? Most students have never wondered how a computation happens. To them, it just does. And so

- being shown / made to view / forced to analyze how, step by step, a computation takes place makes them think about the computational process;
- being the ones in control of which step to undergo makes them aware that several steps are possible; and
- using proof steps to carry out computational steps makes them realize that they are not just reasoning about *what* a program computes: they are reasoning about *how* this computation takes place.

At this point of their learning trajectory, students have outgrown their introductory course about object-oriented programming and realized that there is more to computing than learning to click a "Wingardium Leviosa" icon on the screen to see a feather levitate in a window that has automagically popped open. But still their perception of computing is most likely to be that of layers of program processors: to run a program,

- denotationally, one compiles it into another program and runs this other program, or
- operationally, one interprets it using yet another given program (ultimately the hardware processor).

So everyone is given pause because most likely for the first time, and while they are on the front seat, they see computation defined in terms of something else than computation.

8 AFTERMATH OF SECTION 4: POST-MORTEM ANALYSIS

As a spinoff of Section 4, it makes sense (a) to point out that this particular specification was one of a partial function, (b) to remind one of the option type and its use to implement partial functions as total functions, and (c) to lead the audience towards the following alternative specification, wondering how many mystery functions satisfy it and letting everybody figure it out:

```
Definition specification_of_mystery_function_01'
                                          (mf : nat -> option nat) :=
  mf 0 = None /\ forall n : nat, mf (S n) = Some n.
```

9 HOW TO DESIGN MYSTERY FUNCTIONS

In the author's experience, to be effective, mystery functions must elicit an aha! moment, be they mathematically or computationally inspired. At first, though, they should connect with the comfort zone of the current batch of students. Simple variations on the sample in Sections 3, 4, and 5 do well for a start: the students successfully mimic the proofs and in so doing they familiarize themselves with the vocabulary and the techniques.

In particular, before they have even realized it, the students have put a dozen induction proofs under their belt merely to prove that there is at most one mystery function satisfying a given specification. And after another dozen or two, they have adopted the adjective "routine" next to the noun "induction", not because they have been told (which is always anxiety-inducing), but because they have said it themselves (which is anxiety-repelling, so to speak). After a while, they also realize that they do not actually need to first prove that there is at most one mystery function satisfying a given specification—a sign that they have started to reflect on the overall process.

We can also be manipulative and make allusions since understanding an allusion makes one's audience feel smart. This rhetorical trick can be used to design effective mystery functions. Consider the following specification, for example:

```
Definition specification_of_mystery_function_11 (mf : nat -> nat) :=
  mf 1 = 1 /\ forall i j : nat,
                mf (i + j) = mf i + 2 * i * j + mf j.
```

A mathematically inclined student will recognize the second clause a priori as binomial expansion at rank 2, giving them a head start to identify the mystery function as squaring its argument. A computationally inclined student will recognize this binomial expansion a posteriori, connecting computation and mathematics in their mind.

Likewise, the second clause of the following specification can be identified as the binomial expansion of `n' + 1` at rank 2, giving one a head start to identify the mystery function as also squaring its argument. Alternatively, the two clauses can be identified as the base case and the induction step for summing the first odd natural numbers since this particular specification follows the inductive structure of natural numbers:

```
Definition specification_of_mystery_function_04 (mf : nat -> nat) :=
  mf 0 = 0 /\ forall n' : nat,
                mf (S n') = mf n' + S (2 * n').
```

Again, the identity `forall x : nat, S x = x + 1` proves crucial here, but at any rate the students can be made to observe in passing that adding the first successive odd numbers always yields a square.

Less allusively, specifying a monotonically increasing function makes students come up with distinct examples of such functions to show that the specification is ambiguous, keeping in mind that students remember what they do a lot more than what they are told.

A specification such as the following one makes students realize how the corresponding mystery function enumerates the graph of the factorial function (a familiar example for a beginning functional programmer, and an opportunity to be reminded of what "graph of a function" and "enumerate" mean):

```
Definition specification_of_mystery_function_15
                                          (f : nat -> nat * nat) :=
  f 0 = (0, 1) /\ forall n' : nat,
                  f (S n') = let (x, y) := f n'
                             in (S x, y * S x).
```

Ditto for computing Fibonacci numbers in linear time:

```
Definition specification_of_mystery_function_16
                                          (f : nat -> nat * nat) :=
  f 0 = (0, 1) /\ forall n' : nat,
                  f (S n') = let (x, y) := f n'
                             in (y, x + y).
```

More challengingly, one can use one identity or another about Fibonacci numbers to manufacture the specification of a mystery function:

```
Definition specification_of_mystery_function_17 (mf : nat -> nat) :=
  mf 0 = 0 /\ mf 1 = 1 /\ mf 2 = 1 /\
  forall p q : nat,
    mf (S (p + q)) = mf (S p) * mf (S q) + mf p * mf q.

Definition specification_of_mystery_function_18 (mf : nat -> nat) :=
  mf 0 = 0 /\ mf 1 = 1 /\ mf 2 = 1 /\
  forall n''' : nat,
    mf n''' + mf (S (S (S n'''))) = 2 * mf (S (S n''')).
```

A subsequent exercise about proving one identity or another about Fibonacci numbers as listed in Wikipedia elicits a pleasant surprise when the students realize they have already proved these identities when they studied mystery functions. (Emphasis on "when they realize" because again, students remember what they do a lot better than what they are shown.)

Bringing out a feeling of déjà vu has value since it makes students realize that what they have learned is useful (since they are reusing it). But more than that, a mystery function should make one reflect. For example, basic Coq practice makes one familiar with the lemmas `plus_Sn_m` and `plus_n_Sm`:

```
Lemma plus_Sn_m : forall n m : nat, S n + m = S (n + m).

Lemma plus_n_Sm : forall n m : nat, S (n + m) = n + S m.
```

Together with a base case, these two lemmas make an empowering specification where nested induction is actually justified to prove that there is at most one such mystery function:

```
Definition specification_of_mystery_function_03
                                         (mf : nat -> nat -> nat) :=
  mf 0 0 = 0 /\
  (forall i j: nat, mf (S i) j = S (mf i j)) /\
  (forall i j: nat, S (mf i j) = mf i (S j)).
```

They play a key role to prove that there is at least one such function, provoking a "Hmmm, that's funny" that would have made Isaac Asimov chuckle:

```
Proposition there_is_at_least_one_mystery_function_03 :
  specification_of_mystery_function_03 (fun i j => i + j).
Proof.
  unfold specification_of_mystery_function_03.
  split.
    rewrite -> plus_0_l. reflexivity.
  split.
    exact plus_Sn_m.
  exact plus_n_Sm.
Qed.
```

But otherwise, mystery functions offer a simple vector to make students solve equations (e.g., Cauchy's functional equation below, in the specification of `mystery_function_42` [Aczél 1984]) and write programs (e.g., a maximum function or a comparison predicate):

```
Definition specification_of_mystery_function_42 (mf : nat -> nat) :=
  mf 1 = 42 /\ forall i j : nat,
                mf (i + j) = mf i + mf j.

Definition specification_of_mystery_function_07
                                           (mf : nat -> nat -> nat) :=
  (forall j : nat, mf 0 j = j) /\
  (forall i : nat, mf i 0 = i) /\
  (forall i j k : nat, mf (i + k) (j + k) = (mf i j) + k).

Definition specification_of_mystery_function_08
                                         (mf : nat -> nat -> bool) :=
  (forall j : nat, mf 0 j = true) /\
  (forall i : nat, mf (S i) 0 = false) /\
  (forall i j : nat, mf (S i) (S j) = mf i j).
```

Specifications such as the following ones entice the student to implement the function that halves its argument using successive decrements, either recursively or tail recursively with an accumulator (which requires an induction hypothesis to be strengthened down the line):

```
Definition specification_of_mystery_function_23 (mf : nat -> nat) :=
  mf 0 = 0 /\ mf 1 = 0 /\ forall n'' : nat,
                            mf (S (S n'')) = S (mf n'').

Definition specification_of_mystery_function_24 (mf : nat -> nat) :=
  mf 0 = 0 /\ mf 1 = 1 /\ forall n'' : nat,
                            mf (S (S n'')) = S (mf n'').
```

The student is then in good position to investigate an alternative specification of the halving function, which itself paves the way to the logarithm function in base 2 (where for simplicity $\log_2 0 = 0$):

```
Definition specification_of_mystery_function_13 (mf : nat -> nat) :=
  (forall q : nat, mf (2 * q) = q) /\
  (forall q : nat, mf (S (2 * q)) = q).

Definition specification_of_mystery_function_25 (mf : nat -> nat) :=
  mf 0 = 0 /\ (forall q : nat,
                mf (2 * (S q)) = S (mf (S q)))
  /\
  mf 1 = 0 /\ (forall q : nat,
                mf (S (2 * (S q))) = S (mf (S q))).
```

Incidentally, when proving properties about programs, students appreciate to see the proof structure "main theorem / master lemma" reflect the program structure "main function / helper function" where

- the master lemma is proved by induction because the helper function is structurally recursive and

- the main theorem is a corollary of the master lemma using suitable initial arguments because all the main function does is call the helper function with these initial arguments.

In general, students observe with interest how modularizing programs corresponds to modularizing proofs, and how structural induction in a proof corresponds to structural recursion in a program—again not because they are told, but because they [are made to] observe it. And it is heartwarming, at the oral exam, to see them display ownership of these concepts.

10 MYSTERY FUNCTIONS AS VECTORS TOWARDS OTHER CONCEPTS

As illustrated above with Fibonacci numbers, one can specify a mystery function using a characteristic property, and surf on the problem-solving skills deployed by the students to investigate this function or to expand their proof techniques without having to also expand the Coq vocabulary of their proof tactics (this last point in order to make the students secure in their knowledge, as they become aware that they know enough to solve the problem at hand, a welcome relief to the perpetual (and doubt-inducing) need for new tools to solve new problems).

It also makes sense to point out that specifications can be equivalent (i.e., imply each other, e.g., for `mystery_function_04` and `mystery_function_11` in Section 9), and that some are simpler than others, or at least more familiar than others. For example, are the following specifications equivalent?

```
Definition specification_of_mystery_function_20
                                        (mf : nat -> nat -> nat) :=
  (forall j : nat, mf O j = j) /\
  (forall i j : nat, mf (S i) j = S (mf i j)).

Definition specification_of_mystery_function_21
                                        (mf : nat -> nat -> nat) :=
  (forall j : nat, mf O j = j) /\
  (forall i j : nat, mf (S i) j = mf i (S j)).
```

How do they connect to the specifications of `mystery_function_04` and `mystery_function_11`?

Likewise, it is a small step from mystery functions to abstract data types, i.e., specifications of a type together with operators that satisfy equations.

One can also make the students reflect, e.g., on the very structure of specifications. For example, is the following specification equivalent to the previous one?

```
Definition specification_of_mystery_function_22
                                        (mf : nat -> nat -> nat) :=
  forall i j : nat,
    mf O j = j /\ mf (S i) j = mf i (S j).
```

(Of course it is equivalent. The point here is that it is such a kick to see an undergraduate student playing with the scoping rules of universal quantification over conjunction and explaining the equivalence as such to their compeers. To take the counterpoint of Alfred North Whitehead's aphorism, not knowledge, but knowledge of knowledge, heralds the death of ignorance.)

To close, the following mystery function invites one to reflect on non-inductive specifications, induction proofs (that are defeated by non-inductive specifications), and structurally recursive functions (for which induction proofs go through). The specification embodies the acompositional explanation of how to flatten a binary tree that students are typically exposed to in their introductory course on programming or algorithmics:

```
Inductive tree : Type :=
  | Leaf : nat -> tree
  | Node : tree -> tree -> tree.

Definition specification_of_mystery_function_19 (mf : tree -> tree) :=
  (forall n : nat,
     mf (Leaf n) = Leaf n) /\
  (forall (n : nat) (t : tree),
     mf (Node (Leaf n) t) = Node (Leaf n) (mf t)) /\
  (forall t11 t12 t2 : tree,
     mf (Node (Node t11 t12) t2) = mf (Node t11 (Node t12 t2))).
```

To show that at most one function satisfies this specification, an induction proof fails in the third clause because this third clause is not compositional (on the right-hand side, `mf` is not applied to a proper sub-part of the tree from the left-hand side).

However, to show that the following given function satisfies the specification, an induction proof goes through because this implementation is structurally recursive:

```
Fixpoint mystery_function_19_aux (t a : tree) : tree :=
  match t with
  | Leaf n =>
     Node (Leaf n) a
  | Node t1 t2 =>
     mystery_function_19_aux t1 (mystery_function_19_aux t2 a)
  end.

Fixpoint mystery_function_19 (t : tree) : tree :=
  match t with
  | Leaf n     =>
     Leaf n
  | Node t1 t2 =>
     mystery_function_19_aux t1 (mystery_function_19 t2)
  end.
```

This implementation is typically presented in a subsequent course about functional programming, be it to introduce accumulators in a non-tail-recursive setting, to illustrate normalization by evaluation for monoids [Coquand & Dybjer 1997], or just to present Coq in a hurry [Bertot 2006, Section 4.3].

That `mystery_function_19` (which is compositional) satisfies its specification (which is not compositional) connects what students have learned in their first year with what they have learned in their second year: it is the same proverbial elephant. Down the road, they might learn about the notion of normal form (here: a flat binary tree), about the specification of `mystery_function_19` as providing a small-step semantics for flattening, and about `mystery_function_19` as implementing the corresponding big-step semantics [Danvy 2008, Sections 10 and 11].

From then on the door is open, e.g., to the algebra of lists [Bird & Wadler 1988], and the sky, i.e., the end of the semester, is the limit.

11 CAVEATS

While developing and teaching mystery functions, the author came across the following caveats.

- Some students become uncomfortably aware of holes in their understanding of programming and of proving. They need basic exercises to fill these holes.
- It's obvious, but still: one person's aha! moment is another person's yawn. One should know one's audience and select mystery functions accordingly.

- It's also obvious, but still: new Coq tactics should not be needed to solve a mystery function, or if one is, it should be presented as the learning goal of this particular mystery function.
- After a while, due to simple tiredness, the brain shuts down, the spine takes over, and solving mystery functions becomes a video game: students need to be told to stop playing chicken with Coq and take a break.
- Help should always be given to those who ask for it, but at the price of solving another mystery function—a price that is accepted remarkably well.
- All proofs should be read and commented. Induction proofs that do not use the induction hypothesis should be singled out as such so that they can be downgraded to just-in-case proofs. Mark Twain's advice also applies here: overly complicated proofs should be taken out and shot. At these junctures, Niels Bohr's quote "You are not thinking, you are being logical." comes mightily handy.

12 BACKGROUND AND RELATED WORK

This article was inspired by an exercise in Section 3.1 of Terence Tao's book "Solving Mathematical Problems: A Personal Perspective" [2006]. This exercise resonated with the author's high-school memories of solving exercises in functional analysis [Aczél 1984], and mystery functions were born.

Witness the opening quote in Section 1, the author's take is more idealistic than practical, but then again so was Paul Lockhart's in his "Mathematician's Lament" [2009]. It is closer in spirit to George Polya's "Induction and Analogy in Mathematics" [1954a] and "Patterns of Plausible Inference" [1954b] than to its digest "How to solve it" [1957]. It is not as playful as Raymond Smullyan's puzzles [2015], not as light-hearted as Dan Friedman and Carl's Eastlund "The Little Prover" [2015], not as witty as Gian-Carlo Rota's "Problem Solvers and Theorizers" [1996], not as deep as Isaac Asimov's "Relativity of Wrong" [1989], not as perceptive and colorful as Leo Rosten's "Education of H*Y*M*A*N K*A*P*L*A*N" [Ross 1937], not as dedicated as Rex Page and Ruben Gamboa's "Essential Logic for Computer Science" [2019], and not as comprehensive as Bard Bloom and Alan Fekete's smeagol [1994], Amer Diwan, William M. Waite, Michele H. Jackson, and Jacob Dickerson's PL-detective [2004], and Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler's mystery languages [2017]. It is merely the 5th step of a 12-step journey (a semester) towards helping students become aware and perhaps even appreciative of the isomorphism between programming and proving, using one particular interactive proof assistant. During this 5th step, Coq transitions from being a topic of study to being a tool (here: a domain-specific language for writing proofs) to study other topics (here: solving functional equations).

Students who survived the Western education system so far have grown uncannily good at acquiring new skills and they are experts at figuring out what their teacher wants in order to deliver just that so that they can move on, a strategy otherwise known as "fire and forget." As a result, their knowledge can be unnervingly kaleidoscopic. And so the goal of the approach reported here is not to invent and trademark new and better ways to pasteurize branches of knowledge to make them swallowable and perhaps even digestible. It is to revisit previous skills – here: programming and proving, including their shortcomings – and to unify them (fragments and all) in an unforgettable fire, the furnace of an off-the-shelf proof assistant, in order to empower students with an actionable knowledge that they own and in which they feel secure, so that they say, reflectively, "this makes sense" more often than, dejectedly, "I can't make any sense of this." Due to the law of diminishing returns, unifying existing skills might not be as compelling as acquiring yet another new skill, nor by its nature can this unification replace this acquisition (to generalize anything, one must first know something), but it is a crucial step from learning ("apprendre" in French) to understanding ("comprendre", also in French). Etymologically, "apprendre" and "comprendre" share the root "prendre", which means "to take", "to acquire" and is what every teacher selflessly wants from their students ("make this yours").

Now as a stepping stone, does this 5th step provide a point that is firm enough for a student to lift their world? For some students it appears so. At the very least, everybody survived the furnace in that each of the 250 students mentioned in Section 1 passed the final exam, however much passing the exam has gone from being a measure of success to being the success.

13 CONCLUSION

John von Neumann once quipped that in mathematics, one doesn't understand things, one gets used to them. Once undergraduate students have figured out a whole series of mystery functions, do they understand the concepts of specification, implementation, etc., or have they merely become used to them? The author cannot say, but what he can say is that the students become used to the format of the puzzles very quickly, and that they talk to each other accurately towards solving each of them, using programming and proving skills that have shot up beyond everybody's expectations ("I don't like mathematics, I am bad at it, I've been told so repeatedly, but this is fun, and I am pretty good at it."). Also, as it happens, this knowledge remains actionable not just until the final exam: according to colleagues who teach downstream, some of it is lasting, witness students questioning the specifications they are subsequently given ("does this specification determine a unique program?") or formalizing a new concept. In that sense interactive proof assistants in general, and Coq in particular, do provide students with a new epistemological vista: with Coq in particular, students can come home to roost.

Teachers often say that one can lead a horse to the water, but one can't make it drink. First making the horse thirsty, however, helps. In the present case, the problem solver in each student is tickled, often to the point of galloping back to clamor for more mystery functions, thanks to the functional programming language, the domain-specific language for writing proofs, and the algebraic support provided by a proof assistant such as Coq, and thanks to the infrastructure of Proof General that enables its users to edit at the speed of thought.

Acknowledgments: Thanks are due to Aleš Bizjak, Ranald Clouston, Thomas Dinsdale-Young, Erik Ernst, Thomas Heinze, Aleksandr Karbyshev, Robbert Krebbers, Gianluca Mezzetti, Jan Midtgaard, Michael Raskin, Filip Sieczkowski, and Kasper Svendsen who served as internal evaluators for the oral exams in 2011-2016, and of course

to the 250 students who patiently endured the mystery functions over the years, willy-nilly.

The author is also grateful to Gilles Barthe, David Basin, and Ilya Sergey for their encouragement, a few years back, to document this material.

Additional thanks go to the anonymous reviewers for their feedback and to Ilya Sergey for an insightful round of comments on the first draft.

REFERENCES

Janos Aczél (Ed.). 1984. *Functional Equations: History, Applications and Theory.* D. Reidel Publishing Company.

Isaac Asimov. 1989. The relativity of wrong. *The Skeptical Inquirer* 14, 1 (1989), 35–44.

Yves Bertot. 2006. Coq in a hurry. CoRR. http://arxiv.org/abs/cs/0603118v2.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development.* Springer.

Richard Bird and Philip Wadler. 1988. *Introduction to Functional Programming* (1st ed.). Prentice-Hall International.

Bard Bloom and Alan Fekete. 1994. Self-sufficiency and critical thinking in the programming languages course. *ACM SIGCSE Bulletin* 26, 2 (1994), 9–18.

Thierry Coquand and Peter Dybjer. 1997. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science* 7, 1 (1997), 75–94.

Olivier Danvy. 2008. From Reduction-Based to Reduction-Free Normalization. In *Advanced Functional Programming, Sixth International School (Lecture Notes in Computer Science)*, Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra (Eds.). Springer, 66–164. Lecture notes including 70+ exercises.

Amer Diwan, William M. Waite, Michele H. Jackson, and Jacob Dickerson. 2004. PL-detective: A system for teaching programming language concepts. *Journal on Educational Resources in Computing* 4, 4 (Dec. 2004).

Daniel P. Friedman and Carl Eastlund. 2015. *The Little Prover.* The MIT Press.

Paul Lockhart (Ed.). 2009. *A Mathematician's Lament.* Bellevue Literary Press.

Rex Page and Ruben Gamboa. 2019. *Essential Logic for Computer Science.* The MIT Press.

George Polya. 1954a. *Induction and Analogy in Mathematics.* Mathematics and Plausible Reasoning, Vol. 1. Princeton University Press.

George Polya. 1954b. *Patterns of Plausible Inference.* Mathematics and Plausible Reasoning, Vol. 2. Princeton University Press.

George Polya. 1957. *How to solve it* (second ed.). Princeton University Press.

Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. 2017. Teaching Programming Languages by Experimental and Adversarial Thinking. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 1–9.

Leonard Q. Ross. 1937. *The Education of H*Y*M*A*N K*A*P*L*A*N.* Harcourt, Brace & World Inc.

Gian-Carlo Rota. 1996. *Indiscrete Thoughts.* Birkhaüser.

Raymond Smullyan. 2015. *Reflections.* World Scientific.

Terence Tao. 2006. *Solving Mathematical Problems: A Personal Perspective.* Oxford University Press.

Counting Immutable Beans

Reference Counting Optimized for Purely Functional Programming

Sebastian Ullrich
Karlsruhe Institute of Technology
Germany
sebastian.ullrich@kit.edu

Leonardo de Moura
Microsoft Research
USA
leonardo@microsoft.com

ABSTRACT

Most functional languages rely on some kind of garbage collection for automatic memory management. They usually eschew reference counting in favor of a tracing garbage collector, which has less bookkeeping overhead at runtime. On the other hand, having an exact reference count of each value can enable optimizations such as destructive updates. We explore these optimization opportunities in the context of an eager, purely functional programming language. We propose a new mechanism for efficiently reclaiming memory used by nonshared values, reducing stress on the global memory allocator. We describe an approach for minimizing the number of reference counts updates using borrowed references and a heuristic for automatically inferring borrow annotations. We implemented all these techniques in a new compiler for an eager and purely functional programming language with support for multi-threading. Our preliminary experimental results demonstrate our approach is competitive and often outperforms state-of-the-art compilers.

CCS CONCEPTS

• **Software and its engineering → Runtime environments**; **Garbage collection**;

KEYWORDS

purely functional programming, reference counting, Lean

ACM Reference Format:
Sebastian Ullrich and Leonardo de Moura. 2020. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Although reference counting [Collins 1960] (RC) is one of the oldest memory management techniques in computer science, it is not considered a serious garbage collection technique in the functional programming community, and there is plenty of evidence it is in general inferior to tracing garbage collection algorithms. Indeed, high-performance compilers such as ocamlopt and GHC use tracing garbage collectors. Nonetheless, implementations of several popular programming languages, e.g., Swift, Objective-C, Python, and Perl, use reference counting as a memory management technique. Reference counting is often praised for its simplicity, but many disadvantages are frequently reported in the literature [Jones and Lins 1996; Wilson 1992]. First, incrementing and decrementing reference counts every time a reference is created or destroyed can significantly impact performance because they not only take time but also affect cache performance, especially in a multi-threaded program [Choi et al. 2018]. Second, reference counting cannot collect circular [McBeth 1963] or self-referential structures. Finally, in most reference counting implementations, pause times are deterministic but may still be unbounded [Boehm 2004].

In this paper, we investigate whether reference counting is a competitive memory management technique for purely functional languages, and explore optimizations for reusing memory, performing destructive updates, and for minimizing the number of reference count increments and decrements. The former optimizations in particular are beneficial for purely functional languages that otherwise can only perform functional updates. When performing functional updates, objects often die just before the creation of an object of the same kind. We observe a similar phenomenon when we insert a new element into a pure functional data structure such as binary trees, when we use *map* to apply a given function to the elements of a list or tree, when a compiler applies optimizations by transforming abstract syntax trees, or when a proof assistant rewrites formulas. We call it the *resurrection hypothesis*: many objects die just before the creation of an object of the same kind. Our new optimization takes advantage of this hypothesis, and enables pure code to perform destructive updates in all scenarios described above when objects are not shared. We implemented all the ideas reported here in the new runtime and compiler for the Lean programming language [de Moura et al. 2015]. We also report preliminary experimental results that demonstrate our new compiler produces competitive code that often outperforms the code generated by high-performance compilers such as ocamlopt and GHC (Section 8).

Lean implements a version of the Calculus of Inductive Constructions [Coquand and Huet 1988; Coquand and Paulin 1990], and it has mainly been used as a proof assistant so far. Lean has a metaprogramming framework for writing proof and code automation, where reported here, but one can apply the techniques reported here to general-purpose functional programming languages. users can extend Lean using Lean itself [Ebner et al. 2017]. Improving the performance of Lean metaprograms was the primary motivation for the work

We describe our approach as a series of refinements starting from λ_{pure}, a simple intermediate representation for eager and purely functional languages (Section 3). We remark that in Lean and λ_{pure}, it is not possible to create cyclic data structures. Thus, one of the main criticisms against reference counting does not apply. From λ_{pure}, we obtain λ_{RC} by adding explicit instructions for incrementing (**inc**) and decrementing (**dec**) reference counts, and reusing

IFL'19, September 2019, Singapore
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

memory (Section 4). The inspiration for explicit RC instructions comes from the Swift compiler, as does the notion of *borrowed* references. In contrast to standard (or *owned*) references, of which there should be exactly as many as the object's reference counter implies, borrowed references do not update the reference counter but are *assumed* to be kept alive by a surrounding owned reference, further minimizing the number of **inc** and **dec** instructions in generated code.

We present a simple compiler from λ_{pure} to λ_{RC}, discussing heuristics for inserting destructive updates, borrow annotations, and **inc**/**dec** instructions (Section 5). Finally, we show that our approach is compatible with existing techniques for performing destructive updates on array and string values, and propose a simple and efficient approach for thread-safe reference counting (Section 7).

Contributions. We present a reference counting scheme optimized for and used by the next version of the Lean programming language.

- We describe how to reuse allocations in both user code and language primitives, and give a formal reference-counting semantics that can express this reuse.
- We describe the optimization of using borrowed references.
- We define a compiler that implements all these steps. The compiler is implemented in Lean itself and the source code is available.
- We give a simple but effective scheme for avoiding atomic reference count updates in multi-threaded programs.
- We compare the new Lean compiler incorporating these ideas with other compilers for functional languages and show its competitiveness.

2 EXAMPLES

In reference counting, each heap-allocated value contains a reference count. We view this counter as a collection of tokens. The **inc** instruction creates a new token and **dec** consumes it. When a function takes an argument as an owned reference, it is responsible for consuming one of its tokens. The function may consume the owned reference not only by using the **dec** instruction, but also by storing it in a newly allocated heap value, returning it, or passing it to another function that takes an owned reference. We illustrate our intermediate representation (IR) and the use of owned and borrowed references with a series of small examples.

The identity function *id* does not require any RC operation when it takes its argument as an owned reference.

id x = **ret** *x*

As another example, consider the function *mkPairOf* that takes x and returns the pair (x, x).

mkPairOf x = **inc** *x*; **let** *p* = *Pair x x*; **ret** *p*

It requires an **inc** instruction because two tokens for x are consumed (we will also say that "x is consumed" twice). The function *fst* takes two arguments x and y, and returns x, and uses a **dec** instruction for consuming the unused y.

fst x y = **dec** *y*; **ret** *x*

The examples above suggest that we do not need any RC operation when we take arguments as owned references and consume them exactly once. Now we contrast that with a function that only inspects its argument: the function *isNil xs* returns true if the list *xs* is empty and false otherwise. If the argument *xs* is taken as an owned reference, our compiler generates the following code

isNil xs = **case** *xs* **of**
  (*Nil* → **dec** *xs*; **ret** *true*)
  (*Cons* → **dec** *xs*; **ret** *false*)

We need the **dec** instructions because a function must consume all arguments taken as owned references. One may notice that decrementing xs immediately after we inspect its constructor tag is wasteful. Now assume that instead of taking the ownership of an RC token, we could borrow it from the caller. Then, the callee would not need to consume the token using an explicit **dec** operation. Moreover, the caller would be responsible for keeping the borrowed value alive. This is the essence of *borrowed references*: a borrowed reference does not actually keep the referenced value alive, but instead asserts that the value is kept alive by another, owned reference. Thus, when *xs* is a borrowed reference, we compile *isNil* into our IR as

isNil xs = **case** *xs* **of** (*Nil* → **ret** *true*) (*Cons* → **ret** *false*)

As a less trivial example, we now consider the function *hasNone xs* that, given a list of optional values, returns *true* if *xs* contains a *None* value. This function is often defined in a functional language as

hasNone [] = *false*
hasNone (*None* : *xs*) = *true*
hasNone (*Some x* : *xs*) = *hasNone xs*

Similarly to *isNil*, *hasNone* only inspects its argument. Thus if *xs* is taken as a borrowed reference, our compiler produces the following RC-free IR code for it

hasNone xs = **case** *xs* **of**
  (*Nil* → **ret** *false*)
  (*Cons* → **let** *h* = $\mathbf{proj}_{head}$ *xs*; **case** *h* **of**
    (*None* → **ret** *true*)
    (*Some* → **let** *t* = $\mathbf{proj}_{tail}$ *xs*; **let** *r* = *hasNone t*; **ret** *r*))

Note that our **case** operation does not introduce binders. Instead, we use explicit instructions $\mathbf{proj}_i$ for accessing the head and tail of the *Cons* cell. We use suggestive names for cases and fields in these initial examples, but will later use indices instead. Our borrowed inference heuristic discussed in Section 5 correctly tags *xs* as a borrowed parameter.

When using owned references, we know at run time whether a value is shared or not simply by checking its reference counter. We observed we could leverage this information and minimize the amount of allocated and freed memory for constructor values such as a list *Cons* value. Thus, we have added two additional instructions to our IR: **let** y = **reset** x and **let** z = (**reuse** y **in** $\mathbf{ctor}_i$ $\overline{w}$). The two instructions are used together; if x is a shared value, then y is set to a special reference ■, and the **reuse** instruction just allocates a new constructor value $\mathbf{ctor}_i$ $\overline{w}$. If x is not shared, then **reset** decrements the reference counters of the components of x, and y is set to x. Then, **reuse** reuses the memory cell used by x to store the constructor value $\mathbf{ctor}_i$ $\overline{w}$. We illustrate these two instructions

with the IR code for the list *map* function generated by our compiler as shown in Section 5. The code uses our actual, positional encoding of cases, constructors, and fields as described in the next section.

```
map f xs = case xs of
  (ret xs)
  (let x = proj_1 xs; inc x; let s = proj_2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor_2 y ys); ret r)
```

We remark that if the list referenced by *xs* is not shared, the code above does not allocate any memory. Moreover, if *xs* is a nonshared list of list of integers, then *map* (*map inc*) *xs* will not allocate any memory either. This example also demonstrates it is not a good idea, in general, to fuse **reset** and **reuse** into a single instruction: if we removed the **let** w = **reset** *xs* instruction and directly used *xs* in **reuse**, then when we execute the recursive application *map f s*, the reference counter for *s* would be greater than 1 even if the reference counter for *xs* was 1. We would have a reference from *xs* and another from *s*, and memory reuse would not occur in the recursive applications. Note that removing the **inc** *s* instruction is incorrect when *xs* is a shared value. Although the **reset** and **reuse** instructions can in general be used for reusing memory between two otherwise unrelated values, in examples like *map* where the reused value has a close semantic connection to the reusing value, we will use common functional vocabulary and say that the list is being *destructively updated* (up to the first shared cell).

As another example, a zipper is a technique for traversing and efficiently updating data structures, and it is particularly useful for purely functional languages. For example, the list zipper is a pair of lists, and it allows one to move forward and backward, and to update the current position. The *goForward* function is often defined as

```
goForward ([], bs) = ([], bs)
goForward (x : xs, bs) = (xs, x : bs)
```

In most functional programming languages, the second equation allocates a new pair and *Cons* value. The functions *map* and *goForward* both satisfy our resurrection hypothesis. Moreover, the result of a *goForward* application is often fed into another *goForward* or *goBackward* application. Even if the initial value was shared, every subsequent application takes a nonshared pair, and memory allocations are avoided by the code produced by our compiler.

```
goForward p = case p of
  (let xs = proj_1 p; inc xs;
   case xs of
     (ret p)
     (let bs = proj_2 p; inc bs;
      let c_1 = reset p;
      let x = proj_1 xs; inc x; xs' = proj_2 xs; inc xs';
      let c_2 = reset xs;
      let bs' = (reuse c_2 in ctor_2 x bs);
      let r = (reuse c_1 in ctor_1 xs' bs'); ret r))
```

3 THE PURE IR

Our source language λ_{pure} is a simple untyped functional intermediate representation (IR) in the style of A-normal form [Flanagan et al. 1993]. It captures the relevant features of the actual IR we have implemented and avoids unnecessary complexity that would only distract the reader from the ideas proposed here.

$$
\begin{array}{lll}
w, x, y, z \in \mathit{Var} & & \\
\quad c \in \mathit{Const} & & \\
\quad e \in \mathit{Expr} & ::= c\ \overline{y} \mid \texttt{pap}\ c\ \overline{y} \mid x\ y \mid \texttt{ctor}_i\ \overline{y} \mid \texttt{proj}_i\ x & \\
\quad F \in \mathit{FnBody} & ::= \texttt{ret}\ x \mid \texttt{let}\ x = e;\ F \mid \texttt{case}\ x\ \texttt{of}\ \overline{F} & \\
\quad f \in \mathit{Fn} & ::= \lambda\ \overline{y}.\ F & \\
\quad \delta \in \mathit{Program} & = \mathit{Const} \rightharpoonup \mathit{Fn} &
\end{array}
$$

All arguments of function applications are variables. The applied function is a constant c, with partial applications marked with the keyword **pap**, a variable x, the i-th constructor of an erased datatype, or the special function $\texttt{proj}_i$, which returns the i-th argument of a constructor application. Function bodies always end with evaluating and returning a variable. They can be chained with (non-recursive) **let** statements and branch using **case** statements, which evaluate to their i-th arm given an application of $\texttt{ctor}_i$. As further detailed in Section 5.3, we consider tail calls to be of the form **let** $r = c\ \overline{x}$; **ret** r. A program is a partial map from constant names to their implementations. The body of a constant's implementation may refer back to the constant, which we use to represent recursion, and analogously mutual recursion. In examples, we use $f\ \overline{x} = F$ as syntax sugar for $\delta(f) = \lambda\ \overline{x}.\ F$.

As an intermediate representation, we can and should impose restrictions on the structure of λ_{pure} to simplify working with it. We assume that

- all constructor applications are fully applied by eta-expanding them.
- no constant applications are over-applied by splitting them into two applications where necessary.
- all variable applications take only one argument, again by splitting them where necessary. While this simplification can introduce additional allocations of intermediary partial applications, it greatly simplifies the presentation of our operational semantics. All presented program transformations can be readily extended to a system with n-ary variable applications, which are handled analogously to n-ary constant applications.
- every function abstraction has been lambda-lifted to a top-level constant c.
- trivial bindings **let** $x = y$ have been eliminated through copy propagation.
- all dead **let** bindings have been removed.
- all parameter and **let** names of a function are mutually distinct. Thus we do not have to worry about name capture.

In the actual IR we have implemented[1], we also have instructions for storing and accessing unboxed data in constructor values, boxing and unboxing machine integers and scalar values, and creating literals of primitive types such as strings and numbers. Our

[1] https://github.com/leanprover/lean4/blob/IFL19/library/init/lean/compiler/ir/basic.lean

IR also supports *join points* similar to the ones used in the Haskell Core language [Maurer et al. 2017]. Join points are local function declarations that are never partially applied (i.e., they never occur in **pap** instructions), and are always tail-called. The actual IR has support for defining join points, and a **jmp** instruction for invoking them.

4 SYNTAX AND SEMANTICS OF THE REFERENCE-COUNTED IR

The target language λ_{RC} is an extension of λ_{pure}:

$$e \in Expr \quad ::= \ldots \mid \texttt{reset}\ x \mid \texttt{reuse}\ x\ \texttt{in}\ \texttt{ctor}_i\ \overline{y}$$
$$F \in FnBody ::= \ldots \mid \texttt{inc}\ x;\ F \mid \texttt{dec}\ x;\ F$$

We use the subscripts $_{pure}$ or $_{RC}$ (e.g., $Expr_{pure}$ or $Expr_{RC}$) to refer to the base or extended syntax, respectively, where otherwise ambiguous. The new expressions **reset** and **reuse** work together to reuse memory used to store constructor values, and, as discussed in Section 2, simulate destructive updates in constructor values.

We define the semantics of λ_{RC} (Figures 1 and 2) using a big-step relation $\rho \vdash \langle F, \sigma\rangle \Downarrow \langle l, \sigma'\rangle$ that maps the body F and a mutable *heap* σ under a context ρ to a location and the resulting heap. The context ρ maps variables to locations. A heap σ is a mapping from locations to pairs of values and reference counters. A value is a constructor value or a partially-applied constant. The reference counters of live values should always be positive; dead values are removed from the heap map.

$$l \in Loc$$
$$\rho \in Ctxt \quad = Var \rightharpoonup Loc$$
$$\sigma \in Heap \quad = Loc \rightharpoonup Value \times \mathbb{N}^+$$
$$v \in Value ::= \texttt{ctor}_i\ \overline{l} \mid \texttt{pap}\ c\ \overline{l}$$

When applying a variable, we have to be careful to increment the partial application arguments when copying them out of the **pap** cell, and to decrement the cell afterwards.[2] We cannot do so via explicit reference counting instructions because the number of arguments in a **pap** cell is not known statically.

Decrementing a unique reference removes the value from the heap and recursively decrements its components. **reset**, when used on a unique reference, eagerly decrements the components of the referenced value, replaces them with ■,[3] and returns the location of the now-invalid cell. This value is intended to be used only by **reuse** or **dec**. The former reuses it for a new constructor cell, asserting that its size is compatible with the old cell. The latter frees the cell, ignoring the replaced children.

If **reset** is used on a shared, non-reusable reference, it behaves like **dec** and returns ■, which instructs **reuse** to behave like **ctor**. Note that we cannot simply return the reference in both cases and do another uniqueness check in **reuse** because other code between the two expressions may have altered its reference count.

[2] If the **pap** reference is unique, the two steps can be coalesced so that the arguments do not have to be touched.

[3] which can be represented by any unused pointer value such as the null pointer in a real implementation. In our actual implementation, we avoid this memory write by introducing a `del` instruction that behaves like `dec` but ignores the constructor fields.

Const-App-Full
$$\frac{\delta(c) = \lambda\ \overline{y_c}.\ F \qquad \overline{l} = \overline{\rho(y)} \qquad [\overline{y_c} \mapsto \overline{l}] \vdash \langle F, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle c\ \overline{y}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Const-App-Part
$$\frac{\delta(c) = \lambda\ \overline{y_c}.\ F \qquad \overline{l} = \overline{\rho(y)} \qquad |\ \overline{l}\ | < |\ \overline{y_c}\ | \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \texttt{pap}\ c\ \overline{y}, \sigma\rangle \Downarrow \langle l', \sigma[l' \mapsto (\texttt{pap}\ c\ \overline{l}, 1)]\rangle}$$

Var-App-Full
$$\frac{\sigma(\rho(x)) = (\texttt{pap}\ c\ \overline{l}, \_) \qquad \delta(c) = \lambda\ \overline{y_c}.\ F \qquad l_y = \rho(y) \qquad [\overline{y_c} \mapsto \overline{l}\ l_y] \vdash \langle F, \mathrm{dec}(\rho(x), \mathrm{inc}(\overline{l}, \sigma))\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle x\ y, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Var-App-Part
$$\frac{\sigma(\rho(x)) = (\texttt{pap}\ c\ \overline{l}, \_) \qquad \delta(c) = \lambda\ \overline{y_c}.\ F \qquad l_y = \rho(y) \qquad |\ \overline{l}\ l_y\ | < |\ \overline{y_c}\ | \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle x\ y, \sigma\rangle \Downarrow \langle l', \mathrm{dec}(\rho(x), \mathrm{inc}(\overline{l}, \sigma))[l' \mapsto (\texttt{pap}\ c\ \overline{l}\ l_y, 1)]\rangle}$$

Ctor-App
$$\frac{\overline{l} = \overline{\rho(y)} \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \texttt{ctor}_i\ \overline{y}, \sigma\rangle \Downarrow \langle l', \sigma[l' \mapsto (\texttt{ctor}_i\ \overline{l}, 1)]\rangle}$$

Proj
$$\frac{\sigma(\rho(x)) = (\texttt{ctor}_j\ \overline{l}, \_) \qquad l' = \overline{l}_i}{\rho \vdash \langle \texttt{proj}_i\ x, \sigma\rangle \Downarrow \langle l', \sigma\rangle}$$

Return
$$\frac{\rho(x) = l}{\rho \vdash \langle \texttt{ret}\ x, \sigma\rangle \Downarrow \langle l, \sigma\rangle}$$

Let
$$\frac{\rho \vdash \langle e, \sigma\rangle \Downarrow \langle l, \sigma'\rangle \qquad \rho[x \mapsto l] \vdash \langle F, \sigma'\rangle \Downarrow \langle l', \sigma''\rangle}{\rho \vdash \langle \texttt{let}\ x = e;\ F, \sigma\rangle \Downarrow \langle l', \sigma''\rangle}$$

Case
$$\frac{\sigma(\rho(x)) = (\texttt{ctor}_i\ \overline{l}, \_) \qquad \rho \vdash \langle F_i, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle \texttt{case}\ x\ \texttt{of}\ \overline{F}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Figure 1: λ_{RC} semantics: the λ_{pure} fragment

5 A COMPILER FROM λ_{pure} TO λ_{RC}

Following the actual implementation of our compiler, we will discuss a compiler from λ_{pure} to λ_{RC} in three steps:

(1) Inserting **reset**/**reuse** pairs (Section 5.1)
(2) Inferring borrowed parameters (Section 5.2)
(3) Inserting **inc**/**dec** instructions (Section 5.3)

The first two steps are optional for obtaining correct λ_{RC} programs.

5.1 Inserting destructive update operations

In this subsection, we will discuss a heuristics-based implementation of a function

$$\delta_{reuse} : Const \to Fn_{RC}$$

that inserts **reset**/**reuse** instructions. Given $\texttt{let}\ z = \texttt{reset}\ x$, we remark that, in every control path, z may appear at most once, and in one of the following two instructions: $\texttt{let}\ y = \texttt{reuse}\ z\ \texttt{ctor}_i\ \overline{w}$, or $\texttt{dec}\ z$. We use $\texttt{dec}\ z$ for control paths where z cannot be reused. We implement the function δ_{reuse} as

$$\delta_{reuse}(c) = \lambda\ \overline{y}.\ R(F) \text{ where } \delta(c) = \lambda\ \overline{y}.\ F$$

The function $R(F)$ (Fig. 3) uses a simple heuristic for replacing $\texttt{ctor}_i\ \overline{y}$ expressions occurring in F with $\texttt{reuse}\ w\ \texttt{in}\ \texttt{ctor}_i\ \overline{y}$ where

$$\textsc{Inc}\ \frac{\rho \vdash \langle F, \mathrm{inc}(\rho(x), \sigma)\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle \mathbf{inc}\ x;\ F, \sigma\rangle \Downarrow \langle l', \sigma'\rangle} \qquad \textsc{Dec}\ \frac{\rho \vdash \langle F, \mathrm{dec}(\rho(x), \sigma)\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle \mathbf{dec}\ x;\ F, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

$$\mathrm{inc}(l, \sigma) = \sigma[l \mapsto (v, i+1)] \quad \text{if } \sigma(l) = (v, i)$$

$$\mathrm{inc}(l\ \overline{l'}, \sigma) = \mathrm{inc}(\overline{l'}, \mathrm{inc}(l, \sigma))$$

$$\mathrm{dec}(l, \sigma) = \begin{cases} \sigma & \text{if } l = \blacksquare \\ \sigma[l \mapsto (v, i-1)] & \text{if } \sigma(l) = (v, i), i > 1 \\ \mathrm{dec}(\overline{l'}, \sigma[l \mapsto \bot]) & \text{if } \sigma(l) = (\mathbf{pap}\ c\ \overline{l'}, 1) \\ \mathrm{dec}(\overline{l'}, \sigma[l \mapsto \bot]) & \text{if } \sigma(l) = (\mathbf{ctor}_i\ \overline{l'}, 1) \end{cases}$$

$$\mathrm{dec}(l\ \overline{l'}, \sigma) = \mathrm{dec}(\overline{l'}, \mathrm{dec}(l, \sigma))$$

$$\textsc{Reset-Uniq}\ \frac{\rho(x) = l \qquad \sigma(l) = (\mathbf{ctor}_i\ \overline{l'}, 1)}{\rho \vdash \langle \mathbf{reset}\ x, \sigma\rangle \Downarrow \langle l, \mathrm{dec}(\overline{l'}, \sigma[l \mapsto (\mathbf{ctor}_i\ \blacksquare^{|\overline{l'}|}, 1)])\rangle}$$

$$\textsc{Reset-Shared}\ \frac{\rho(x) = l \qquad \sigma(l) = (\_, i) \qquad i \neq 1}{\rho \vdash \langle \mathbf{reset}\ x, \sigma\rangle \Downarrow \langle \blacksquare, \mathrm{dec}(l, \sigma)\rangle}$$

$$\textsc{Reuse-Uniq}\ \frac{\rho(x) = l \qquad \sigma(l) = (\mathbf{ctor}_j\ \blacksquare^{|\overline{y}|}, 1) \qquad \overline{\rho(y)} = \overline{l''}}{\rho \vdash \langle \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y}, \sigma\rangle \Downarrow \langle l, \sigma[l \mapsto (\mathbf{ctor}_i\ \overline{l''}, 1)]\rangle}$$

$$\textsc{Reuse-Shared}\ \frac{\rho(x) = \blacksquare \qquad \rho \vdash \langle \mathbf{ctor}_i\ \overline{y}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

Figure 2: λ_{RC} semantics cont.

$$R : \mathit{FnBody}_{pure} \to \mathit{FnBody}_{RC}$$
$$R(\mathbf{let}\ x = e;\ F) = \mathbf{let}\ x = e;\ R(F)$$
$$R(\mathbf{ret}\ x) = \mathbf{ret}\ x$$
$$R(\mathbf{case}\ x\ \mathbf{of}\ \overline{F}) = \mathbf{case}\ x\ \mathbf{of}\ \overline{D(x, n_i, R(F_i))}$$
where n_i = #fields of x in i-th branch

$$D : \mathit{Var} \times \mathbb{N} \times \mathit{FnBody}_{RC} \to \mathit{FnBody}_{RC}$$
$$D(z, n, \mathbf{case}\ x\ \mathbf{of}\ \overline{F}) = \mathbf{case}\ x\ \mathbf{of}\ \overline{D(z, n, F)}$$
$$D(z, n, \mathbf{ret}\ x) = \mathbf{ret}\ x$$
$$D(z, n, \mathbf{let}\ x = e;\ F) = \mathbf{let}\ x = e;\ D(z, n, F)$$
if $z \in e$ or $z \in F$
$$D(z, n, F) = \mathbf{let}\ w = \mathbf{reset}\ z;\ S(w, n, F)$$
otherwise, if $S(w, n, F) \neq F$ for a fresh w
$$D(z, n, F) = F \quad \text{otherwise}$$

$$S : \mathit{Var} \times \mathbb{N} \times \mathit{FnBody}_{RC} \to \mathit{FnBody}_{RC}$$
$$S(w, n, \mathbf{let}\ x = \mathbf{ctor}_i\ \overline{y};\ F) = \mathbf{let}\ x = \mathbf{reuse}\ w\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y};\ F$$
if $|\ \overline{y}\ | = n$
$$S(w, n, \mathbf{let}\ x = e;\ F) = \mathbf{let}\ x = e;\ S(w, n, F) \quad \text{otherwise}$$
$$S(w, n, \mathbf{ret}\ x) = \mathbf{ret}\ x$$
$$S(w, n, \mathbf{case}\ x\ \mathbf{of}\ \overline{F}) = \mathbf{case}\ x\ \mathbf{of}\ \overline{S(w, n, F)}$$

Figure 3: Inserting `reset`/`reuse` pairs

w is a fresh variable introduced by R as the result of a new **`reset`** operation. For each arm F_i in a **`case`** x **`of`** $\overline{F}$ operation, the function R requires the arity n of the corresponding matched constructor. In the actual implementation, we store this information for each arm when we compile our typed frontend language into λ_{pure}. The auxiliary functions D and S implement the *dead* variable search and *substitution* steps respectively. For each **`case`** operation, R attempts to insert **`reset`**/**`reuse`** instructions for the variable matched by the **`case`**. This is done using D in each arm of the **`case`**. Function $D(z, n, F)$ takes as parameters the variable z to reuse and the arity n of the matched constructor. D proceeds to the first location where z is dead, i.e. not used in the remaining function body, and then uses S to attempt to find and substitute a *matching* constructor $\mathbf{ctor}_i\ \overline{y}$ instruction with a **`reuse`** w **`in`** $\mathbf{ctor}_i\ \overline{y}$ in the remaining code. If no matching constructor instruction can be found, D does not modify the function body.

As an example, consider the *map* function for lists

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; let s := proj2 xs;
   let y = f x; let ys = map f s;
   let r = ctor2 y ys; ret r)
```

Applying R to the body of *map* , we have D looking for opportunities to **`reset`**/**`reuse`** *xs* in both **`case`** arms. Since *xs* is unused after **`let`** s = $\mathbf{proj}_2$ *xs*, S is applied to the rest of the function, looking for constructor calls with two parameters. Indeed, such a call can be found in the let-binding for r. Thus, function D successfully inserts the appropriate instructions, and we obtain the function described in Section 2. Now, consider the list *swap* function that swaps the first two elements of a list. It is often defined as

```
swap [] = []
swap [x] = [x]
swap (x : y : zs) = y : x : zs
```

In λ_{pure}, this function is encoded as

```
swap xs = case xs of
  (ret xs)
  (let t1 = proj2 xs; case t1 of
    (ret xs)
    (let h1 = proj1 xs;
     let h2 = proj1 t1; let t2 = proj2 t1;
     let r1 = ctor2 h1 t2; let r2 = ctor2 h2 r1; ret r2))
```

By applying R to *swap*, we obtain

```
swap xs = case xs of
  (ret xs)
  (let t1 = proj2 xs; case t1 of
    (ret xs)
    (let h1 = proj1 xs; let w1 = reset xs;
     let h2 = proj1 t1 ; let t2 = proj2 t1 ;
     let w2 = reset t1 ; let r1 = reuse w2 in ctor2 h1 t2 ;
     let r2 = reuse w1 in ctor2 h2 r1 ; ret r2))
```

Similarly to the *map* function, the code generated for the function *swap* will *not* allocate any memory when the list value is not shared. This example demonstrates that our heuristic procedure can avoid memory allocations even in functions containing many nested **case** instructions. The example also makes it clear that we could further optimize our λ_{RC} by adding additional instructions. For example, we can add an instruction that combines **reset** and **reuse** into a single instruction and is used in situations where **reuse** occurs *immediately* after the corresponding **reset** instruction such as in the example above where we have **let** w_2 = **reset** t_1; **let** r_1 = **reuse** w_2 **in ctor**$_2$ h_1 t_2,

5.2 Inferring borrowing signatures

We now consider the problem of inferring borrowing signatures, i.e. a mapping $\beta : Const \rightharpoonup \{\mathbb{O}, \mathbb{B}\}^*$, which for every function should return a list describing each parameter of the function as either $\mathbb{O}$wned or $\mathbb{B}$orrowed. Borrow annotations can be provided manually by users (which is always safe), but we have two motivations for inferring them: avoiding the burden of annotations, and making our IR a convenient target for other systems (e.g., Coq, Idris, and Agda) that do not have borrow annotations.

If a function f takes a parameter x as a borrowed reference, then at runtime x may be a shared value even when its reference counter is 1. Thus, we must never mark x as borrowed if it is used by a **let** y = **reset** x instruction. We also assume that each $\beta(c)$ has the same length as the corresponding parameter list in $\delta(c)$.

Partially applying constants with borrowed parameters is also problematic because, in general, we cannot statically assert that the resulting value will not escape the current function and thus the scope of borrowed references. Therefore we extend δ_{reuse} to the program δ_β by defining a trivial wrapper constant $c_\mathbb{O} := c$ (we will assume that this name is fresh) for any such constant c, set $\beta(c_\mathbb{O}) := \overline{\mathbb{O}}$, and replace any occurrence of **pap** c $\overline{y}$ with **pap** $c_\mathbb{O}$ $\overline{y}$. The compiler step given in the next subsection will, as part of the general transformation, insert the necessary **inc** and **dec** instructions into $c_\mathbb{O}$ to convert between the two signatures.

Our heuristic is based on the fact that when we mark a parameter as borrowed, we reduce the number of RC operations needed, but we also prevent **reset** and **reuse** as well as primitive operations from reusing memory cells. Our heuristic collects which parameters and variables should be owned. We say a parameter x should be owned if x or one of its projections is used in a **reset**, or is passed to a function that takes an owned reference. The latter condition is a heuristic and is not required for correctness. We use it because the function taking an owned reference may try to reuse its memory cell. A formal definition is given in Fig. 4. Many refinements are possible, and we discuss one of them in the next section. Note that if a call is recursive, we do not know which parameters are owned, yet. Thus, given $\delta(c) = \lambda\overline{y}.\ b$, we infer the value of $\beta(c)$ by starting with the approximation $\beta(c) = \mathbb{B}^n$, then we compute $S = collect_\mathbb{O}(b)$, update $\beta(c)_i := \mathbb{O}$ if $y_i \in S$, and repeat the process until we reach a fix point and no further updates are performed on $\beta(c)$. The procedure described here does not consider mutually recursive definitions, but this is a simple extension where we process a block of mutually recursive functions simultaneously. By applying our heuristic to the *hasNone* function described before, we obtain $\beta(hasNone) = \mathbb{B}$. That is, in an application *hasNone xs*, *xs* is taken as a borrowed reference.

$$
\begin{aligned}
&collect_\mathbb{O} : FnBody_{RC} \to 2^{Vars}\\
&collect_\mathbb{O}(\textbf{let } z = \textbf{ctor}_i\ \overline{x}\ ;\ F) = collect_\mathbb{O}(F)\\
&collect_\mathbb{O}(\textbf{let } z = \textbf{reset } x\ ;\ F) = collect_\mathbb{O}(F) \cup \{x\}\\
&collect_\mathbb{O}(\textbf{let } z = \textbf{reuse } x \textbf{ in ctor}_i\ \overline{x}\ ;\ F) = collect_\mathbb{O}(F)\\
&collect_\mathbb{O}(\textbf{let } z = c\ \overline{x}\ ;\ F) = collect_\mathbb{O}(F) \cup \{x_i \in \overline{x} \mid \beta(c)_i = \mathbb{O}\}\\
&collect_\mathbb{O}(\textbf{let } z = x\ y;\ F) = collect_\mathbb{O}(F) \cup \{x, y\}\\
&collect_\mathbb{O}(\textbf{let } z = \textbf{pap } c_\mathbb{O}\ \overline{x};\ F) = collect_\mathbb{O}(F) \cup \{\overline{x}\}\\
&collect_\mathbb{O}(\textbf{let } z = \textbf{proj}_i\ x;\ F) = collect_\mathbb{O}(F) \cup \{x\} \text{ if } z \in collect_\mathbb{O}(F)\\
&collect_\mathbb{O}(\textbf{let } z = \textbf{proj}_i\ x;\ F) = collect_\mathbb{O}(F) \text{ if } z \notin collect_\mathbb{O}(F)\\
&collect_\mathbb{O}(\textbf{ret } x) = \emptyset\\
&collect_\mathbb{O}(\textbf{case } x \textbf{ of } \overline{F}) = \bigcup_{F_i \in \overline{F}} collect_\mathbb{O}(F_i)
\end{aligned}
$$

Figure 4: Collecting variables that should not be marked as borrowed

5.3 Inserting reference counting operations

Given any well-formed definition of β and δ_β, we finally give a procedure for correctly inserting **inc** and **dec** instructions.[4]

$$
\begin{aligned}
&\delta_{RC}(c) : Const \to Fn_{RC}\\
&\delta_{RC}(c) = \lambda\ \overline{y}.\ \mathbb{O}^-(\overline{y}, C(F, \beta_l)) \quad \text{where } \delta_\beta(c) = \lambda\ \overline{y}.\ F,\\
&\qquad\qquad \beta_l = [\overline{y} \mapsto \beta(c), \ldots \mapsto \mathbb{O}]
\end{aligned}
$$

The map $\beta_l : Var \to \{\mathbb{O}, \mathbb{B}\}$ keeps track of the borrow status of each local variable. For simplicity, we default all missing entries to $\mathbb{O}$.

In general, variables should be incremented prior to being used in an *owned context* that consumes an RC token. Variables used in any other (*borrowed*) context do not need to be incremented. Owned references should be decremented after their last use. We use the following two helper functions to conditionally add RC instructions (Fig. 5) in these contexts:

- $\mathbb{O}^+_x$ prepares x for usage in an owned context by incrementing it. The increment can be omitted on the last use of an owned variable, with V representing the set of live variables after the use.

$$
\begin{aligned}
&\mathbb{O}^+_x(V, F, \beta_l) = F && \text{if } \beta_l(x) = \mathbb{O} \wedge x \notin V\\
&\mathbb{O}^+_x(V, F, \beta_l) = \textbf{inc } x;\ F && \text{otherwise}
\end{aligned}
$$

[4] We will tersely say that a variable x "is incremented/decremented" when an **inc**/**dec** operation is applied to it, i.e. the RC of the referenced object is incremented/decremented at runtime.

$$
\begin{array}{l}
C : \mathit{FnBody}_{RC} \times (\mathit{Var} \to \{\mathbb{O}, \mathbb{B}\}) \to \mathit{FnBody}_{RC} \\
C\,(\texttt{ret}\ x,\ \beta_l) = \mathbb{O}^+_x(\emptyset, \texttt{ret}\ x,\ \beta_l) \\
C\,(\texttt{case}\ x\ \texttt{of}\ \overline{F},\ \beta_l) = \texttt{case}\ x\ \texttt{of}\ \overline{\mathbb{O}^-(\overline{y}, C(F, \beta_l), \beta_l)} \\
\qquad\qquad \text{where } \{\overline{y}\} = \mathrm{FV}(\texttt{case}\ x\ \texttt{of}\ \overline{F}) \\
C\,(\texttt{let}\ y = \texttt{proj}_i\ x;\ F,\ \beta_l) = \texttt{let}\ y = \texttt{proj}_i\ x;\ \texttt{inc}\ y;\ \mathbb{O}^-_x(C(F, \beta_l), \beta_l) \\
\qquad\qquad \text{if } \beta_l(x) = \mathbb{O} \\
C\,(\texttt{let}\ y = \texttt{proj}_i\ x;\ F,\ \beta_l) = \texttt{let}\ y = \texttt{proj}_i\ x;\ C\,(F, \beta_l[y \mapsto \mathbb{B}]) \\
\qquad\qquad \text{if } \beta_l(x) = \mathbb{B} \\
C\,(\texttt{let}\ y = \texttt{reset}\ x;\ F,\ \beta_l) = \texttt{let}\ y = \texttt{reset}\ x;\ C\,(F, \beta_l) \\
C\,(\texttt{let}\ z = c\ \overline{y}\ ;\ F,\ \beta_l) = C_{app}(\overline{y}, \beta(c), \texttt{let}\ z = c\ \overline{y};\ C(F, \beta_l), \beta_l) \\
C\,(\texttt{let}\ z = \texttt{pap}\ c\ \overline{y}\ ;\ F,\ \beta_l) = C_{app}(\overline{y}, \beta(c), \texttt{let}\ z = \texttt{pap}\ c\ \overline{y};\ C(F, \beta_l), \beta_l) \\
C\,(\texttt{let}\ z = x\ y;\ F,\ \beta_l) = C_{app}(x\ y, \mathbb{O}\ \mathbb{O}, \texttt{let}\ z = x\ y;\ C(F, \beta_l), \beta_l) \\
C\,(\texttt{let}\ z = \texttt{ctor}_i\ \overline{y}\ ;\ F,\ \beta_l) = C_{app}(\overline{y}, \overline{\mathbb{O}}, \texttt{let}\ z = \texttt{ctor}_i\ \overline{y};\ C(F, \beta_l), \beta_l) \\
C\,(\texttt{let}\ z = \texttt{reuse}\ x\ \texttt{in}\ \texttt{ctor}_i\ \overline{y}\ ;\ F,\ \beta_l) = \\
\quad C_{app}(\overline{y}, \overline{\mathbb{O}}, \texttt{let}\ z = \ \texttt{reuse}\ x\ \texttt{in}\ \texttt{ctor}_i\ \overline{y};\ C(F, \beta_l), \beta_l)
\end{array}
$$

$$
\begin{array}{l}
C_{app} : \mathit{Var}^n \times \{\mathbb{O}, \mathbb{B}\}^n \times \mathit{FnBody}_{RC} \times (\mathit{Var} \to \{\mathbb{O}, \mathbb{B}\}) \to \mathit{FnBody}_{RC} \\
C_{app}(y\ \overline{y'}, \mathbb{O}\ \overline{b}, \texttt{let}\ z = e;\ F, \beta_l) = \\
\quad \mathbb{O}^+_y(\overline{y'} \cup \mathrm{FV}(F), C_{app}(\overline{y'}, \overline{b}, \texttt{let}\ z = e;\ F, \beta_l), \beta_l) \\
C_{app}(y\ \overline{y'}, \mathbb{B}\ \overline{b}, \texttt{let}\ z = e;\ F, \beta_l) = \\
\quad C_{app}(\overline{y'}, \overline{b}, \texttt{let}\ z = e;\ \mathbb{O}^-_y(F, \beta_l), \beta_l) \\
C_{app}([], [], \texttt{let}\ z = e;\ F, \beta_l) = \texttt{let}\ z = e;\ F
\end{array}
$$

Figure 5: Inserting `inc`/`dec` instructions

- $\mathbb{O}^-_x$ decrements x if it is both owned and dead. $\mathbb{O}^-(\overline{x}, F, \beta_l)$ decrements multiple variables, which may be needed at the start of a function or **`case`** branch.

$$
\begin{array}{ll}
\mathbb{O}^-_x(F, \beta_l) = \texttt{dec}\ x;\ F & \text{if } \beta_l(x) = \mathbb{O} \wedge x \notin \mathrm{FV}(F) \\
\mathbb{O}^-_x(F, \beta_l) = F & \text{otherwise} \\
\mathbb{O}^-(x\ \overline{x'}, F, \beta_l) = \mathbb{O}^-(\overline{x'}, \mathbb{O}^-_x(F, \beta_l), \beta_l) & \\
\mathbb{O}^-([], F, \beta_l) = F &
\end{array}
$$

Applications are handled separately, recursing over the arguments and parameter borrow annotations in parallel; for partial, variable and constructor applications, the latter default to $\mathbb{O}$.

Examples

We demonstrate the behavior of the compiler on two application special cases. The value of β_l is constant in these examples and left implicit in applications.

(1) Consuming the same argument multiple times

$$
\begin{array}{l}
\beta(\mathsf{c}) := \mathbb{O}\ \mathbb{O} \\
\quad \beta_l := [\mathsf{y} \mapsto \mathbb{O}] \\
C(\texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = C_{app}(\mathsf{y}\ \mathsf{y}, \mathbb{O}\ \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ C(\texttt{ret}\ \mathsf{z})) \\
\quad = C_{app}(\mathsf{y}\ \mathsf{y}, \mathbb{O}\ \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, C_{app}(\mathsf{y}, \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z})) \\
\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{z}\}, C_{app}([], [], \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}))) \\
\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{z}\}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z})) \\
\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = \texttt{inc}\ \mathsf{y};\ \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}
\end{array}
$$

Because y is dead after the call, it needs to be incremented only once, *moving* its last token to c instead.

(2) Borrowing and consuming the same argument

$$
\begin{array}{l}
\beta(\mathsf{c}) := \mathbb{B}\ \mathbb{O} \\
\quad \beta_l := [\mathsf{y} \mapsto \mathbb{O}] \\
C(\texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = C_{app}(\mathsf{y}\ \mathsf{y}, \mathbb{B}\ \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ C(\texttt{ret}\ \mathsf{z})) \\
\quad = C_{app}(\mathsf{y}\ \mathsf{y}, \mathbb{B}\ \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = C_{app}(\mathsf{y}, \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \mathbb{O}^-_{\mathsf{y}}(\texttt{ret}\ \mathsf{z})) \\
\quad = C_{app}(\mathsf{y}, \mathbb{O}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{dec}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\},\ C_{app}([], [], \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{dec}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z})) \\
\quad = \mathbb{O}^+_{\mathsf{y}}(\{\mathsf{y}, \mathsf{z}\}, \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{dec}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}) \\
\quad = \texttt{inc}\ \mathsf{y};\ \texttt{let}\ \mathsf{z} = \mathsf{c}\ \mathsf{y}\ \mathsf{y};\ \texttt{dec}\ \mathsf{y};\ \texttt{ret}\ \mathsf{z}
\end{array}
$$

Even though the owned parameter comes after the borrowed parameter, the presence of y in the **`dec`** instruction emitted when handling the first parameter makes sure we do not accidentally move ownership when handling the second parameter, but copy y by emitting an **`inc`** instruction.

Preserving tail calls

A *tail call* $\texttt{let}\ r = c\ \overline{x};\ \texttt{ret}\ r$ is an application followed by a **`ret`** instruction. Recursive tail calls are implemented using *goto*s in our compiler backend. Thus, it is highly desirable to preserve them as we transform λ_{pure} into λ_{RC}. However, the previous example shows that our function for inserting reference counting instructions may insert **`dec`** instructions after a constant application, and consequently, destroy tail calls. A **`dec`** instruction is inserted after a constant application $\texttt{let}\ r = c\ \overline{x}$ if $\beta(c)_i = \mathbb{B}$ and $\beta_l(x_i) = \mathbb{O}$ for some $x_i \in \overline{x}$. That is, function c takes the i-th parameter as a borrowed reference, but the actual argument is owned. As an example, consider the following function in λ_{pure}.

$$
\begin{array}{l}
f\,x = \texttt{case}\ x\ \texttt{of} \\
\quad (\texttt{let}\ r = \texttt{proj}_1\ ;\ \texttt{ret}\ r) \\
\quad (\texttt{let}\ y_1 = \texttt{ctor}_1\ ;\ \texttt{let}\ y_2 = \texttt{ctor}_1\ y_1\ ;\ \texttt{let}\ r = f\,y_2\ ;\ \texttt{ret}\ r)
\end{array}
$$

The compiler from λ_{pure} to λ_{RC}, infers $\beta(f) = \mathbb{B}$, and produces

$$
\begin{array}{l}
f\,x = \texttt{case}\ x\ \texttt{of} \\
\quad (\texttt{let}\ r = \texttt{proj}_1\ x;\ \texttt{inc}\ r;\ \texttt{ret}\ r)
\end{array}
$$

```
(let y1 = ctor1 ; let y2 ctor1 y1 ;
 let r = f y2 ; dec y2 ; ret r)
```

which does not preserve the tail call `let r = f y2; ret r`. We addressed this issue in our real implementation by refining our borrowing inference heuristic, marking $\beta(c)_i = \mathbb{O}$ whenever c occurs in a tail call `let r = c x̄; ret r` where $\beta_l(x_i) = \mathbb{O}$. This small modification guarantees that tail calls are preserved by our compiler, and the following λ_{RC} code is produced for f instead

```
f x = case x of
  (let r = proj1 x; inc r; dec x; ret r)
  (dec x; let y1 = ctor1 ; let y2 = ctor1 y1 ;
   let r = f y2 ; ret r)
```

6 OPTIMIZING FUNCTIONAL DATA STRUCTURES FOR `reset/reuse`

In the previous section, we have shown how to automatically insert **reset** and **reuse** instructions that minimize the number of memory allocations at execution time. We now discuss techniques we have been using for taking advantage of this transformation when writing functional code. Two fundamental questions when using this optimization are: Does a **reuse** instruction now guard my constructor applications? Given a `let y = reset x` instruction, how often is x not shared at runtime? We address the first question using a simple static analyzer that when invoked by a developer, checks whether **reuse** instructions are guarding constructor applications in a particular function. This kind of analyzer is straightforward to implement in Lean since our IR is a Lean inductive datatype. This kind of analyzer is in the same spirit of the *inspection-testing* package available for GHC [Breitner 2018]. We cope with the second question using runtime instrumentation. For each `let y = reset x` instruction, we can optionally emit two counters that track how often x is shared or not. We have found these two simple techniques quite useful when optimizing our own code. Here, we report one instance that produced a significant performance improvement.

Red-black trees

Red-black trees are implemented in the Lean standard library and are often used to write proof automation. For the purposes of this section, it is sufficient to have an abstract description of this kind of tree, and one of the re-balancing functions used by the insertion function.

```
Color = R | B
Tree a = E | T Color (Tree a) a (Tree a)
balance1 v t (T _ (T R l x r1) y r2) = T R (T B l x r1) y (T B r2 v t)
balance1 v t (T _ l1 y (T R l2 x r)) = T R (T B l1 y l2) x (T B r v t)
balance1 v t (T _ l y r) = T B (T R l y r) v t
insert (T B a y b) x = balance1 y b (insert a x) if x < y and a is red
...
```

Note that the first two $balance_1$ equations create three T constructor values, but the patterns on the left-hand side use only two T constructors. Thus, the generated IR for $balance_1$ contains T constructor applications that are not guarded by *reuse*, and this fact can be detected at compilation time. Note that even if the result of (*insert a x*) contains only nonshared values, we still have to allocate one constructor value. We can avoid this unnecessary memory allocation by inlining $balance_1$. After inlining, the input value (*T B a y b*) is reused in the $balance_1$ code. The final generated code now contains a single constructor application that is not guarded by a *reuse*, the one for the equation:

```
insert E x = T R E x E
```

The generated code now has the property that if the input tree is not shared, then only a single new node is allocated. Moreover, even if the input tree is shared we have observed a positive performance impact using **reset** and **reuse**. The recursive call (*insert a x*) always returns a nonshared node even if x is shared. Thus, $balance_1$ *y b* (*insert a x*) always reuses at least one memory cell at runtime.

There is another way to avoid the unnecessary memory allocation that does not rely on inlining. We can chain the T constructor value from *insert* to $balance_1$. We accomplish this by rewriting $balance_1$ and *insert* as follows

```
balance1 (T _ _ v t) (T _ (T R l x r1) y r2) = T R (T B l x r1) y (T B r2 v t)
balance1 (T _ _ v t) (T _ l1 y (T R l2 x r)) = T R (T B l1 y l2) x (T B r v t)
balance1 (T _ _ v t) (T _ l y r) = T B (T R l y r) v t
insert (T B a y b) x = balance1 (T B E y b) (insert a x) if x < y and a is red
```

Now, the input value (*T B a y b*) is reused to create value (*T B E y b*) which is passed to $balance_1$. Note that we have replaced a with E to make sure the recursive application (*insert a x*) may also perform destructive updates if a is not shared. This simple modification guarantees that $balance_1$ does not allocate memory when the input trees are not shared.

7 RUNTIME IMPLEMENTATION

7.1 Values

In our runtime, every value starts with a header containing two tags. The first tag specifies the value kind: **ctor**, **pap**, **array**, **string**, **num**, **thunk**, or **task**. The second tag specifies whether the value is single-threaded, multi-threaded, or *persistent*. We will describe how this kind is used to implement thread safe reference counting in the next subsection. The kinds **ctor** and **pap** are used to implement the corresponding values used in the formal semantics of λ_{pure} and λ_{RC}. The kinds **array**, **string**, and **thunk** are self explanatory. The kind **num** is for arbitrary precision numbers implemented using the GNU multiple precision library (GMP). The **task** value is described in the next subsection.

Values tagged as single- or multi-threaded also contain a reference counter. This counter is stored in front of the standard value header. We will primarily focus on the layout of **ctor** values here because it is the most relevant one for the ideas presented in this paper. A $\texttt{ctor}_i$ value header also includes the constructor index i, the number of pointers to other values and/or boxed values, and the number of bytes used to store scalar unboxed values such as machine integers and enumeration types. In a 64-bit machine, the **ctor** value header is 16 bytes long, twice the size of the header used in OCaml to implement the corresponding kind of value. After the header, we store all pointers to other values and boxed values,

and then all unboxed values. Thus, in a 64 bit machine, our runtime uses 32 bytes to implement a *List Cons* value: 16 bytes for the header, and 16 bytes for storing the list head and tail. The unboxed value support has restrictions similar to the ones found in GHC. For example, to pass an unboxed value to a polymorphic function we must first box it.

Our runtime has built-in support for array and string operations. Strings are just a special case of arrays where the elements are characters. We perform destructive updates when the array is not shared. For example, given the array write primitive

$$Array.write : Array\ \alpha \to Nat \to \alpha \to Array\ \alpha$$

the function application $Array.write\ a\ i\ v$ will destructively update and return the array a if it is not shared. This is a well known optimization for systems based on reference counting [Jones and Lins 1996], nonetheless we mention it here because it is relevant for many applications. Moreover, destructive array updates and our **reset**/**reuse** technique complement each other. As an example, if we have a nonshared list of integer arrays xs, $map\ (Array.map\ inc)\ xs$ destructively updates the list and all arrays. In the experimental section we demonstrate that our pure quick sort is as efficient as the quick sort using destructive updates in OCaml, and the quick sort using the primitive ST monad in Haskell.

7.2 Thread safety

We use the following basic task management primitives to develop the Lean frontend.

$$Task.mk : (Unit \to \alpha) \to Task\ \alpha$$
$$Task.bind : Task\ \alpha \to (\alpha \to Task\ \beta) \to Task\ \beta$$
$$Task.get : Task\ \alpha \to \alpha$$

The function *Task.mk* converts a closure into a **task** value and executes it in a separate thread, $Task.bind\ t\ f$ creates a **task** value that waits for t to finish and produce result a, and then starts $f\ a$ and waits for it to finish. Finally, $Task.get\ t$ waits for t to finish and returns the value produced by it. These primitives are part of the Lean runtime, implemented in C++, and are available to regular users.

The standard way of implementing thread safe reference counting uses memory fences [Schling 2011]. The reference counters are incremented using an atomic fetch and add operation with a relaxed memory order. The relaxed memory order can be used because new references to a value can only be formed from an existing reference, and passing an existing reference from one thread to another must already provide any required synchronization. When decrementing a reference counter, it is important to enforce that any decrements of the counter from other threads are visible before checking if the object should be deleted. The standard way of achieving this effect uses a *release* operation after dropping a reference, and an *acquire* operation before the deletion check. This approach has been used in the previous version of the Lean compiler, and we have observed that the memory fences have a significant performance impact even when only one thread is being executed. This is quite unfortunate because most values are only touched by a single execution thread.

We have addressed this performance problem in our runtime by tagging values as *single-threaded*, *multi-threaded*, or *persistent*. As the name suggests, a single-threaded value is accessed by a single thread and a multi-threaded one by one or more threads. If a value is tagged as single-threaded, we do not use any memory fence for incrementing or decrementing its reference counter. Persistent values are never deallocated and do not even need a reference counter. We use persistent values to implement values that are created at program initialization time and remain alive until program termination. Our runtime enforces the following invariant: from persistent values, we can only reach other persistent values, and from multi-threaded values, we can only reach persistent or multi-threaded values. There are no constraints on the kind of value that can be reached from a single-threaded value. By default, values are single-threaded, and our runtime provides a $markMT(o)$ procedure that tags all single-threaded values reachable from o as multi-threaded. This procedure is used to implement $Task.mk\ f$ and $Task.bind\ x\ f$. We use $markMT(f)$ and $markMT(x)$ to ensure that all values reachable from these values are tagged as multi-threaded *before* we create a new task, that is, while they are still accessible from only one thread. Our invariant ensures that *markMT* does not need to visit values reachable from a value already tagged as multi-threaded. Thus values are visited at most once by *markMT* during program execution. Note that task creation is not a constant time operation in our approach because it is proportional to the number of single-threaded values reachable from x and f. This does not seem to be a problem in practice, but if it becomes an issue we can provide a primitive $asMT\ g$ that ensures that all values allocated when executing g are immediately tagged as multi-threaded. Users would then use this flag in code that creates the values reachable by $Task.mk\ f$ and $Task.bind\ x\ f$.

The reference counting operations perform an extra operation to test the value tag and decide whether a memory fence is needed or not. This additional test does not require any synchronization because the tag is only modified before a value is shared with other execution threads. In the experimental section, we demonstrate that this simple approach significantly boosts performance. This is not surprising because the additional test is much cheaper than memory fences on modern hardware. The approach above can be adapted to more complex libraries for writing multi-threaded code. We just need to identify which functions may send values to other execution threads, and use *markMT*.

8 EXPERIMENTAL EVALUATION

We have implemented the RC optimizations described in the previous sections in the new compiler for the Lean programming language. We have implemented all optimizations in Lean, and they are available online [5]. At the time of writing, the compiler supports only one backend where we emit C++ code. We chose C++ just for convenience because the Lean runtime is implemented in C++. We are currently working on an LLVM backend for our compiler. To test the efficiency of the compiler and RC optimizations, we have devised a number of benchmarks[6] that aim to replicate common tasks performed in compilers and proof assistants. All timings are arithmetic means of 50 runs as reported by the `temci` benchmarking tool [Bechberger 2016][7], executed on a PC with an i7-3770 Intel

[5] https://github.com/leanprover/lean4/tree/master/library/init/lean/compiler/ir
[6] https://github.com/leanprover/lean4/tree/IFL19/tests/bench
[7] Detailed reports are available at https://pp.ipd.kit.edu/~ullrich/report

CPU and 16 GB RAM running Ubuntu 18.04, using Clang 7.1.0 for compiling the Lean runtime library as well as the C++ code emitted by the Lean compiler.

- `deriv` and `const_fold` implement differentiation and constant folding, respectively, as examples of symbolic term manipulation where big expressions are constructed and transformed. We claim they reflect operations frequently performed by proof automation procedures used in theorem provers.
- `rbmap` stress tests the red-black tree implementation from the Lean standard library. The benchmarks `rbmap_10` and `rbmap_1` are two variants where the we perform updates on shared trees.
- `frontend` is the new frontend we are developing for the next version of Lean. Its parser and macro expander are written purely in Lean (approximately 8000 lines of code), while it is interfacing with the old C++ implementation for elaboration. We are planning to eventually rewrite the elaborator in Lean as well. The new frontend is just 20% slower than the old one written in C++, but it is more powerful and supports user customizations that are not handled by the old one. For example, the new parser implements infinite lookahead while the old parser uses single token lookahead.
- `qsort` it is the basic quicksort algorithm for sorting arrays.
- `binarytrees` is taken from the Computer Languages Benchmarks Game[8]. This benchmark is a simple adaption of Hans Boehm's `GCBench` benchmark[9]. The Lean version is a translation of the fastest, parallelized Haskell solution, using `Task` in place of the Haskell `parallel` API.
- `unionfind` implements the union-find algorithm which is frequently used to implement decision procedures in automated reasoning. We use arrays to store the *find* table, and thread the state using a state monad transformer

We have tested the impact of each optimization by selectively disabling it and comparing the resulting runtime with the base runtime (Fig. 6):

- *-reuse* disables the insertion of **`reset`**/**`reuse`** operations
- *-borrow* disables borrow inference, assuming that all parameters are owned. Note that the compiler must still honor borrow annotations on builtins, which are unaffected.
- *-ST* uses atomic RC operations for all values

The results show that the new **`reset`** and **`reuse`** instructions significantly improve performance in the benchmarks `const_fold`, `rbmap`, and `unionfind`. The borrowed inference heuristic provides significant speedups in benchmarks `binarytrees` and `deriv` benchmarks.

We have also directly translated some of these programs to other statically typed, functional languages: Haskell, OCaml, and Standard ML (Fig. 7). For the latter we selected the compilers MLton [Weeks 2006], which performs whole program optimization and can switch between multiple GC schemes at runtime, and MLKit, which combines Region Inference and garbage collection [Hallenberg et al. 2002]. While not primarily a functional language, we have also included Swift as a popular statically typed language using reference counting. For `binarytrees`, we have used the original files and compiler flags from the fastest Benchmark Game implementations. For Swift, we used the second-fastest, safe implementation, which is much more comparable to the other versions than the fastest one completely depending on unsafe code. The Benchmark Game does not include an SML version. For `qsort`, the Lean code is pure and relies on the fact that array updates are destructive if the array is not shared. The Swift code behaves similarly because Swift arrays are copy-on-write. All other versions use destructive updates, using the *ST* monad in the case of Haskell.

	base	*-reuse*	*-borrow*	*-ST*
`binarytrees`	$1.00\tilde{}$	$0.99\tilde{}$	$1.22\tilde{}$	$1.13\tilde{}$
`deriv`	1.00	0.97	$1.17\tilde{}$	$1.24\tilde{}$
`const_fold`	1.00	1.53	0.97	1.07
`frontend`	$1.00\tilde{}$	$1.04\tilde{}$	$1.04\tilde{}$	$1.29\tilde{}$
`qsort`	1.00	1.14	1.01	1.15
`rbmap`	1.00	$3.09\tilde{}$	$1.09\tilde{}$	$1.42\tilde{}$
`rbmap_10`	$1.43\tilde{}$	$3.11\tilde{}$	$1.44\tilde{}$	$2.03\tilde{}$
`rbmap_1`	$4.78\tilde{}$	$5.31\tilde{}$	$4.53\tilde{}$	$7.13\tilde{}$
`unionfind`	$1.00\tilde{}$	$1.27\tilde{}$	$0.98\tilde{}$	$1.96\tilde{}$
geom. mean	$1.24\tilde{}$	$1.69\tilde{}$	$1.29\tilde{}$	$1.65\tilde{}$

Figure 6: Lean variant benchmarks, normalized by the base run time (`rbmap` for `rbmap_*`). Digits whose order of magnitude is no larger than that of twice the standard deviation are marked by squiggly lines.

While the absolute runtimes in Fig. 7 are influenced by many factors other than the implementation of garbage collection that make direct comparisons difficult, the results still signify that both our garbage collection and the overall runtime and compiler implementation are very competitive. We initially conjectured the good performance was a result of reduced cache misses due to reusing allocations and a lack of GC tracing. However, the results demonstrate this is not the case. The only benchmark where the code generated by our compiler produces significantly fewer cache misses is `rbmap`. Note that Lean is 4x faster than OCaml on `const_fold` even though they both trigger a similar number of cache misses per second. The results suggest that Lean code is often faster in the benchmarks where the code generated by other compilers spends a significant amount of time performing GC. Using `const_fold` as an example again, Lean spends only 13% of the runtime deallocating memory, while OCaml spends 91% in the GC. This comparison is not entirely precise since it does not include the amount of time Lean spends updating reference counts, but it seems to be the most plausible explanation for the difference in performance. The results for `qsort` are surprising, the Lean and Swift implementations outperforms all destructive ones but MLton. We remark that MLton and Swift have a clear advantage since they use arrays of unboxed machine integers, while Lean and the other compilers use boxed values. We did not find a way to disable this optimization in MLton or Swift to confirm our conjecture. We believe this benchmark demonstrates that our compiler allows programmers to write efficient pure code that uses arrays and hashtables. For `rbmap`, Lean is much faster

[8] https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html
[9] http://hboehm.info/gc/gc_bench/

	Lean 4			GHC 8.6.5			ocamlopt 4.07.1			MLton 20180207			MLKit 4.3.18			Swift 5.0.1		
	Time	Del	CM	Time	GC	CM	Time	GC	CM	Time	GC	CM	Time	GC	CM	Time	GC	CM
binarytrees	1.00	4%	14	2.91	71%	14	1.28	—	18	—	—	—	—	—	—	5.31	58%	10
deriv	1.00	28%	16	2.09	42%	5	1.31	75%	6	0.86	22%	19	3.75	58%	21	3.27	45%	6
const_fold	1.00	15%	22	2.28	59%	7	4.03	90%	6	0.92	29%	31	3.74	63%	14	5.11	53%	12
qsort	1.00	11%	0	1.70	1%	0	1.34	33%	0	0.54	0%	0	3.17	0%	0	0.65	0%	0
rbmap	1.00	2%	4	2.14	36%	5	0.87	31%	5	2.66	29%	33	5.62	60%	10	6.70	58%	1
rbmap_10	1.43	12%	15	12.49	87%	14	1.63	58%	9	2.92	29%	33	8.08	72%	16	7.30	54%	3
rbmap_1	4.78	27%	29	12.47	87%	13	8.02	88%	13	3.81	39%	30	13.07	83%	33	10.96	48%	14

Figure 7: Cross-language benchmarks. The measurements include wall clock time (normalized by the Lean base run time), GC time (in percent, as reported by the respective compiler), and last-level cache misses (CM, in million per second, as reported by `perf stat`). For Swift, we measure time spent in inc, dec, and deallocation runtime functions as GC time using `perf`. For Lean, the former are always inlined, so we can only measure object deletion time.

than all other systems except for OCaml. We imagined this would only be the case when the tree was not shared. Then we devised the two variants `rbmap_10` and `rbmap_1` which save the current tree in a list after every tenth or every insertion, respectively. The idea is to simulate the behavior of a backtracking search where we store a copy of the state before each case-split. As expected, Lean's performance decreases on these two variants since the tree is now a shared value, and the time spent deallocating objects increases substantially. However, Lean still outperforms all systems but MLton on `rbmap_1`. In all other systems but MLton and Swift, the time spent on GC increases considerably. Finally, we point out that MLton spends significantly less time on GC than the other languages using a tracing GC in general.

9 RELATED WORK

The idea of representing RC operations as explicit instructions so as to optimize them via static analysis is described as early as Barth [1977]. Schulte [1994] describes a system with many features similar to ours. In general, Schulte's language is much simpler than ours, with a single list type as the only non-primitive type, and no higher-order functions. He does not give a formal dynamic semantics for his system. He gives an algorithm for inserting RC instructions that, like ours, has an on-the-fly optimization for omitting **`inc`** instructions if a variable is already dead and would immediately be decremented afterwards. Schulte briefly discusses how RC operations can be minimized by treating some parameters as "nondestructive" in the sense of our borrowed references. In contrast to our inference of borrow annotations, Schulte proposes to create one copy of a function for each possible destructive/nondestructive combination of parameters (i.e. exponential in the number of (non-primitive) parameters) and to select an appropriate version for each call site of the function. Our approach never duplicates code.

Introducing destructive updates into pure programs has traditionally focused on primitive operations like array updates [Hudak and Bloss 1985], particularly in the functional array languages Sisal [McGraw et al. 1983] and SaC [Scholz 1994]. Grelck and Trojahner [2004] propose an `alloc_or_reuse` instruction for SaC that can select one of multiple array candidates for reuse, but do not describe heuristics for when to use the instruction. Férey and Shankar [2016] describe how functional update operations explicit in the source language can be turned into destructive updates using the reference counter. In contrast, Schulte [1994] presents a "reusage" optimization that has an effect similar to the one obtained with our **`reset`**/**`reuse`** instructions. In particular, it is independent of a specific surface-level update syntax. However, his optimization (transformation $T14$) is more restrictive and is only applicable to a branch of a **`case`** x if x is dead at the beginning of the branch. His optimization cannot handle the simple *swap* described earlier, let alone more complex functions such as the red black tree rebalancing function $balance_1$.

While not a purely functional language, the Swift programming language[10] has directly influenced many parts of our work. To the best of our knowledge, Swift was the first non-research language to use an intermediate representation with explicit RC instructions, as well as the idea of (safely) avoiding RC operations via "borrowed" parameters (which are called "+0" or "guaranteed" in Swift), in its implementation. While Swift's primitives may also elide copies when given a unique reference, no speculative destructive updates are introduced for user-defined types, but this may not be as important for an impure language as it is for Lean. Parameters default to borrowed in Swift, but the compiler may locally change the calling convention inside individual modules.

Baker [1994] describes optimizing reference counting by use of *two* pointer kinds, a standard one and a *deferred increment* pointer kind. The latter kind can be copied freely without adding RC operations, but must be converted into the standard kind by incrementing it before storing it in an object or returning it. The two kinds are distinguished at runtime by pointer tagging. Our borrowed references can be viewed as a static refinement of this idea. Baker then describes an extended version of deferred-increment he calls *anchored* pointers that store the stack level (i.e. the lifetime) of the standard pointer they have been created from. Anchored pointers do not have to be converted to the standard kind if returned from a stack frame above this level. In order to statically approximate this extended system, we would need to extend our type system with support for some kind of *lifetime annotations* on return types as

[10] https://developer.apple.com/swift/

featured in Cyclone [Jim et al. 2002] and Rust [Matsakis and Klock 2014].

Choi et al. [2018] describe a thread safe reference counting scheme for the Swift programming language that also tries to minimize the number of atomic operations. In their approach, each value has a header containing the ID of the thread T that allocated the value, and two reference counters: a shared one that requires atomic operations, and another one that is only updated by T. Our approach is simpler for a runtime like ours, where it is easy to find all values reachable from a given value, and there are clear places where values transition from the single-threaded world to the multi-threaded one. We suspect this is not the case for Swift, and the additional complexity is needed.

10 CONCLUSION

We have explored reference counting as a memory management technique in the context of an eager and pure functional programming language. Our preliminary experimental results are encouraging and show our approach is competitive with state-of-the-art compilers for functional languages and often outperform them. Our resurrection hypothesis suggests there are many opportunities for reusing memory and performing destructive updates in functional programs. We have also explored optimizations for reducing the number of reference counting updates, and proposed a simple and efficient technique for implementing thread safe reference counting.

We barely scratched the surface of the design space, and there are many possible optimizations and extensions to explore. We hope our λ_{pure} will be useful in the future as a target representation for other purely functional languages (e.g., Coq, Idris, Agda, and Matita). We believe our approach can be extended to programming languages that support cyclic data structures because it is orthogonal to traditional cycle-handling techniques. Finally, we are working on a formal correctness proof of the compiler described in this paper, using a type system based on intuitionistic linear logic to model owned and borrowed references.

ACKNOWLEDGMENTS

We are very grateful to Thomas Ball, Johannes Bechberger, Christiano Braga, Sebastian Graf, Simon Peyton Jones, Daan Leijen, Tahina Ramananandro, Nikhil Swamy, Max Wagner and the anonymous reviewers for extensive comments, corrections and advice.

REFERENCES

Henry G. Baker. 1994. Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures. *SIGPLAN Not.* 29, 9 (Sept. 1994), 38–43. https://doi.org/10.1145/185009.185016

Jeffrey M. Barth. 1977. Shifting Garbage Collection Overhead to Compile Time. *Commun. ACM* 20, 7 (July 1977), 513–518. https://doi.org/10.1145/359636.359713

Johannes Bechberger. 2016. Besser Benchmarken. https://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit https://github.com/parttimenerd/temci.

Hans-J. Boehm. 2004. The Space Cost of Lazy Reference Counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 210–219. https://doi.org/10.1145/964001.964019

Joachim Breitner. 2018. A promise checked is a promise kept: Inspection Testing. *arXiv preprint arXiv:1803.07130* (2018).

Jiho Choi, Thomas Shull, and Josep Torrellas. 2018. Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 35, 12 pages. https://doi.org/10.1145/3243176.3243195

George E. Collins. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM* 3, 12 (Dec. 1960), 655–657. https://doi.org/10.1145/367487.367501

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Inform. and Comput.* 76, 2-3 (1988), 95–120.

Thierry Coquand and Christine Paulin. 1990. Inductively Defined Types. In *COLOG-88 (Tallinn, 1988)*. Lecture Notes in Comput. Sci., Vol. 417. Springer, Berlin, 50–66.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, 2015, Proceedings*. 378–388.

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP (Sept. 2017). https://doi.org/10.1145/3110278

Gaspard Férey and Natarajan Shankar. 2016. Code Generation Using a Formal Model of Reference Counting. In *Proceedings of the 8th International Symposium on NASA Formal Methods - Volume 9690 (NFM 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 150–165. https://doi.org/10.1007/978-3-319-40648-0_12

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Clemens Grelck and Kai Trojahner. 2004. Implicit memory management for SAC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL*, Vol. 4. 335–348.

Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press. Berlin, Germany.

Paul Hudak and Adrienne Bloss. 1985. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*. ACM, New York, NY, USA, 300–314. https://doi.org/10.1145/318593.318660

Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*. 275–288.

Richard Jones and Rafael D Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 482–494. https://doi.org/10.1145/3062341.3062380

J. Harold McBeth. 1963. Letters to the Editor: On the Reference Counter Method. *Commun. ACM* 6, 9 (Sept. 1963), 575–. https://doi.org/10.1145/367593.367649

James McGraw, Stephen Skedzielewski, Stephen Allan, D Grit, R Oldehoeft, J Glauert, I Dobes, and P Hohensee. 1983. *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1.* Technical Report. Lawrence Livermore National Lab., CA (USA).

Boris Schling. 2011. *The Boost C++ Libraries*. XML Press.

Sven-Bodo Scholz. 1994. Single Assignment C - Functional Programming Using Imperative Style. In *In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia*.

Wolfram Schulte. 1994. Deriving Residual Reference Count Garbage Collectors. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*. Springer-Verlag, London, UK, 102–116.

Stephen Weeks. 2006. Whole-program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML (ML '06)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1159876.1159877

Paul R. Wilson. 1992. Uniprocessor garbage collection techniques. In *Memory Management*, Yves Bekkers and Jacques Cohen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–42.

Interpreting Task Oriented Programs on Tiny Computers

Mart Lubbers
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
mart@cs.ru.nl

Pieter Koopman
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
pieter@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

Small Microcontroller Units (MCUs) drive the omnipresent Internet of Things (IoT). These ubiquitous devices are small, cheap, and energy efficient. However, they are not very computationally powerful and lack an OS. Hence it is difficult to apply high level abstractions and write software that stays close to the design.

Task Oriented Programming (TOP) is a novel paradigm for creating multi-user collaborative systems. A program consists of tasks—descriptions of what needs to be done. Tasks always return an observable value and can be combined and transformed with task combinators.

MTask is an embedded Domain Specific Language (eDSL) to program MCUs following the TOP paradigm. Previous work has described the mTask language, a static C code generator, and how to integrate mTask with TOP servers.

This paper shows that for dynamic IoT applications, tasks must be sent at runtime to the devices for interpretation. It describes in detail *how* to compile specialized IoT TOP tasks to bytecode and *how* to interpret them on devices with very little memory.

These additions allow the creation of complete, dynamic IoT applications arising from a single source using a mix of iTasks and mTask tasks. Details such as serialization and communication are captured in simple abstractions.

KEYWORDS

Internet of Things, Functional Programming, Distributed Applications, Task Oriented Programming, Clean

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2020. Interpreting Task Oriented Programs on Tiny Computers. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

IoT consists of tiny devices that sense, act, and communicate with each other and with the world. The IoT is often visualized as a layered system [5]. An example of an IoT application that touches every layer is a smart-home hub. A smart-home hub usually consists of a coordinating server and several clients. This server coordinates the devices and offers an interface to the user, e.g. via a display and buttons or a web interface. Typical clients have hard-coded tasks or they receive them from a server on demand. Some clients contain interfaces to interact with the user directly and start tasks as well.

In IoT applications, clients are often heterogeneous collections of microcontrollers all with their own peculiarities, language of choice and hardware interfaces. The hardware needs to be cheap, small and energy efficient. As a result, the MCUs used to power these devices have little computational power, a soupçon of memory, and little communication bandwidth. Typically the devices do not run a full-fledged OS but statically compiled firmware. This firmware is often written in an imperative language and needs to be flashed to the program memory. This greatly reduces the flexibility for dynamic systems where tasks are created on the fly and executed on demand. While devices are getting a bit faster, smaller, and cheaper, they keep these properties to an extent. In this paper we show how these problems can be solved by dynamically sending TOP code to the MCU to be interpreted.

The TOP paradigm is a declarative paradigm where a program consists of tasks. Tasks represent collaborative work that needs to be done by people and computer systems. The iTasks framework is the reference implementation for TOP [20]. Given the task specification, iTasks generates a multi-user web interface to, guide the work based on the current state of affairs.

Tasks are generated on the fly and can be combined to form compound tasks. Moreover, it is possible to tailor-make tasks at runtime for specific work that needs to be done. A task's execution is modelled as an interactive rewrite system. As a consequence, work is automatically divided into slices and parallelisation in the form of automatic interleaving comes for free. When designing a program for an IoT device, the jobs that need to be performed on the device fit the TOP paradigm very well [18]. Tasks that devices need to perform are frequently parallel and adapted to the current needs of the system. An example of this adaptability and parallel nature for our smart-home hub is a multi-room thermostat. Different tasks are required to be executed at the same time, e.g. measuring the temperature in the room, controlling the heater and reacting to user input both on the client *and* on the server. For example when devices in the rooms contain buttons to locally set the target temperature. Furthermore, straightforwardly compiling iTasks to code suitable for IoT devices is not possible since the devices are simply not powerful enough. Unfortunately, writing multi-threaded applications using actual threads is strenuous because of the restricted memory capacity and lack of an OS. There are ways of simulating

IFL'19, September 2020, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

threads in imperative languages on MCUs. Unfortunately they all have downsides such as spaghetti code, the lack of local state, and compile-time fixed thread timings (see Section 7).

Previous work showed the TOP language mTask [14]. The mTask language is an eDSL hosted in Clean that aids writing the device part of an IoT application using TOP. The mTask language contains an important subset of TOP such as tasks, task combinators and Shared Data Sources (SDSs). It allows interaction with peripherals following the TOP paradigm. However, this version only supported generating static C code and there was no built-in communication with a server.

In real-life TOP applications, tasks are generated on the fly and combined at will using combinators. The program memory for an MCU is not suitable for rapid reprogramming when a task needs to be executed. We therefore resort to interpretation. With interpretation, tasks are sent at runtime to the device from a server to be executed on demand. In previous work we have shown how a simplified imperative version of mTask could be integrated with iTasks [17]. Small, imperative tasks could be lifted to iTasks tasks and could access iTasks SDSs. They were compiled to bytecode and interpreted on the device. However, there was no support for task oriented programming on the client.

1.1 Research contribution

For dynamic applications, generating code at runtime for interpretation on the device is necessary. In this paper we show in detail how to compile mTask tasks to bytecode. This bytecode can be interpreted on MCUs with very little memory and processing power. The programmer does not have to worry about details such as serialization, communication, or rapidly changing programs using precious write cycles of the program memory. With these additions, complete, dynamic IoT applications can be constructed from a single source using a mix of iTasks and mTask tasks.

Our solution accomplishes this by (1) describing mTask's semantics more formally, (2) providing a concrete compilation scheme for compiling mTask tasks to bytecode for an abstract machine, and (3) describing this abstract machine for interpreting bytecode generated for TOP tasks.

Section 2 presents an overview of TOP in general while Section 3 introduces the mTask language. Section 4 presents the bytecode backend infrastructure—i.e. the integration with iTasks, the compiler and the Runtime System (RTS), and presents a demonstrative example. The compilation rules are given in Section 5 followed by the rewriting rules in Section 6. The paper concludes with related work, conclusions and future work (Sections 7 to 9).

2 Task Oriented Programming

TOP is a programming paradigm for specifying distributed systems modelling collaborations between people and machines. *Tasks* are the basic building blocks of the program and behave a bit like communicating threads. They represent the work that needs to be done and they can be combined and transformed to form compound tasks. The full set of available combinators and their semantics for the reference implementation can be found in [21]. Tasks are event-driven, stateful rewrite functions that yield, after each step, a three-state *task value*. Task values are observable by other tasks and may change over time. The task value is either *no value*, an *unstable* or *stable* value. The allowed task value transitions are shown in Figure 1. The events that drive the execution of a task can be anything ranging from user input to clocks, hardware events or even the task itself requesting a subsequent execution step.

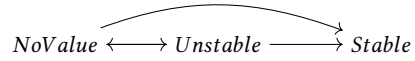


Figure 1: A state transition diagram for task values.

Task values can be observed and acted upon by other tasks if they are combined sequentially or in parallel. Furthermore, data can be shared using SDS. SDSs provide a general abstraction over any data. They can be accessed through three tasks that retrieve, store or modify the data (`get`, `set`, and `upd` respectively). While tasks in parallel are executed interleaved, access to SDSs is always atomic, i.e. exactly one rewrite step. SDSs can be combined and transformed using combinators similar to those used for tasks.

3 MTASK LANGUAGE

The core of the mTask system is the mTask language—a multi-backend class-based eDSL. The classes are type-constructor classes and therefore a backend implementing a class is a type of the form `v t` where `v` is the actual backend. The phantom type `t` represents the type of the construction. Not all types are suitable for MCUs, for example they have to be serializable and bounded. To enforce this, the type `t` must have instances for the `type` class collection containing these constraints.

The classes for expressions—i.e. arithmetic functions, conditional expressions and tuples—are listed in Listing 1. Some of the functions are oddly named (e.g. `+.`) to avoid name conflicts with Clean's functions. There is no need for loop control due to support for tail-call optimized recursive functions and tasks. The `lit` function fulfills a special role of the language: it allows lifting host language values to the mTask domain. For tuples there is a useful default instance (`topen`) to convert a function with an mTask tuple as an argument to a function with a tuple of mTask values as an argument.

```
class arith v where
    lit          :: t → v t | type t
    (+.) infixl 6 :: (v t) (v t) → v t | basicType, +, zero t
    ...
class cond v where
    If :: (v Bool) (v t) (v t) → v t | type t
class tupl v where
    first  :: (v (a, b))  → v a      | type a & type b
    second :: (v (a, b))  → v b      | type a & type b
    tupl   :: (v a) (v b) → v (a, b) | type a & type b

    topen :: (v (a, b) → c) (v a, v b) → c
    topen f x = f (first x, second x)
```

Listing 1: The mTask classes for simple expressions.

3.1 Functions

Functions are supported in the eDSL, albeit with some limitations. All user-defined mTask functions are typed by Clean expressions

to ensure type safety. All functions are defined using the multi-parameter typeclass `fun`. The first parameter (`a`) of the typeclass is the shape of the argument and the second parameter (`v`) is the backend (Listing 2). Functions may only be defined at the top level and to constrain this, the `Main` type is introduced to box a program.

One implementation for the `fun` class is defined for every arity. So for a function from `a` to `b`, the **instance** `fun (T a) T | type a` is used. The listing gives example instances for arities zero to two for backend `T`. Defining the different arities as tuples of arguments forbids the use of curried functions. To illustrate the use, the factorial function is given as an example. The `type` constraint on the function arguments forbids the use of higher order functions. This Clean function will construct the program that will calculate the factorial of the given argument. All mTask constructions used in the `factorial` function must be defined as class constraints on the backend type variable. This creates quite a bit of clutter as the number of class constraints increases rapidly. The class collection `mtask` can be used to avoid long lists of constraints.

```
:: Main a = {main :: a}
:: In a b = In infix 0 a b
class fun a v where
    fun :: ((a → v s) → In (a → v s) (Main (v u)))
        → Main (v u) | type s & type u

:: T a // a backend
instance fun () T
instance fun (T a) T | type a
instance fun (T a, T b) T | type a & type b

class mtask v | arith v & cond v & ... & fun () v & fun (v Int) v & ...

factorial :: Int → Main (v Int) | mtask v
factorial x =
    fun λfac=(
       λi→If (i <=. lit 0)
           (lit 1) (i *. fac (i -. lit 1))
    ) In {main=fac (lit x)}
```

Listing 2: The mTask classes for functions definitions.

3.2 Tasks

Tasks are viewed as trees with leafs and forks. Basic tasks are the leafs and often represent a side effect such as hardware access. Task combinators are the forks and transform one or more tasks to a single transformed task.

Task values in mTask are represented by the same type as in iTasks (Listing 3). To lift a value in the expression domain to the task domain, the basic task `rtrn` is used. The resulting task will forever yield the given value as a stable task value.

```
:: MTask v t :== v (TaskValue t)
class rtrn v where rtrn    :: (v t) → MTask v t | type t
```

Listing 3: The mTask classes for basic tasks.

Interaction with the General Purpose Input/Output (GPIO) pins, and other peripherals for that matter, is also captured in basic tasks. For each type of pin, there is a read and a write task that, given the pin, executes the action. The class for analogue GPIO pin access is shown in Listing 4. The `readA` task constantly yields the value of the analogue pin as an unstable task value. The `writeA` writes the given value to the given pin once and returns the written value as a stable task value.

```
:: APin = A0 | A1 | A2 | A3 | ...
:: DPin = D0 | D1 | D2 | D3 | ...
class aio v where
    readA  :: (v APin)          → MTask v Int
    writeA :: (v APin) (v Int) → MTask v Int
```

Listing 4: The mTask classes for GPIO access.

3.3 Task Combinators

There are two flavours of task combinators. With sequential combinators, tasks are executed after each other. With parallel combinators, the tasks are executed at the same time and the resulting task values are combined.

The step combinator (`>>*.`) is the Swiss army knife of sequential combination (Listing 5). The value that the left-hand side of the combinator yields is matched against all task continuations (`Step v t u`) on the right-hand side, i.e. the right-hand side tasks *observe* the task value. If one of the continuations yields a new task, the combined task continues with it, pruning the left-hand side. All other sequential combinators are derived from the step combinator as default instances but their implementation can be overridden to for example provide a more efficient implementations. For example, the `>>=.` combinator is very similar to the monadic bind: it continues if and only if a stable value is yielded. The `>>~.` combinator continues when any value, stable or unstable, is yielded. The `>>|.` and `>>..` combinators are variants of the aforementioned combinators that do not take the value into account.

```
class step v where
    (>>*.) infixl 1 :: (MTask v t) [Step v t u] → MTask v u | type u & type t

    (>>=.) infixl 0 :: (MTask v t) ((v t) → MTask v u) → MTask v u | ...
    (>>=.) m f = m >>*. [IfStable (λ_→lit True) f]
    (>>~.) infixl 0 :: (MTask v t) ((v t) → MTask v u) → MTask v u | ...
    (>>~.) m f = m >>*. [IfValue (λ_→lit True) f]
    (>>|.) infixl 0 :: (MTask v t) (MTask v u) → MTask v u | ...
    (>>|.) m f = m >>=. λ_→f
    (>>..) infixl 0 :: (MTask v t) (MTask v u) → MTask v u | ...
    (>>..) m f = m >>~. λ_→f

:: Step v t u
    = IfValue    ((v t) → v Bool) ((v t) → MTask v u)
    | IfStable   ((v t) → v Bool) ((v t) → MTask v u)
    | IfUnstable ((v t) → v Bool) ((v t) → MTask v u)
    | IfNoValue                           (MTask v u)
    | Always                              (MTask v u)
```

Listing 5: The mTask classes for sequential task combinators.

The following listing shows an example of a step in action. The `readPinBin` function will produce an mTask task that will classify the value of an analogue pin into four bins. It also shows that the nature of embedding allows the host language to be used as a macro language.

```
readPinBin :: Main (MTask v Int) | mtask v
readPinBin = {main=readA A2 >>*.
    [  IfValue (λx→x <. lim) λ_→rtrn (lit bin)
    \\ lim←[64,128,192,256]
    &  bin←[0..]]
```

Listing 6: An example task using sequential combinators.

In contrast to iTasks—that has one supercombinator for all parallel combinations—there are only two parallel combinators (Listing 7). The conjunction combinator `.&&.` combines the task values to a tuple. The disjunction combinator `.||.` combines them into a single task value, giving preference to the *most stable* value (See Section 6.4). The listing shows an example of querying two pins at the same time, returning the one with the highest value.

```
class .&&. v where
    (.&&.) infixr 4 v :: (MTask v a) (MTask v b) → MTask v (a, b) | type a ...
class .||. v where
    (.||.) infixr 3 v :: (MTask v a) (MTask v a) → MTask v a | type a

maxPins :: APin APin → Main (MTask v Int) | mtask v
maxPins p1 p2 = {main=
        readA p1 .&&. readA p2
    >>~. topen λ(l, r)→rtrn (If (l >. r) l r)}
```

Listing 7: The mTask classes for parallel task combinators.

Finally there are also miscellaneous combinators. For example, the `rpeat` function forever executes the argument task. When the argument task is stable, it reinstates it and starts all over again. The `delay` task waits for the specified amount of time. This task yields no value until the given time has elapsed, then it will yield the numer of milliseconds it overshot as a stable task value. To demonstrate them, the `blink` program is given that toggles the given pin every 500 milliseconds. The functionality of `rpeat` can also be simulated using recursive functions as shown in the `blinkFun` task.

```
class rpeat v where
    rpeat :: (MTask v a) → MTask v () | type a
class delay v where
    delay :: (v Int) → MTask v t | type t

blink :: DPin → Main (MTask v ()) | mtask v
blink p = {main=rpeat (
        delay (lit 500) >>|. writeD (lit True) p
    >>|. delay (lit 500) >>|. writeD (lit False) p)}

blinkFun :: DPin → Main (MTask v Bool) | mtask v
blinkFun p =
    fun λblink=(
        λst→delay (lit 500) >>|. writeD st p >>=. blink o Not
    ) In {main=blink (lit True)}
```

Listing 8: The mTask classes for repeat and delay.

3.4 Shared Data Sources

In mTask it is also possible to share data between tasks type safely using SDSs. Similar to functions, SDSs can only be defined at the top level. They are well-typed parts of the monadic state.

The `sds` class contains the functions for defining and accessing SDSs. With the `sds` function, local SDSs can be defined. They are also typed by functions in the host language to ensure type safety. The other functions in the class are for creating `get` and `set` tasks. The `getSds` returns a task that constantly emits the value of the SDS as an unstable task value; `setSds` writes the given value to the task and re-emits it as a stable task value when it is done.

Listing 9 the definitions and an artificial example showing a task that mirrors a pin value to another pin using an SDS.

```
class sds v where
    sds    :: ((v (Sds t)) → In t (Main (MTask v u)))
        → Main (MTask v u) | type t & type u
    getSds :: (v (Sds t))       → MTask v t | type t
    setSds :: (v (Sds t)) (v t) → MTask v t | type t

localvar :: Main (MTask v ()) | mtask v
localvar = sds λx=42 In {main= rpeat (readA D13 >>~. setSds x)
                        .||.  rpeat (getSds x >>~. writeD D1)}
```

Listing 9: The mTask classes for SDS tasks.

The `liftsds` class below is used to allow iTasks SDSs to be accessed from within mTask tasks and vice versa. The function has a similar type as `sds` and creates an mTask SDS from an iTasks SDS so that it can be accessed using the class functions from the `sds` class. Listing 10 shows an example of this where an iTasks SDS is used to control an LED on a device. During task execution, the server and the device get notified when the SDS is modified.

```
:: Shared a // an iTasks SDS
class liftsds v | sds v where
    liftsds :: ((v (Sds t)) → In (Shared t) (Main (MTask v u)))
        → Main (MTask v u) | type t & type u

lightSwitch :: (Shared Bool) → Main (MTask v ()) | mtask v & liftsds v
lightSwitch s = liftsds λx=s In {main=rpeat (getSds x >>~. writeD D13)}
```

Listing 10: The mTask class for lifting iTasks SDSs.

4 BYTECODE COMPILER INFRASTRUCTURE

This section presents the infrastructure surrounding the bytecode backend. This is where tasks are compiled, sent and executed during runtime of the iTasks server to a specific device. The supporting Clean functions for this are given in Listing 11.

The `withDevice` function offers access to the device with the given specification. The first argument of the function, which is a type implementing the `channelSync` class, contains the information for maintaining a connection with the device. At present, there are instances for `channelSync` for types representing a TCP or serial connection and a simulator. The resulting task connects the device and ascertains that the connection is set up, kept up and closed down on completion. After the connection is set up, the second argument—the task doing something with a device—is executed.

Within the argument task—besides executing iTasks tasks—the `liftmTask` task can be used. This function lifts an mTask task to an iTasks task using the specified device so that the mTask task can be fully utilized within the iTasks system. The `BCInterpret` type houses the backend and therefore implements the mTask classes. It adheres to the compilation scheme given in Section 5. The mTask constructions will have the form `Main (MTask BCInterpret u)`. Under the hood, this function creates a task that executes the compiler, sends the generated bytecode, listens to messages from the device and watches the lifted SDSs. The task value of the mTask task is observable from iTasks because the task is now a regular iTasks task. Furthermore, lifted SDSs can be accessed and used for communication. In a traditional setting, all these things—such as communication, data sharing, task scheduling—have to be done by hand. In contrast, `liftmTask` automatically does it all.

```
:: MTDevice //Abstract device representation
:: Channels //Communication channels

class channelSync a :: a (Shared Channels) → Task ()
withDevice :: a (MTDevice → Task b) → Task b | iTask b & channelSync a
```

```
liftmTask :: (Main (MTask BCInterpret u)) MTDevice → Task u | iTask, type u
```

Listing 11: Functions integrating mTask with iTasks.

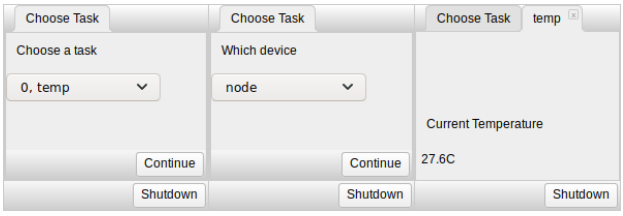
4.1 Example

This subsection presents a toy home automation program (Listing 12) to illustrate the language and its integration with an iTasks server. It consists of a web interface for the user to control the tasks. The tasks may be executed on either of two connected devices: an Arduino UNO, connected via a serial port and an *ESP8266* based prototyping board called NodeMCU, connected via TCP over WiFi.

Lines 1–2 show the specification for the devices followed by lines 4–8 containing the actual task. This task first connects the devices (lines 5–6). Followed by a `parallel` task that is visualized as a tabbed window with a shutdown button to terminate the program (lines 7–8). The `chooseTask` task (lines 10–21) allows the user to pick a task and send it to the specified device. Tasks are picked from the `tasks` list (lines 23–38). For example, the `temperature` task shows the current temperature to the user. When the temperature changes, the Digital Humidity and Temperature sensor (DHT) sensor reports it and the task value for the `temperature` task changes. This change in task value is reflected to the iTasks server and the task value of the `liftmTask` task changes respectively. The task is lifted to an iTasks task and the `>&>` iTasks combinator transforms the task value into an SDS that can be displayed to the user using `viewSharedInformation`. A screenshot of the temperature task is given in Figure 2.

```
1  arduino = {TTYSettings | zero & devicePath="/dev/ttyACM0"}
2  nodeMCU = {TCPSettings | host="192.168.0.1", port=8123}
3
4  autoHome :: Task ()
5  autoHome = withDevice arduino λdev1→
6             withDevice nodeMCU λdev2→
7      parallel [(Embedded, chooseTask dev1 dev2)] [] <<@ ArrangeWithTabs True
8      >>* [OnAction (Action "Shutdown") (always (shutDown 0))]
9
10 chooseTask :: MTDevice MTDevice (SharedTaskList ()) → Task ()
11 chooseTask dev1 dev2 stl = tune (Title "Choose Task") $ forever
12     $            enterChoice "Choose a task" [] (zip2 [0..] (map fst tasks))
13     >>= λ(i, n)→enterChoice "Which device"  [] ["arduino", "node"]
14     >>= λdevice→appendTask Embedded (mkTask n i device) stl
15     >>| chooseTask dev1 dev2 stl
16 where
17     mkTask n i device _
18         # dev = if (device == "node") dev2 dev1
19         = ((snd (tasks !! i) $ dev)
20             >>* [OnAction (Action "Close") $ always $ treturn ()]) <<@ Title n
21
22 tasks :: [(String, MTDevice → Task ())]
23 tasks =
24     [ ("temp", λdev→
25         liftmTask (DHT D6 DHT22 λdht→{main=temperature dht}) dev
26         >&> viewSharedInformation "Current Temperature"
27             [ViewAs λi→toString (toReal (fromMaybe 0 i) / 10.0) +++ "C"]
28         @! ())
29     , ("lightswitch", λdev→
30         withShared False λsh→
31             liftmTask (lightswitch sh) dev
32             -|| updateSharedInformation "Switch" [] sh
33     , ("factorial", λdev→
34             updateInformation "Factorial of what" [] 5
35         >>= λi→liftmTask (factorial i) dev
36         >>= viewInformation "result" []
37         @! ())
38     ]
37  , ... ]
```

Listing 12: An example of a home automation program.



(a) Select task **(b) Select device** **(c) View result**

Figure 2: Screenshots of the example program in action.

4.2 Runtime System

The RTS is designed to run on systems with as little as 2kB of RAM and therefore aggressive memory management is vital. Not all firmwares for MCUs support heaps and—when they do—the allocation functions often leave holes when not used in a Last In First Out strategy. Therefore the RTS allocates a chunk of memory in the global data segment with its own memory manager tailored to the needs of mTask. The size of this block can be changed in the configuration of the RTS if necessary. On an Arduino UNO that only boasts 2kB of RAM, this size can be about 1500 bytes.

In memory, the data for tasks grows from the bottom up and an interpreter stack is located on top of it. Values in the interpreter are always stored on the stack, even tuples. As a consequence, this segment may grow and—e.g. when a new task is received—the stack has to move. This never happens within execution because communication is always processed before execution. Task trees grow from the top down as in a heap. This approach allows for flexible ratios, i.e. many tasks and small trees or few tasks and big trees.

The event loop of the RTS is executed repeatedly and consists of three distinct phases.

4.2.1 Communication. In the first phase, the communication channels are processed. SDS updates are processed immediately, tasks are stored and will be initialized later on.

4.2.2 Execution. The second phase consists of executing tasks. The RTS executes all tasks in a round robin fashion. If a task is not initialized, the bytecode of the main function is interpreted to produce the initial task tree. The rewriting engine uses the interpreter when needed, e.g. to calculate the step continuations. The rewriter and interpreter use the same stack to store intermediate values. Rewriting steps are small so that interleaving results in seamingly parallel execution. Both rewriting and initialization are atomic operations in the sense that no processing on SDSs is done other than SDS operations from the task itself. The host is notified if a task value is changed after a rewrite step. In this phase new task tree nodes may be allocated.

4.2.3 Memory Management. The third and final phase is memory management in which stable tasks, and unreachable task tree nodes are removed. If a task needs to be removed, all tasks with higher memory addresses are moved down as much as possible. For task trees—stored in the heap—the RTS already marks tasks as trash during rewriting so only compacting is necessary which is a single linear pass. This is possible because there is no sharing or cycles in task trees and pointers to parents are stored.

4.3 Instruction Set

The instruction set is a fairly standard stack machine instruction set. Listing 13 shows the Clean type housing the instructions. Table 1 gives the detailed semantics for every instruction. Type synonyms are used to provide insight on the arguments of the instructions. One notable instruction is the `MkTask` instruction. This constructs a task tree node and pushes a pointer to it on the stack.

```
:: ArgWidth  :== UInt8          :: ReturnWidth :== UInt8
:: Depth     :== UInt8          :: Num         :== UInt8
:: SdsId     :== UInt8          :: PerId       :== UInt8
:: JumpLabel :== UInt16

:: BCInstr
    = BCJumpF JumpLabel | BCJump JumpLabel | BCJumpSR ArgWidth JumpLabel
    | BCLabel JumpLabel
    | BCReturn ReturnWidth ArgWidth | BCTailcall ArgWidth ArgWidth JumpLabel
    | BCArg ArgWidth | BCStepArg UInt16 UInt8
    | BCIsStable | BCIsUnstable | BCIsNoValue | BCIsValue
    | BCPush String | BCPop Num | BCRot Depth Num | BCDup | BCPushPtrs
    | BCAdd | BCSub | BCMult | BCDiv | BCAnd | BCOr | BCNot
    | BCEq | BCNeq | BCLe | BCGe | BCLeq | BCGeq
    | BCMkTask BCTaskType

:: BCTaskType
    = BCStable1 | BCStable2 | ... | BCUnstable1 | BCUnstable2 | ...
    | BCReadD | BCWriteD | BCReadA | BCWriteA
    | BCRepeat | BCDelay | BCTAnd | BCTOr
    | BCSdsGet SdsId | BCSdsSet SdsId
    | BCStep ArgWidth JumpLabel
    //Peripherals
    | BCDHTTemp PerId | BCDHTHumid PerId
    | ...
```

Listing 13: The type housing the instruction set.

4.4 Compiler

The bytecode compiler backend for the mTask language is a stateful writer monad. The state contains: the bytecode for the main expression, the context of arguments, a function dictionary, streams for fresh labels and SDS identifiers, an SDS dictionary containing both local (*Left*) and lifted (*Right*) SDSs (See Section 3.4) and a list of peripherals. Executing the bytecode compiler does not result in usable bytecode immediately. After execution of the monad the following processing steps are applied: all tail call `BCReturn` instructions are optimized to `BCTailCall`; the functions are concatenated before the main expression[1]; redundant instructions are removed; the labels are resolved to actual memory addresses; all lifted SDSs are queried for their initial value. The result—bytecode, SDS specifications and peripherals specification—can then be sent to the device for execution.

[1]In this way the labels are still correct when removing the bytecode for the main function to save space

```
:: BCInterpret a :== StateT BCState (WriterT [BCInstr] Identity) a
:: BCState =
    { bcs_mainexpr     :: [BCInstr]
    , bcs_context      :: [BCInstr]
    , bcs_functions    :: Map JumpLabel BCFunction
    , bcs_freshlabel   :: JumpLabel
    , bcs_freshsds     :: UInt8
    , bcs_sdses        :: Map UInt8 (Either String255 (Shared String255))
    , bcs_hardware     :: [BCPeripheral]
    }
:: BCFunction =
    { bcf_instructions :: [BCInstr]
    , bcf_argwidth     :: UInt8
    , bcf_returnwidth  :: UInt8
    }
```

Listing 14: The type for the bytecode backend.

5 COMPILATION RULES

The compilation scheme is divided in three schemes. When something is surrounded by $\|$, e.g. $\|a_i\|$, it denotes the number of stack cells required to store it. Some schemes have a *context* r as an argument which contains information about the location of the arguments in scope. More information is given in the schemes requiring such arguments.

Scheme	Description
$\mathcal{E}[\![e]\!]\ r$	Produces the value of expression e given the context r and pushes it on the stack. The result can be a basic value or a pointer to a task.
$\mathcal{F}[\![e]\!]$	Generates the bytecode for functions.
$\mathcal{S}[\![e]\!]\ r\ w$	Generates the function for the step continuation given the context r and the width w of the left-hand side task value.

5.1 Expressions

Almost all expression constructions are compiled using $\mathcal{E}$. The argument of $\mathcal{E}$ is the context (See Section 5.2). Values are always placed on the stack; tuples are unpacked. Function calls, function arguments and tasks are also compiled using $\mathcal{E}$ but are explained later.

$$\mathcal{E}[\![\texttt{lit}\ e]\!]\ r = \texttt{BCPush (bytecode e)};$$
$$\mathcal{E}[\![e_1 \texttt{+}.\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r; \texttt{BCAdd};$$
Similar for other binary operators
$$\mathcal{E}[\![\texttt{Not}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \texttt{BCNot};$$
Similar for other unary operators
$$\mathcal{E}[\![\texttt{If}\ e_1\ e_2\ e_3]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \texttt{BCJmpF}\ l_1; \mathcal{E}[\![e_2]\!]\ r; \texttt{BCJmp}\ l_2; \texttt{BCLabel}\ l_1; \mathcal{E}[\![e_3]\!]\ r; \texttt{BCLabel}\ l_2;$$
Where l_1 and l_2 are fresh labels
$$\mathcal{E}[\![\texttt{tupl}\ e_1\ e_2]\!]\ r = \mathcal{E}[\![e_1]\!]\ r; \mathcal{E}[\![e_2]\!]\ r;$$
$$\mathcal{E}[\![\texttt{first}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \texttt{BCPop}\ w;$$
Where w is the width of the left value
$$\mathcal{E}[\![\texttt{second}\ e]\!]\ r = \mathcal{E}[\![e]\!]\ r; \texttt{BCRot}\ w_1\ (w_1 + w_2); \texttt{BCPop}\ w_2;$$
Where w_1 is the width of the left and

w_2 of the right value

Translating $\mathcal{E}$ is very straightforward, it basically means executing the monad. Almost always, the type of the backend is not used, i.e. it is a phantom type. To still have the functions return the correct type, the `tell`` helper is used. This function is similar to the writer monad's `tell` function but is casted to the correct type. Listing 15 shows the implementation for the arithmetic and conditional expressions. Note that r, the context, is not an explicit argument but stored in the state.

```
instance arith BCInterpret where
    lit   t   = tell` [BCPush $ toByteCode{|*|} t]
    (+.)  a b = a >>| b >>| tell` [BCAdd]
    ...
instance cond BCInterpret where
    If c t e = freshlabel >>= λelselabel→freshlabel >>= λendiflabel→
        c >>| tell` [BCJumpF elselabel] >>|
        t >>| tell` [BCJump endiflabel,BCLabel elselabel] >>|
        e >>| tell` [BCLabel endiflabel]
```

Listing 15: Backend implementation for the arithmetic and conditional classes.

5.2 Functions

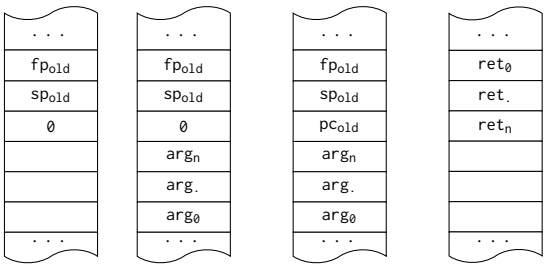
Compiling the functions themselves occurs in $\mathcal{F}$, which generates bytecode for the complete program by iterating over the functions and ending with the main expression. When compiling the body of the function, the arguments of the function are added to the context so that the addresses can be determined when referencing arguments. The main expression is a special case of $\mathcal{F}$ since it neither has arguments nor something to continue with. As a result, it is simply compiled with $\mathcal{E}$.

$$\begin{aligned}\mathcal{F}[\![main = m]\!] &= \mathcal{E}[\![m]\!]\ [];\\ \mathcal{F}[\![f\ \ a_0 \ldots a_n = b\ \texttt{In}\ m]\!] &= \texttt{BCLabel}\ \ f;\\ &\quad\ \texttt{BCLabel}\ \ l_2;\\ &\quad\ \mathcal{E}[\![b]\!]\ \ [\langle f, i\rangle, i \in \{\Sigma_{i=0}^{n}\|a_i\|..0\}];\\ &\quad\ \texttt{BCReturn}\ \ \|b\|\ \ n;\\ &\quad\ \mathcal{F}[\![m]\!];\end{aligned}$$

A function call starts by pushing the stack and frame pointer, and making space for the program counter (Figure 3a) followed by evaluating the arguments in reverse order (Figure 3b). On executing `BCJumpSR`, the program counter is set and the interpreter jumps to the function (Figure 3c). When the function returns, the return value overwrites the old pointers and the arguments. This occurs right after a `BCReturn` (Figure 3d). Putting the arguments on top of pointers and not reserving space for the return value uses little space and facilitates tail call optimization.

Calling a function and referencing function arguments are an extension to $\mathcal{E}$ as shown below. Arguments may be at different places on the stack at different times (see Section 5.4) and therefore the exact location always has to be determined from the context using `findarg`[2] Compiling argument a_{fi}, the ith argument in function f,

[2] `findarg [l':r] l = if (l == l`) 0 (1 + findarg r l)`



(a) `BCPushPtrs` **(b) Arguments** **(c)** `BCJumpSR` **(d)** `BCReturn`

Figure 3: The stack layout during function calls.

consists of traversing all positions in the current context. Arguments wider than one stack cell are fetched in reverse to preserve the order.

$$\begin{aligned}\mathcal{E}[\![f(a_0, \ldots, a_n)]\!]\ \ r &= \texttt{BCPushPtrs};\\ &\quad\ \mathcal{E}[\![a_n]\!]\ \ r; \mathcal{E}[\![a_{\ldots}]\!]\ \ r; \mathcal{E}[\![a_0]\!]\ \ r;\\ &\quad\ \texttt{BCJumpSR}\ \ n\ \ f;\\ \mathcal{E}[\![a_{fi}]\!]\ \ r &= \texttt{BCArg}\ \text{findarg}(r, f, i)\ \text{ for all }\ i \in \{w \ldots v\};\\ v &= \Sigma_{j=0}^{i-1}\|a_{fj}\|\\ w &= v + \|a_{fi}\|\\ w &= v + \|a_{fi}\|\end{aligned}$$

Translating the compilation schemes for functions to Clean is not as straightforward as other schemes due to the nature of shallow embedding. The `fun` class has a single function with a single argument. This argument is a Clean function that—when given a callable Clean function representing the mTask function—will produce `main` and a callable function. To compile this, the argument must be called with a function representing a function call in mTask. Listing 16 shows the implementation for this as Clean code. To uniquely identify the function, a fresh label is generated. The function is then called with the `callFunction` helper function that generates the instructions that correspond to calling the function. That is, it pushes the pointers, compiles the arguments, and writes the `JumpSR` instruction. The resulting structure (`g In m`) contains a function representing the mTask function (`g`) and the `main` structure to continue with. To get the actual function, `g` must be called with representations for the argument, i.e. using `findarg` for all arguments. The arguments are added to the context and `liftFunction` is called with the label, the argument width and the compiler. This function executes the compiler, decorates the instructions with a label and places them in the function dictionary together with the metadata such as the argument width. After lifting the function, the context is cleared again and compilation continues with the rest of the program.

```
instance fun (BCInterpret a) BCInterpret | type a where
    fun def = {main=freshlabel >>= λfunlabel→
        let (g In m) = def λa→callFunction funlabel (byteWidth a) [a]
        in  addToCtx funlabel zero (argwidth def)
        >>| liftFunction funlabel (argwidth def)
            (g (findArgs funlabel zero (argwidth def))) Nothing
        >>| clearCtx >>| m.main
```

```
    }

callFunction :: JumpLabel UInt8 [BCInterpret b] → BCInterpret c | ...
liftFunction :: JumpLabel UInt8 (BCInterpret a) (Maybe UInt8) → BCInterpret ()
```

Listing 16: The backend implementation for functions.

5.3 Tasks

Task trees are created with the `BCMkTask` instruction that allocates a node and pushes it to the stack. It pops arguments from the stack according to the given task type. An extension of $\mathcal{E}$ follows that gives this compilation scheme (except for the step combinator, see Section 5.4).

$$
\begin{aligned}
\mathcal{E}[\![\texttt{rtrn}\ e]\!]\ r\ w &= \mathcal{E}[\![e]\!]\ r\ w; \texttt{BCMkTask BCStable}_{\|e\|};\\
\mathcal{E}[\![\texttt{unstable}\ e]\!]\ r\ w &= \mathcal{E}[\![e]\!]\ r\ w; \texttt{BCMkTask BCUnstable}_{\|e\|};\\
\mathcal{E}[\![\texttt{readA}\ e]\!]\ r\ w &= \mathcal{E}[\![e]\!]\ r\ w; \texttt{BCMkTask BCReadA};\\
\mathcal{E}[\![\texttt{writeA}\ e_1\ e_2]\!]\ r\ w &= \mathcal{E}[\![e_1]\!]\ r\ w; \mathcal{E}[\![e_2]\!]\ r\ w; \texttt{BCMkTask BCWriteA};\\
\mathcal{E}[\![\texttt{readD}\ e]\!]\ r\ w &= \mathcal{E}[\![e]\!]\ r\ w; \texttt{BCMkTask BCReadD};\\
\mathcal{E}[\![\texttt{writeD}\ e_1\ e_2]\!]\ r\ w &= \mathcal{E}[\![e_1]\!]\ r\ w; \mathcal{E}[\![e_2]\!]\ r\ w; \texttt{BCMkTask BCWriteD};\\
\mathcal{E}[\![\texttt{delay}\ e]\!]\ r\ w &= \mathcal{E}[\![e]\!]\ r\ w; \texttt{BCMkTask BCDelay};\\
\mathcal{E}[\![\texttt{rpeat}\ e]\!]\ r\ w &= \mathcal{E}[\![e]\!]\ r\ w; \texttt{BCMkTask BCRepeat};\\
\mathcal{E}[\![e_1\texttt{.||.}e_2]\!]\ r\ w &= \mathcal{E}[\![e_1]\!]\ r\ w; \mathcal{E}[\![e_2]\!]\ r\ w; \texttt{BCMkTask BCOr};\\
\mathcal{E}[\![e_1\texttt{.\&\&.}e_2]\!]\ r\ w &= \mathcal{E}[\![e_1]\!]\ r\ w; \mathcal{E}[\![e_2]\!]\ r\ w; \texttt{BCMkTask BCAnd};
\end{aligned}
$$

This literally translates to Clean code by just writing the correct `BCMkTask` instruction as exemplified in Listing 17.

```
instance rtrn BCInterpret where rtrn m = m >>| tell` [BCMkTask BCStable]
```

Listing 17: The backend implementation for `rtrn`.

5.4 Step combinator

The `step` construct is a special type of task because the task value of the left-hand side may change over time. Therefore, the continuation tasks on the right-hand side are *observing* this task value and acting upon it. In the compilation scheme, all continuations are first converted to a single function that has two arguments, namely the stability of the task and the value of the task. This function either returns a pointer to a task tree or fails (denoted by $\bot$). It is special because in the generated function, the task value of a task can actually be inspected. Furthermore, it is a lazy node in the task tree: the right-hand side may yield a new task tree after several rewrite steps (i.e. it is allowed to create infinite task trees using step combinators). The function is generated using the $\mathcal{S}$ scheme that requires two arguments: the context r and the width of the left-hand side so that it can determine the position of the stability which is added as an argument to the function. The resulting function is basically a list of if-then-else constructions to check all predicates one by one. Some optimization is possible in this checking but currently not implemented.

$$
\begin{aligned}
\mathcal{E}[\![t_1\texttt{>>*.}t_2]\!]\ r\ w = {} & \mathcal{E}[\![a_{f\,i}]\!]\ r, \langle f, i\rangle \in r; \texttt{BCMkTask BCStable}_{\|r\|};\\
& \mathcal{E}[\![t_1]\!]\ r;\\
& \texttt{BCMkTask BCAnd};\\
& \texttt{BCMkTask BCStep}\ (\mathcal{S}[\![t_2]\!]\ (r + [\langle l_s, i\rangle])\ \|t_1\|);\\
\mathcal{S}[\![[]]\!]\ r\ w = {} & \texttt{BCPush}\ \bot; \texttt{`}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}[\![\texttt{IfValue}\ f\ t : cs]\!]\ r\ w = {} & \texttt{BCArg}(\|r\| + w); \texttt{BCIsNoValue};\\
& \mathcal{E}[\![f]\!]\ r; \texttt{BCAnd};\\
& \texttt{BCJmpF}\ l_1;\\
& \mathcal{E}[\![t]\!]\ r; \texttt{BCJmp}\ l_2;\\
& \texttt{BCLabel}\ l_1; \mathcal{S}[\![cs]\!]\ r\ w;\\
& \texttt{BCLabel}\ l_2;\\
& \textit{Where } l_1 \textit{ and } l_2 \textit{ are fresh labels}\\
& \textit{Similar for } \texttt{IfStable} \textit{ and } \texttt{IfUnstable}\\
\mathcal{S}[\![\texttt{IfNoValue}\ t : cs]\!]\ r\ w = {} & \texttt{BCArg}(\|r\| + w); \texttt{BCIsNoValue};\\
& \texttt{BCJmpF}\ l_1;\\
& \mathcal{E}[\![t]\!]\ r; \texttt{BCJmp}\ l_2;\\
& \texttt{BCLabel}\ l_1; \mathcal{S}[\![cs]\!]\ r\ w;\\
& \texttt{BCLabel}\ l_2;\\
& \textit{Where } l_1 \textit{ and } l_2 \textit{ are fresh labels}\\
\mathcal{S}[\![\texttt{Always}\ f : cs]\!]\ r\ w = {} & \mathcal{E}[\![f]\!]\ r;
\end{aligned}
$$

First the context is evaluated. The context contains arguments from functions and steps that need to be preserved after rewriting. The evaluated context is combined with the left-hand side task value by means of a `.&&.` combinator to store it in the task tree so that it is available after a rewrite. This means that the task tree is be transformed as follows:

```
t1 >>= λv1→t2 >>= λv2→t3 >>= ...
//is transformed to
t1 >>= λv1→rtrn v1 .&&. t2 >>= λv2→rtrn (v1, v2) .&&. t3 >>= ...
```

The translation to Clean is given in Listing 18.

```
instance step BCInterpret where
    (>>*.) lhs cont
        //Fetch a fresh label and fetch the context
        =   freshlabel >>= λfunlab→gets (λs→s.bcs_context)
        //Generate code for lhs
        >>= λctx→lhs
        //Possibly add the context
        >>| tell` (if (ctx =: []) []
                //The context is just the arguments up till now in reverse
                (  [BCArg (UInt8 i)\\i←reverse (indexList ctx)]
                ++ map BCMkTask (bcstable (UInt8 (length ctx)))
                ++ [BCMkTask BCTAnd]
                ))
        //Increase the context
        >>| addToCtx funlab zero lhswidth
        //Lift the step function
        >>| liftFunction funlab
                //Width of the arguments is the width of the lhs plus the
                //stability plus the context
                (one + lhswidth + (UInt8 (length ctx)))
                //Body label ctx width continuations
                (contfun funlab (UInt8 (length ctx)))
                //Return width (always 1, a task pointer)
                (Just one)
        >>| modify (λs→{s & bcs_context=ctx})
        >>| tell` [BCMkTask $ instr rhswidth funlab]

toContFun :: JumpLabel UInt8 → BCInterpret a
toContFun steplabel contextwidth
    = foldr tcf (tell` [BCPush fail]) cont
where
    tcf (IfStable f t)
        = If ((stability >>| tell` [BCIsStable]) &. f val)
            (t val >>| tell` [])
    ...
    stability = tell` [BCArg $ lhswidth + contextwidth]
```

```
val = retrieveArgs steplabel zero lhswidth
```

Listing 18: Backend implementation for the step class.

5.5 Shared Data Sources

The compilation scheme for SDS definitions is a trivial extension to $\mathcal{F}$ since there is no code generated as seen below.

$$\mathcal{F}[\![\texttt{sds}\ \ x = i\ \ \texttt{In}\ \ m]\!] = \mathcal{F}[\![m]\!];$$
$$\mathcal{F}[\![\texttt{liftsds}\ \ x = i\ \ \texttt{In}\ \ m]\!] = \mathcal{F}[\![m]\!];$$

The SDS access tasks have a compilation scheme similar to other tasks (See Section 5.3). The `getSds` task just pushes a task tree node with the SDS identifier embedded. The `setSds` task evaluates the value, lifts that value to a task tree node and creates an SDS set node.

$$\mathcal{E}[\![\texttt{getSds}\ \ s]\!]\ \ r = \texttt{BCMkTask (BCSdsGet r);}$$
$$\mathcal{E}[\![\texttt{setSds}\ \ s\ \ e]\!]\ \ r = \mathcal{E}[\![e]\!]\ \ r; \texttt{BCMkTask BCStable}_{\|e\|}\texttt{;}$$
$$\texttt{BCMkTask (BCSdsSet r);}$$

While there is no code generated in the definition, the bytecode compiler is storing the SDS data in the `bcs_sdses` field in the compilation state. The SDSs are typed as functions in the host language so an argument for this function must be created that represents the SDS on evaluation. For this, a compiler is created that emits this identifier. When passing this compiler to the function, the initial value of the SDS is returned. This initial value is stored as a bytecode encoded value in the state and the compiler continues with the rest of the program.

Compiling `getSds` is a matter of executing the `BCInterpret` representing the SDS, which yields the identifier that can be embedded in the instruction. Setting the SDS is similar: the identifier is retrieved and the value is written to put in a task tree so that the resulting task can remember the value it has written. Lifted SDSs are compiled in a very similar way. The only difference is that there is no initial value but an iTasks SDS when executing the Clean function. A lens on this SDS converting `a` from the `Shared a` to a `String255`—a bytecode encoded version—is stored in the state. This encoding and decoding is in principle unsafe, nonetheless, the type safety of the language should make it safe. Upon sending the mTask task to the device, the initial values of the lifted SDSs are fetched to make the SDS specification complete.

```
:: Sds a = Sds Int
instance sds BCInterpret where
    sds def = {main = freshsds >>= λsdsi→
            let sds = modify (λs→{s & bcs_sdses='Data.Map'.put sdsi
                        (Left (toByteCode{|*|} t)) s.bcs_sdses})
                    >>| pure (Sds sdsi)
                (t In e) = def sds
            in e.main}
    getSds f   = f >>= λ(Sds i)→ tell` [BCMkTask (BCSdsGet (fromInt i))]
    setSds f v = f >>= λ(Sds i)→v >>| tell`
        (  map BCMkTask (bcstable (byteWidth v))
        ++ [BCMkTask (BCSdsSet (fromInt i))])
```

Listing 19: Backend implementation for the SDS classes.

6 TASK REWRITING

This section shows the rewriting rules for the mTask tasks. Tasks are rewritten every cycle and one rewrite cycle is generally very fast. This results in seemingly parallel execution of the tasks because the rewrite steps are interleaved. Rewriting is a destructive process that actually modifies the task tree nodes in memory and marks nodes that have become unreachable. The emitted task value is stored on the stack and therefore only available during rewriting.

6.1 Basic Tasks

The `rtrn` and `unstable` tasks always rewrite to themselves and have no side effects. The GPIO interaction tasks do have side effects. The `readA` and `readD` tasks will query the given pin every rewrite cycle and emit it as an unstable task value. The `writeA` and `writeD` tasks write the given value to the given pin and immediately rewrite to a stable task of the written value.

6.2 Delay and Repetition

The `delay` task stabilizes once a certain amount of time has been passed by storing the finish time on initialization. In every rewrite step it checks whether the current time is bigger than the finish time and if so, it emits the number of milliseconds that it overshot as a stable task value. The `rpeat` task combinator rewrites the argument until it becomes stable. Rewriting is a destructive process and therefore the original task tree must be saved. As a consequence, on installation, the argument is cloned and the task rewrites the clone.

6.3 Sequential Combination

First the left-hand side of the step task is rewritten. The resulting value is passed to the continuation function. If the continuation function returns a pointer to a task tree, the task tree rewrites to that task tree and marks the original left-hand side as trash. If the function returns $\perp$, the step is kept unchanged. The step itself will always yield a *no value*.

6.4 Parallel Combination

There are two parallel task operation combinators available. The `.&&.` only becomes stable when both sides are stable. The `.||.` becomes stable when one of the sides is stable.

The combinators first rewrite both sides and then merge the task values according to the semantics given as a Clean function in Listing 20.

```
(.&&.) :: (TaskValue a) (TaskValue b) → TaskValue (a, b)
(.&&.) (Value lhs stab1) (Value rhs stab2) = Value (lhs, rhs) (stab1 && stab2)
(.&&.) _                 _                 = NoValue

(.||.) :: (TaskValue a) (TaskValue a) → TaskValue a
(.||.) lhs=:(Value _ True) _                 = lhs
(.||.) (Value lhs _)       rhs=:(Value _ True) = rhs
(.||.) NoValue             rhs               = rhs
(.||.) lhs                 _                 = lhs
```

Listing 20: Task value semantics for the parallel combinators.

6.5 Shared Data Sources Tasks

The `BCSdsGet` node always rewrites to itself. It will read the actual SDS embedded and emit the value as an unstable task value.

Setting an SDS is a bit more involved because after writing, it emits the value written as a stable task value. The `BCSdsSet` node contains the identifier for the SDS and a task tree that, when rewritten, emits the value to be set as a stable task value. The naive approach would be to just rewrite the `BCSdsSet` to a node similar to the `BCSdsGet` but only with a stable value. However, after writing the SDS, its value might have changed due to other tasks writing it, and then the `setSDS`'s stable value may change. Therefore, the `BCSdsSet` node is rewritten to the argument task tree which is always represents constant stable value. In future rewrites, the constant value node will emit the value that was originally written.

The client only knows whether an SDS is a lifted SDS, not to which iTasks SDS it is connected. If the SDS is modified on the device, it will send an update to the server.

7 RELATED WORK

7.1 Functional Programming on IoT devices

Haenisch et al. showed that Functional Programming (FP) for IoT leads to less code complexity and better maintainability [12]. For example, Microscheme is a purely functional programming language for the Arduino that runs on the little Arduino UNO [23]. To mitigate the memory issues, direct compilation is used and some elements like garbage collection and first class closures are omitted. Furthermore it only supports a single thread of execution.

Functional Reactive Programming (FRP) [7] bears similarities to TOP. Juniper, FRP for MCUs, translates the code to C++ [13]. In this way even the small Arduino UNO can be programmed with parallel tasks. It differs from the mTask approach in the sense that there is no automatic communication nor interpretation.

Others have approached FRP on MCUs without the computing power constraint in mind. For example, using many frameworks such as *Kafka* and *NodeJS*, sensor networks can be programmed without having to know the details of the communication protocols [19]. Also, de Troyer et al. used the FRP language *Elixir* that runs on the *Erlang* Virtual Machine (VM) to create distributed IoT applications with ease [24]. These approaches allow the creation of multithreaded IoT applications with automatic communication similar to TOP. However, a lot more computing power and memory is required to run them and all approaches forbid dynamic task sending. Smaller devices can participate but they become simpler data relay devices instead.

7.2 Interpretation on IoT Devices

Typical MCUs have limited memory which makes interpretation difficult. However, many solutions for interpreted or tethered control of the device have been made.

Lightweight scripting languages for MCUs are amply available, for example micropython, LUA and *EveryLite* [16]. All these solutions require either a more substantial amount of memory or only support single threaded operation.

Another example of this is the Firmata protocol that allows remote control of the peripherals via a host machine [22]. Implementations for this in a functional language are also available such as hArduino [8]. Grebe et al. created Haskino—using a remote monad—C++ like code could be interpreted on the Arduino [10]. Later they extended this traditional model and support multi tasking [11]. The language is still imperative and communication between the threads has to be defined explicitly. In mTask, the interleaving only happens on the task level, making SDS communication automatically thread-safe.

Reprogramming MCUs is also possible without interpretation and even wirelessly using Over the Air Update (OTA) programming. However, this wears out the programming memory. For example Bachelli et al. described such a system in which code can be updated on demand [1, 2].

7.3 Multitasking on IoT Devices

Multitasking or multithreading is generally difficult on MCUs because of the limited resources. There are solutions using (Real Time) OSs available to support multithreading on MCUs. Examples are TinyOS [15], Qduino [4] or ERIKA [3]. They all have relatively high hardware requirements (e.g. 32kB RAM) and often do not support interpretation or automatic communication with a server.

Multitasking on MCUs is also possible using manual interleaving [9]. It often results in explicitly simulating state machines, is work intensive, error prone and results in hard to maintain code when using it for more than simple tasks. To mitigate the issues, protothreads are available [6]. Protothreads allow, by clever use of macros, automatic interleaving of stackless threads that are defined in a single function. However, they are not interpreted, and cannot span multiple functions.

7.4 Multitasking in a Functional Language

Suspending and resuming calculations in FP programs in general is possible by using call/CC techniques [25]. The programmer defines where the code can be suspended and with some wrapper code, multiple tasks can be interleaved. Tasks in mTask are modelled as rewrite systems and can automatically interleaved by interleaving the rewriting steps. Furthermore, thread safe communication is provided out of the box. Many other methods of multitasking in functional languages rely on the thread support of the OS and are therefore not suitable in memory scarce environments.

8 CONCLUSION

The mTask system is a novel programming environment for developing all layers of dynamic IoT applications from a single source following the TOP paradigm. The language is implemented as a multi-backend, class-based shallowly eDSL. Its bytecode backend offers run-time compilation for tailor-made IoT tasks that can be executed on tiny microcontrollers. The given compilation scheme describes exactly how tasks are compiled to bytecode. Devices are prepared with an RTS that takes care of the communication between the server and the client as it schedules and executes tasks. It runs on devices with as little as 2kB memory since the RTS only uses about 500 bytes of memory, leaving the rest for tasks, the stack and the heap. Due to the nature of rewriting, parallel execution can

be simulated by interleaving the rewrite steps of tasks and subtasks. Furthermore, mTask tasks can be lifted to iTasks tasks to create applications with tasks both on the server and the device. An mTask task can access iTasks SDSs with regular SDS constructs from the mTask language. Programmers can achieve this without requiring knowledge about the underlying protocols and communication techniques, resulting in very compact code. The iTasks task combinator set can then be used on mTask tasks mixed with iTasks tasks, albeit via the server, to create applications spanning all IoT layers. With the addition of the bytecode backend, the mTask system allows the creation of complete, dynamic IoT applications. In this type of application, tasks change rapidly and are constructed using runtime information while not wearing out the program memory. It has been shown for smaller examples, experimental results for bigger applications is future work still.

9 FUTURE WORK

Executing arbitrary code received from a server is a security concern. While the language is quite general, it is still restricted to operate solely on the device itself. Communication with the peripherals is not spelled out using dynamically sent code which mitigates the risk a bit. In the current system, the device cannot determine if it is communicating with a legitimate server. Some IoT communication protocols such as ZigBee have checks for this built in. Security by design is often argued to be a much stronger starting point. Therefore it would be fruitful to investigate the security problems to identify the weak points and improve thereupon.

Tasks are constructed at runtime, but always with Clean as the host language. Followup research could be to see whether it is possible to create a type-safe iTasks editor for mTask tasks so that truly any task can be created out of thin air.

In terms of language features, SDSs are very powerful in TOP. In iTasks, lenses can be applied and SDSs can be transformed. The mTask system would benefit from this as well. It is worth investigating to see which type of lenses and transformations are possible in the mTask ecosystem. Furthermore, the combinators could be enriched with more direct interaction with the peripherals. For example, the step combinator in iTasks contains options for buttons.

The execution model could also be enriched. The iTasks system only rewrites tasks when an event occurred that is of interest to it. It would be interesting to incorporate this as well in the mTask RTS. Events might arise from interrupts, but also timers or triggers from the host server. As an extension, interrupts might be caught using an `IfInterrupt` continuation in the `step` combinator. Furthermore, we would like to investigate how these interrupts can be modeled in mTask and what the behaviour in the rewriting system would be. It might be that the semantics for interrupts are similar to that of exceptions in iTasks.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their invaluable comments.

REFERENCES

[1] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. 2018. Reprogramming Low-end IoT Devices from the Cloud. In *2018 3rd Cloudification of the Internet of Things (CIoT)*. IEEE, 1–6.

[2] Emmanuel Baccelli, Joerg Doerr, Shinji Kikuchi, Francisco Padilla, Kaspar Schleiser, and Ian Thomas. 2018. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE PerCom 2018*.

[3] Pasquale Buonocunto, Alessandro Biondi, and Pietro Lorefice. 2014. Real-time multitasking in Arduino. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE, Pisa, 1–4. https://doi.org/10.1109/SIES.2014.7087331

[4] Zhuoqun Cheng, Ye Li, and Richard West. 2015. Qduino: A multithreaded arduino system for embedded computing. In *Real-Time Systems Symposium, 2015 IEEE*. IEEE, 261–272.

[5] Li Da Xu, Wu He, and Shancang Li. 2014. Internet of things in industries: a survey. *Industrial Informatics, IEEE Transactions on* 10, 4 (2014), 2233–2243.

[6] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. 2005. Using protothreads for sensor node programming. In *Proceedings of the REALWSN*, Vol. 5.

[7] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 263–273.

[8] Levent Erkok. 2016. hArduino by LeventErkok. https://leventerkok.github.io/hArduino/

[9] Loe Feijs. 2013. Multi-tasking and Arduino : why and how?. In *Design and semantics of form and movement. 8th International Conference on Design and Semantics of Form and Movement (DeSForM 2013)*, L. L. Chen, T. Djajadiningrat, L. M. G. Feijs, S. Fraser, J. Hu, S. Kyffin, and D. Steffen (Eds.). Wuxi, China, 119–127.

[10] Mark Grebe and Andy Gill. 2016. Haskino: A remote monad for programming the arduino. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 153–168.

[11] Mark Grebe and Andy Gill. 2019. Threading the Arduino with Haskell. In *Trends in Functional Programming*, David Van Horn and John Hughes (Eds.). Springer International Publishing, Cham, 135–154.

[12] Till Haenisch. 2016. A case study on using functional programming for internet of things applications. *Athens Journal of Technology & Engineering* 3, 1 (2016).

[13] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, 8–16.

[14] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, Vienna, Austria, 1–11. https://doi.org/10.1145/3183895.3183902

[15] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and others. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

[16] Zhenying Li, Xiaohui Peng, Lu Chao, and Zhiwei Xu. 2018. EveryLite: A Lightweight Scripting Language for Micro Tasks in IoT Systems. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 381–386.

[17] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2018. Task Oriented Programming and the Internet of Things. In *Proceedings of the 30th Symposium on the Implementation and Application of Functional Programming Languages*. ACM, Lowell, MA, 83–94. https://doi.org/10.1145/3310232.3310239

[18] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Multitasking on Microcontrollers using Task Oriented Programming. In *2019 42st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, Opatija, Croatia, 1842–1846.

[19] Haidong Lv, Xiaolong Ge, Hongzhi Zhu, Zhiwei Yuan, Zhen Wang, and Yongkang Zhu. 2018. Designing of IoT Platform Based on Functional Reactive Pattern. In *2018 International Conference on Computer Science, Electronics and Communication Engineering (CSECE 2018)*. Atlantis Press.

[20] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices* 42, 9 (2007), 141–152.

[21] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*. ACM, 195–206.

[22] Hans-Christoph Steiner. 2009. Firmata: Towards Making Microcontrollers Act Like Extensions of the Computer.. In *NIME*. 125–130.

[23] Ryan Suchocki and Sara Kalvala. 2015. Microscheme: Functional programming for the Arduino. In *2014 SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*.

[24] Christophe Troyer, de, Jens Nicolay, and Wolfgang Meuter, de. 2018. Building IoT Systems Using Distributed First-Class Reactive Programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 185–192. https://doi.org/10.1109/CloudCom2018.2018.00045

[25] Mitchell Wand. 1999. Continuation-based multiprocessing revisited. *Higher-Order and Symbolic Computation* 12, 3 (1999), 283–283.

Instr.	Args	Semantics	sp	pc
Return	rw aw	st[fp-aw-3+i] = st[fp+i] **for all** i∈{0..rw} fp = st[fp-aw-2]	st[fp-aw-3+rw]	st[fp-aw-1]
JumpF	jl		sp-1	st[sp-1] ? pc+1 : jl
Jump	jl			jl
JumpSR	aw jl	st[sp-i-1] = pc+2 fp = sp		jl
Tailcall	w_1 w_2 jl	rotate (w_1+3+w_2,w_2) fp = fp-w_1+w_2 **where** w_1 is the width of the caller, w_2 the width of the callee	fp	jl
Arg	a	st[sp] = st[fp-1-a]	sp+1	pc+2
Push	n s	st[sp+i] = s[i] **for all** i∈{0..n}	sp+n	pc+2+n
Pop	n		sp-n	pc+2
Rot	d n	rotate (d, n) **for all** i∈{0..n}	sp-n	pc+3
Dup		st[sp] = st[sp-1]	sp-1	pc+1
PushPtrs		st[sp] = sp st[sp+1] = fp st[sp+2] = 0	sp+3	pc+1
UnOp		st[sp-1] = ⋄ st[sp-1]		pc+1
BinOp		st[sp-2] = st[sp-2] ⊕ st[sp-1]	sp-1	pc+1
MkTask	$Stable_n$	st[sp-n-1] = node (stable, st[sp-1], ..., st[sp-n-1])	sp-n+1	pc+2
	$Unstable_n$	st[sp-n-1] = node (unstable, st[sp-1], ..., st[sp-n-1])	sp-n+1	pc+2
	ReadD	st[sp-1] = node (readd, st[sp-1])		pc+2
	ReadA	st[sp-1] = node (reada, st[sp-1])		pc+2
	Repeat	st[sp-1] = node (repeat, st[sp-1])		pc+2
	Delay	st[sp-1] = node (delay, st[sp-1])		pc+2
	WriteD	st[sp-2] = node (writed, st[sp-1], st[sp-2])	sp-1	pc+2
	WriteA	st[sp-2] = node (writea, st[sp-1], st[sp-2])	sp-1	pc+2
	And	st[sp-2] = node (and, st[sp-1], st[sp-2])	sp-1	pc+2
	Or	st[sp-2] = node (or, st[sp-1], st[sp-2])	sp-1	pc+2
	SdsSet i	st[sp-1] = node (sdsset,i, st[sp-1])		pc+3
	SdsGet i	st[sp] = node (sdsget, i)	sp+1	pc+3
	DHTTemp i	st[sp-1] = node (dhttemp, i)	sp+1	pc+3
	DHTHumid i	st[sp-1] = node (dhthumid, i)	sp+1	pc+3
	Step aw jl	st[sp-1] = node (step, aw, jl, st[sp-1])	sp-1	pc+5

Table 1: Semantics for all instructions

A Functional Approach to Accelerating Monte Carlo based American Option Pricing

Wojciech Michal Pawlak
Department of Computer Science,
University of Copenhagen
SimCorp Technology Labs
Denmark
wmp@di.ku.dk,wmpk@simcorp.com

Martin Elsman
Department of Computer Science,
University of Copenhagen
Denmark
mael@di.ku.dk

Cosmin Eugen Oancea
Department of Computer Science,
University of Copenhagen
Denmark
cosmin.oancea@diku.dk

ABSTRACT

We study the feasibility and performance efficiency of expressing a complex financial numerical algorithm with high-level functional parallel constructs. The algorithm we investigate is a least-square regression-based Monte-Carlo simulation for pricing American options. We propose an accelerated parallel implementation in Futhark, a high-level functional data-parallel language. The Futhark language targets GPUs as the compute platform and we achieve a performance comparable to, and in particular cases better than, an implementation optimized by NVIDIA CUDA engineers. In absolute terms, we can price a put option with 1 million simulation paths and 100 time steps in 20ms on a NVIDIA Tesla V100 GPU. Furthermore, the high-level functional specification is much more accessible to the financial-domain experts than the low-level CUDA code, thus promoting code maintainability and facilitating algorithmic changes.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; **Massively parallel algorithms**; **Massively parallel and high-performance simulations**; Parallel programming languages; • **Applied computing** → **Economics**; • **Computer systems organisation** → *Multicore architectures*;

KEYWORDS

High-Performance Computing, Parallel (GPU) Programming, Functional Programming, Compilers, Computational Finance, Derivative Pricing

ACM Reference Format:
Wojciech Michal Pawlak, Martin Elsman, and Cosmin Eugen Oancea. 2020. A Functional Approach to Accelerating Monte Carlo based American Option Pricing. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Pricing American options is a fundamental business case in the financial services sector, because such financial instruments are widely traded in the derivative markets. American options can be exercised at any time between the present date and the time to maturity. This aspect puts them in contrast to European options, that can only be exercised at their maturity. In the usual case, the option holder is expected to exercise the option as soon as it is more profitable to do so rather than wait until its expiration. Effectively, the value of an American option is the value achieved by exercising it at the optimal time. This embedded optimisation (optimal stopping) problem is the main challenge. As there is no general closed-form formula solutions [20], it is necessary to approximate the option value accurately with a numerical simulation. This is a substantial and time-consuming computational effort. It has to be significantly reduced to become acceptable for time-critical applications in financial practice. Therefore it is a compelling case for accelerating the computation on highly parallel hardware, such as GPUs.

Currently, the most efficient accelerated simulations are implemented in dedicated languages and frameworks like CUDA [1, 18, 32], MPI [13], OpenMP [38] and other technologies [12]. The challenge with these implementations is the poor expressibility, which makes them inaccessible to domain experts. A specialist developer has to be appointed to implement and maintain such kind of code. An interaction between domain experts and developers is needed every time a change in the algorithm is required. It also results in code that is difficult to maintain. On top of that, the developer needs to be aware of the low-level properties of the underlying hardware architecture, and have good knowledge of code transformation aimed at optimizing potentially-conflicting factors such as locality of reference and degree of parallelism.

We propose a functional approach to the implementation of an accelerated option pricing model. The use of high-level parallel constructs lets us express the algorithm in an intuitive manner, without the implementation concerns of mapping the code to the architecture. The Futhark language and the optimizing compiler behind it makes this possible [24]. The main contributions of this work are the following:

(1) We present a high-level data-parallel implementation of the Longstaff-Schwartz algorithm for pricing American options using Monte Carlo Simulation with Least-Square Regression (abbreviated LSMC) [29]. The implementation serves as a reference implementation and can easily be ported to other functional languages. Moreover, the algorithm makes explicit the available parallelism by using high-level data-parallel constructs.
(2) We give a detailed description of the algorithmic changes required to achieve an efficient parallel implementation of the LSMC algorithm.

IFL'19, September 2019, Singapore
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(3) We present an optimised efficient version of the algorithm and describe how the original algorithm is rewritten in Futhark to achieve performance results that in most cases matches a CUDA version, which is a hand-tuned implementation of the algorithm, written in a dedicated low-level programming model (by CUDA engineers). In addition, we present specific cases, in which Futhark version achieves up to 2× speedup over the CUDA version.

The remainder of the paper is organised as follows. In Section 2, we examine the problem solved by our application and describe the high-level algorithm for American option pricing. In Section 3, we describe the Futhark language and show how American option pricing is written in Futhark. In Section 4, we demonstrate how we can turn the inefficient version of the algorithm into a more efficient version targeting GPUs. We also describe the design and implementation choices that were necessary for obtaining an efficient version of the algorithm. Section 5 presents and discusses the main experimental results. Section 6 reviews the related work on the topic and Section 7 summarizes the main findings of out work.

2 APPLICATION AND ALGORITHM

We begin by introducing the algorithm structure and challenges that create a background for and motivate our work.

2.1 Monte Carlo Simulation and American Option Pricing

An option contract is defined by its payoff function. For the vanilla-type options like *calls* and *puts*, it compares the strike price K and the current asset spot price S and thereby determines the cashflow of an option. The strike price K is an agreed fixed price, at which the option holder can buy (in case of a call) or sell (in case of a put) the underlying asset. The spot price S is a price of the underlying asset like stock or commodity, that the option derives its value from. S varies over the time. This progression in time is random and thus can be described by a stochastic process. As a rule such it is defined by a stochastic differential equation (SDE), which cannot be solved directly and, thus, motivate use of a numerical method like a simulation.

In practice, Monte Carlo (MC) simulation is the most widely used and robust method for solving general SDE problems, and option pricing problems in particular. Its popularity in the quantitative finance community is due to (1) its ease of implementation and parallelisation and (2) because it allows pricing of complex instruments that cannot be solved with deterministic methods like finite difference or lattice models. Monte Carlo is the only numerical method that can be used for multi-factor pricing in dimensions greater than four. For instance, pricing options that are based on many underlying assets leads to a problem too complex to express with a grid, a discretisation of PDEs in many dimensions [19].

The Monte Carlo Simulation method is based on two theorems of probability theory. The first one is the Strong Law of Large Numbers, which guarantees the convergence of a certain series of independent random numbers having the same distribution to a value of an integral. The second is the Central Limit Theorem, which determines the convergence rate of the first law [4, 34].

In a standard Monte Carlo simulation, the paths of the state variables are simulated forward in time, which is, in particular, the case for pricing European options. Given an option payoff, a forward (future) price is determined for each path at the maturity date. To estimate the present price of the instrument, the future cashflows (prices in different points in future time) have to be discounted to the present time using some established discount rate. This is due to the time value of money concept that staates that money available now is worth more than the identical sum in the future. Finally, an unbiased estimate of the current option price is a mean average of these prices.

In contrast, the pricing of American options happens backward in time. First the optimal exercise price is determined at the maturity and then it is recursively propagated and discounted backward in time using dynamic programming until the current time. In this way, the current price is estimated. Although an American option can be exercised at any time, the exercise times are discretised and restricted to a fixed set of times (e.g., daily or monthly). The overall goal here is to provide an approximation of the optimal stopping rule that maximizes the value of the American option.

At the maturity $t = T$, the option holder will exercise the option if it is in-the-money (ITM), that is, if the value of the option, if exercised, will generate a positive cash flow. For instance, in case of a call option spot price higher than the strike price is preferable, while for put option the inverse is favorable. For any other time t_i, the holder needs to choose whether to exercise the option or to continue to hold it. The option value is at the maximum if the exercise happens as soon as the immediate exercise cash flow is greater than or equal to the continuation value, the discounted expected option value at the next instance in time. However, this continuation value at any given time t_i is not known, so it needs to be estimated.

In practice, we want to price large portfolios of options through simulation within seconds for real-time decision making. That is why GPUs, which allow for a high degree of parallelism due to its massive number of cores, are a good fit for such workloads.

2.2 The Longstaff-Schwartz Algorithm

Several authors have proposed the use of regression to estimate continuation values from simulated paths and thereby enable American option pricing through simulation. It is, especially, the initial studies of [9], the most renown [29] as well as [37], who performed similar studies at the same time. We choose the Longstaff-Schwartz approach [29] for our implementation for two reasons. It is the algorithm with the most widespread adoption in the financial industry and it is based on an easily-parallelisable Monte Carlo simulation. In addition, we mention that the authors of [15] prove the convergence of the Longstaff-Schwartz algorithm and analyse the dependence of its convergence rate on the number of simulated paths.

2.2.1 Least-Square Regression. To start with, we turn our focus to the core challenge of the algorithm — the estimation of the continuation values C_i at each time step i. Let us assume that S_{ij} is an asset spot price at time step i on a path j. Each continuation value $C_i(S_{ij})$ is the regression of the option value in the next time step on the value of S_{ij} in the current time step and path. The procedure is to approximate C_i by a linear combination of basis functions of the

current state and use regression to estimate the best coefficients for this approximation. The approximation accuracy depends on the choice of functions used in the regression.

Following the Longstaff-Schwartz approach, we apply an ordinary least-squares regression across the simulated paths at any given time t_i to estimate the continuation value C_i.

The least-square regression is a method for finding approximate solutions of over-determined[1] systems of linear equations by minimizing the sum of the squares of the errors in the equations [8]. However, since the decision to exercise the option is relevant only when the option is in-the-money, we regress only paths that are in-the-money. This choice results in an improved algorithmic efficiency without negative impact on accuracy. Both the convergence of the algorithm and how the algorithm converges with the number of simulated paths were analysed in [15].

To begin with, the ITM paths are parametrised using a quadratic polynomial $\beta_0 + \beta_1 S_{ij} + \beta_2 S_{ij}^2$, where S_{ij} is a variable — an asset spot price at a time step i, here for some ITM path j. In particular, each term of a polynomial, a monomial with basis $1, x, x^2, \ldots$, is a basis function. We chose to use up to second order polynomials in the basis. Having said that, it is necessary to mention that this choice is usually left as a parameter to the algorithm. In contrast, we deliberately settle on a particular type of basis function. this makes it possible to implement specialized versions of matrix transformations, and as a result enable performance optimisations. The number of used polynomials matches the number of different time steps i, because one regression is performed for each step. A point cloud of asset spot prices $S_i j$, distributed for each path across the time steps, is obtained from a simulation. The regression problem is then to find, separately for each of these time steps, the best fit in terms of β coefficients and basis functions for a quadratic polynomial. We discuss this procedure further in Section 4.

Essentially, we deal with an overdetermined system of equations, because the number of equations are larger than the number of unknown coefficients (i.e., 3 β coefficients). For the sake of presentation, let us assume we solve a system of equations of form $Ax = b$. We minimize the objective function $\| Ax-b \|^2$ by finding a variable vector $\hat{x}$ from all possible choices of x. In other words, it is a least squares approximate solution to $\| A\hat{x} - b \|^2 \leqslant \| Ax - b \|^2$. In our case, we assume that the number of ITM paths itm is significantly larger than the number of the basis functions (3). It follows that A is a tall data matrix of size $itm \times 3$ and b is a data column vector of size itm. The variable of this system of equations is the column vector x of size 3. In particular, A is a matrix of powers of asset spot prices S_i built from the chosen polynomial. b is a vector of cash flows $\hat{V}_i$ dependent on the payoff function $p(S_i)$, specific for an option that is priced. Finally, x is a vector of polynomial coefficients β_k, that we want to fit with the least squares method. We arrive at the following system of equations:

$$\begin{bmatrix} 1 & S_0 & S_0^2 \\ & \cdots & \\ 1 & S_j & S_j^2 \\ & \cdots & \\ 1 & S_{itm-1} & S_{itm-1}^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} p(S_0) \\ \cdots \\ p(S_j) \\ \cdots \\ p(S_{itm-1}) \end{bmatrix} \tag{1}$$

[1]There are more equations to solve than variables to choose.

The textbook solution for solving this least-square problem is to multiply both sides by A^T, resulting in $A^T A\hat{x} = A^T b$. Since $A^T A$ is a square matrix, we can now multiply both sides with its inverse[2] resulting in the unique solution: $\hat{x} = (A^T A)^{-1} A^T b$. The matrix $(A^T A)^{-1} A^T$ is the pseudo-inverse of the matrix A, denoted $A^\dagger$. The approach to constructing $A^\dagger$ is the main algorithmic challenge and the source for optimizations. There exist different methods to build $A^\dagger$. For the first naive implementation, we use the formula directly, applying matrix multiplication, transpositions, and inversion on A. In Section 4.2, we present an efficient algorithm for a pseudo-inverse construction, adapted for massive parallelism offered by GPUs.

2.2.2 Algorithmic Structure. The generic structure of a simulation algorithm that employs a linear least squares regression according to Longstaff-Schwartz algorithm [19] can be summarized as follows:

We assume as input we are given the number of time steps m, the number of paths n, and the option-specific payoff function $p_i(S_{ij})$, where S_{ij} is an asset spot price at time step i and path j. Payoff p_i is discounted back from the maturity T (time step $m-1$) to current time (time step i).

A generic linear combination of basis functions is denoted by a polynomial function $\psi_i : \mathbb{R}^r \to \mathbb{R}$ and constant coefficients β_{ik}, where r is the highest degree of the chosen polynomial with each term being a basis function $k = 0, \ldots, r-1$. Moreover, $\beta_i = [\beta_{i0}, \ldots, \beta_{im-1}]$ and $\psi(S_i)^T = [\psi_0(S_i), \ldots, \psi_{m-1}(S_i)]^T$.

(1) Generate a matrix $W(n, m)$ of random numbers drawn from a standard normal distribution.
(2) Using W, simulate (by *forward induction*) n independent paths $S_{0j}, \ldots, S_{m-1j}, j = 0, \ldots, n-1$ of Geometric Brownian Motion stochastic processes for the underlying asset prices.
(3) At the last step $m-1$ (at maturity T), compute the option value $\hat{V}_{mj} = p_m(S_{mj}), j = 0, \ldots, n-1$ applying the payoff function p at the last step $m-1$.
(4) Apply *backward induction* for each step $i = m-2, \ldots, 1$ to compute cashflows:
   (a) Select the ITM paths.
   (b) Build the matrix ψ_i from asset prices S_i and the right hand side cashflows vector $\hat{V}_{i+1}$ only for the ITM paths for the least-square linear equation $\psi_i(S_i)\beta_i = \hat{V}_{i+1}(S_{i+1})$.
   (c) Use regression to calculate $\hat{\beta}_i$ by solving a pseudo-inverse
   $$\psi(S_i)^\dagger = (\psi(S_i)^T \psi(S_i))^{-1} \psi(S_i)^T$$
   in
   $$\hat{\beta}_i = \psi(S_i)^\dagger \hat{V}_{i+1}(S_{i+1}).$$
   (d) Approximate the continuation function $\hat{C}_i(S_i) = \hat{\beta}_i \psi(S_i)^T$.
   (e) Decide to early-exercise based on the value of the continuation function $\hat{C}_i$ for each ITM path j:
   $$\hat{V}_{ij} = \begin{cases} p_i(S_{ij}), & p_i(S_{ij}) \geqslant \hat{C}_i(S_{ij}); \\ \hat{V}_{i+1,j}, & p_i(S_{ij}) < \hat{C}_i(S_{ij}). \end{cases} \tag{2}$$
(5) Return $\hat{V}_0 = (\hat{V}_{10} + \cdots + \hat{V}_{1n-1})/n$ discounted to time step $i = 0$.

[2]A fundamental assumption of least squares method is that the columns of matrix A are linearly independent, therefore $A^T A$ is always invertible.

3 THE FUTHARK LANGUAGE

Futhark is a statically typed parallel functional array language. The language is based on an ML or Haskell style syntax and is equipped with a number of second-order array combinators (SOACs), such as `map`, `reduce`, `scan`, and `filter`. The Futhark surface language features a higher-order module system [17], polymorphism, and limited support for higher-order functions [26].[3] Just as higher-order functions and modules are eliminated entirely at compile time, using a no-overhead approach, arrays of records (or tuples) are turned into records of flat arrays. The language also features a uniqueness type system and explicit sequential **loop** constructs, which, together with support for array-updates, allows for implementing imperative-like algorithm in a functional style.

The Futhark compiler supports aggressive fusion of parallel constructs [23], and specialized code generators for key parallel operators, such as map-scan and (segmented) map-reduce compositions [22, 28]. Essentially, since all available parallelism is assumed to be explicit in the program, Futhark can be seen as a "sequentializing" compiler that attempts to efficiently sequentialize the parallelism in excess of what the hardware can support, thus enabling opportunities for locality-of-reference optimizations.

In this sense, the compiler supports a code transformation, named *moderate flattening* [24], that translates arbitrarily-nested, but *regular*[4] parallelism to a flat form that can be straightforwardly and efficiently mapped to the underlying GPU hardware, in the common case. (Whereas perfectly-nested parallel constructs can easily be translated to flat parallelism, it is not straightforward to do so with imperfectly-nested constructs.)

Furthermore, the Futhark compiler implements a notion of *incremental flattening* [25], which generates multiple code versions in situations where the optimal optimization strategy is sensitive to the input dataset.[5] These are autotuned globally, resulting in one program that offers (in most cases) quasi-optimal performance for all datasets.

However, given a problem exhibiting irregular parallelism, the Futhark programmer is required to implement the flattening strategy by hand, which is not necessarily straightforward. The programmer may choose to use a padding approach to flattening (and thereby treat all problems to be of the same size). Such an approach might not be work efficient,[6] although it might be the most efficient strategy in practice. An alternative strategy is for the programmer to apply a full-flattening approach, which leads to a work efficient algorithm but which may not be efficient in practice.

3.1 Data-Parallel Functional Notation

We use (i) `[n]`α to denote the type of an array whose `n` elements have type α, (ii) `[`a_1`,...,`a_n`]` to denote an array literal, and (iii)

```
1   iota n = [0,...,n-1]
2   replicate n v = [v, ..., v] -- v repeated n times
3   map f [a_1,...,a_n] = [f a_1,..., f a_n]
4   map2 g [a_1,...,a_n][b_1,...,b_n]=[g a_1 b_1,...,g a_n b_n]
5   reduce ⊙ e_⊙ [a_1,...,a_n] = a_1 ⊙...⊙ a_n
6
7   scan^inc ⊙ e_⊙ [a_1,...,a_n] = [a_1,..., a_1 ⊙...⊙a_n]
8   scan^exc ⊙ e_⊙ [a_1,...,a_n]=[e_⊙,a_1,...,a_1 ⊙...⊙a_n-1]
9   sgmscan^inc ⊙ e_⊙ [...,1, 0, ...,0, 1, ...]
10  [...,a_1^k,a_2^k,...,a_n^k, a_1^k+1,...] =
11  [...,a_1^k,a_1^k⊙a_2^k,...,a_1^k⊙...⊙a_n^k, a_1^k+1,...]
12
13  scatter [a_0, a_1, a_2, a_3, ...,a_n-1]
14  [2, -1, 0, 3]
15  [b_0, b_1, b_2, b_3] =
16  [b_2, a_1, b_0, b_3, ...,a_n-1]
17
```

Figure 1: Data-Parallel Array Constructors and Combinators

`(a,b)` to denote a tuple (record) value. Applying a function `f` on two arguments `a` and `b` is written as `f a b` (without any parenthesis or commas).

The semantics of several key parallel operators are presented in figure 1: `iota` applied to integer `n` creates the array containing elements from `0` to `n-1`, and `replicate n v` creates an array of length `n` whose elements are all `v`. `map` produces a result array by applying its function argument `f` to each element of its input array. The function can be declared in the program or can be an anonymous (lambda) function; for example `map (`λ`x->x+1) arr` adds one to each element of `arr`. Similarly, `map2` applies its function argument to (corresponding) elements from its array arguments. `reduce` successively applies a binary-*associative* operator $\oplus$ to all elements of its input array ($e_\oplus$ denotes the neutral element).

`scan` [5] is similar to `reduce`, except that it produces an array of length `n` containing all prefix sums of its input array: the inclusive scan (`scan`^{inc}) starts from the first element of the array, and the exclusive scan (`scan`^{exc}) starts from the neutral element. Segmented scan (`sgmscan`) has the semantics of a `scan` applied on each subarray of an irregular array of subarrays. The latter is represented (i) by a flag array made of zeroes and ones in which a one denotes the start of a (new) subarray, and (ii) by a matching-length flat array containing all elements of all subarrays. For example, flag `[1,0,1,0,0,0,1]` denotes an array with three rows, having two, four and one elements, respectively. Segmented scan can be implemented as a `scan` with a modified operator.

The last operator, `scatter x is vs` updates in place the array `x` at indices contained in array `is` with the values contained in array `vs`, except that out-of-bounds indices are ignored (not updated). For example, in figure 1, value b_1 was not written in the result because its index `-1` is out of bounds.

Finally, the notation supports the usual unary/binary operators and (normalized) `let` bindings, which have the form `let a=`e_1 `let b=`e_2`...in` e_n and are similar to a block of statements followed by a return denoted by keyword `in`. In-place updates to array elements are allowed and are written as `let arr[i]=x`. The notation supports `if` expressions, of form `if c then e2 else e3` and semantics similar to the C ternary operator `c? e1 : e2`, and loop expressions

[3] Functions may not be stored in arrays or returned by conditional expressions.
[4] By "regular" parallelism we mean that the size of the inner-parallel operators is invariant to the outer-parallel nest.
[5] For example, the optimal GPU code for dense matrix-multiplication depends on the sizes of the matrix dimensions: If enough parallelism is available in the outer two parallel levels, then the dot-product dimension should be sequentialized, thus enabling various tiling strategies that optimize temporal locality. Otherwise, the inner dimension should also be executed in parallel.
[6] A parallel algorithm is *work efficient* if the work (i.e., the number of operations) performed by the algorithm is of the same asymptotic complexity as the work performed by the best known sequential algorithm that solves the same problem.

of the form: `loop (x) for i<n do e`. Here, `x` is a loop-variant variable that is initialized for the first iteration with an in-scope variable bearing the same name. `loop` executes iterations `i` from `0` to `n-1`, and the result of the loop-body expression `e` provides the value of `x` for the next iteration.

3.2 An Implementation of the Naive Algorithm

We first present a Futhark implementation of the naive LSMC algorithm, as specified in the original paper. The Futhark function lsmc_naive, which implements the main part of the algorithm is listed in Figure 2. The function takes as arguments (1) a two-dimensional array containing generated paths, (2) the maturity time (T) as a year fraction, (3) a risk-free interest rate r used for discounting, and (4) a payoff function pFun.

4 DESIGN AND IMPLEMENTATION

First of all, we want to emphasize that, in our Futhark implementation of LSMC, we follow the same algorithmic choices as taken by NVIDIA in their CUDA implementation [16]. However, as we could not find any published material on the exact linear algebra transformations that are undertaken, we specify the process in detail in the following sections. Therefore, we present the algorithmic consideration and an efficient approach to an implementation of a financial algorithm for a widespread case of Monte Carlo simulation for American Option Pricing, which is frequently reimplemented across financial institutions. This is, to our knowledge, the first state-of-the-art high-level approach to this particular problem available to the public.

4.1 Main Considerations

The main inefficiency of the naive (straightforward) implementation of the LSMC algorithm is that $A^{\dagger} = (A^T A)^{-1} A^T$ is fully computed at each time step. We optimize this by means of an algorithmic refinement which aims to separate the part of the computation of $A^{\dagger}$ that is intrinsically dependent within the time-step loop from the one that is not. The latter, independent part can thus be precomputed in parallel before the time-step loop is entered. The algorithmic change consists, at a very high level, of working with a QR decomposition of $A = QR$, where the R matrices have small dimensionality (3×3 in our case) and can be efficiently precomputed in parallel for all time steps. With this, the computation inside the time-step loop is reduced to $A^{\dagger} = R^{-1}Q^T$.

Furthermore, the Q^T matrix is not actually manifested in memory, but rather computed on the fly from the sample matrix and fused in the multiplication with R^{-1}. This requires some redundant computation, but significantly decreases the number of accesses to global memory, which are orders of magnitude slower than scalar arithmetic. Finally, the sample matrix is computed in transposed layout in order to optimize spatial locality (i.e., coalesced accesses to global memory on GPU).

4.2 Building the Pseudo-Inverse Efficiently

We take our design goals into consideration and change the algorithm to adhere to parallel computation on GPUs. This approach was first adapted in the original CUDA implementation by NVIDIA that we are trying to match in performance [16, 30].

```
1  let lsmc_naive [paths] [steps]
2      (Ss: [paths][steps]real)
3      (T: real) (r: real) (pFun: real → real)
4    : real =
5    let Sst = transpose Ss
6    let dt = T / (real (steps-1))
7    -- compute discount factors
8    let disc = map (λi → exp(r*real(-(i+1))*dt))
9                   (iota (steps-1))
10   -- prepare initial payoffs
11   let Ps = map (λji →
12               let j = ji / steps
13               let i = ji % steps
14               in if i < steps-1 then zero
15                  else pFun(Ss[j,i])
16               ) (iota (paths*steps))
17   -- iteratively update the payoffs going backwards
18   let (Ps, _) =
19     loop (Ps, h) = (Ps, steps - 1)
20     while h >= 1 do
21       -- compute paths that are in-the-money
22       let pickedpaths =
23          filter (λj → pFun(Sst[h,j]) > zero)
24                 (iota paths)
25       -- prepare for and perform regression
26       let Y = map (λj →
27         map (λi → disc[i]*Ps[j*steps + h+i])
28             (iota (steps-h))
29         |> reduce (+) zero) pickedpaths
30       let Xt = map (λi →
31                     map (λj → Sst[h,j] ** (real i)
32                         ) pickedpaths
33                    ) (iota 3)
34       let X = transpose Xt
35       let beta = Mat.matvecmul_row
36           (Mat.matmul Xt X |> Mat.inv)
37           (Mat.matvecmul_row Xt Y)
38       let exVals =
39           map (λj → pFun(Sst[h,j])) pickedpaths
40       let contVals = map (λj →
41           let sst = Sst[h,j]
42           in (.1) <|
43              loop (racc,sacc) = (0.0,1.0) for k < 3 do
44                 (racc + sacc * regY[k],
45                  sacc * sst)
46         ) pickedpaths
47       let (updInds, updVals) = unzip <|
48         map (λji →
49           let j = ji / steps
50           let i = ji % steps
51           let j' = pickedpaths[j]
52           in if (contVals[j] < exVals[j]) then
53                if (i != h) then (j' * steps + i, zero)
54                else (j' * steps + i, exVals[j])
55              else (-1, zero)
56         ) (iota ((length pickedpaths)*steps))
57       let Ps = scatter Ps updInds updVals
58       in (Ps, h - 1)
59   -- compute the discounted mean
60   let prices =
61     map (λj →
62           map (λi → disc[i]* Ps[j*steps + i+1])
63               (iota (steps-1))
64           |> reduce (+) zero
65         ) (iota paths)
66   in Stats.mean prices
```

Figure 2: Futhark code for the naive LSMC algorithm.

For the sake of brevity, we assume that we work with one system of equations $Ax = b$, although one per each time step i needs to be solved. We follow the standard practice by applying Singular Value Decomposition (SVD) $A = U\Sigma V^T$ to reduce the dimensions of A and build the Moore-Penrose pseudo-inverse of a tall matrix $A^\dagger = V\Sigma^{-1}U^T$. Next, $A^\dagger$ is used to compute the solution $\hat{x} = V\Sigma^{-1}U^T b$.

Yet, our goal is to build $A^\dagger$ efficiently. This is due to the fact that we need to construct one matrix A for each time step. Thereby it is simply too expensive in execution time and memory required to naively compute SVD of A. To remedy this problem, we start with a QR decomposition of $A = QR$, and use the fact that R is much smaller than A. We specialize the algorithm to work with a 3-degree parametrization in terms of a quadratic polynomial. Therefore, R is of size 3×3 in our case. We compute SVD of $R = U_R \Sigma_R V_R^T$ to build the SVD of $A = QU_R\Sigma_R V_R^T$.

The QR factorization is typically performed using householder transformation. However, the naive application of it would mean $3 \times n \times n$ memory access as that many elements would need to be updated. The efficient solution comes from the observation that, in our case, matrix R can be constructed using only 8 scalars:

$$S_0, S_1, S_2, \sum_{i=0}^{itm-1} S_i^0, \sum_{i=0}^{itm-1} S_i^1, \sum_{i=0}^{itm-1} S_i^2, \sum_{i=0}^{itm-1} S_i^3, \sum_{i=0}^{itm-1} S_i^4,$$

where S_i is the asset spot price on the ITM path i. Three first scalars are asset spot prices for the first ITM paths found when traversing the paths from path 0. The first sum can be translated to a total number of ITM paths found. The remaining four sums are consecutive powers $(1, \ldots, 4)$ of spot prices associated with each found ITM path. These scalars are prepared as part of SVD preparation. We implement a custom function that uses these scalars to build the matrix R.

Furthermore, thanks to QR factorization we get a simple formula for the pseudo-inverse. We use the fact that A is left-invertible, so its columns are linearly independent. We have

$$A^T A = (QR)^T (QR) = R^T Q^T QR = R^T R,$$

so

$$A^\dagger = (A^T A)^{-1} A^T = (R^T R)^{-1} (QR)^T = R^{-1} R^{-T} R^T Q^T = R^{-1} Q^T.$$

The final equation that we solve in the main loop for each time step is then as follows: $\hat{x} = R^{-1}Q^T b$. The R^{-1} is precomputed for each time step before the loop using SVD — $R^{-1} = V_R \Sigma_R^{-1} U_R^T$. The orthogonal matrix $Q^T = R^{-T} A^T$ does not need to be stored for each time step and can instead be computed on-the-fly. We again use the fact that (1) R^{-1} is by now already precomputed using SVD and (2) matrix A can itself be computed on-the-fly, where, for our case, each row comprises 3 elements: $1, S_{ij}, S_{ij}^2$, i.e., it can be computed from vector S_i that consists of asset spot prices for ITM paths for a given time step i. Naturally, they need to be processed in transformed form.

4.3 Optimised Algorithm

We do not claim any contributions to this algorithm and instead closely follow the implementation proposed by NVIDIA [16, 30]. We focus on the goal to match the performance of this public benchmark implementation. The obtained algorithm outlined in Figure 3 is implemented using a nested composition of sequential **loop**s and parallel **map**, **reduce**, and **scan** constructs.

```
1   -- Path Generation
2   map(n)
3     loop(m)
4   transpose
5   -- SVD Preparation
6   map(m)
7     loop(n)
8       scan(chunk)
9     map(n) |> reduce(n)
10    map(n)
11  map(m)
12  -- Main Regression Loop
13  loop(m)
14    map(n)
15    map(n)
16  reduce(n)
```

Figure 3: The high-level view of the implemented optimised algorithm structure presented as a combination of parallel constructs. It consists of 3 parts with n denoting the number of paths and m denoting the number of time steps. The `transpose` function performs matrix transposition.

The code in Figure 4 demonstrates the main lsmc_opt function of the optimised algorithm. The function takes as arguments (1) number of time steps m, (2) number of paths n, (3) a function to verify if the option is ITM is_itm (4) a payoff function payoff, (5) the time step size dt as a fraction of year, (6) initial asset spot price at the current day S0, (7) a risk-free interest rate r used for discounting, (8) volatility sigma, (9) seed for Random Number Generator and 2 helper parameters that determine the amount of computation that should be performed sequentially. The function returns the calculated option price.

In the next sections, we give a detailed description of the implementation and optimisation involved in 3 main parts of the algorithm—path generation in Section 4.3.1, SVD preparation in Section 4.3.2, and main regression loop in Section 4.3.3. Each section is accompanied with a code listing presenting a Futhark implementation of a function that implements the given part.

4.3.1 Path Generation. The computational effort of a Monte Carlo simulation is determined by the number of paths and time steps. A large number of paths n, usually 100.000 to 1.000.000, needs to be generated to obtain an accurate value approximation [19]. In American option pricing case, the number of time steps m is bound to the number of early-exercise opportunities and is usually much smaller than n. Path generation part consists of two sub-parts: random number generation and path generation.

For the first part, we use a minimum standard pseudo-random number generator (RNG) in a parallel skip-ahead fashion. We use Newer "Minimum standard" Minimum standard RNG for this purpose. After seeding the RNG, we draw $m \times n$ random samples by splitting the RNG into n sub-RNGs for each path and than sequentially drawing a sample for each time step of the given path. The samples are independent from each other. For the process, that we want to simulate, we need samples drawn from Gaussian (standard

```
1  let lsmc_opt (m: i32) (n: i32)
2      (is_itm: real → i32)
3      (payoff: real → real) (dt: real)
4      (S0: real) (r: real) (sigma: real) (seed: i32)
5      (min_itm: i32) (CHUNK: i32)
6    : real =
7    -- Path Generation
8    let paths = generate_samples_and_paths
9        prng_seed m n S0 dt r sigma payoff
10   -- SVD Preparation
11   let Sst = transpose paths
12   let (svds, all_otms) = prepare_svds Sst is_itm
13       min_itm CHUNK
14   -- Main Regression Loop
15   let expmrdt = exp(-r*dt)
16   let (cashflows, _) =
17     loop (cashflows, i) = (Sst[m - 1], m-2)
18     while i >= 0 do
19       let betas = compute_betas is_itm svds[i] Sst[i]
20           cashflows all_otms[i]
21       let new_cashflows = update_cashflows payoff
22           expmrdt betas Sst[i] all_otms[i] cashflows
23     in (new_cashflows, i - 1)
24   in expmrdt * (reduce (+) zero cashflows) / n
```

Figure 4: Futhark code for the main regression loop.

normal) probability distribution. We achieve it in two steps. First, we draw samples from uniform probability distribution using the RNG. Afterwards, we use the *Inverse Normal Cumulative Density Function* (CDF) to produce normally-distributed samples out of the generated uniforms. As there exists no exact formula for Inverse Normal CDF, we need to use an approximation algorithm. We implement the *Beasley-Springer-Moro* algorithm known for its speed and accuracy following the procedure described in [19].

For the simulation purposes, every sample needs to be turned into an asset spot price instance at every simulation time step (or early-exercise opportunity). We chose the standard *Geometric Brownian Motion GBM*(r, σ^2) with a mean (drift) equal to the risk-free interest rate r and variance (diffusion) equal to a square of volatility σ^2. We use the generated normally-distributed samples to simulate a stochastic process. In practice, as the process is a Markov chain, we know that the current step is independent of the past realizations of the process.

In our case, we deal with a one stochastic factor — the underlying asset spot price, as the American option, that we price, is only dependent on one underlying variable. This means the paths are independent from each other. This allows us to parallelise the generation efficiently across the paths by having one thread generate one whole path. Nevertheless, we want to strike that our assumption of one underlying is not a limitation to parallelism in the implementation. Many stochastic processes could be generated in parallel as long as we would adjust the simulation for the correlations between the stochastic variables, a necessary step in practice.

The code in Figure 5 presents a compact Futhark implementation of path generation. For each path and step, UnifRealDist.rand function draws first a single random number from a uniform distribution. Then function computeGbmNormalStep transforms that number to a one standard normal distribution and computes a current GBM step.

```
1  let computeGbmNormalStep (drift: real) (vol: real)
2      (x: real) : real =
3    drift * exp(vol * (NormRealDist.invNormalCdf x))
4
5  let generate_samples_and_paths (seed: i32)
6      (m: i32) (n: i32)
7      (S0: real) (dt: real) (r: real) (sigma: real)
8      (payoff: real → real)
9    : [n][m]real =
10   let rng = minstd_rand.rng_from_seed [seed]
11   let std_dist = (0.0, 1.0)
12   let rngs = minstd_rand.split_rng n rng
13   let drift = exp((r-0.5*sigma*sigma)*dt)
14   let dtSigma = sigma * mysqrt(dt)
15   in map (λr →
16     let path = replicate m 0.0
17     let (path', _, _) =
18       loop (path, rng, acc) = (path, r, 1.0)
19       for i < m do
20         let (rng, num) = UnifRealDist.rand std_dist rng
21         let W = computeGbmNormalStep drift dtSigma num
22         let acc' = acc * W
23         let v = acc' * S0
24         let v' =
25           if i < m - 1
26           then v
27           else payoff v
28         let path[i] = v'
29         in (path, rng, acc')
30     in path'
31   ) rngs
```

Figure 5: Futhark code for path generation.

Function NormRealDist.invNormalCdf approximates the Inverse Normal CDF of a uniform sample. At the last time step the cashflow is known, as this is the last time the option can be exercised or not. Therefore, the value is set to the payoff.

Performance Enabler. We gain most performance here by fusing the random sample generation and path generation together into one step like in lines 20 – 28 in the code in Figure 5. We perform these actions for each step based on the observation that we work on the same array for both actions as well as each sample is independent from the all other ones. This way we read from and write to the device global memory only once and thereby save the redundant intermediate memory accesses that are costly to execute compared to compute instructions.

4.3.2 SVD Preparation. This part is run before entering the main regression loop and cover the main algorithmic optimization. The main advantage of this approach is that SVD for R in each time step can be processed in parallel. As R is small, the intermediate variables easily fit into registers of one streaming multiprocessor (SM). The number of SVDs to prepare depends on the number of time steps m. This part is compute-intensive, but at the same time the parallelism is limited by the fact that m is usually much smaller than n. A sequential loop is needed to find the first three ITM paths to get the asset spot prices to build R matrix. In the body of prepare_svds, the eight required scalars are gathered and computed. They are subsequently passed to svd_3x3 that performs the QR decomposition and SVD decomposition for R and R^{-1}. It start with assembling the

R matrix from the 8 scalars and afterwards uses the iterative Jacobi method to determine the inverse matrix R^{-1} The function returns 6 upper elements of matrix R and 6 upper elements of the inverse matrix R^{-1} as the matrices are orthogonal.

Performance Enabler. The key to performance here is the fact that we not only perform computation in parallel on the outer level across all time steps m, but we also enable inner parallelism in computation for each of the time steps itself.

First of all, SVD preparation benefits significantly from the intra-group parallelism in the first map (lines 5 – 31 in the code in Figure 6), that finds the first three ITM paths. The spot prices on these paths are needed for construction of matrix R. It works by taking the CHUNK paths at one time and working on them in parallel. it is achieved by a combination of parallel constructs like **maps**, **reduces**, an exclusive scan scanExc and **scatter**. CHUNK parameter depends the level of intra parallelism here. The smaller its value, the faster this part performs. However, it cannot be smaller than a number of itm paths that are needed for constructing R SVDs. In our implementation, it is determined by min_itm parameter and fixed to 4, one more than dimension size of R. The unsafe keyword is a hint to the Futhark compiler to not generate any dynamic checks for boundary or array size conditions in the enclosed expression. This is sometimes necessary for optimized parallel execution.

The next beneficial optimization is the application of segmented reduction in the lines 33 – 48 that enables parallelism in gathering the remaining 5 scalars.

Afterwards, a map in the line 49 works in parallel across time steps, but internally, for each time step, it uses the matching scalars to compute the partial in a sequential manner in the call to svd_3x3.

4.3.3 Main Regression Loop. This part is where the least squares system of equations is regressed, the continuation value is computed and the cashflow per each time step is updated. It can be seen in the code in Figure 4 in lines 16 – 23. This loop has $m-1$ iterations. The computation in the loop is greatly simplified thanks to the SVD preparation that is performed before entering the loop as most of the sequential computation was performed there. The code in Figure 7 shows the implementation. Function compute_betas computes β coefficients required for regression through a multiplication of pseudo-inverse $A^{\dagger}$ and cashflow vector. Afterwards, the update_cashflows estimates a payoff based on βs for each path and determines continuation value for each of them comparing the estimated payoff with a current payoff of the option.

Performance Enabler. Thanks to the SVD preparation before start of the main regression loop, the computational work in each loop iteration is significantly reduced. The other reason is the reduced size of the matrices that are being processed. All the rest is performed in parallel across n paths. As presented in code in Figure 7, compute_betas use a **map** (line 13 followed by a **reduce** (line 21. The update_cashflows follows with one more **map** in line 32. The performance of the loop is highly depenedent on the number of time steps, as they need to be processed sequentially, because of data dependency betweeen consecutive time steps.

```
1  let prepare_svds [m] [n] (Sst: [m][n]real)
2      (is_itm: real → i32)
3      (min_itm: i32) (CHUNK: i32)
4    : ([m][12]real, [m]i32) =
5    let svds = map (λSs →
6      let svds = replicate 12 zero
7      -- loop to find 3 first ITM paths
8      let (svds, _, _, _) = unsafe
9        loop (svds, found_paths, path_offs, exit)
10           = (svds, 0, 0, false)
11       while (!exit && found_paths < 3
12           && path_offs < n) do
13         let Ss_chunked = map (λi →
14             if i + path_offs < n
15             then Ss[i+path_offs]
16             else zero) (iota CHUNK)
17         let itms = map is_itm Ss_chunked
18         let exit = reduce (&&) true <| map (==0i32)
19           itms
20        let scn_ms = scanExc (+) 0 itms
21         let tot_sum = scn_ms[CHUNK-1] + itms[CHUNK-1]
22         let inds = map2 (λin_m sm →
23             if in_m == 1i32 && found_paths+sm < 3
24             then found_paths+sm
25             else -1)
26           itms scn_ms
27         let svds = scatter svds inds Ss_chunked
28         let found_paths = found_paths + tot_sum
29         in (svds, found_paths, path_offs+CHUNK, exit)
30     in svds
31     ) Sst
32   let (ms, sums, all_otms) = unzip3 <|
33     map (λSs →
34       let itms = map is_itm Ss
35       let ms = reduce_comm (+) 0i32 itms
36       let sums = map2 (λin_m S →
37         if in_m == 1i32
38         then (S, S*S, S*S*S, S*S*S*S)
39         else (zero, zero, zero, zero)
40       ) itms Ss
41       |> reduce_comm tuple4_sum_op
42           (zero, zero, zero, zero)
43       let all_otm =
44         if (ms < min_itm)
45         then 1i32
46         else 0i32
47       in (ms, sums, all_otm)
48     ) Sst
49   let svds = map3 svd_3x3 ms sums svds
50   in (svds, all_otms)
```

Figure 6: Futhark code for SVD preparation.

4.4 Motivation for Functional Approach

As a final remark to presented Futhark code listings, we would like to motivate some virtues of functional programming approach that can be especially valuable for non-technical domain experts. Anyone, who programs in parallel low-level languages targeting multi-core architectures on a regular basis, will make an immediate observation that there is no explicit kernel setup or device memory management. Code is agnostic to the underlying platform and most of the performance optimizations are applied by the aggressive optimisation and parallisation compiler that applies non-trivial code analysis. This means that the same Futhark code base is portable

```
let compute_betas [n] (is_itm: real → i32)
    (svds: [SLOTS]real) (Ss: [n]real)
    (cashflows: [n]real) (all_otm: i32)
  : (real, real, real) =
  if all_otm == 1i32
  then (zero, zero, zero)
  else
    let R00 = svds[0]
    -- ...
    -- code omitted
    -- Initalize R and W matrices from svds
    -- and compute inverse of R and W
    in map2 (λS i →
      -- Compute Qis. The elements of the Q matrix
      -- in the QR decomposition.
      -- ...
      let cashflow = if (is_itm S) == 1i32
          then cashflows[i] else zero
      in (WI0*cashflow, WI1*cashflow, WI2*cashflow)
    ) Ss (iota n)
        |> reduce tuple3_sum_op (zero, zero, zero)

let update_cashflows [n]
    (payoff: real → real)
    (expmrdt: real)
    (beta: (real, real, real))
    (Ss: [n]real)
    (all_otm: i32)
    (cashflows: [n]real)
  : [n]real
  =
  map2 (λS path →
    let old_cashflow = expmrdt * cashflows[path]
    in if all_otm == 1i32
      then old_cashflow
      else
        let cur_payoff = payoff S
        let (beta0, beta1, beta2) = beta
        let estimated_payoff =
          (beta0 + beta1 * S + beta2 * S * S) * expmrdt
        in
          if cur_payoff <= estimated_payoff
          then old_cashflow
          else cur_payoff
  ) Ss (iota n)
```

Figure 7: Futhark code for the main regression loop. It consists of a computation of βs for assessing the continuation value with a subsequent update of the cash flows.

across architectures can be compiled to a sequential code running on one CPU core as well as another one running on a massively-parallel GPU. Moreover, when writing Futhark, one focuses more on expressing algorithms using and combining the Second-Order Array Combinators (SOACs) that mimic the higher-order functions found in conventional functional languages. In Futhark, they have sequential semantics, but permit parallel execution, and as such are compiled to parallel code. This means that Futhark focuses more on compilation to high-performance parallel code, but still is expressive enough for implementation of non-trivial programs. The purely functional nature of Futhark allows the compiler to apply more high-level optimisation. In terms of performance portability, Futhark support nested parallelism, so the code can be further autotuned to support efficiently all the available parallelism exposed by the hardware. This is rather a cumbersome, and often impossible, task to perform manually.

5 EXPERIMENTAL RESULTS

In this section, we present and discuss the results for different tests. We validate the accuracy of the simulations and measure the performance of the implementation comparing it against other established benchmarks. We run the experiments on a Linux system with a 26-core 2-way HT Intel Xeon Platinum 8167M CPU (2.00GHz), 754 GB DDR RAM and NVIDIA Tesla **V100** SXM2 GPU (2688 Volta *FP64* cores, 16 GB HBM2) using CUDA 10.1.

5.1 Accuracy

To start with, we compare the pricing results with an established benchmark to validate correctness of our implementation. Table 1 presents comparison of different implementations of American Option Pricing. We are pricing an American put option with a fixed strike and constant risk-free rate. The remaining parameters vary as specified in the original paper by Longstaff-Schwartz [29]. Columns FDM and LSMC stand for the results from the original paper. The two remaining columns comprise results for CUDA implementation that we used as a benchmark algorithm, and our Futhark implementation. That said, we mention that the original LSMC simulation uses antithetic sampling, which cuts the number of random samples in half that leads to reduced variance of the sample paths as well as improvement in the overall accuracy of the simulation.

The simulation results compared to FDM method have a low error. The difference between simulation results varies slightly for different sets of parameters, but, in general, they are insignificant and are the outcome of using different RNGs. The same is valid for both the original CUDA and Futhark implementations.

5.2 Performance Test Case

The pricing test case is presented in Table 2. It is an example of a typical put option that is ITM on the calculation day.

The performance results are presented in Table 3. The correctness is validated against the benchmark binomial tree numerical method for the same problem. We want to obtain a value that is as close as possible to this benchmark. Futhark compiler is able to translate high-level functional language to different backend parallel languages. We test the CUDA (**V1**) and OpenCL (**V2**) backends, and observe a slightly faster (1.9 ms) execution time with **V2**. In terms of the speedups, **Ref** is 1.45× faster than **V2**. Furthermore, we do not observe large discrepancies in terms of partial execution times among the 3 main parts of the algorithm. This fact demonstrates that Futhark auto-generated low-level code is similar in complexity and on par in performance with one that is hand-tuned. The slight differences (3-4%) in **Path** part might be due to choice and an internal implementation of RNGs. We have used a different RNG in comparison to **Ref**, which uses CURAND_RNG_PSEUDO_MRG32K3, a member of the Combined Multiple Recursive family of pseudo-random number generators. The 6% overhead in **Main** is caused by a larger number of memory copies compared to **Ref**. The effect is amplified, because the loop has 99 iterations.

Table 1: Comparison between results from the original paper (finite difference method (FDM) and LSMC) and LSMC implementations in CUDA and Futhark. The strike price of the put option is 40 and the risk-free rate is 0.06. The remaining parameters are as indicated. All LSMC simulations are done with 100.000 paths and 50 time steps per year.

S_0	σ	T	FDM	LSMC	CUDA	Futhark
36	0.20	1	4.478	4.472	4.460	4.465
36	0.20	2	4.840	4.821	4.821	4.826
36	0.40	1	7.101	7.091	7.077	7.092
36	0.40	2	8.508	8.488	8.514	8.518
38	0.20	1	3.250	3.244	3.232	3.239
38	0.20	2	3.745	3.735	3.736	3.739
38	0.40	1	6.148	6.139	6.131	6.147
38	0.40	2	7.670	7.669	7.670	7.661
40	0.20	1	2.314	2.313	2.307	2.313
40	0.20	2	2.885	2.879	2.873	2.878
40	0.40	1	5.312	5.308	5.290	5.319
40	0.40	2	6.920	6.921	6.914	6.909
42	0.20	1	1.617	1.617	1.613	1.612
42	0.20	2	2.212	2.206	2.205	2.205
42	0.40	1	4.582	4.588	4.578	4.590
42	0.40	2	6.248	6.243	6.231	6.234
44	0.20	1	1.110	1.118	1.104	1.104
44	0.20	2	1.690	1.675	1.682	1.680
44	0.40	1	3.948	3.957	3.945	3.952
44	0.40	2	5.647	5.622	5.628	5.637

Table 2: Set of model and simulation parameters for the American option pricing. We provide an option price obtained from a different numerical method (binomial tree) for reference.

Model Parameters	**Value**
Option Type, Payoff	Put, $\max(K - S)$
Initial Spot price (S_0)	80.0
Strike price (K)	90.0
Time to maturity (T)	1 year
Risk free rate (r)	5%
Volatility (σ)	30%
Simulation Parameters	
Time steps/Early Exercise dates	100
Paths	1.000.000
Ref. value (Binomial Tree)	13.804

5.3 Performance Scalability

5.3.1 Fixed number of paths, various number of time steps. As a next step, we test the scalability behaviour of Futhark implementation **V2** in case of varying number of time steps and paths. These are the main parameters that determine the runtime performance of a Monte Carlo simulation. Consequently, we reuse the test case from Table 2 and compare against the benchmark **Ref** for different combinations of these two simulation parameters.

Table 3: Execution times for the test case. Ref is the original CUDA benchmark, while V1 is Futhark compiled to OpenCL and V2 is Futhark compiled to CUDA. Both total and partial execution times for each part of the algorithm are shown. Execution times are given in *ms*. The runtimes are averaged based on 250 runs. *Path* stands for Path Generation part, *SVD* — SVD Preparation, and *Main* for the Main Regression Loop. In Δ column, we compare the speedups against the slowest runtime. The obtained values are presented in *Val* column.

	Path	SVD	Main	Total	Δ	Val
Ref	4.7 (30%)	1.8 (12%)	8.9 (58%)	**15.4**	1.45×	13.778
V1	5.9 (26%)	2.1 (10%)	14.3 (64%)	**22.3**	1.00×	13.789
V2	5.4 (27%)	1.9 (9%)	13.1 (64%)	**20.4**	1.09×	13.789

First of all, we fix the number of paths to a relatively high number 1.000.000 and test against 4 different time step numbers. High number allows for massive parallelism across the path dimension and thus full utilisation of the GPU hardware. Figure 8 shows the results of this experiment. Contributions of three algorithmic parts are distinguished and add up to a total runtime for each tested case.

The main observation is that for low number of time steps up to ~ 50 time steps Futhark **V2** is faster than benchmark CUDA **Ref**. In particular, for very few time steps like 10, it is around 2× faster. The difference diminishes with increasing number of time steps to match at 50 time steps. For more time steps, **Ref** becomes faster than **V2** as shown, e.g., 1.45× faster for the case in Table 3.

The another observation is that for many time steps, the **Main** part becomes the main bottleneck and takes most of the computational time as can be seen in relative performance in the Figure 9. It takes 50% for 10 time steps, but more than 60% for 100. Obviously,

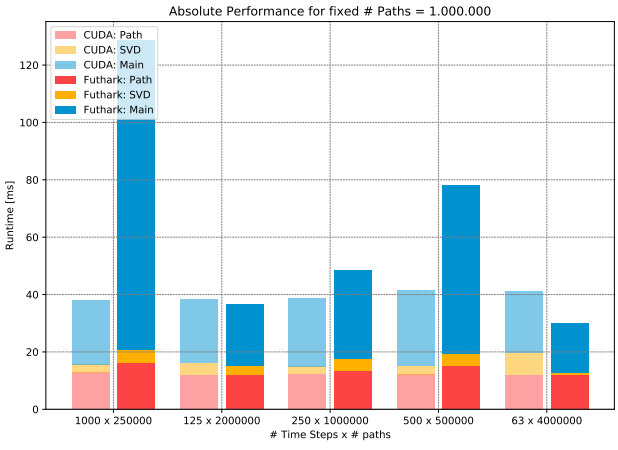


Figure 8: Runtime comparison of CUDA (Ref) and Futhark (V2). Absolute performance is presented for a fixed number of 1.000.000 paths.Runtime is given in *ms*.

this is caused by the amount of computation in the main regression loop that increases with each additional time step added. We compare **Ref** CUDA implementation to auto-generated low-level code produced for **V2** and identify that the reason for degraded Futhark performance is the non-efficient host-device memory communication and accesses. The particular issue is due to a Futhark compiler rule on how to treat operations on scalar variables. An exact situation can be seen in calls to compute_betas and update_cashflows functions in 7, where a tuple of βs is passed between functions. The general rule, plausible in most cases, forces operations on such scalars to be executed on the host and thus the device-to-host memory transfer needs to be initiated. On GPU architectures such transfers introduce significant delays in execution. In our special case, these transfers are redundant and should be omitted, while all intermediate data should be kept on the device. This is exactly what is done in **Ref** and is the reason it becomes faster in this part for larger numbers of time steps. Unfortunately, the compiler has no way to infer this information automatically from the code, and thus needs to be hinted what to do. We are aware of this limitation and plan to address it. This is an encouraging case, disregarding the memory transfers, our implementation seems to deal with the parallel computational work in **Main** more efficiently than **Ref**. This behaviour is specifically observable on the cases with few time steps. Therefore, we conclude that is the part that needs more code optimisation effort, and we claim that the compiler can be adjusted for such a special case. We mention that the performance gap between **Ref** and **V2** widens for large numbers of time steps due to the above issue.

The last observation is that the contribution of **SVD** part decreases with more time steps, while the **Path** increases. The former seems to more efficiently implemented in **V2** as it is always faster than one in **Ref**. The latter is due to the different RNG types and their implementations used in the two compared codes.

5.3.2 Various number of time steps and paths. Figure 10 presents the results for different combinations of time steps and paths. The ratio between them is kept so, that the required work as well as

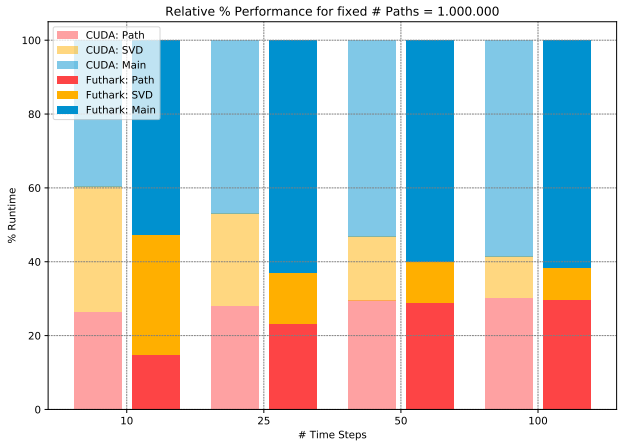


Figure 9: Runtime comparison of CUDA (Ref) and Futhark (V2). Relative (percentage contributions for each part) performance is presented for a fixed number of 1.000.000 paths.

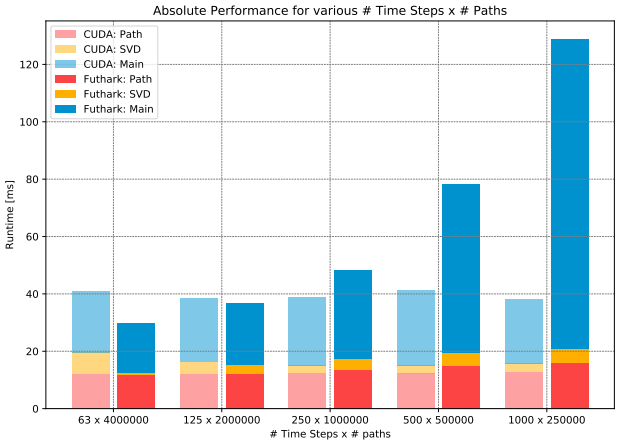


Figure 10: CUDA (Ref) and Futhark (V2) are compared in terms of runtimes. Absolute performance is presented for different combinations of number of time steps and paths. Runtime is given in *ms*.

memory requirements are constant. These cases saturate the memory available on **V100**. We can see that the impact of the time steps on the overall runtime is higher than the number of paths. Again this is caused by the computations in **Main** loop, that need to be run one step at a time. The performance of *Ref* is stable, while Futhark is affected by the redundant memory transfers. In addition, we can see that for low number of time steps, here 63 and 125, **V2** is faster than **Ref**.

6 RELATED WORK

The most efficient implementations of Monte Carlo based American option pricing are implemented in low-level dedicated data-parallel languages and frameworks, such as CUDA [1, 18, 32, 38]. Other efficient parallel implementations are based on task-parallel approaches [12**?**], which are suitable for multi-core architectures. We are not aware of any accelerated implementations of American option pricing using functional languages.

Previous work has investigated the use of Futhark for implementing Monte Carlo based European option pricing [2, 31]. Whereas European option pricing is simpler than (and a special case of) American option pricing, the previous work covered a number of advanced features that the present work does not consider. In particular, the previous work on European option pricing considered European options with multiple underlyings, Sobol sequence generation [21], and options that are so-called *path dependent*, meaning that the price of an option not only depends on the value of the underlying at maturity, but also on intermediate values of the underlying. Using the module language features of Futhark, we believe it will be possible to parameterise the implementation in such a way that multiple underlyings and path dependence are features that are supported by the American pricing engine (we consider such advanced features future work). It is also straightforward to replace the pseudo-random number generation used with Sobol sequence generation.

There are quite a few functional language approaches aiming at generating efficient data-parallel GPGPU code for applications written using high-level array language constructs. Such high-level approaches include Obsidian [14, 35, 36] and Accelerate [11], which are both domain specific languages embedded in Haskell, but, which do not feature arbitrary nested parallelism. Approaches that support arbitrary nested parallelism includes the seminal work on flattening of nested parallelism in NESL [6, 7], which was extended to operate on a richer set of values in Data-parallel Haskell [10], and the work on data-only flattening [39]. However, such general compiler-based flattening is challenging to implement efficiently in practice, particularly on GPUs [3]. Other promising attempts at compiling NESL to GPUs include Nessie [33], which is still under development, and CuNesl [39], which aims at mapping different levels of nested parallelism to different levels of parallelism on the GPU, but lacks critical optimisations such as fusion.

7 CONCLUSION

In this work, we present the results of the accelerated implementation of a well-known LSMC algorithm for a common financial use case of American Option Pricing. We choose to use a high-level functional approach to the implementation, express the algorithm using succinct parallel constructs and let the optimising compiler auto-generate an efficient parallel code that targets massively parallel hardware. For this purpose, we use the Futhark language and address GPUs as a suitable compute platform. We demonstrate that with this approach it is possible to achieve the execution times that are, in general, at the same level as the hand-tuned implementations in dedicated languages like CUDA, but there exist cases, where the implementation beats the benchmark by up to 2×. This promising finding motivates further work on code improvements as we have already identified the performance issue and means to remedy it.

We consider the high-level functional specification as being much more suitable and accessible for the financial-domain experts than the low-level dedicated code, that is usually written by expert software developers. Its expressibility and modularity enables code maintainability, hiding the implementation details targeting particular parallel architecture, and instead turning focus to algorithmic and domain-specific consideration. It also facilitates algorithmic changes, so prevalent in financial industry, e.g. due to a multitude of financial instruments traded in the global markets.

This work has a potential to be integrated as a module in a larger risk management system aimed at large investment portfolios. Such modular and fast pricing capabilities enriched with, e.g., sensitivity calculations, can be combined into standard Value-at-Risk (VaR) or more involved Value Attribution (xVA) risk portfolio analytics [27].

REFERENCES

[1] L.A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. 2014. Pricing derivatives on graphics processing units using Monte Carlo simulation. *Concurrency and Computation: Practice and Experience* 26, 9 (June 2014), 1679–1697. https://doi.org/10.1002/cpe.2862

[2] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (June 2016), 27 pages.

[3] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the Gpu. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/2364527.2364563

[4] Patrick Billingsley. 2012. *Probability and Measure*. John Wiley & Sons. Google-Books-ID: a3gavZbxyJcC.

[5] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.

[6] Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge.

[7] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Communications of the ACM (CACM)* 39, 3 (1996), 85–97.

[8] Stephen Boyd and Lieven Vandenberghe. 2018. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares* (1st ed.). Cambridge University Press.

[9] Jacques F. Carriere. 1996. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance: Mathematics and Economics* 19, 1 (Dec. 1996), 19–30. https://doi.org/10.1016/S0167-6687(96)00004-2

[10] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*. 10–18.

[11] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proc. of the sixth workshop on Declarative aspects of multicore programming*. ACM, 3–14.

[12] Ching-Wen Chen, Kuan-Lin Huang, and Yuh-Dauh Lyuu. 2015. Accelerating the least-square Monte Carlo method with parallel computing. *The Journal of Supercomputing* 71, 9 (Sept. 2015), 3593–3608. https://doi.org/10.1007/s11227-015-1451-7

[13] A. R. Choudhury, A. King, S. Kumar, and Y. Sabharwal. 2008. Optimizations in financial engineering: The Least-Squares Monte Carlo method of Longstaff and Schwartz. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536290

[14] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Work. on Decl. Aspects of Multicore Prog DAMP*. 21–30.

[15] Emmanuelle Clément, Damien Lamberton, and Philip Protter. 2002. An analysis of a least squares regression method for American option pricing. *Finance and Stochastics* 6, 4 (Oct. 2002), 449–471. https://doi.org/10.1007/s007800200071

[16] Julien Demouth. 2014. Monte-Carlo Simulation of American Options with GPUs. http://on-demand.gputechconf.com/gtc/2014/presentations/S4784-monte-carlo-sim-american-options-gpus.pdf. Presentation at NVIDIA GPU Technology Conference.

[17] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 97 (July 2018), 30 pages.

[18] Massimiliano Fatica and Everett Phillips. 2013. Pricing American Options with Least Squares Monte Carlo on GPUs. In *Proceedings of the 6th Workshop on High Performance Computational Finance (WHPCF '13)*. ACM, New York, NY, USA, 5:1–5:6. https://doi.org/10.1145/2535557.2535564 event-place: Denver, Colorado.

[19] Paul Glasserman. 2004. *Monte Carlo methods in financial engineering*. Springer, New York.

[20] Espen Gaarder Haug. 2007. *The Complete Guide to Option Pricing Formulas* (2nd ed.). McGraw-Hill Education, New York.

[21] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2018. Modular Acceleration: Tricky Cases of Functional High-Performance Computing. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC '18)*. ACM, New York, NY, USA.

[22] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, NY, USA, 17–24. https://doi.org/10.1145/2935323.2935326

[23] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 47–58.

[24] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[25] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

[26] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming (TFP'18)*.

[27] John Hull. 2018. *Risk management and financial institutions* (5th ed.). John Wiley & Sons, Inc, Hoboken, New Jersey.

[28] Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

[29] Francis A. Longstaff and Eduardo S. Schwartz. 2001. Valuing American Options by Simulation: A Simple Least-Squares Approach. *The Review of Financial Studies* 14, 1 (Jan. 2001), 113–147. https://doi.org/10.1093/rfs/14.1.113

[30] NVIDIA. 2014. NVIDIA Developer Blog Code Samples repository at GitHub. https://github.com/NVIDIA-developer-blog/code-samples/tree/master/posts/american-options.

[31] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial Software on GPUs: Between Haskell and Fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364474.2364484

[32] Gilles Pagès and Benedikt Wilbertz. 2012. GPGPUs in computational finance: massive parallel computing for American style options. *Concurrency and Computation: Practice and Experience* 24, 8 (2012), 837–848. https://doi.org/10.1002/cpe.1774

[33] John Reppy and Nora Sandler. 2015. Nessie: A NESL to CUDA Compiler. Presented at the *Compilers for Parallel Computing Workshop (CPC '15)*. Imperial College, London, UK.

[34] Albert N. Shiryaev. 2016. *Probability-1* (3rd ed.). Springer-Verlag, New York. https://www.springer.com/gp/book/9780387722054

[35] Joel Svensson. 2011. *Obsidian: GPU Kernel Programming in Haskell*. Ph.D. Dissertation. Chalmers University of Technology.

[36] Joel Svensson, Mary Sheeran, and Koen Claessen. 2011. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages (IFL'08)*. Springer-Verlag, Berlin, Heidelberg, 156–173. http://dl.acm.org/citation.cfm?id=2044476.2044485

[37] J. N. Tsitsiklis and B. Van Roy. 2001. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks* 12, 4 (July 2001), 694–703. https://doi.org/10.1109/72.935083

[38] Shuai Zhang, Zhao Wang, Ying Peng, Bertil Schmidt, and Weiguo Liu. 2017. Mapping of option pricing algorithms onto heterogeneous many-core architectures. *The Journal of Supercomputing* 73, 9 (Sept. 2017), 3715–3737. https://doi.org/10.1007/s11227-017-1968-z

[39] Yongpeng Zhang and Frank Mueller. 2012. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP'12)*. IEEE Computer Society, Washington, DC, USA, 340–349.

A New View on Parser Combinators

Pieter Koopman
pieter@cs.ru.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

Rinus Plasmeijer
rinus@cs.ru.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

ABSTRACT

Parser combinators offer a concise and fast way to produce reasonably efficient parsers. The combinator libraries themselves can be small and provide an elegant application of functional programming techniques. They are one of the success stories in functional programming that are also ported to many other languages.

In this paper, we illustrate that we can make the parser combinators more general by modeling them as a tagless domain specific language. The idea is to replace the ordinary combinators by a set of type constructor classes. By making different implementations of this class we can assign various interpretations of one and the same grammar specification. The set of type classes makes the DSL type-safe and extendable without needing to change existing parts and implementations. This enables us to make multiple interpretations, views, of the specified grammar. In this paper we show views for deterministic parsing, nondeterministic parsing, generating possible parse trees produced by the grammar without needing the corresponding input, generating inputs accepted by the grammar, adapting the grammar rules such that the parser combinators can handle left-recursion and so on. This makes our multi-view parser combinators more powerful than the existing approaches.

CCS CONCEPTS

• **Software and its engineering** → **Parsers**; **Domain specific languages**.

KEYWORDS

Parser Combinators, Domain Specific Language, Tagless

ACM Reference Format:
Pieter Koopman and Rinus Plasmeijer. 2020. A New View on Parser Combinators. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Parser combinators are one of the success stories of functional programming. A small library of operators enables the construction of parsers in a simple way. It is a perfect tool to construct a parser for programming languages and external domain specific languages. The topic is part of many courses in functional programming. Due to these successes of the libraries, the concept is also ported to non-functional languages like Java, Go, C#, F#, Python and many more. Libraries like Parsec show that the concept of parser combinators is much more than a useful toy, they can be used to generate industrial-strength parsers. Moreover, parser combinators have their own page in Wikipedia.

Parser combinators have a long tradition, starting with Bruge in 1975 [3]. The combinator expression of the parser directly corresponds to grammar implemented. The full host language can be used as "macro" language for the grammar to be implemented as well as the parse tree generated. An additional advantage of combinator based parsers is that no external tools are needed for the construction of the parsers.

As a consequence of the success of parser combinators and the ease to build a new library meeting certain requirements, there are very many libraries, each of them with their advantages and disadvantages. Each library has its interface and associated use. Often the user has to choose a library perhaps before all requirements are known and is tied to the library even if the needs change during development, e.g., for a liberal library allowing many grammars to a fast library with special requirements on the grammar. In this paper, we define an embedded domain specific language, DSL, to specify parsers and parse tree generation. Using a tagless DSL [4], with the same architecture as the mTask language for Task Oriented Programming of the IoT [12, 18], we can have many interpretations of the same DSL for grammars and associated parse trees. We show that using such a DSL not only allows different parsing algorithms for the specified grammar but also allow different possibilities; in this paper we show how to generate parse trees without needing input, generate inputs that will be accepted by the grammar and show the grammar without the clutter from parse tree generation.

Well known tools for parser generation are YACC and LEX and their variants like Bison, Flex etc. [10, 17]. These tools use a DSL for the specification of the grammar to be recognized and the parse tree to be generated, or the tokens to be produced as an intermediate step. The advantage of having such a tailor-made tool is that it can have a sophisticated implementation that checks and transforms the grammar to obtain an efficient implementation. The disadvantage is that a fixed external tool is used. When we need something new,

like position-dependent code for the offside rule, it might be hard to implement this with the existing tools.

A parser combinator library is much easier to adapt for special needs than a standalone tool. This contributes to the success of these libraries but also resulted in a huge number of libraries. Most of these libraries have their own advantages. Some are small and simple, and hence easy to adapt for new features like position-dependent code. Other libraries can model non-deterministic grammars and happily start backtracking if a particular branch of the grammar fails to recognize some input. Typically using the list of the successes pattern [26]. The price to be paid for the ability to handle nondeterministic grammars is that parsing can be inefficient. As a reaction, other libraries impose restrictions on the grammar encoded, like no left-recursion and left factoring [1, 19, 21], but offer an efficient parser in return, e.g., [13, 16]. The library Parsec is probably the most famous example [16]. It was originally developed for Haskell, but is ported to languages like C#, F#, Go, Java, Javascript, Lisp, OCaml, Python, R, and Rust. Yet other libraries offer better error messages if the input does not fit the grammar implemented, can do error correction by finding the smallest change of the input needed to be recognized by the parser, are able to handle left-recursive grammars in some way, transform the grammar to optimize parsing and so on [2, 5, 9].

Also, the architecture of the libraries is different. Some libraries have their own ad-hoc set of combinators for parsing, other combinators are based on arrows [8] or monads [27]. Using an arrow or monad based approach has the advantage that many of the combinators and their use will be familiar for programmers. The monadic approach is more general than the arrow-based design since it allows context-sensitive parsers.

In Section 2 we define a new DSL used to describe grammars and parse tree generation. For users of Monadic parsers libraries it will look rather familiar, but it offers many interpretations instead of just one. Next, we define two implementations to generate parsers. The deterministic implementation (Section 3) is efficient, but only succeeds if the grammar obeys conditions like the right ordering of alternatives and the absence of left-recursion. The nondeterministic implementation (Section 4) might be slower but is much more liberal to the grammars accepted. In Section 5 we show how we can easily handle left-recursive grammars by limiting the recursion depth of the nonterminals. We do not require any special action of the user of the library. Other implementations can be added in the same way.

To show that the DSL can be used in completely different ways we show how to generate parse trees that can be generated by the parser in Section 6. Such trees can, for instance, be useful to test other parts of a compiler. We do not need the inputs corresponding to the generated parse trees. Similarly, we can also generate inputs that will be accepted by the parser. This interpretation of the DSL is shown in Section 7. Next, we show how we can obtain a textual version of the grammar from the parser in Section 8. Finally, we discuss the possibilities to transform the grammar automatically in Section 9. Such a transformation would, for instance, transform a left-recursive grammar to non-left-recursive version. This is appealing to increase the efficiency of parsing. Unfortunately, the availability of arbitrary transformation of the parse result makes this in general impossible. Only by invoking severe restrictions on the (intermediate) parse tree produced such a transformation would be feasible.

The contribution of this paper is that it shows how the representation parser combinator library as type class-based DSL can be used to construct type-safe parsers that have multiple useful interpretations of the specified parser.

2 THE PARSER COMBINATOR DSL

Parser combinators libraries have a long tradition in functional programming. The general idea is that there is a small number of primitive parsers. For instance, parsers to recognize integer denotations, keywords and identifiers. Next, there are combinators to compose more complex parsers form simpler parsers. Typical combinators denote the sequential composition of simpler parsers or the choice between two or more parsers. We will introduce the basic parsers and combinators used in this paper on-the-fly.

For the implementation of the parser DSL we use Clean in a version from June 2019 from https://clean.cs.ru.nl/Clean. However, we do not use any specific language aspects of Clean, nor very new features. Any functional language with higher-order functions, lazy evaluation and type constructor classes will do.

As primitive parsers, we introduce just three classes. The primitive `int` recognizes an integer denotation in the input or fails. The `idn` tries to recognize an identifier, typically an alphabetic character followed by a sequence of characters or digits. The `lit` recognizes the literal described by the given string or fails. It is convenient to introduce a class `eof` to detect the end of the file. This is useful to ensure that a parser consumes the entire input. The type variable `v` in those classes describes the view.

```
class int v :: v Int
class idn v :: v String
class lit v :: String → v String
class eof v :: v Bool
```

For a full-fledged parser library, we need more primitives, like denotations for doubles and other basic types. In this paper, we stick to this limited set that is sufficient to demonstrate the possibilities.

Our parser combinators are monad based like most libraries. This implies that sequences of grammar components can be composed by the monadic operations `<*>`, `>>=` and `>>|`[1]. The choice between alternative parts of a grammar is indicated by the operator `<|>` for the class `Alternative`.

We make a small deviation from the usual monadic definitions. Our sequence operator `>>|` is not defined as a fixed

[1] In Haskell the sequence operator >>| is called >>. In Clean >> denotes a bitwise shift of integers, just like Java, C++ and C.

definition based on ≫=, but defined as an own class with a default implementation[2].

```
class (>>|) infixl 1 m :: (m a) (m b) → m b | Monad m
instance >>| m where (>>|) ma mb = ma ≫= λ_ → mb
```

If we do not define an instance for a specific class this default will be used. This gives us the possibility to provide a tailor-made implementation for some types. We will use this in Section 8.

For convenience we introduce two additional constructs; `alt` contains a list of alternatives and `list` turns a list of parsers into a parser that returns the list of their values. Both operators are defined again as a class with a default implementation.

```
class alt v :: ([v b] → v b) | Alternative v
instance alt v where alt = foldr (<|>) empty

class list v :: ([v a] → v [a]) | Functor, pure, <*> v
instance list v where
  list = foldr (λa x.fmap cons a <*> x) (pure [])
```

Using the monadic style has as advantage over the arrow style that we can write context-sensitive parsers in the monadic style. The difference is indicated by the type of the sequence operator[3].

```
seqMonad :: (Parser a) (a → Parser b) → Parser b
seqArrow :: (Parser a) (Parser (a→b)) → Parser b
```

Using the primitives we can make a context sensitive parser `context` that recognizes the first identifier twice. The `idn` recognizes an arbitrary identifier. The recognized identifier value is given as argument to the parser `lit` recognizing the second occurrence of the literal.

```
context :: v String | gram v
context = idn ≫= lit
```

This context sensitivity illustrates that the constructed parser are not restricted to context free grammars.

The class `gram` used here as type constraint is just the collection of grammar classes introduced in this paper. It is often more convenient to use this collection than to list all classes the are actually used.

Although nonterminals in the grammar can be defined by functions in the host language Clean, we have good reasons to define a primitive for these definitions in our parser DSL. When the nonterminals are an explicit element of the DSL we can implement it as we need instead of using the default interpretation of functions from the host language. In Section 5 we use this to make a recursive descent parser that can handle left-recursive grammars.

In this definition `v` is again the view and `t` the type of the nonterminal defined. The provided default implementation just feeds the body of the definition to every occurrence of nonterminal. The type `In` defines the infix constructor `In`. Basically, this is just a tuple in a somewhat different syntax; the constructor `In` replaces the comma from the tuple and the parentheses can be omitted. This implies that we can use `a In b` similar to `(a,b)`.

```
class nt v t :: (t → In t (v a)) → v a
instance nt v t where nt f = let (p In b) = f p in b

:: In a b = In infix 0 a b
```

The class `nt` behaves similar to let-definitions. The advantage of having definitions as a type constructor class in our DSL is that we can change the semantics of the definitions.

A very simple example illustrates the use of the pseudo keyword `In`. We define a nonterminal named `diamond` and define a parser to recognize two of them. A `diamond` is defined as a sequence of the literals `<` and `>`.

```
diamonds :: v String | gram v & nt v (v String)
diamonds =
  nt λdiamond → ((lit "<" >>| lit ">") In (diamond >>| diamond))
```

Note that we use a lambda-expression as function to bind the name `diamond` properly. The identifier `diamond` in the DSL is an ordinary identifier in the host language Clean. Hence, it is type-checked by the compiler like any other identifier to guarantee type correctness statically. By omitting superfluous parentheses, replacing the → in the lambda-expression by a =, changing the layout somewhat and using type derivation this definition can equally be written as:

```
diamonds =
  nt λdiamond = lit "<" >>| lit ">"
  In diamond >>| diamond
```

A typical example of nonterminal definitions is a grammar that describes nested identifiers where each identifier is matched by its identical twin at the other side of the nesting. The result of this parser is the nesting depth matching identifiers.

```
nested :: v Int | gram v & nt v (v Int)
nested =
  nt λn = (idn ≫= λv. n ≫= λd. lit v >>| pure (d + 1))
          <|> pure 0
  In n
```

This example shows that the nonterminal `n` can be used recursively. It is very well possible to use arguments for nonterminals introduced in this way.

For instance, the grammar that specifies an integer value followed by the recognized number of identifiers can be specified by `nIdn`. The argument `n` of `rep` is the number of identifiers to recognize. The result is the list of recognized identifiers.

```
nIdn :: v [String] | gram v & nt v (Int → v [String])
nIdn =
  nt λrep = (λn.if (n=0) (pure []) (fmap cons idn <*> rep (n-1)))
  In int ≫= rep
```

The definitions of nonterminals can be nested just like any other expression. This is illustrated by the definition `exp` for simple numerical expressions.

```
exp :: (v Expr) | gram v & nt v (v Expr)
exp =
  nt λbasic.pure Int <*> int In
  nt λterm.(pure Op <*> basic <*> lit "*" <*> term) <|> basic In
  nt λexp. (pure Op <*> term <*> lit "-" <*> exp) <|> term In
  exp
```

[2] In Haskell one separates the types of function arguments by a → and writes class constraints like `Monad m` in front of the function type instead of after it.

[3] In Haskell these types are written as:

```
seqMonad :: Parser a → (a → Parser b) → Parser b
seqArrow :: Parser a → Parser (a → b) → Parser b
```

There are three issues with this definition. First of all the binding direction of the operators is right-associative, while the common mathematical semantics prescribes association to the left. The simplest solution is to swap the arguments for the nonterminal `term` to a left recursive grammar rule: `nt λterm. (pure Op <*> term <*> lit "*" <*> basic) <|> basic`. The same applies to the multiplication, but since that operator is commutative it is less of a problem.

The next issue is that this left recursive formulation contains a repeated part `basic`. It can be the start of a `term` followed by a literal `*` and a `term`, as well as a term on its own. This yields an overly verbose grammar. This is a common pattern in grammars and hence we define a special operator, `?>=`, for this. We define this operator again as a class with a default implementation. It yields either a parsed function applied to the parsed argument, or just the parsed function argument.

```
class (?>=) infixl 1 m :: (m (a→a)) (m a) → m a
                        | Monad, Alternative m
instance ?>= m where (?>=) f x = f <*> x <|> x
```

This allows us to define the nonterminal for terms concisely as `nt λterm.pure Op <*> term <*> lit "*" ?>= basic`.

The final problem with the definition of nonterminals is that their scope is limited to the body and the expression after the `In`. This implies that we cannot define that an expression between parentheses is also a `basic` since `exp` is not in scope inside `basic`. This can be easily solved by defining multiple nonterminals at once. In this example we also allow the addition operator.

```
exp2 :: v Expr | gram v & nt v (v Expr, v Expr, v Expr)
exp2 =
  nt λ(basic, term, expr) =
    (pure (λl e r.e) <*> lit "(" <*> expr <*> lit ")" <|>
     pure Int <*> int                                          // basic
    ,pure Op <*> term <*> lit "*" ?>= basic                    // term
    ,pure Op <*> expr <*> (lit "-" <|> lit "+") ?>= term       // expr
    ) In expr
```

Since the grammar rules for `term` and `expr` are left recursive we need a clever implementation that prevents unbounded recursive calls. We will discuss that in Section 5.

3 DETERMINISTIC PARSING

The simplest implementation of the parser combinators makes a deterministic parser. Either the input is recognized as specified in the grammar, or parsing fails. The type used for this view of our parser combinators is:

```
:: DParse s a = DParse ([s]→Maybe (a, [s]))
```

In this type `s` is the type of input symbols and `a` is the result type of the items produced by the parser combinator. The input is a list of symbols of type `[s]`. Since parsers can fail the result is a maybe tuple of the result and the remaining input. The implementation of the monadic operators is completely standard. In this paper, we will omit the implementation of these operators unless it is worthwhile to say something about it.

The class `next` produces the next input symbol (if it exists). The implementation for `DParse` is simple.

```
class next v :: v s s

instance next DParse where
  next = DParse λs.case s of
    []    = Nothing
    [a:x] = Just (a, x)
```

The implementation of the basic parser combinators is straightforward. Whenever necessary we assume strings as input symbols in this paper. Assuming a concrete type for input symbols is only required for the basic combinators like `int`, other combinators are independent of the type of tokens.

```
instance int (DParse String) where
  int = next >>= λs.let i = toInt s in s == toString i ? pure i
instance idn (DParse String) where
  idn = next >>= λ s. size s> 0 && isAlpha s.[0] ? pure s
instance lit (DParse String) where
  lit x = next >>= λs.x == s ? pure x

(?) infix 1 :: Bool (v a) → (v a) | Alternative v
(?) b m = if b m empty
```

For all other operators the default implementations suffice.

For deterministic grammars, this implementation works fine and efficiently. For nondeterministic grammars, this implementation produces the first successful parse.

A famous example of nondeterminism is the dangling else of conditional expressions with an optional else part. For the input **`if c then s if c then s else s`** the else part can be assigned to the first as well as the second if. A parser for conditionals with an optional else part in our grammar DSL is:

```
conditional =
  nt λcondition = tip <*> lit "c" In
  nt λstatement =
    alt
      [node "C1" <*> list [keyword "if", condition
                          ,keyword "then", statement]
      ,node "C2" <*> list [keyword "if", condition
                          ,keyword "then", statement
                          ,keyword "else", statement]
    ,tip <*> lit "s"
    ] In pure K <*> statement <*> eof
```

The parse tree constructed by this parser is a general Rose tree. Apart from the datatype we define some convenience functions to obtain concise parser expressions. This Rose tree is used at several places in this paper to hold parse results.

```
:: Rose a = Rose a [Rose a]

bin :: v ((Rose a) a (Rose a) → Rose a) | pure v
bin = pure λl o r.Rose o [l,r]

tip :: v (a → Rose String) | toString a & pure v
tip = pure λa.Rose (toString a) []

keyword :: String → v (Rose String) | pure, lit, <*> v
keyword k = tip <*> lit k

leaf :: a → b (a → Rose a) | pure b
leaf a = pure λb.Rose a [Rose b []]

node :: a → b ([Rose a] → Rose a) | pure b
node a = pure λl.Rose a l
```

Parsing input `["if","c","then","if","c","then","s","else","s"]` with the grammar `conditional` in this deterministic view fails. When we swap the alternative C1 and C2 in the parser `conditional` it recognizes this input successfully. The else part is bound to the second condition.

This shows that the deterministic parser can handle non-deterministic grammars in a limited way. The programmer has to order the alternatives carefully to ensure success in most situations. Based on this order of alternatives the parse will produce one of the possible parse results or fail.

To select this interpretation of the grammar expressions and apply it to an input, we define the function `DParse`. The type of this function as well as the constructor `DParse` select the required view.

```
dparse :: (DParse s a) [s] → Maybe (a, [s])
dparse (DParse p) input = p input
```

4 NONDETERMINISTIC PARSING

A non-deterministic view makes the development of parser much easier because the user does not have to arrange the grammar rules such that the first result found is the required result. The nondeterministic parser combinators produce all possible results and the desired result can be selected later. A typical requirement on the results is that it consumes the entire input. Often we only want the first result that consumes the entire input. Producing all results instead of a single result has a runtime penalty, in the worst case, it can have exponential complexity. For well-designed parsers, the parse result that consumes the entire input is produced in a reasonable time. The potential exponential complexity can usually be avoided in practice.

We use Wadler's idea to replace failure by a list of successes [26]. The basic idea for the implementation is to replace the type `DParse` by `Parse`.

```
:: Parse s a = Parse ([s] → [(a, [s])])
```

A nondeterministic parser is a function form a list of inputs symbol to a list of parse results. Each parse result is a tuple of the actual result and the remaining input.

In the next section, we reuse the nondeterministic parsing view and adapt it to handle left-recursive grammars. To be prepared for this we replace the input by a record containing the input as well as some administration that will become clear in the next section.

```
:: Parse s a = Parse ((ParseState s) → [(a, ParseState s)])

:: ParseState s =
  {input   :: [s]
  ,freshId :: ID
  ,rules   :: Map ID Int
  ,length  :: Int
  }
:: ID := Int
```

The first step is to implement the monadic operators for this type. This is again completely standard and omitted for brevity. For the added operators we use the default implementation. This leaves just the basic parser operations to be implemented. As a basis, we use a tailor-made instance of `next`. Using this the implementation of the basic parsing combinators becomes identical to their deterministic equivalent.

```
instance next Parse where
  next = Parse λs.case s.input of
    []    = []
    [a:x] = [(a, {s & input = x, length = s.length - 1})]

instance int (Parse String) where
  int = next >>= λs.let i = toInt s in s == toString i ? pure i
instance idn (Parse String) where
  idn = next >>= λ s. size s> 0 && isAlpha s.[0] ? pure s
instance lit (Parse String) where
  lit x = next >>= λs.x == s ? pure x
```

Parsing the ambiguous conditional example from the previous section in this view produces the required list with two possible parses. In the first result, the dangling else-part is assigned to the second conditional and in the second result, it is recognized as part of the first conditional.

To select this view, we introduce a function `parse`. In this function we select only the parse results that consume the entire input.

```
parse :: (Parse s a) [s] → [a]
parse (Parse p) input
  = [ a
    \\ (a, s) ← p { input = input
                  , freshId = 0
                  , rules = newMap
                  , length = length input
                  }
    | isEmpty s.input]
```

5 LEFT-RECURSIVE GRAMMARS

In a left-recursive grammar rule, a nonterminal is encountered recursively in its definition without consuming any input. Many languages can conveniently be described by a left-recursive grammar. The numerical expressions from Section 2 are a well-known example.

Left-recursion is a notorious source of nontermination in many parsers. In both of the previous views, a left-recursive grammar causes an infinite amount of recursive calls. This yields a non-terminating program or a runtime error like stack overflow.

There are well-known transformations of grammars to remove left-recursion. We illustrate this with a simple example of expressions containing only integers and the subtraction operator. The left recursive version is

```
eLeft :: v Expr | gram v & nt v (v Expr)
eLeft =
  nt λi = pure Int <*> int In
  nt λe = (pure Op <*> e <*> lit "-" <*> i) <|> i In e
```

In a simple approach we replace this by a right recursive variant.

```
eRight :: v Expr | gram v & nt v (v Expr)
eRight =
  nt λi = pure Int <*> int In
  nt λe = (fmap Op i <*> lit "-" <*> e) <|> i In e
```

Although this obviously recognizes the same inputs, the constructed parse trees are different. The input `3 - 2 - 1` is recognized by `eLeft` as `(3 - 2) - 1` while parsing it with `eRight` recognizes this as `3 - (2 - 1)`. The first result evaluates to 0, corresponding with the common mathematical interpretation of the input, while the second one evaluates to 2. Hence, the different parse tree of `eRight` is wrong.

The left-corner transformation is an algorithm to transform possible left-recursive grammars to right-recursive grammars [2, 9, 19, 21]. It solves the non-termination problem for left-recursive grammars, but there are two major disadvantages.

First, the parse tree changes and the semantics of the language changes accordingly. This is illustrated by the `eRight` example above. Second, the transformed grammar tends to become much bigger than the original grammar. Most implementations try to reduce the size of the generated grammar in some way.

A way to prevent problems with left-recursion is by offering library support to create the desired parse tree and avoid the left-recursion trap. For our parsers we add a primitive `kleene` to mimic the Kleene star operator: repeat the given parser as often as needed and collect the results in a list. This is used in the combinator `chainl` that is parameterized by parsers for the elements and operator.

```
kleene :: (v a) → v [a] | Alternative v
kleene p = fmap cons p <*> (kleene p) <|> pure []

chainl :: (v a) (v (a → a → a)) → v a | Alternative v
chainl p op
  = fmap (foldl (λl (o,r).o l r)) p <*>
    (kleene (fmap (λf a.(f,a)) op <*> p))

eChain :: v Expr | gram v
eChain
  = chainl (pure Int <*> int) (fmap (λo l r.Op l o r) (lit "-"))
```

This solves the problem with left-recursion. The price to be paid is that the user of the library has to spot left-recursion in his grammar and replace it with an application of `chainl`.

One of the attractive features of parser combinator libraries is that one can easily write parsers by directly following the grammars of the language to be recognized. This is seriously compromised by the requirement to replace left-recursion by right-recursion or an application of `chainl`. Fortunately, there are other solutions. Frost gives an overview of possible solutions before introducing a new efficient algorithm [6]. All approaches mark the (possible left-recursive) grammar rules in one way or another to limit the recursion depth. One approach is to make a memoization table for each rule at each input position. Recursion is stopped when no new items can be added to the table, e.g. [7]. Memoization has as additional advantage that it improves the efficiency by preventing the recomputation of parse results in different contexts. There is several related approaches to make parser combinators that can handle arbitrary grammars using advanced data-structures, see e.g. [20, 25]. The data-structures are used to represent all, relevant, derivations.

Our approach is to limit the maximum recursion depth to the length of the remaining input. The idea is that each application of a grammar rule should consume at least one input token. Kuno seems to be the first to use this idea [15]. The implementation of this algorithm very simple compared to the approaches mentioned above. Our approach is very memory efficient, but also less time efficient.

In our DSL for grammars, nonterminals are defined as a function in the `nt` construct. This implies that we can control the function calls by defining an appropriate instance of `nt`. As a simple demonstration of the possibilities, we will limit the number of function recursive calls of each nonterminal to the length of the remaining input. The determination of the length of the input does not need to be very accurate, any upper bound will do. For instance, we can use the remaining length of the input, or even the total length of the input, in characters as a safe approximation of the number of remaining input tokens. To limit the number of function calls we use the additional fields of the `ParseState` defined in Section 4. We assign a fresh identifier to each nonterminal and maintain a mapping from identifiers to allowed calls in the `ParseState`. The instances of `nt` for a single parser and a tuple of parses are given below. Other instances can be added by need following the same pattern.

```
instance nt (Parse s) (Parse s a) where
  nt f = fresh >>= λi. let (p In b) = f (safe i p) in b
instance nt (Parse s) (Parse s a, Parse s b) where
  nt f =
    fresh >>= λi.
    fresh >>= λj.
    let ((p, q) In b) = f (safe i p, safe j q) in b
```

The helper functions to generate a fresh `ID` and to check if a function call is safe are:

```
fresh :: v s ID | freshl v & Functor (v s)
fresh = Parse λs=:{freshId = i}.[(i, {s & freshId = i + 1})]

safe :: ID (Parse s a) → Parse s a
safe i p=:(Parse f) = Parse check where
  check s = case get i s.rules of
    Nothing = f {s & rules = put i s.length s.rules}
    Just c  | c < 0
      = [] // terminate, too many recursive calls
      = f {s & rules = decr i s.rules}
```

These definitions works well to parse any mutual and left-recursive grammar that requires no more than one recursive call to consume one input token. By multiplying the upper bound by n they work well for grammars requiring n recursive calls to consume a single token. They work perfect for the grammar `exp` for mathematical expressions introduced above.

```
exp :: v Expr | gram v & nt v (v Expr,v Expr,v Expr)
exp =
  nt λ(basic, term, expr) =
    (pure Int <*> int
     <|> pure (λl e r.e) <*> lit "(" <*> expr <*> lit ")"
    ,pure Op <*> term <*> lit "*" ?>= basic
    ,pure Op <*> expr <*> (lit "-" <|> lit "+") ?>= term
    ) In expr
```

Also grammars from natural languages can be expressed in this way. A famous example by Tomita [24] is a grammar for sentence in a natural language.

```
tomita =
  nt λ(verb, det, noun, prep, np, pp, vp, s) =
    (leaf "verb" <*> lit "saw"
    ,leaf "det" <*> alt (map lit ["a", "the"])
    ,leaf "noun" <*> alt (map lit ["i", "man", "park", "bat"])
    ,leaf "prep" <*> alt (map lit ["in", "with"])
    ,node "np" <*> alt (map list [[noun], [det, noun], [np, pp]])
    ,node "PP" <*> list [prep, np]
    ,node "vp" <*> list [verb, np]
    ,node "s" <*> alt (map list [[np, vp],[s, pp]])
    ) In
    s
```

The result is a `Rose` tree as defined in Section 3. Our implementation happily recognizes sentences like `["i", "saw", "a", "man", "in", "the", "park", "with", "a", "bat"]`. This example has 5 results with the parser described in this section. The deterministic parser from Section 3 fails with a runtime error due to the left-recursion in `np` and `s`.

Frost et al. use the same example to show that illustrate that their parser combinators can handle left recursion [6, 7]. They avoid infinite recursion by building a table with partial parse results for every input position. Only when a new result is added to this table their parsers go into the recursion. The effect on termination is identical. The advantage of their approach is that is is more efficient in time complexity. Our approach needs to recompute these results, but is much simpler and does not need the huge memoization tables. This is yet another example of the time/space tradeoff. In their library, the user needs to indicate memoization points in their parsers. When their algorithm would be implemented in our approach these memoization points can be added automatically as illustrated above.

Another idea to enforce termination might be to allow a new call of a nonterminal only when at least one token has been consumed since the previous occurrence of this nonterminal. Such a mechanism can be implemented very similar to the upper-bound on the number of function calls. This would have the advantage that we can stop on a much lower number of recursive calls and hence produce a more efficient parser. Unfortunately, allowing no recursive function calls without consuming input between them is too restrictive in many situations. For our grammar `eLeft` and the input `["3","-","2","-","1"]` we need two recursive calls of `e`. The deepest one recognizes `["3"]`, the next one `[3,"-","2"]` and the final one `[0p 3 "-" 2,"-","1"]`. This example does show that the threshold can in many situations be lower than the length of the input. In most real-life situations we can use heuristics to limit it much more to recognize "reasonable" inputs. For small inputs, the given limit works fine despite the awful complexity.

The advantage of our approach is that it works for every nonterminal. It is not required that the user applies some mark, like the library of Frost demands. Adding and maintaining the marks is very cheap. Whenever the left-recursive grammar appears to be too slow in practice we can transform it to a more efficient grammar without having to change the parsing framework. Apart from removing left-recursion, one might want to apply techniques like left-factoring and switch to deterministic parsing for top performance within the same framework [1].

In the same style, we can make other parser implementations as new views of our parser combinators. For instance, to produce proper error messages or doing error correction by inserting and deleting tokens to find a parse tree. See, for instance [23] for inspiration.

6 PARSE TREE GENERATION

Apart from various parsing algorithms, we can assign completely different meanings to our parser combinator expressions. In this paper we just list some possibilities in the coming sections, the possibilities are endless and can be added independently of each other.

Parsers are typically the first step in a program. For instance, in a compiler, the input program is parsed, checked, transformed and finally produces an equivalent program in a different language. To test the later stages of a compiler, it is very convenient to have a large number of test cases available that can be produced by the parser. In this section, we define a view of our parser combinators that produces these trees without requiring any input.

To produce these parse trees we define a datatype `Trees` that can be used as an instance of our classes of parser combinators. This view just contains a list of results `[a]`, no state is needed in this view.

```
:: Trees s a = Trees [a]
```

As usual we define the monadic operators for this type. The instance for `Alternative` is the only slightly interesting one. The resulting trees of both alternatives of the choice operator are merged.

```
instance Alternative (Trees s) where
  empty = Trees []
  <|> (Trees p) (Trees q) = Trees (merge q p)

merge :: [a] [a] → [a]
merge [a:x] y = [a:merge y x]
merge []    y = y
```

To ensure well-definedness we have to assume an order for the recursive alternatives of the choice operator `<|>`. Without loss of generality we assume that the recursive case (if it occurs) is the first argument of this operator. The function `merge` mixes the results of both alternatives. Better test cases would be obtained by a skew merge order as used in modern versions of Gast [11, 14]. This is omitted for brevity.

The instances for our basic parser combinators generate positive integers and single-character identifiers. This can easily be changed to more sophisticated values if that would be required.

```
instance int (Trees s) where int = Trees [0..]
instance idn (Trees s) where idn = Trees (map toString ['a'..'z'])
instance lit (Trees s) where lit x = Trees [x]
```

To prevent runtime problems with cycle-in-spine errors we add tailor-made definitions for the definition of nonterminals like:

```
instance nt (Trees s) (Trees s a, Trees s b) where
  nt f = let ((p0, p1) In q) = f (p0, p1) in q
```

Semantically, this definition is equivalent to default instance `instance nt v t where nt f = let (p In b) = f p in b` from Section 2. Operationally, the default instance produces a cycle in spine error. The tailor-made definition adds just the amount of laziness required to execute this cycle smoothly.

This is sufficient to generate parse trees that can be produced by the grammars specified by parser combinators. For all other classes in our library, the default implementation of the class is sufficient. The first 10 parse trees of type `Exp` defined by the grammar `exp` from Section 2 are:

```
[(Int 0)
,(Op (Int 0) "-" (Int 0))
,(Op (Int 0) "*" (Int 0))
,(Op (Int 0) "-" (Op (Int 0) "-" (Int 0)))
,(Int 1)
,(Op (Int 0) "-" (Op (Int 0) "*" (Int 0)))
,(Op (Int 0) "*" (Op (Int 0) "*" (Int 0)))
,(Op (Int 0) "-" (Op (Int 0) "-" (Op (Int 0) "-" (Int 0))))
,(Int 2)
```

```
,(Op (Int 0) "-" (Int 1))
]
```

Duplicated parse trees will occur when there is more than one way to produce a given parse tree. For instance, the grammar `exp2` will contain duplicated trees expressions since the fragment `pure (λl e r.e) <*> lit "(" <*> expr <*> lit ")"` allows any expression to produces directly, as well as enclosed in brackets. Since these are different production rules in the grammar both results will be obtained. Whenever necessary these duplicates can be filtered for the result.

This generation works for parse results of any type. The first 10 results for the grammar `nested` from Section 2 are `[0,1,2,3,4,5,6,7,8,9]`. Note that there is only a single production rule for each result. Hence any value will be unique, although there are many inputs to generate such a value.

Note that the generated parse trees are the ones described by the grammar. Generating expressions based on the type definition of `Exp` by Gast would generate more instances of the datatype, for instance with arbitrary strings as the operator in the expressions [14]. The generation of elements of the AST is QuickCheck would suffer from very similar limitations; the generation looks only at the type definitions while generating based on the parser will only generate the instances that can be produced by the parsers. For instance, here the strings representing the operators are just the values `"-"` and `"*"` occurring in the grammar. Any String can be generated as an operator if we only look at the type. Similarly, generating integers with Gast instead of generating the numbers that can be a result of `nested` will produce negative numbers and values like max-int and min-int. The trees produced based on the grammar are typically much better test cases for the rest of the compiler.

7 INPUT GENERATION

In the same spirit, we can define a view that produces inputs accepted by the grammar specified. Although there is a close connection between generated parse trees and generated inputs, there are also some significant differences. With a nondeterministic parser, a single input can produce multiple parse trees. Depending on the grammar there might be different inputs that produce the same parse tree, obvious differences are additional whitespace and parentheses.

Just like all views we start with a datatype to be used as instance of the classes in the parser library.

```
:: Inputs s a = Inputs [(a,[s])]
```

Here `[s]` is the input corresponding to a given result `a`. The entire view makes a list of possibilities to generate result `a` and the corresponding input `[s]`. This view is kind of the inverse of `Parse`. In `Parse` we construct the result `a` based on the given input. Here we construct the possible parse results and the corresponding input based on the grammar rules encoded as combinator expression. One might wonder why it is necessary to produce a value `a` as well as an input `[s]` when we are only interested in the inputs. The type `a` is used in context-sensitive parsers to determine what input symbol of type `s` will be recognized. A very simple example is the parser `context = idn >>= lit` from Section 2 where `a` and `s` are bound to `String`. Hence, the types `a` and `s` are both needed.

Like usual we define the monadic operators for this type. The choice operator is again the only interesting one.

```
instance Alternative (Inputs s) where
    empty = Inputs []
    <|> (Inputs p) (Inputs q) = Inputs (merge q p)
```

The instances for the basic parser combinators are:

```
instance int (Inputs String) where
  int = Inputs [(i, [toString i]) \\ i ← [0..]]
instance idn (Inputs String) where
  idn = Inputs [let s = toString i in (s, [s]) \\ i ← ['a'..'z']]
instance lit (Inputs String) where
  lit x = Inputs [(x, [x])]
```

Just like in the previous section we need a tailor made instance for `nt` to avoid problems with cycles in spine for (left) recursive definitions.

```
instance nt (Inputs s) (Inputs s a0, Inputs s a1) where
  nt f = let ((p0, p1) In q) = f (p0, p1) in q
```

For all other definitions we use the default definitions of the classes. The first ten inputs generated for `exp2` are:

```
[["0"]
,["0","+","0"]
,["0","*","0"]
,["0","+","0","*","0"]
,["(","0",")"]
,["0","-","0"]
,["0","*","(","0",")"]
,["0","+","(","0",")"]
,["1"]
,["0","-","0","*","0"]]
```

The first ten inputs generated for `nested` are:

```
[[]
,["a","a"]
,["a","a","a","a"]
,["b","b"]
,["a","a","a","a","a","a"]
,["b","a","a","b"]
,["c","c"]
,["a","b","b","a"]
,["b","a","a","a","a","b"]
,["c","a","a","c"]]
```

As expected there are several inputs generated that correspond to the parse tree with value 1. For instance the generated inputs `["a","a"]`, `["b","b"]` and `["c","c"]` in the first ten inputs would all produce parse result 1. Also for other depths, we obtain different inputs corresponding to that parse result. This shows that the view of the previous section produces better test cases (no duplicates) if we are interested in the parse trees as test cases for the rest of the compiler. By replacing the fair `merge` with a skew variant we can choose again between a preference between varying the input symbols or the length of the input.

This illustrates that generating inputs and parse trees is really different. Parsing such an input will produce a valid parse tree, but many duplicated parse trees will be obtained if we would make them by parsing generated inputs. Hence, generating parse trees deserves its own view.

8 PRETTY PRINTING THE GRAMMAR

In yet another view we might be interested in the grammar specified. Not only the construction of parse trees might clutter the view on grammar. More important is that the host language of the grammar DSL, here Clean, can be used as powerful "macro" language to construct grammars. Constructing grammars by expressions in the host language is very convenient, but from time to time one might wonder which grammar is actually specified.

To reveal the grammar without the functions constructing the parse tree we define a new view of the parser combinators producing a textual representation of just the grammar. Since our parser combinators are just functions in the host language and can be context-dependent this is a little tricky at some points. Without using very ugly tricks we have no control over what the functions in the host language do with the recognized tokens.

Moreover, we cannot check what happens in the context-dependent parts of the grammar. This would require either a runtime analysis of the code in Clean or applying all relevant inputs to the parser. Neither of these options is feasible. Nevertheless, we can give a very good impression of the grammar.

To understand the grammar recognized by a parser expression we do not need to know the parse tree constructed. This implies that we want skip the code associated to parse tree constructions as much as possible. To handle context-dependent parts we chose some result that can be produced by the initial part of the grammar and use that in all applied occurrences. Although this produces only the grammar corresponding to this value, it gives typically a very good impression of the grammar. When we would also lift the code controlling the context-sensitive parts to the grammar DSL we can produce a complete description of these context-sensitive parts. This is done in our mTask system [12, 18], but considered to be overkill for parser combinators.

We start by defining a data type for the instances of the combinators for this interpretation. The state contains a grammar-ID, `grmID`, used to name fresh nonterminals as well as descriptions of the grammar as a list of strings. We use a continuation funtion `print` instead of a plain list of strings to avoid a lot of inefficient, $O(n)$, appends.

```
:: Grammar s a = Grammar (GState → (a, GState))
:: GState =
    { grmID :: ID
    , print :: [String] → [String]
    }
```

To write some string to the `print` function we define a function `writeR` with a result and a function `write` without a proper result. We use the value `undef` as placeholder for the undefined result.

```
writeR :: String a → Grammar s a
writeR l a = Grammar λs.(a,  {s & print = λc. s.print [l:c]})

write :: String → Grammar s a
write s = writeR s undef
```

The instances of `Functor` and `pure` are used in the production of parse trees. Since we are not interested in the generation of parse trees the instances of these combinators leave the `print` component untouched.

```
instance Functor (Grammar s) where
  fmap f (Grammar g) = Grammar λs.let (a,t) = g s in (f a, t)
instance pure    (Grammar s) where pure a = Grammar λs.(a, s)
```

For operators like bind and sequence the situation is slightly more complex. On one hand, we want to show the grammar components specified by these operators, on the other hand, we want to use these operators in the definition of other operators. In those other situations, we do not want any changes of `print`. To achieve this we define new operators that does not change the `print`, but are otherwise equivalent with the usual operators.

An additional complication is that these operators are sometimes used to construct parse trees and in other situations to compose grammar elements. In order to use operators like bind without changing `print` we introduce special versions of operators that leave the `print` untouched. For the other operators, we use a heuristic to determine if the operator is used to generate a parse tree or to compose grammar components. When an operand produces output the operation is assumed to compose grammar components. Otherwise, it is assumed to produce only parse trees. Since `print` is a function, it is slightly tricky to determine if it produces output. For some grammar `Grammar f` we determine if it produce output by applying it to a `GState` with value `{grmID = 0, print = id}`. The function `f` will change this `GState`. From the obtained state we select the `print` function and apply it to an empty list of strings. When the result is also empty the grammar component does not contribute to the output and we assume that it is only there to produce a syntax tree. The function `ifEmpty` performs this test to select grammar view `t` or `e`. The functions `onEmpty` and `onNotEmpty` are convenient abbreviations.

```
ifEmpty :: (Grammar s a) (Grammar s a) (Grammar s a)→Grammar s a
ifEmpty (Grammar f) t e
  = if (isEmpty ((snd (f {grmID = 0, print = id})).print [])) t e

onEmpty :: (Grammar s a) (Grammar s a) → Grammar s a
onEmpty p q = ifEmpty p q p

onNotEmpty :: (Grammar s a) (Grammar s a) → Grammar s a
onNotEmpty p q = ifEmpty p p q
```

Using this machinery we can define both versions of the bind operator as where `>>-` is just the operator $\gg=$ without producing output.

```
(>>-) infixl 1 :: (Grammar s a) (a → Grammar s b) → Grammar s b
(>>-) (Grammar p) f = Grammar λs.
    let (a, s2) = p s
        (Grammar q) = f a
    in q s2

instance Monad (Grammar s) where
  bind g f = onEmpty g (write "." >>!g) >>- λa.
             onNotEmpty (f a) (write "≫= " >>! f a)
```

In this view we use that fact that `>>|` is an operator on its own instead of a macro definition to define a tailor made version as well as an equivalent leaving the `print` untouched. The

operator >>! is the equivalent of >>| that does not generate output.

```
instance >>| (Grammar s) where
  (>>|) ma mb = onEmpty ma (write ".") >>!
                onNotEmpty mb (write ">>| " >>! mb)

(>>!) infixl 1 :: (Grammar s a) (Grammar s b) → Grammar s b
(>>!) (Grammar f) (Grammar g) = Grammar (g o snd o f)
```

The `Alternative` is supposed to define always a choice between grammar rules. Hence, there is always something added to the `print`. When the `print` of one of the alternatives is empty we add `"."` to denote the empty grammar production.

```
instance Alternative (Grammar s) where
  empty = write "empty "
  <|> f g = onEmpty f (write ".") >>! write "<|> " >>!
            onEmpty g (write ". ")
```

For the other combinators, we add similar implementations that show the application of the operator instead of using the default implementation.

For the basic parser operations, we define instances that show their application. They produce a result that will be used in a context-sensitive grammar. In all other situations, this result will be ignored in the end.

```
instance int (Grammar s) where
  int = writeR "int " 7
instance idn (Grammar s) where
  idn = fresh >>- λi.let v = "i" + toString i
                    in  writeR ("idn (" + v + ") ") v
instance lit (Grammar String)
  where lit x = write ("lit λ"" + x +"λ" ")
```

For the nonterminal definition we also make a tailor-made implementation instead of default implementation. For any applied occurrence of the nonterminal, we use a generated name instead of the body. We show two instances as examples. The first one is the definition of a single nonterminal value. In the second definition, this nonterminal has an integer argument, it is actually a function definition. Unfortunately, we lose the names of the original nonterminals and substitute them by generated names like `nt0` and `nt1`.

```
instance nt (Grammar s) (Grammar s a) where
  nt f = freshl 1 >>- λl.
         let [var0:_] = map varName l;
             (p In b) = f (write var0)
          in  write ("nt \\" + var0 + " λn = ") >>!
              p >>! write " InλnL  " >>! b

instance nt (Grammar s) (Int→Grammar s a) where
  nt f = freshl 2 >>- λ[i0, i1:_].
         let var0 = varName i0;
             a0   = "a" + toString i1
                (p In b) = f (λx.write (".." + var0 + ".. "))
         in  write ("nt \\" + var0 + " = λn  \\" + a0 + ".") >>!
             p 1 >>! write "InλnL  " >>! b
```

In the same style we define tailor made instance for our other parser combinators, (`<*>`, `alt`, `list`, `?>=`, `chainl`, ..) that show the application of the parser combinator instead using its semantics from the default implementation.

After these lengthy preparations, we can finally show the grammar corresponding to parsers. The grammar corresponding to the parser `exp2` is shown as:

```
nt λ(nt0 ,nt1 ,nt2 ) =
  (lit "(" <*> nt2 <*> lit ")" <|> int , // nt0
  ,nt1 <*> lit "*" ?>= nt0 , // nt1
  ,nt2 <*> lit "-" <|> lit "+" ?>= nt1 , // nt2
  ) In nt2
```

To demonstrate that the host language can be used to generate grammars we construct a flexible way to generate infix expressions with various priorities inspired by Swierstra [23]. The grammar is defined as:

```
instance + Expr where + x y = Op x "+" y
instance - Expr where - x y = Op x "-" y
instance * Expr where * x y = Op x "*" y

op :: (a,String) → b a | lit b & Functor b
op (sem,symbol) = fmap (λo.sem) (lit symbol)

ops :: ([(a,String)] → b a) | alt b & Alternative b & lit b
ops = alt o map op

eChain :: v Expr | gram v
eChain = foldl chainl (fmap Int int)
                      (map ops [[(*,"*")],[(+,"+"),(-,"-")]])
```

The advantage of this approach is that is easy and flexible to add operators in the grammar, even adding a new priority level is achieved by just adding an element to the list of operators `[[(*,"*")],[(+,"+"),(-,"-")]]`.

The grammar recognized by `eChain` is displayed as:

```
(chainl ((chainl (int ) (alt [lit "*" ] )))
        (alt [lit "+" , lit "-" ] ))
```

9 OPTIMIZATION OF GRAMMARS

After the successes of making the various instances of the grammars, it is tempting to implement an optimizing view of grammars. As a simple example, we consider a grammar of statements that are a conditional without else part, a conditional with an else part or the literal `s`. The recognized grammar is show by the tooling from the previous section as:

```
nt λnt0
 = lit "c"  In
  nt λnt1
 = alt [list [lit "if", nt0, lit "then", nt1]
       ,list [lit "if", nt0, lit "then", nt1, lit "else", nt1]
       ,lit "s"]  In
  nt1 <*> eof
```

It seems obvious that we can optimize the first two alternatives of `nt1` of this parser by left factoring the fragment `list [lit "if" , nt0 , lit "then" , nt1 ]`. Unfortunately, the arbitrary functions that might occur in parsers make this impossible. Although these fragments recognize the same input, they cannot simply be folded together since different parse trees might be generated from the identical grammar parts.

When we would restrict ourselves to a single (intermediate) parse result type it is possible to implement optimizations like left factoring and the left corner transform in the same style as [2]. Since this is less general than the parsers in the rest of this paper and requires a significant amount of details we do not elaborate on this. Whenever desired the grammars must be transformed manually to the desired form.

The optimizations of a tagless query language as described by Suzuki et al. might be adapted to our grammar language [22]. This nontrivial amount of work is left as future work.

10 DISCUSSION

In this paper, we demonstrate that parsers can be formulated such that multiple interpretations of a single parser definition are possible. To achieve this goal we defined parser combinators that enable us to specify parsers as a tagless DSL. This is achieved by defining the parser combinators as a type constructor class instead of a single view of such a class. For each new interpretation, view in DSL terms, we define in principle a new instance of the parser combinators. We have shown that for many views we only have to define the monadic operators as well as the basic parsers. For many other combinators, we can use a default implementation that works for most views. For a full-fledged production library, we need additional convenience combinators. We have not treated them here since our goal was to demonstrate the principle rather than making a complete library.

Representing a parser as a DSL is the key allow different interpretations, views, of the parsing rules. In this paper, we used a tagless embedding by embedding of the DSL by representing it as a set of type constructor classes. A more basic representation of a multi-view DSL is an initial embedding; represent the parsing rules as a datatype. To specify the result types of the parsers correctly we would need to a Generalized Algebraic Datatype, GADT. We consider GADTs more complex than type constructor classes, there is a good reason why they were introduced later and their implementation is more cumbersome. A big advantage of the tagless embedding is that it is easy to introduce new well-typed identifiers in the DSL. This is demonstrated by the class `nt` in Section 2. Moreover, a DSL based on datatypes cannot be extended without touching all existing code. In our tagless representation, we can simply add a class and the desired instances without changing exiting parts of the DSL. The compiler of the host language checks statically if every required part is defined. We used this to introduce our DSL step by step. Finally, the tagless formulation is more concise and often more efficient. A representation based on datatypes needs a traversal of the datatypes that can be avoided in a tagless DSL. A disadvantage of a tagless DSL is that the type error messages can be somewhat intimidating. It requires some training to understand them quickly. Those more complex type error messages do not outweigh the advantages of the tagless representation of the DSL for grammars.

In this paper, we introduced interpretations for deterministic and nondeterministic parsing. The advantage of the deterministic interpretation is that it is rather efficient, while the latter can handle ambiguous grammars. The literature shows how the efficiency of these implementations can be optimized, e.g., [13, 23].

We have introduced a new way to handle arbitrary left-recursive grammars. We changed the function definitions corresponding to nonterminals such that there is a fixed upper bound on the recursion depth (the length of the remaining input). This approach does not require any action from the user and is much simpler than existing methods like the transformation of the grammar [2, 9] or maintaining tables with possible parse results at all positions of the input [6]. Since we do not need no tables our approach uses very limited amounts of memory. The price to be paid is that the implementation can be slower.

In other views, we generated parse trees that might be generated by the parser without the need to supply the corresponding input. We also demonstrated that the inputs accepted by the grammar can be generated efficiently.

Finally, we developed an instance of the combinators that showed that grammar accepted by the parser concisely. Since we want to get rid of the parse tree generations this interpretation is somewhat longer than the preceding ones.

The combination of these interpretations showed that we achieved our goal. We have introduced a set of parser combinators that is concise and strongly typed, but also suited for more interpretations than just a single way of parsing.

ACKNOWLEDGMENTS

To authors want to thank Peter Achten for very useful feedback on drafts of this paper. With Mart Lubbers we had inspiring discussions about the design and implementation of type constructor class-based DSLs. The gained insights were very useful for the system described in this paper.

The anonymous reviewers provided very useful feedback and questions that helped to improve this paper.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. 2010. Typed Transformations of Typed Grammars: The Left Corner Transform. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 51 – 64. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[3] William H. Burge. 1975. *Recursive programming techniques*. Addison-Wesley.

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543. https://doi.org/10.1017/S0956796809007205

[5] Nils Anders Danielsson. 2010. Total Parser Combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/1863543.1863585

[6] Richard A. Frost and Rahmatullah Hafiz. 2006. A New Top-down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time. *SIGPLAN Not.* 41, 5 (May 2006), 46–54. https://doi.org/10.1145/1149982.1149988

[7] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-recursive Grammars. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages (PADL'08)*. Springer-Verlag, Berlin, Heidelberg, 167–181. http://dl.acm.org/citation.cfm?id=1785754.1785766

[8] John Hughes. 2005. Programming with Arrows. In *Advanced Functional Programming*, Varmo Vene and Tarmo Uustalu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–129.

[9] Mark Johnson and Brian Roark. 2000. Compact Non-left-recursive Grammars Using the Selective Left-corner Transform and Factoring. In *Proceedings of the 18th Conference on Computational Linguistics - Volume 1 (COLING '00)*. Association for Computational Linguistics, 355–361.

[10] Stephen C. Johnson. 1979. *Yacc: Yet Another Compiler-Compiler*. Technical Report 32. Bell Laboratories Computing Science.

[11] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. 2003. Gast: Generic Automated Software Testing. In *Implementation of Functional Languages*, Ricardo Peña and Thomas Arts (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–100.

[12] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop, RWDSL@CGO 2018, Vienna, Austria, February 24-24, 2018*. 4:1–4:11. https://doi.org/10.1145/3183895.3183902

[13] Pieter Koopman and Rinus Plasmeijer. 1999. Efficient Combinator Parsers. In *Implementation of Functional Languages*, Kevin Hammond, Tony Davie, and Chris Clack (Eds.). 120–136.

[14] Pieter W. M. Koopman and Rinus Plasmeijer. 2005. Generic generation of the elements of data types. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*. 163–178.

[15] Susumu Kuno. 1965. The Predictive Analyzer and a Path Elimination Technique. *Commun. ACM* 8, 7 (July 1965), 453–462. https://doi.org/10.1145/364995.365689

[16] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World* (technical report uu-cs-2001-35, universiteit utrecht ed.). Technical Report UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007.

[17] John Levine, Tony Mason, and Doug Brown. 1992. *Lex & Yacc, 2Nd Edition* (second ed.). O'Reilly.

[18] Mart Lubbers, Pieter W. M. Koopman, and Rinus Plasmeijer. 2018. Task Oriented Programming and the Internet of Things. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA, September 5-7, 2018*. 83–94.

[19] Robert C. Moore. 2000. Removing Left Recursion from Context-free Grammars. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference (NAACL 2000)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 249–255. http://dl.acm.org/citation.cfm?id=974305.974338

[20] Tom Ridge. 2014. Simple, Efficient, Sound and Complete Combinator Parsing for All Context-Free Grammars, Using an Oracle. In *SLE*.

[21] D. J. Rosenkrantz and P. M. Lewis. 1970. Deterministic Left Corner Parsing. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory (Swat 1970)*. IEEE Computer Society, 139–152.

[22] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, Safely-extensible and Efficient Language-integrated Query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, New York, NY, USA, 37–48.

[23] S. Doaitse Swierstra. 2009. *Combinator Parsing: A Short Tutorial*. Springer Berlin Heidelberg, Berlin, Heidelberg, 252–300.

[24] Masaru Tomita. 1985. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA.

[25] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2018. GLL Parsing with Flexible Combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2018)*. ACM, New York, NY, USA, 16–28. https://doi.org/10.1145/3276604.3276618

[26] Philip Wadler. 1985. How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*. 113–128.

[27] Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, 24–52.

Type Debugging with Counter-Factual Type Error Messages Using an Existing Type Checker

Kanae Tsushima
National Institute of Informatics
Japan
k.tsushima@nii.ac.jp

Olaf Chitil
University of Kent
United Kingdom
oc@kent.ac.uk

Joanna Sharrad
University of Kent
United Kingdom
jks31@kent.ac.uk

ABSTRACT

The cause of a type error can be very difficult to find for the Hindley-Milner type system. Consequently many solutions have been proposed, but they are hardly used in practice. Here we propose a new solution that provides easy-to-understand messages, because it is based on the counter-factual type error messages that programers are already familiar with, and that is easy to implement, because it reuses an existing type checker as a subroutine. We transform an ill-typed program into a well-typed program with additional λ-bound variables. The types of these λ-bound variables yield actual and counter-factual type information. That type information plus intended types added as type annotations direct the search of the type debugger.

ACM Reference Format:
Kanae Tsushima, Olaf Chitil, and Joanna Sharrad. 2020. Type Debugging with Counter-Factual Type Error Messages Using an Existing Type Checker. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Hindley-Milner type system is a foundation for most statically typed functional programming languages, such as ML, OCaml and Haskell. This type system has many benefits, but it does make type debugging hard: if a program is not well-typed, it is difficult for the programmer to locate the cause of the type error, that is, to determine where to change the program and how.

Many solutions to the problem have been proposed in the literature (see Section 8). Here we propose a new solution with two distinctive advantages: It is easy to understand for the functional programmer, because it appears to be only a minor extension of the type error messages they are already familiar with. It is easy to implement, because it does not require the implementation of a new type checker, but instead reuses any existing one as a subroutine.

Consider the following ill-typed OCaml program[1] and the type error message produced by the OCaml compiler:

```
let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

Error: This expression has type
    float
but it should be an expression of type
    string
```

The message identifies the underlined expression `2.0` in the program as the location of the type error. The message gives two different types for the expression `2.0`: its actual type and an expected type. The expected type is determined by the context of `2.0`, the rest of the program. As the expected type is different from the actual type, it is a counter-factual type. The message basically says that if the expression `2.0` was replaced by some expression of the expected type, then this part of the program would be well-typed (there might be further type errors elsewhere). Indeed, replacing `2.0` by any string, for example `"2.0"`, produces a well-typed program.

So if the type error message identified the type error location correctly, then the message with its actual and expected type is very helpful. However, the subexpression `2.0` might be correct and the programmer might have confused the string concatenation operator `^` with the floating point exponentiation operator `**`. In that case a type error message like the following would have been helpful:

```
let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

Error: This expression has type
    string -> string -> string
but it should be an expression of type
    'a -> float -> 'b
```

In this paper we first show how to produce such counter-factual type error messages for *all* potential locations of type errors. Although a program may contain many potential type error locations, we assume that in practice a program contains a large, well-typed part, which provides a context to limit the number of potential type error locations and which yields informative counter-factual types.

There are still too many potential type error locations for showing them all, embedded in counter-factual type error messages, to the programmer. Hence the second part of our proposal is to run an interactive debugger to find the correct error location. The programmer only has to state whether an actual type or expected type agrees with their intentions. Sometimes both types do not agree with the programmer's intention. In that case the programmer can

[1] Library function `List.map : ('a -> 'b) -> 'a list -> 'b list` applies its first argument, a function, to each element of its second argument, a list.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

also enter another type. In the example session below the input of the user is given in italics.

```
Interactive counter-factual type debugger.

1. Choose your intended type for this expression.

let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

A: float
B: string
Your choice (C: another type): A

2. Choose your intended type for this expression.

let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

A: string -> string -> string
B: 'a -> float -> 'b
Your choice (C: another type): B

3. Type error located:

let f n lst = List.map (fun x -> x ^ n) lst in
f 2.0

This expression has type
    string -> string -> string
but it should be an expression of type
    'a -> float -> 'b
```

After two questions the type debugger identifies the correct type error location. The debugger gives the full counter-factual type error message so that the programmer can determine how to correct the error.

In this paper we make the following contributions:

- We define a new method for enumerating all counter-factual type error messages for an ill-typed program. The type checker of a real functional programming system is too complex to be easily replaced by something new. Hence we do not develop a new type checking algorithm, but instead describe an algorithm that reuses any existing type checker, which is treated as a black box. (Sections 2 and 3).
- We provide an argument that it is sufficient to only consider leafs in the abstract syntax tree of the ill-typed program as potential type error locations. (Section 2).
- We extend our method to produce counter-factual type error messages also for expressions that have to be polymorphic for the program to become well-typed (Section 4).
- The time complexity of our basic method is too high to be used for medium-sized or large programs. Hence we show how we can first execute a known type error slicing algorithm to then obtain counter-factual type error messages more quickly. (Section 5)
- We define an interactive debugger that identifies the correct type error location building on our method for enumerating counter-factual type error messages (Section 6).
- We evaluate our prototype implementation with a number of ill-typed programs. (Section 7).

2 IDEA: COUNTER-FACTUAL TYPES FROM AN EXISTING TYPE CHECKER

We reuse an existing type checker by giving it many variations of our ill-typed program to check. Viewing the type checker as a black box means that we expect the type checker to tell us only whether the program is well-typed or ill-typed. If the program is well-typed, then the type checker shall also tell us the type of the given program. If the program is ill-typed, then we demand no further information. In particular we do not use any details of the type checker's own error message(s).

2.1 Only Leaves as Potential Error Locations

In all our previous examples potential type error locations were simple variables or constants, not more complex expressions. In other words, a location was a leaf of the abstract syntax tree, never an inner node of the abstract syntax tree.

Our method works for both leaves and inner nodes of an abstract syntax tree, but here we argue that to debug all type errors it is sufficient to consider only leaf locations.

Consider the following ill-typed example:

```
(fun (x, y) -> if x then y else y + 1) [true; 1]
```

We assume that the programmer intended the argument to be the tuple `(true, 1)`. So arguably the whole argument `[true; 1]` is the type error location. However, because we consider only leaves in the abstract syntax tree as potential type error locations, we will instead offer the list constructor itself as type error location and produce the message

```
(fun (x, y) -> if x then y else y + 1) [true; 1]

Error: This expression has type
    'a -> 'a -> 'a list
but it should be an expression of type
       bool -> int -> bool * int
```

This message states the error more indirectly. The message should make the programmer consider replacing `[..;..]` by the function `fun x -> fun y -> (x,y)`. Subsequently simplifying `(fun x -> fun y -> (x,y)) true 1` to the expression `(true,1)` should be trivial.

In the worst case a leaf-location message may suggest to substitute a complex function that rearranges a whole subtree of the abstract syntax tree. Here "rearrange" is the keyword, because the arguments are usually not superfluous but needed. Limiting ourselves to leaves in the syntax tree has the advantage that the expression identified as erroneous is always simple.

In general we can limit ourselves to leaves as potential type error locations, because any program without any leaves[2] is well-typed. Only the use occurrences of variables, data constructors and constants lead to type constraints that cause type errors.

[2]More precisely: any program where we replaced every leaf by a different fresh variable.

2.2 Single- and Multiple-Location Type Errors

Consider the ill-typed expression `[1;2;3.]`, which mixes integers and floating point numbers in a list. Two different single locations could be the cause of ill-typedness:

- The subexpression `3.` may be the type error location. Replacing it by an integer constant such as 3 would yield the well-typed expression `[1;2;3]`.
- The list constructor `[..;..;..]` (cf. next subsection) may be the type error location. Replacing it by, for example, a tuple `(..,..,..)` would yield the well-typed expression `(1,2,3.)`.

We call both *single-location type errors*.

Another way to correct the ill-typed expression would be to simultaneously replace `1` by `1.` and `2` by `2.`, yielding the well-typed expression `[1.;2.;3.]`. We call this a *multiple-location type error*, more precisely a *2-location type error*.

In general, a k-location type error states that replacing the expressions at k distinct locations in the program makes the program well-typed.

In practice, ill-typed programs often contain many independent type errors. However, we are not aware of any agreed definition of when two type errors are independent and when they are not. Our definition of k-location type error handles programs with many type errors, no matter whether there exist dependencies amongst these k locations or not.

Our method can handle k-location type errors for any k. However, to simplify the discussion, we will in most of this paper only consider searching for a single location that causes a program to be ill-typed. In practice, even when we enumerate all type errors, it is sensible to first enumerate all single location type errors, then 2-location type errors, etc. It is very likely that the programmer will find the actual cause of ill-typedness early in this list and thus does not have to look at errors that include numerous locations.

2.3 Program Syntax and the Syntax Tree

In the previous example we considered `[..;..;..]` as an atomic syntactic construct, which has the type `'a -> 'a -> 'a -> 'a list` and semantically denotes a function constructing a list from three arguments (`fun x -> fun y -> fun z -> [x; y; z]`). Desugaring this list construction into uses of the list constructor `(::)` and the empty list is not desirable, because desugared expressions in error messages would be confusing for the programmer. Similarly `if..then..else..` is an atom of type `bool -> 'a -> 'a -> 'a`.

In general we treat most syntactic constructs of a programming language as constants with fixed types. In the syntax tree of a program they are leafs.

2.4 Determining Potential Error Locations with Expected Types

Let us consider the simple program `1.0 + 2.0`. Because the operator `+` demands integer numbers but `1.0` and `2.0` are floating point numbers, the program is ill-typed. Internally the program is represented as an application of the operator `+` to two arguments, that is, `(@ (+) 1.0 2.0)`.

The program has three leaf locations. We investigate for each leaf whether it is a potential single type error location by type checking a corresponding variant of our program:

```
(1) fun hole -> @ hole 1.0 2.0
(2) fun hole -> @ (+) hole 2.0
(3) fun hole -> @ (+) 1.0 hole
```

So we replaced a potential type error location by a new variable `hole` and added a λ-binding for the variable to the whole program. The λ-binding ensures that the program has no free variables and allows us to obtain a type for `hole` from the type checker.

We run the existing type checker on each of the three program variants. Programs (2) and (3) are ill-typed. Hence replacing the variable `hole` by any expression of any type would not make the program well-typed. Consequently the locations `1.0` and `2.0` are not potential single type error locations.

Program (1) is well-typed and its inferred type is `(float -> float -> 'a) -> 'a`. So the type of the variable `hole` is `float -> float -> 'a`. Consequently `(+)` is a potential type error location and we can produce the following message:

```
1. + 2.

Here should be an expression of type
    float -> float -> 'a
```

2.5 Obtaining Actual Types Too

The error message above does not yet include the actual type of `+`. We can also obtain that type if we do not simply replace a potential type error location by a variable `hole`, but instead apply a variable `hole` to the potential type error location. Again we λ-bind the variable `hole`. So instead of the 3 program variants listed before, we type check the following 3 variants:

```
(1) fun hole -> @ (hole (+)) 1.0 2.0
(2) fun hole -> @ (+) (hole 1.0) 2.0
(3) fun hole -> @ (+) 1.0 (hole 2.0)
```

As before, only variant (1) is well-typed. For this variant the inferred type is `((int -> int -> int) -> (float -> float -> 'a)) -> 'a`. So the type of the variable `hole` is `(int -> int -> int) -> (float -> float -> 'a)`, which contains both the actual and the expected type of the potential type error location. Thus we can produce the complete message:

```
1. + 2.

Error: This expression has type
    int -> int -> int
but it should be an expression of type
    float -> float -> 'a
```

Naturally we could have obtained the actual type of `+` by just type checking the program `(+)`. However, in general a potential type error location may not be a predefined function or data constructor, but some variable that is λ- or let-bound in the ill-typed program. Our method of applying a variable `hole` to the potential type error location works in all situations.

3 OBTAINING COUNTER-FACTUAL TYPES FOR THE SIMPLY-TYPED λ-CALCULUS

In this section we formalise our idea for a small core functional language, the simply-typed lambda calculus $\lambda^{\rightarrow}$. Whereas in the preceding section we focussed on 1-location type errors, we now consider the general case of k-location counter-factual type errors. Recall that a k-location counter-factual type error states that replacing the expressions at k distinct locations in the program by expressions of different, expected types, makes the program well-typed.

The syntax and types of $\lambda^{\rightarrow}$ are shown in Figure1. The constants include numeric literals, tuple data constructors and list data constructors. Each constant and each variable has location information, l, which uniquely identifies each occurrence in the program.

Figure 2 gives the algorithm for obtaining counter-factual type errors for $\lambda^{\rightarrow}$. The capitalised type-writer font functions (`GENVAR` etc.) are external.

The last function, *get_cft_nloc*, obtains for a given program M and number k all k-location counter-factual type errors. The function *get_cft_nloc* uses the function `LEAFLOCS` to determine all (leaf) locations of the program, the function `SUBSETS` to determine all subsets of k elements, and applies the *get_cft* function with the standard function `MAP` to every subset (represented as list).

The function *get_cft* obtains for a term M and list of locations L a counter-factual type error, namely an actual and expected type for each location. However, it returns the empty list if no such counter-factual type error exists for the given arguments. Therefore function *get_cft_nloc* usually returns many empty lists. The function *get_cft* uses the functions *pierce* and *infer*.

The function *pierce* takes an expression and list of locations. It inserts new variables as holes at the given locations in the expression and also returns the list of new variables. The function `GENVAR` creates a fresh variable from a given location.

The function *infer* takes an expression with k "holes" and the list of "hole" variables; it returns a k-location counter-factual type error, if for the given input there is such a type error. Otherwise, it returns the empty list. The function `INFER` is an existing type checker. It takes an expression and returns its inferred type. However, if the expression is ill-typed, the type checker raises an exception `TYPE_ERROR`. In the definition of the function *infer* that exception is caught by the `try ... with` construct. Finally, the function `GET_LOC` obtains from a variable the location information given at its creation. Note that if type checking succeeds, then every "hole" variable will have a type of the shape *actual type* $\rightarrow$ *expected type*.

4 OBTAINING COUNTER-FACTUAL TYPES FOR THE LET-POLYMORPHIC λ-CALCULUS

Problem. The method presented in the preceding two sections does not work with let polymorphism. Consider the following ill-typed example:

```
let id = (fun lst -> List.iter
                       (fun x -> x) lst) in
  (id [1;3;4], id [true])
```

Because the type of `List.iter` is `('a -> unit) -> 'a list -> unit`, the type of `id` must be `unit list -> unit`. However, we pass two non `unit` lists (`[1;3;4] : int list` and `[true] : bool list`) to `id`, so type checking fails. We assume that the use of `List.iter` is the source of the type error: the programmer intended to use the function `List.map` instead, which has type `('a -> 'b) -> 'a list -> 'b list`, and thus makes the whole program well-typed.

Let us now apply our old method to the program (changed parts are underlined):

```
(fun hole ->
let id = (fun lst -> (hole List.iter)
                       (fun x -> x) lst) in
  (id [1;3;4], id [true]))
```

$$
\begin{array}{rcll}
(M : term) & ::= & c^l & \text{(constant)} \\
 & | & x^l & \text{(variable)} \\
 & | & \texttt{fun}\ x \mapsto M & \text{(abstraction)} \\
 & | & \texttt{@}\, M_1\, M_2 & \text{(application)} \\
(l : loc) & ::= & \textit{location information} & \\
(\tau : typ) & ::= & b & \text{(type variable)} \\
 & | & \text{int, bool, ...} & \text{(type constants)} \\
 & | & \tau_1 \rightarrow \tau_2 & \text{(function type)}
\end{array}
$$

Figure 1: The terms and types of $\lambda^{\rightarrow}$

$$
\begin{array}{l}
\textit{pierce} : \textit{term} * (\textit{loc list}) \rightarrow \textit{term} * (\textit{var list}) \\
\textit{pierce}[\![c^l]\!]_L = \\
\quad \texttt{if}\ l \in L\ \texttt{then let}\ u = \texttt{GENVAR}\ l\ \texttt{in}\ (\texttt{@}\, u\, c^l, [u]) \\
\qquad \texttt{else}\ (c^l, []) \\
\textit{pierce}[\![x^l]\!]_L = \\
\quad \texttt{if}\ l \in L\ \texttt{then let}\ u = \texttt{GENVAR}\ l\ \texttt{in}\ (\texttt{@}\, u\, x^l, [u]) \\
\qquad \texttt{else}\ (x^l, []) \\
\textit{pierce}[\![\texttt{fun}\ x \mapsto M]\!]_L = \\
\quad \texttt{let}\ (\texttt{M}', us) = \textit{pierce}[\![M]\!]_L\ \textit{in}\ (\texttt{fun}\ x \mapsto M', us) \\
\textit{pierce}[\![\texttt{@}\, M_1\, M_2]\!]_L = \\
\quad \texttt{let}\ (M_1', us_1) = \textit{pierce}[\![M_1]\!]_L\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us_2) = \textit{pierce}[\![M_2]\!]_L\ \texttt{in}\ (\texttt{@}\, M_1'\, M_2', us_1 + us_2)
\end{array}
$$

$$
\begin{array}{l}
\textit{infer} : \textit{term} * (\textit{var list}) \rightarrow (\textit{loc} * \textit{typ} * \textit{typ})\ \textit{list} \\
\textit{infer}[\![M]\!]_{[u_1;\cdots;u_n]} = \\
\quad \texttt{try}(\texttt{let}\ ((\tau_1 \rightarrow \tau_1') \cdots \rightarrow (\tau_n \rightarrow \tau_n') \rightarrow \tau) = \\
\qquad \texttt{INFER}\ (\texttt{fun}\ \ u_1 \mapsto \cdots \mapsto \texttt{fun}\ u_n \mapsto M)\ \texttt{in} \\
\qquad [(\texttt{GET\_LOC}\ u_1, \tau_1, \tau_1'); \cdots ; (\texttt{GET\_LOC}\ u_n, \tau_n, \tau_n')]) \\
\quad \texttt{with TYPE\_ERROR} \mapsto [] \\
\\
\textit{get\_cft} : \textit{term} * (\textit{loc list}) \rightarrow (\textit{loc} * \textit{typ} * \textit{typ})\ \textit{list} \\
\textit{get\_cft}[\![M]\!]_L = \\
\quad \texttt{let}\ (M', locs) = \textit{pierce}[\![M]\!]_L\ \texttt{in}\ \textit{infer}\ [\![M']\!]_{locs} \\
\\
\textit{get\_cft\_nloc} : \textit{term} * \textit{int} \rightarrow ((\textit{loc} * \textit{typ} * \textit{typ})\ \textit{list})\ \textit{list} \\
\textit{get\_cft\_nloc}[\![M]\!]_k = \\
\quad \texttt{MAP}\ (\lambda L.\ \textit{get\_cft}[\![M]\!]_L)\ (\texttt{SUBSETS}\ (\texttt{LEAFLOCS}\ M)\ k)
\end{array}
$$

Figure 2: Obtaining counter-factual types for $\lambda^{\rightarrow}$

We expect to obtain counter-factual types from this transformed program. However, it is ill-typed! In the original program `List.iter` has a polymorphic type, but the substituted expression `(hole List.iter)` has a monomorphic type, because the variable `hole` is λ-bound in the program. Therefore the let-bound variable `id` is monomorphic too and its two different uses have to cause a type error.

Solution. To preserve the polymorphism of let-bound variable `id`, we move the binding for `hole` under the definition of `id`.

```
let id = (fun hole -> (fun lst ->
           (hole List.iter) (fun x -> x) lst))
```

This transformation increases the number of `id`'s arguments. Therefore, we add additional variables `hole1` and `hole2` to each occurrence of `id` as the first argument.

```
let id = (fun hole -> (fun lst ->
           (hole List.iter) (fun x -> x) lst)) in
  (id hole1 [1;3;4], id hole2 [true])
```

In this program the fresh variables `hole1` and `hole2` are not bound. Therefore we add lambda bindings for them and obtain the following program as the final result:

```
fun hole1 -> fun hole2 ->
let id = (fun hole -> (fun lst ->
           (hole List.iter) (fun x -> x) lst)) in
  (id hole1 [1;3;4], id hole2 [true])
```

We infer the type of this transformed program using an existing type checker and obtain the expected types and actual types of the occurrence of `List.iter`. From the types of `hole1` and `hole2` we know that one of the expected types is `('b -> 'b) -> int list -> 'c` and another one is `('d -> 'd) -> bool list -> 'e`. Using these types, we can produce the following counter-factual type error message:

```
let id = (fun lst -> List.iter (fun x -> x) lst) in
  (id [1;3;4], id [true])

Error: This expression has type
    (('a -> unit) -> 'a list -> unit)
but it should be an expression with types
    ('b -> 'b) -> int list -> 'c
    ('d -> 'd) -> bool list -> 'e
```

We note that a counter-factual type error sometimes has to explicitly require that the replacement for an expression is polymorphic. Our message states this requirement by listing several types for the replacement.

Fixed Points. In OCaml the let-rec-binding is also polymorphic. However, we can translate an OCaml program such as

```
let rec exponential x =
  if x = "0" then 1
             else exponential (x - 1) in
   exponential 5
```

into our language as follows:

```
let exponential =
  (fix (fun f -> fun x ->
    if x = "0" then 1 else f (x - 1))) in
  exponential 5
```

$$
\begin{array}{rcll}
(M : term) & ::= & c^l & \text{(constant)} \\
& | & x^l & \text{(variable)} \\
& | & \texttt{fun}\ x \mapsto M & \text{(abstraction)} \\
& | & \texttt{@}\, M_1\, M_2 & \text{(application)} \\
& | & \texttt{let}\ x = M_1\ \texttt{in}\ M_2 & \text{(let expression)} \\
& | & \texttt{fix M} & \text{(fixed point)}
\end{array}
$$

Figure 3: The terms of λ^{let}

$$
\begin{array}{l}
pierce : term * (loc\ list) \to term * var\ list \\
pierce[\![\texttt{let}\ x = M_1\ \texttt{in}\ M_2]\!]_L = \\
\quad \texttt{let}\ (M_1', [u_1; \cdots; u_n]) = pierce[\![M_1]\!]_L\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us') = add[\![M_2]\!]_{(x,n)}\ \texttt{in} \\
\quad \texttt{let}\ (M_2'', us'') = pierce[\![M_2']\!]_L\ \texttt{in} \\
\quad (\texttt{let}\ x = \texttt{fun}\ u_1 \mapsto \cdots \mapsto \texttt{fun}\ u_n \mapsto M_1'\ \texttt{in}\ M_2'', us' + us'') \\
pierce[\![\texttt{fix}\, M]\!]_L = \\
\quad \texttt{let}\ (M', us) = pierce[\![M]\!]_L\ \texttt{in}\ (\texttt{fix}\, M', us)
\end{array}
$$

$$
\begin{array}{l}
add : term * var * int \to term * var\ list \\
add[\![c^l]\!]_{(y,n)} = (c^l, []) \\
add[\![x^l]\!]_{(y,n)} = \\
\quad \texttt{if}\ \ y \neq x\ \texttt{then}\ (x^l, []) \\
\qquad \texttt{else let}\ u_1 = \texttt{GENVAR}\ l\ \texttt{in} \\
\qquad\qquad \vdots \qquad \vdots \\
\qquad\qquad \texttt{let}\ u_n = \texttt{GENVAR}\ l\ \texttt{in} \\
\qquad\qquad (\texttt{@}\, x\, u_1\ \ldots\ u_n, [u_1; \cdots; u_n]) \\
add[\![\texttt{fun}\ x \mapsto M]\!]_{(y,n)} = \\
\quad \texttt{if}\ y = x\ \texttt{then}\ (\texttt{fun}\ x \mapsto M, []) \\
\qquad \texttt{else let}\ (M', us') = add[\![M]\!]_{(y,n)}\ \texttt{in} \\
\qquad\quad (\texttt{fun}\ x \mapsto M', us') \\
add[\![\texttt{@}\, M_1\, M_2]\!]_{(y,n)} = \\
\quad \texttt{let}\ (M_1', us_1) = add[\![M_1]\!]_{(y,n)}\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us_2) = add[\![M_2]\!]_{(y,n)}\ \texttt{in}\ (\texttt{@}\, M_1'\, M_2', us_1 + us_2) \\
add[\![\texttt{let}\ x = M_1\ \texttt{in}\ M_2]\!]_{(y,n)} = \\
\quad \texttt{let}\ (M_1', us_1) = add[\![M_1]\!]_{(y,n)}\ \texttt{in} \\
\quad \texttt{let}\ (M_2', us_2) = add[\![M_2]\!]_{(y,n)}\ \texttt{in} \\
\quad \texttt{if}\ us_1 = []\ \texttt{then}\ (\texttt{let}\ x = M_1'\ \texttt{in}\ M_2', us_2) \\
\qquad \texttt{else let}\ [u_1; \cdots; u_m] = us_1\ \texttt{in} \\
\qquad\quad \texttt{let}\ (M_2'', us_2') = add[\![M_2']\!]_{(x,m)}\ \texttt{in} \\
\qquad\quad (\texttt{let}\ x = \texttt{fun}\ u_1 \mapsto \cdots \texttt{fun}\ u_m \mapsto M_1'\ \texttt{in}\ M_2'' \\
\qquad\quad , us_2 + us_2') \\
add[\![\texttt{fix}\, M]\!]_{(y,n)} = \\
\quad \texttt{let}\ (M', us') = add[\![M]\!]_{(y,n)}\ \texttt{in}\ (\texttt{fix}\, M', us')
\end{array}
$$

Figure 4: Obtaining counter-factual types for λ^{let} (new cases only)

The fixed point construct is monomorphic. `fix` simply has type `(('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)` and does not require any special treatment. Only the let-binding is polymorphic and needs to be handled as described before. For example, when we consider the occurrence of `"0"` in the program as a potential

type error location, we pass the following transformed program to the type checker:

```
(fun hole1 ->
let exponential =
  (fun hole -> (fix (fun f -> fun x ->
    if x = (hole "0") then 1 else f  (x - 1)))) in
  exponential hole1 5 )
```

From the inferred type for `hole1` we then produce the counter-factual type error message:

```
let exponential =
  (fix (fun f  -> fun x ->
    if x = "0" then 1 else f (x - 1))) in
  exponential 5

Error: This expression has type
    string
but it should be an expression of type
    int
```

Formalisation. Figure 3 shows the syntax of the let-polymorphic λ-calculus λ^{let}. The types are the same as for $\lambda^{\rightarrow}$ in Figure 1. We do not need to include type schemas amongst the types. Type schemas are used during type inference for inferring polymorphic types. Because we use an existing compiler's type checker and its inferred types, our type debugger never sees any type schemas.

Figure 4 extends our algorithm of Figure 2 to obtaining counter-factual types for λ^{let}. Recall the purpose of function *pierce*: It takes an expression and list of locations; then it inserts new variables as holes at the given locations in the expression and also returns the list of new variables. For a let-expression it uses the new function *add* to add new n argument variables to every occurrence of a target variable y. The function *add* returns both the transformed expression and the list of new argument variables. Note that GENVAR is not a pure function but creates a fresh variable for every call.

5 OBTAINING COUNTER-FACTUAL TYPES MORE EFFICIENTLY

In Sections 3 and 4 we introduced a method to obtain counter-factual types using an existing type checker. However, the time complexity of the naive implementation of this method (*get_cft_nloc* in Figure 2) is too high to be used for anything but short toy programs. Hence we consider in this section how to improve the algorithm to make the method applicable to real programs.

Time complexity of the naive implementation. Assume we have a program of size n. Any leaf in the syntax tree could be a single location type error. So to find all single location errors, we have to call the type checker $O(n)$ times with a variant of the program. There are $O(n^2)$ location pairs that could be 2-location type errors. In general, the naive implementation described in Sections 3 and 4 calls the type checker $O(n^k)$ times to find all k-location counter-factual type errors. This exponential time complexity is clearly unacceptable in practice.

Type constraints and program slices. To improve the time complexity, we now build on established work on type error slicing, which is also discussed in Section 8. A slice is a part of a program, possibly including holes. Every part of a program gives rise to a constraint on the types of the whole program. For a fixed program there is a bijection between its slices and the subsets of its type constraints. We assume that we have an ill-typed program P and its set of type constraints is p. Because P is an ill-typed program, $ill\_typed(p)$ holds.

If we assume that the original type constraints of the holed part (for counter-factual types) are p_{cft}, then the type constraints of the transformed program are $p \setminus p_{\text{cft}}$. The aim of this paper is to find p_{cft} that satisfies the following property:

$$well\_typed(p \setminus p_{\text{cft}}) \; and \; \forall p' \subset p_{\text{cft}}.\; ill\_typed(p \setminus p')$$

The second part ensures that p_{cft} is minimal; a minimal set is removed from the program to make the program well-typed.

Our concern is that it is sometimes hard to find p_{cft}. The key observation to solve this problem is that this property is similar to the property of type error slices [6]. Let p_{slice} be the type constraints of a type error slice of P. Then p_{slice} satisfies the following property:

$$ill\_typed(p_{\text{slice}}) \; and \; \forall p' \subset p_{\text{slice}}.\; well\_typed(p')$$

From this definition we know that if we have a type error slice p_{slice}, we can obtain many p_{cft}s of p_{slice} easily. Therefore, our improved approach is the following. First we obtain a type error slice p_{slice}. Any non-empty subset of p_{slice} is p_{cft} of p_{slice}. To obtain p_{cft} of p, first we choose a part of p_{slice} as p_{cft} and restore each element of $p \setminus p_{\text{slice}}$ to p_{slice} or p_{cft}. Then we can obtain p_{cft} of p. If we apply this method to each subset of p_{slice}, then we can obtain several counter-factual types of the program P.

An example. Let us see how our improved approach works with a concrete example:

```
(fun x -> x +. 1) 3
```

The underlined part is a type error slice of the program. Every part of a type error slice can be a hole for a counter-factual type error. Thus we can choose either `+.` or `1` as a `hole`.

Choice 1. First we pierce 1, that is, apply the *pierce* function to the program with the location of 1.

```
(fun hole -> (fun x -> x +. (hole 1)) 3)
```

We might expect this program to be well-typed, but it is not, because the types of `3` and `+.` conflict. This is because if $well\_typed(p_{\text{slice}} \setminus x)$ holds, $well\_typed(p \setminus x)$ does not always hold. To avoid this problem, we should pierce the parts that are not included in the type error slice by `holes` as follows:

```
(fun hole -> (fun h1 -> (fun h2 ->
  (fun x -> h1 +. (hole 1)) h2)))
```

Then we obtain a counter-factual type error and generate the following error message:

```
(fun x -> x +. 1) 3
Error: This expression has type
    int
but it should be an expression of type
    float
```

This approach, making counter-factual types of type error slices, seems to work well. However, this is a counter-factual type of this type error slice, it is not always a counter-factual type of the whole

program. To see the problem, let us consider another choice of the original program.

Choice (+.). Following our method, we obtain:

```
(fun hole -> (fun h1 -> (fun h2 ->
  (fun x -> (hole (+.)) h1 1) h2)))
```

The parts that are not included in the type error slice are replaced by holes too. Because this program is well-typed, we can produce the following message:

```
(fun x -> x +. 1) 3
Error: This expression has type
    float -> float -> float
but it should be an expression of type
    'a -> int -> 'b
```

Because this is a counter-factual type error for this type error slice, the expected type is more general than the expected type of the whole program, `int -> int -> 'c`. However, it is very easy to expand the upper program to obtain the counter-factual types of the whole program. We just restore removed program fragments that do not conflict. In this example, we can restore all removed program fragments, `x` and `3`.

```
(fun hole -> (fun x -> (hole (+.)) x 1) 2)
```

This program is well-typed and satisfies the property of counter-factual type errors of the whole program.

The algorithm. We need a type error slicer using existing compiler's type checker. Schilling [12] computes type error slices by increasing program fragments until the program is ill-typed. Because the last added program fragment relates to a type error, we can obtain type error slices by collecting them. We can emulate his approach using the function *pierce* of Figures 2 and 4. Figure 5 shows the functions for type error slicing, Figure 6 shows the function for restoring, and the main function for improved obtaining counter-factual types is given in Figure 7.

Slicing function increases the number of program fragments and check whether the program fragments are enough to be ill-typed. The function $slicing'$ choose a location randomly from L using PICKUP_ONE. Because L is the locations of holes, $L_{\text{slice}} \setminus l$ means increasing the l's program fragment (= decreasing the number of holes). If the program constructed by program M and location information $L_{\text{slice}} \setminus l$ is ill-typed, the last added location relates to the type error. Therefore $slicing'$ returns the last added location. The function *slicing* gathers the return value by $slicing'$. It stops when the gathered locations are enough to be ill-typed.

The function *restore* restores the removed parts by type error slicing. It receives locations of counter-factual types for a type error slice and checks whether each location makes the whole program well-typed or not. If well-typedness had kept after addition of a location, it does not relate to a type error.

The function *get_cfts_of_slice* is the main function, obtaining counter-factual types. First, it obtains a type error slice using *slicing*. Because L_{slice} includes the locations of a type error slice, we can use $L \setminus L\_slice$ for obtaining a type error slice using *pierce*. We add a part of L_{slice} to it for obtaining its counter-factual type, and restore the L_{cft} and infer its type. We apply this method to each L_{cft} and obtain several counter-factual types.

$$
\begin{array}{l}
slicing' : term * (loc\ list) * (loc\ list) \to loc \\
slicing'[\![M]\!]_{(L, L_{\text{slice}})} = \\
\quad \texttt{let}\ l = \texttt{PICKUP\_ONE}\ L\ \texttt{in} \\
\quad \texttt{if}\ infer(M, L_{\text{slice}} \setminus l) = [] \\
\qquad \texttt{then}\ slicing'[\![M]\!]_{(L \setminus l, L_{\text{slice}} \setminus l)} \\
\qquad \texttt{else}\ l
\end{array}
$$

$$
\begin{array}{l}
slicing : term * (loc\ list) * (loc\ list) \to loc\ list \\
slicing[\![M]\!]_{(L, L_{slice})} = \\
\quad \texttt{if}\ infer(M, L_{slice}) = []\ \texttt{then}\ L_{slice} \\
\qquad \texttt{else let}\ l = slicing'[\![M]\!]_{(L, L_{slice})}\ \texttt{in} \\
\qquad\quad slicing[\![M]\!]_{(L \setminus l, L_{slice} \setminus l)}
\end{array}
$$

Figure 5: Type error slicing

$$
\begin{array}{l}
restore : term * (loc\ list) * (loc\ list) \to loc\ list \\
restore[\![M]\!]_{(L_{\text{cft}}, L_{\text{acc}})} = \\
\quad \texttt{if}\ L_{\text{cft}} = []\ then\ L_{\text{acc}} \\
\qquad \texttt{else let}\ l = \texttt{PICKUP\_ONE}\ (L_{\text{cft}})\ \texttt{in} \\
\qquad\quad \texttt{if}\ infer(M, L_{\text{cft}} \setminus l) = [] \\
\qquad\qquad \texttt{then}\ restore[\![M]\!]_{(L_{\text{cft}} \setminus l, L_{\text{acc}})} \\
\qquad\qquad \texttt{else}\ restore[\![M]\!]_{(L_{\text{cft}} \setminus l, L_{\text{acc}} + l)}
\end{array}
$$

Figure 6: Restoring

$$
\begin{array}{l}
get\_cfts\_of\_slice : term * (loc\ list) \to ((loc * typ * typ)\ list)\ list \\
get\_cfts\_of\_slice[\![M]\!]_L = \\
\quad \texttt{let}\ L_{\text{slice}} = slicing[\![M]\!]_{(L, L)}\ \texttt{in} \\
\quad \texttt{let}\ L_{\text{cft}}s = \texttt{MAP}(\lambda l.\ (L \setminus L_{\text{slice}}) + l)\ L_{\text{slice}}\ \texttt{in} \\
\quad \texttt{let}\ M's = \texttt{MAP}\ (\lambda L_{\text{cft}}.\ restore[\![M]\!]_{(L_{\text{cft}}, [])})\ L_{\text{cft}}s\ \texttt{in} \\
\quad \texttt{MAP}\ (\lambda M'.\ infer[\![(M', [l])]\!])\ M's
\end{array}
$$

Figure 7: Improved obtaining counter-factual types

Computational complexity. Let n be the program size. The complexity of type error slicing is $O(n^2)$ and the complexity of restoring is $O(n)$ for each p_{cft}. Therefore the whole complexity is $O(n^2)$. If we need k locations for $k > 1$, then the complexity is lower than the naive use of Sections 3 and 4.

6 AN INTERACTIVE TYPE DEBUGGER

In this section, we describe an interactive debugger that uses the enumerated counter-factual type error messages to find the correct type error location. Usually the debugger shows only a few counter-factual type error messages to the programmer.

6.1 When is the correct location found?

The purpose of the interactive type debugger is to determine a subexpression in the program that needs to be changed to obtain a well-typed program. The debugging process is driven by the programmer's intention, the types that the programmer intends certain subexpressions to have.

To clarify the idea, let us consider some examples.

```
Choose your intended type for this expression.

 (1 +. 2)

A: float -> float -> float
B: int -> int -> int
Your choice: (C: another type): B
```

In this case `+.` is the correct type error location, because the actual type of `+.` and the programmer's intended type are in conflict.

```
Choose your intended type for this expression.

let f = (+.) in (f 1 2)

A: float -> float -> float
B: int -> int -> int
Your choice (C: another type): B
```

This program is very similar to the previous example. However, the cause of the type error has not yet been located. The highlighted expression `f` is not the cause, but the expression `(+.)`. Replacing `(+.)` by `(+)` makes the program well-typed.

These examples indicate when the cause of a type error has been found. During the debugging process the type debugger collects the programmer's intentions. The cause of a type error is located when there is a conflict amongst the stated programmer's intentions.

6.2 The debugging process.

We reconsider the following program from the introduction:

```
let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ n) lst)) in
f 2.0
```

In this program we have four potential single counter-factual type error locations: `f` (in `f 2.0`), `2.0`, `^` and `n` (in `x ^ n`). Our interactive type debugger does not simply enumerate these four locations with actual and expected types and leaves it to the programmer to find the location that causes the type error. Instead the type debugger starts by randomly selecting one counter-factual type error message. Let us assume that the selected message is

```
Choose your intended type for this expression.

let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ n) lst)) in
f 2.0

A: float
B: string
Your choice (C: another type):
```

Choice A. The programmer states that the actual type, `float`, is the intended type. The debugger adds this information to the original ill-typed program by adding a type annotation:

```
let f = (fun n -> (fun lst ->
       List.map (fun x -> x ^ (n:float)) lst)) in
f 2.0
```

For this annotated program we can obtain only one counter-factual type error that does not include `(n:float)`. Because `^` and `(n:float)` have a type conflict, we have to remove either `^` or `(n:float)` for the program to be well-typed. However, the debugger already asked about `n`, hence `^` must be the type error location:

```
Type error located:
let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ n) lst)) in
f 2.0

This expression has type
    string -> string -> string
but it should be an expression of type
    'a -> float -> 'b
```

In conclusion, we started with four potential type error locations, but after one question the debugger has identified the correct type error location.

Choice B. The programmer states that the expected type, `string`, is the intended type.

Note that just because the expected type is the intended type, `n` is not necessarily the cause of the type error. The types of variables are often forced by other parts of the program[3].

To judge whether the `n` is the cause of the type error or not, the type debugger generates the following program with several holes and passes it to the type checker:

```
(fun h5 -> fun h6 -> fun h7 -> fun h8 -> fun h9 ->
  fun h10 ->
    let f = fun h1 -> fun h2 -> fun h3 -> fun h4 ->
       fun n -> fun lst ->
         h1 (fun x -> h2 h3 (n:string)) h4 in
         h5 h6 h7 h8 h9 h10)
```

This is the original program with every leaf replaced by a different variable, except for the sub term `n` and its type annotation. Because this program is well-typed, the debugger knows that it has to continue considering other candidates as cause of the type error.

The debugger adds the intended type as a type annotation and produces counter-factual type error messages for the program:

```
let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ (n:string))
             lst)) in
f 2.0
```

There are only two single-location counter-factual type error messages, one for `f` and one for `2.0`. Note that `^` is no longer a candidate.

The debugger randomly chooses one of the two counter-factual type error messages:

```
Choose your intended type for this expression.

let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ (n:string))
             lst)) in
f 2.0

A: float
B: string
```

[3]This does not mean that the cause of a type error has to be in another part. The cause is sometimes a variable itself. In this example, if the programmer replaced `n` by another variable that has type `string`, then the whole program would be well-typed.

```
Your choice (C: another type): B
```

Let us assume that the programmer states that the expected type B is correct. Integrating this intention into the program the debugger internally produces

```
let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ (n:string))
             lst)) in
f (2.0:string)
```

Again the debugger has to check whether the cause has been found or it needs to continue asking questions. Hence again it replaces every leaf by a different variable excluding the terms already asked about and their type annotations. It passes the following program to the type checker:

```
(fun h5 -> fun h6 -> fun h7 -> fun h8 -> fun h9 ->
let f = fun h1 -> fun h2 -> fun h3 -> fun h4 ->
  fun n -> fun lst ->
  h1 (fun x -> h2 h3 (n:string)) h4 in
  h5 h6 h7 h8 h9 (2.0:string))
```

This program is ill-typed and hence the last counter-factual type error did indeed locate the fault:

```
Type error located:

let f = (fun n -> (fun lst ->
           List.map (fun x -> x ^ (n:string))
             lst)) in
f 2.0

This expression has type
    float
but it should be an expression of type
    string
```

Type annotations. Type annotations reduce type error candidates very effectively. This raises the question whether, to find the cause of a type error, the programmer could not simply add themselves type annotations about their intentions to a program. Sometimes this does work, but there is no guarantee. Type checking algorithms are free to handle type annotations in many different ways. When the OCaml type checker traverses a type annotation, it often already inferred the type of the annotated term. Hence OCaml often identifies the correctly annotated term as the cause of the type error. Because counter-factual types are produced independently of the order of type unification, type annotations work more effectively than with standard type checking.

6.3 The Type Error Debugging Algorithm

We extend the object language with type annotated terms in Figure 8. For simplicity we assume that annotated terms are only introduced by the debugger[4].

The definitions for additional syntax to obtain the counter-factual types are shown in Figure 9. In the definition of *pierce* we do never remove type annotations, because we assume type annotations introduced by the debugger are always correct.

$$(M : \mathit{term}) \quad ::= \quad (M : \tau) \qquad \text{(type annotated term)}$$

Figure 8: Additional syntax for type annotation

$$\begin{array}{l}
\mathit{pierce} : \mathit{term} * (\mathit{loc}\ \mathit{list}) \to \mathit{term} * (\mathit{loc}\ \mathit{list}) \\
\mathit{pierce}[\![(M : \tau)]\!]_L = \texttt{let}\ (M', us) = \mathit{pierce}[\![M]\!]_L\ \texttt{in}\ (M' : \tau, us) \\
\\
\mathit{add} : \mathit{term} * \mathit{term} * (\mathit{loc}\ \mathit{list}) \to \mathit{term} * (\mathit{loc}\ \mathit{list}) \\
\mathit{add}[\![(M : \tau)]\!]_{(f, us)} = \\
\quad \texttt{let}\ (M', us') = \mathit{add}[\![M]\!]_{(f, us)}\ \texttt{in}\ ((M' : \tau), us')
\end{array}$$

Figure 9: Additional definitions for obtaining counter-factual type errors

$$\begin{array}{l}
\mathit{annot} : \mathit{term} * \mathit{loc} * \mathit{typ} \to \mathit{term} \\
\mathit{annot}[\![c^l]\!]_{(l', \tau)} = \texttt{if}\ l = l'\ \texttt{then}\ (\texttt{c}^l : \tau)\ \texttt{else}\ \texttt{c}^l \\
\mathit{annot}[\![x^l]\!]_{(l', \tau)} = \texttt{if}\ l = l'\ \texttt{then}\ (x^l : \tau)\ \texttt{else}\ x^l \\
\mathit{annot}[\![\texttt{fun}\ x \mapsto M]\!]_{(l', \tau)} = \texttt{fun}\ x \mapsto \mathit{annot}[\![M]\!]_{(l', \tau)} \\
\mathit{annot}[\![\texttt{@}\ M_1\ M_2]\!]_{(l', \tau)} = \texttt{@}\ \mathit{annot}[\![M_1]\!]_{(l', \tau)}\ \mathit{annot}[\![M_2]\!]_{(l', \tau)} \\
\mathit{annot}[\![\texttt{let x = M}_1\ \texttt{in M}_2]\!]_{(l', \tau)} = \\
\quad \texttt{let}\ x = \mathit{annot}[\![M_1]\!]_{(l', \tau)}\ \texttt{in}\ \mathit{annot}[\![M_2]\!]_{(l', \tau)} \\
\mathit{annot}[\![\texttt{fix}\ M]\!]_{(l', \tau)} = \texttt{fix}\ \mathit{annot}[\![M]\!]_{(l', \tau)} \\
\\
\mathit{debug} : \mathit{term} * ((\mathit{loc} * \mathit{typ})\ \mathit{list}) * (\mathit{loc}\ \mathit{list}) \to \mathit{unit} \\
\mathit{debug}[\![M]\!]_{(\mathit{asked}, \mathit{remaining})} = \\
\quad \texttt{if}\ \mathit{infer}(\mathit{pierce}[\![M]\!]_{\mathit{remaining}}) = [] \\
\qquad \texttt{then}\ ()\quad (*\ \text{located at last question}\ *) \\
\qquad \texttt{else let}\ \mathit{lst} = \mathit{get_cft_of_slice}[\![M]\!]_{\mathit{remaining}}\ \texttt{in} \\
\qquad\quad \texttt{let}\ (l, \tau, \tau') = \texttt{PICKUP_ONE}\ \mathit{lst}\ \texttt{in} \\
\qquad\quad \texttt{let}\ \tau_{\text{intended}} = \texttt{ASK}[\![M]\!]_{(l, \tau, \tau')}\ \texttt{in} \\
\qquad\quad \texttt{let}\ M' = \mathit{annot}[\![M]\!]_{(l, \tau_{\text{intended}})}\ \texttt{in} \\
\qquad\quad \mathit{debug}[\![M']\!]_{(l, \tau_{\text{intended}})::\mathit{asked}, \mathit{remaining} \setminus l)}
\end{array}$$

Figure 10: Interactive type debugger using type annotations

The definitions for a type debugger using counter-factual types and type annotations are shown in Figure 10. The function *annot* adds the user's intentions as type annotations. It receives a program, the target location l', and the user's intended type τ for l'. For constants and variables, it adds a type annotation if its location is equal to the target location. For other constructors we call *annot* recursively to add a type annotation in a sub-expressions.

The function *debug* is an interactive type debugger. It receives a closed ill-typed program M, a list of already asked locations with their intended types *asked* and not asked locations *remaining*. First, we check whether the already asked parts of the annotated program M cause a type conflict or not. If they cause a type conflict, then the type debugging process ends. Otherwise, we need to continue asking questions. For asking questions, we obtain counter-factual types

[4] The programmer's annotated terms might be the cause of a type error, but the answers to the debugger cannot be the cause of a type error, provided the debugger's questions are answered correctly. Thus, if we want to introduce the programmer's type annotated terms, we should distinguish between these two forms of annotations.

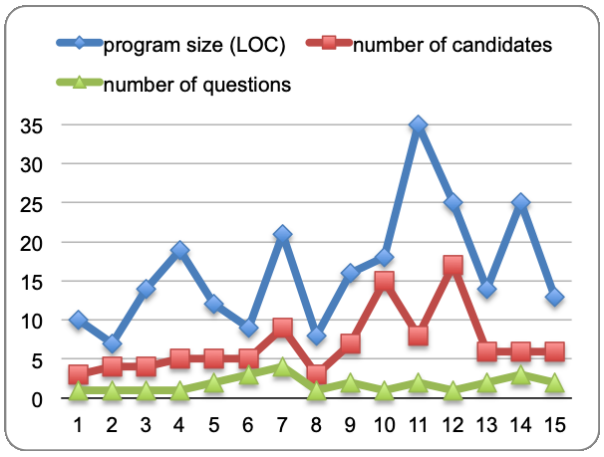


Figure 11: Search space of our prototype

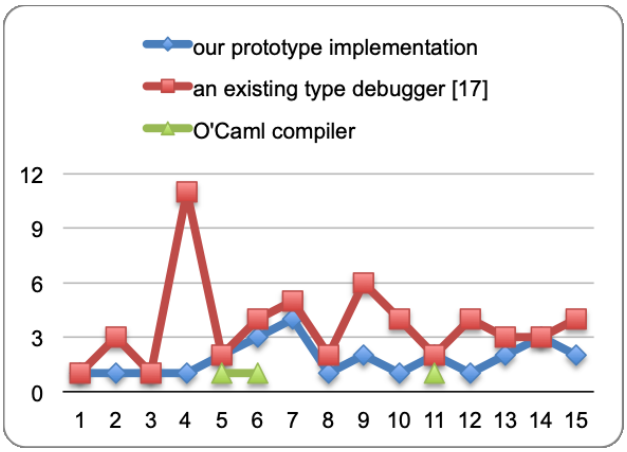


Figure 12: Question numbers of different type debuggers

using *get_cft_of_slice* and ask the programmer about it. The external function `ASK` receives the program M and its counter-factual type, asks the programmer and returns the programmer's intended type τ_{intended}. After that, the type debugger adds the correct intended annotation to the program and obtains a new annotated program `M'`. Because the cause of the type error is not located yet, we call *debug* again with the updated information. To start type debugging, we call *debug* with the original ill-typed program M, an empty asked list and a list of all locations of M.

7 EVALUATION

We built a prototype implementation of our type debugging method. It is embedded in OCaml 4.01.0. We compared it with two other implementations: OCaml 4.01.0 itself and an existing interactive type debugger [17], which is also embedded in OCaml 4.01.0. We use 15 small ill-typed programs from two sources: student's programs from an OCaml introductory course and test code for Skalpel's online demonstration [11].

How was the search space reduced by our approach? Figure 11 gives for each program its size, the number of candidates for the first question and the number of questions needed for locating the fault. We can learn two things: Program parts that are unrelated to type errors are often successfully removed. If a type error includes many conflicts, then the number of candidates can be large, but the numbers of questions needed is smaller, often substantially so.

For removing the unrelated candidates, the user's interactive input can help much. In Test 11 the number of candidates is 8 and our prototype asks about the type of `::`. With the general type of `::`, the number of candidates for the second question is 7. However, if the user inputs a specific type (e.g., `int → int list → int list`), then the number of the candidates for the second question becomes 1. Thus we know that the type annotations of Figure 10 remove candidates substantially.

Is the prototype better than an existing type debugger? In Figure 12 we compare the number of questions with an existing interactive type debugger [17] and the OCaml compiler. Because the OCaml compiler is not interactive, we only record when its error message locates the source of the type error correctly (Test 5, 6 and 11). We see that our prototype locates the source of a type error quicker than the older debugger. This is because the older debugger asks about many parts of the program, but our type debugger focusses on parts related to the type error. The OCaml compiler can sometimes locate the error source, but not always.

Program size (LOC)	Error line (LOC)	Time (seconds)
10	35	1.01
22	37	2.69
122	5	2.43
122	39	2.85
122	107	3.66
482	413	6.92

Table 1: Runtime measurements

Is runtime reasonable for interactive use? Table 1 shows the runtimes of our prototype debugger for different programs. These programs were made by injecting a type error into a well-typed program. The runtime time depends on the program size and error location in the program; it varies substantially. Because our implementation is a prototype, we expect to achieve speed improvements in the future. At least our prototype works in reasonable time for small programs (around 100 LOC).

8 RELATED WORK

A wealth of papers have been published on type error debugging since the 1980's. Here we focus on a few.

New Type Checking Algorithms. Existing functional programming systems use variants of the standard type checking algorithm $\mathcal{W}$ by Milner and Damas [5]. Algorithm $\mathcal{W}$ traverses the abstract syntax tree of the program and eagerly solves type constraints by unification. When type unification fails, the currently inspected expression is reported as type error location together with its actual and expected type. Thus the reported type error location depends

on the order in which $\mathcal{W}$ solves type constraints, that is, traverses subexpressions and unifies types.

Many improved type checking algorithms have been proposed. Wand [19] extends $\mathcal{W}$ to keep track of the history of how type variables are instantiated and shows this history for conflicting types when unification fails. Lee and Yi [9] propose to use the algorithm $\mathcal{M}$. They show that $\mathcal{M}$ finds type conflicts earlier than algorithm $\mathcal{W}$ and thus $\mathcal{M}$ reports a smaller expression as error location. Yang et al. [20] describe a new algorithm $\mathcal{IEI}$ that combines the algorithm $\mathcal{M}$ with unification of type assumption environments. Their implementation enumerates several error messages with counter-factual types like our method. However, $\mathcal{IEI}$ focuses on a type conflict in function applications. Therefore their approach does not enumerate all counter-factual type errors of the ill-typed program. In contrast to all these algorithms, our approach is completely independent of the order of traversing the syntax tree or solving type constraints.

Reusing an Existing Type Checker. Braßel [1] was the first to propose using an existing type checker for type debugging. His experimental tool TypeHope automatically corrects type errors for the functional logic language Curry. The type debugger Seminal [10] takes this idea further. It replaces erroneous parts with various syntactically correct similar expressions, and sees if they type check. If they do, the replacements are displayed as candidates for fixing the type error.

Chen and Erwig [2] already note that usually there exist too many different expression changes that correct a type error; most of these expression changes do not agree with the programmer's intentions. Hence, unlike Seminal [10] but like Chen and Erwig [2], our aim is to suggest type changes and leave it to the programmer to select the appropriate expressions.

Sharrad et. al. [13] take the idea of using an existing type checker as a black box even further by not transforming a syntax tree but just the lines of the original program text. They apply the delta-debugging method to locate a line or lines of code that cause the error. Scaling up to large programs is under current development.

Counter-Factual Types. Most existing compilers for Hindley-Milner-based programming languages produce type error messages with counter-factual types. Also several of the early proposals for improving type error debugging use counter-factual types. However, Chen and Erwig [2] were the first to clearly identify the concept of counter-factual types and argue their usefulness for type error debugging. They propose to assist type debugging by automatically enumerating all potential type error locations with counter-factual types. So a message suggests changing the actual type of a given program location to the counter-factual type. Chen and Erwig use a new type checking algorithm based on variational types. Because these variational types are monomorphic, they restrict counter-factual types to be monomorphic as well. Potential type error locations that require polymorphic types are not identified. In contrast, our approach produces in such a case a list of expected monomorphic types, which could also be transformed into a single expected polymorphic type.

Interactive Type Debugging. Interactive type debugging systems have been proposed to enable the programmer to include their intentions in the search for the error location. Chitil [4] developed an algorithmic debugger for type debugging, using a compositional type inference algorithm. Based on his work, Tsushima and Asai [17] designed an algorithmic type debugger for OCaml that uses the compiler's own type checker rather than a tailor-made type checking algorithm. Algorithmic debugging guarantees to find a type error location correctly, but answering the questions of an algorithmic debugger requires a good understanding of types, especially intended types, by the programmer.

Chen and Erwig [3] propose an interactive type debugger based on enumerating counter-factual type errors. Like our interactive type debugger it determines the correct type error location and gives its counter-factual type. However, the interaction is different in that the programmer does not choose between types like in our debugger but has to enter intended types for given subexpressions. Internally Chen and Erwig's type debugger relies on the use of their variational types [2] to quickly reduce the number of type error location candidates.

Type Error Slicing. The aim of type error slicing is to determine a minimal slice of an ill-typed program which contains all the program parts responsible for the ill-typedness. To make the program well-typed, a change within this type error slice is required. Underlying type error slicing is the observation that every program fragment corresponds to a constraint on the types of subexpressions of the program. For an ill-typed program the complete set of its type constrains is unsatisfiable. A minimal unsatisfiable set of type constraints can be computed that then corresponds to a minimal type error slice of the program.

Haack and Wells [6] define and implement type error slicing for ML using their own type checking algorithm which solves annotated type constraints. Later Rahli et al. [11] extend this work to a much larger fragment of ML. Independently Stuckey, Sulzmann, and Wazny [14][15][16] develop the idea of finding the source of type errors by solving constraints expressed with constraint handling rules (CHRs). Their type debugger called Chameleon implements its own type inference based on CHRs. The debugger can explain type errors and inferred types by showing relevant slices. Their work covers many advanced language features, such as type-annotation, Haskell-style overloading and generalised algebraic data types (GADTs).

Later Schilling [12] obtains type error slices for a large subset of Haskell using the compiler's type checker as a black box. Tsushima and Asai [18] improve Schilling's type error slicer using weights. If a type error slice has conflicts with several parts of the program, then it is more likely to be the cause of ill-typedness. Hence weighting such conflicts enables choosing better slices.

The advantage of type error slicing is that the process is fully automatic and the programmer does not have to answer any questions. On the other hand even minimal slices can still be relatively big and slices do not explain an error or how to correct it. Although looking very different to the programmer, type error slicing and our approach are closely related, as Section 5 demonstrates.

Heuristics for type debugging. Heuristics based on a large corpus of ill-typed programs can help with finding the correct type error location and providing more helpful type error messages. Heeren and Hage [8] use a constraint-based type checker to flexibly vary

the order of solving constraints based on a heuristic. Their approach sometimes but not always produces good error messages, depending on whether a specific type error is considered by the heuristic or not. Hage and Heeren [7] define heuristics for type debugging using their students' programs. They focus on helping non-expert programmers. Wherever our interactive type debugger currently makes a random choice it could instead be guided by a heuristic. With experience in practice the messages themselves could be made more helpful in future.

9 CONCLUSION AND FUTURE WORK

In this paper we proposed a new method for enumerating counter-factual type error messages for all potential type error locations in a program, reusing an existing type checker. Our method also fully supports the polymorphic Hindley-Milner type system in that it handles the situation where a subexpression of a program needs to be replaced by a polymorphic expression. The basic idea underlying our method is very simple: we introduce `holes` in the program and obtain their types. We introduced an improved method for obtaining counter-factual type error messages using type error slicing. Its computational complexity is $O(n^2)$ where n is the program's size. We married the enumeration of counter-factual type error messages with a new interactive debugger that locates the correct type error. Type annotations are at the heart of this interactive debugger.

We both gave informal descriptions with examples of all algorithms and defined them formally for the Hindley-Milner-typed λ-calculus with recursion. We also implemented it for a small subset of OCaml.

The advantage of this work is twofold. First, our method does not depend on any algorithm of inferring types and it requires only the inferred types. Hence our method can be applied to many programming languages based on Hindley-Milner type system, regardless of which type inference algorithm is used by their compilers. Second, it is easy to implement. To implement a type checker that returns exactly the same type as the compiler's type checker would be tedious and error-prone.

We have several plans for future work:

We want to explore whether replacing the program slicing by our delta-debugging implementation [13] improves the speed of our type error debugger.

We want to apply our idea to advanced language features. Practical functional programming languages include numerous features, for example module and object systems, for which we still have to define our method. Thus we expect to prove the scalability of our approach.

Finally, we want to develop heuristics for our type debugger. Which potential type error location should the debugger ask about first? Are some locations more likely to be correct than others? Currently we have no answers to these questions. We first need to implement a type debugger that covers a large subset of OCaml. We will use it with many programmers and then collect their ill-typed programs and how they corrected them.

REFERENCES

[1] Braßel, B. "Typehope: There is hope for your type errors," *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'04)*, (2004).

[2] Chen, S., M. Erwig. "Counter-Factual Typing for Debugging Type Errors," *Proceedings of the 41th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'14)*, (2014).

[3] Chen, S., M. Erwig. "Guided Type Debugging," *Proceedings of the 12th International Symposium on Functional and Logic Programming (FLOPS'14)* pp. 35–51 (2014).

[4] Chitil, O. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors," *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming (ICFP'01)*, pp. 193–204 (2001).

[5] Damas, L., R. Milner. "Principal type-schemes for functional programs," *Proceedings of the 9th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'82)*, pp. 207–212 (1982).

[6] Haack, C., J. B. Wells. "Type Error Slicing in Implicitly Typed Higher-Order Languages," *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP'03)*, Volume 50 Issue 1-3 (2004).

[7] Hage, J., and B. Heeren. "Heuristics for Type Error Discovery and Recovery," *Proceedings of the 12th International Workshop on Implementation and Application of Functional Languages (IFL'07)*, Lecture Notes in Computer Science Volume 4449, pp. 199-216 (2007).

[8] Heeren, B., D. Leijen., J. Hage. "Helium, for Learning Haskell," *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell'03)*, pp. 62–72 (2003).

[9] Lee, O., K. Yi. "Proofs about a Folklore let-polymorphic Type Inference Algorithm," *ACM Transactions on Programming Languages and Systems*, pp. 707-723 (1998).

[10] Lerner, B. S., M. Flower, D. Grossman, C. Chambers. "Searching for Type-Error Messages," *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pp. 425–434 (2007).

[11] Rahli, V., J. B. Wells, J. Pirie and F. Kamareddine. "Skalpel: a type error slicer for standard ML," *Electronic Notes in Theoretical Computer Science*, Vol. 312, pp. 197–213 (2015).

[12] Schilling, T. "Constraint Free Type Error Slicing," *Proceedings of the 12th international conference on Trends in Functional Programming (TFP'11)*, pp. 1–16 (2012).

[13] Sharrad, J., O. Chitil and M.Wang. "Delta Debugging Type Errors with a Blackbox Compiler," *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018)*, pp. 13–24 (2019).

[14] Stuckey, P. J., M. Sulzmann, J. Wazny. "Interactive type debugging in Haskell," *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell'03)*, pp. 72–83 (2003).

[15] Stuckey, P. J., M. Sulzmann, J. Wazny. "Improving type error diagnosis," *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (Haskell'04)*, pp. 80–91 (2004).

[16] Stuckey, P. J., M. Sulzmann, J. Wazny. "Type Processing by Constraint Reasoning," *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS'06)*, pp. 1–25 (2006).

[17] Tsushima, K., K. Asai. "An Embedded Type Debugger," *Proceedings of the 24th International Workshop on Implementation of Functional Languages (IFL'12)*, pp. 190–206, (2013).

[18] Tsushima, K., and K. Asai. "A weighted type-error slicer (in Japanese)," *Journal of Computer Software*, Vol. 31, No. 4, pp. 131–148 (2014).

[19] Wand, M. "Finding the Source of Type Errors," *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'86)*, pp. 38–43 (1986).

[20] Yang, J., G. Michaelson, P. Trinder, J. B. Wells. "Improved Type Error Reporting," *Proceedings of 12th International Workshop on Implementation of Functional Languages (IFL'00)*, pp. 71-86 (2000)

Lazy Interworking of Compiled and Interpreted Code for Sandboxing and Distributed Systems

Camil Staps
info@camilstaps.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

John van Groningen
johnvg@cs.ru.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

Rinus Plasmeijer
rinus@cs.ru.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

ABSTRACT

More and more applications rely on the safe execution of code unknown at compile-time, for example in the implementation of web browsers and plugin systems. Furthermore, these applications usually require some form of communication between the added code and its embedder, and hence a communication channel must be set up in which values are serialized and deserialized. This paper shows that in a functional programming language we can solve these two problems at once, if we realize that the execution of extra code is nothing more than the deserialization of a value which happens to be a function. To demonstrate this, we describe the implementation of a serialization library for the language Clean, which internally uses an interpreter to evaluate added code in a separate, sandboxed environment. Remarkable is that despite the conceptual asymmetry between "host" and "interpreter", lazy interworking must be implemented in a highly symmetric fashion, much akin to distributed systems. The library interworks on a low level with the native Clean program, but has been implemented without any changes to the native runtime system. It can therefore easily be ported to other programming languages.

We can use the same technique in the context of the web, where we want to be able to share possibly lazy values between a server and a client. In this case the interpreter runs in WebAssembly in the browser and communicates seamlessly with the server, written in Clean. We use this in the *iTasks* web framework to handle communication and offload computations to the client to reduce stress on the server-side. Previously, this framework cross-compiled the Clean source code to JavaScript and used JSON for communication. The interpreter has a more predictable and better performance, and integration is much simpler because it interworks on a lower level with the web server.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; • **Information systems** → *Browsers*; • **Computer systems organization** → *Client-server architectures.*

IFL'19, September 2020, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

interpreters, functional programming, laziness, sandboxing, WebAssembly

ACM Reference Format:
Camil Staps, John van Groningen, and Rinus Plasmeijer. 2020. Lazy Interworking of Compiled and Interpreted Code for Sandboxing and Distributed Systems. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The execution of untrusted code, or code that is unknown to the executable when it is started, has become paramount in a large number of different contexts. The most common example is the web browser, which must run JavaScript and WebAssembly code that can interact with the web page but should under no circumstance crash the browser or the rendering engine.

Another use case can be found in plugin systems. We may for example imagine a compiler which can load plugins for various language extensions, where plugins can provide functions for transforming the abstract syntax tree. In this case the plugin has to interwork on an even more fine-grained level with the host program, interacting closely with the same data types.

A similar need is felt in workflow systems like the *iTasks* framework for Task-Oriented Programming [24, 25]. Applications written in this framework center around various tasks and the workflows that users walk through to execute them. Currently, applications must be recompiled to add new possible workflows. However, if we could dynamically add code to a running executable, the iTasks server could remain online during updates.

In this paper, we shall focus on the runtime safety aims required for plugin and workflow systems: we assume that the plugin is trusted (which, in these contexts, can be verified using other techniques, such as digital signatures), so do not need to protect against reading sensitive data and the like, but still need to prevent it from crashing the host program.

In many of these contexts, we furthermore want the "added code" or the "plugin" to be able to be distributed in a platform-independent manner: there are no different JavaScript versions for different binary platforms, nor would one expect to have to download a different compiler plugin on Windows than on Linux. This complicates the matter, because it means we cannot simply distribute machine code which can be linked dynamically.

Lastly, in a lazy functional programming language, we expect that the interface between the host program and the added code is lazy as well. For instance, it should in principle be possible to have

a plugin in some system which computes an infinite list of integers, as long as the host only requests a finite amount of them.

1.1 The solution in a nutshell

Pure functional programming languages are at least theoretically ideally suited for the distribution of unknown code: because functions are first-class citizens, the distribution of code is essentially a special case of serialization. Hence we do not need to treat functions separately (as, for instance, Java does with its remote method invocation interface). Instead, we can design a serialization library which automatically deserializes functions as expected.

Unfortunately, so far, implementers of serialization libraries have taken a traditional, "data-only" approach to serialization. This can be seen in the many JSON libraries that are available for virtually every programming language. In a lazy language, serializing a value to JSON can lead to an amount of work disproportionate to the size of the value in the heap, because the value must be evaluated first. Moreover, shared pointers are lost when serializing to such flat formats, and serialization may not even finish when cyclic or infinite data structures are encountered. Another approach to serialization is to make a copy of the graph without evaluating it [8]. However, because this graph may contain references to the machine code that is needed to further evaluate it, it can only be deserialized by the exact same executable, or, when using symbolic addresses, executables that already contain the relevant code [12, 20]. These solutions, while useful in some distributed systems, are therefore insufficient for the use cases described above.

We describe a new solution. To the outside, it looks like a serialization library, with built-in support for functions. Internally, it builds on existing functionality to copy graphs [20]. However, in addition to the string representation of the graph, serialized expressions now also contain a bytecode which can be used to evaluate it. The serialization library includes an interpreter for this bytecode. This way we combine the compiled code of the native host program with the interpreted code of deserialized expressions.

Our library has the following properties. First, it is cross-platform:[1] the serialized representation of a value is independent of the execution platform, and can be deserialized anywhere else, including by executables that do not contain the source code needed to evaluate them. Second, after deserialization, it is possible to handle serialized expressions in the same way as native values, i.e., without constantly using special functions to access them. Third, deserialization of functions happens in a sandbox and cannot crash the host program. As will become apparent, the second and third requirements exclude each other. Both are implemented, but the programmer must choose whether seamless integration or sandboxing is used.

Lastly, we should note that although our library requires low-level access to the Clean heap to serialize graphs and copy nodes between the native heap and the interpreter heap, it was not necessary to change Clean's native runtime system. We therefore believe our implementation can easily be ported to other contexts.

[1] The interpreter must be available on the target platform. Since it is written in C and only ca. 100 lines of assembly code are needed to connect native Clean with the interpreter, new platforms are easily supported.

1.2 Web applications

We have also successfully applied our interpreter in the context of the web. A main issue here is that the frontend and the backend of a web application are usually written in different languages. The frontend has to be executed in the browser, and is therefore limited to very few languages (primarily JavaScript). For programmers it is however much more convenient to write the entire application in one language. For this reason there exist many frameworks that let one compile server-side languages to JavaScript.

However, compiling functional programming languages for the browser is difficult, because they typically rely heavily on features like deep recursion and non-local goto's that cannot easily be mimicked in browser languages. These issues can be worked around, for example by using continuation-passing style ("trampolined code") and simulating a heap, which is done when Clean is compiled to JavaScript through Sapl [10] or when Haskell is compiled to WebAssembly by Asterius [16]. However, each trick that is used to work around these issues creates a bigger gap between the native runtime system and that used in the browser. Hence, it becomes more difficult to use the graph copying mechanism for communication between the server and the client, and performance becomes more and more unpredictable compared to that of native code.

Therefore we take a different approach here. We cross-compile the interpreter used in the serialization library to WebAssembly, so that it can be used to interpret functional programs in the browser. Interworking is then established between the server, which runs the native version of the executable, and the client, which interprets the same functional language and interacts with the web page to create an interactive web application.

1.3 Organisation

The next section describes the ABC machine, the abstract machine that our interpreter simulates. Section 3 describes the (de)serialization library and the interworking with the compiled host program. In section 4 we describe how the same interpreter is used in the browser. Section 5 provides benchmarks for various implementations of the ABC interpreter. We finish by describing related work and summarizing our conclusions in sections 6 and 7.

2 THE ABC MACHINE

The ABC machine [19, pp. 35–56] is an abstract imperative stack machine designed for the execution of lazy functional programs. The Clean compiler generates ABC code, from which actual machine code is generated. Clean's graph rewriting semantics [26, §1.1] are directly mirrored in the architecture of the ABC machine in that it has a heap which contains the graph that is being rewritten. Additionally it uses three stacks, which contain return addresses, basic values (e.g. integers), and references to nodes in the graph.

2.1 Compilation of Clean

Each Clean function is compiled to a sequence of ABC instructions with a number of entry points. Which entry point is used depends on the context in which the function is called. The most basic entry point is the *strict* entry: this entry point is used when the function result is guaranteed to be needed; the arguments are passed on the stack, and the strict arguments have been evaluated. Other

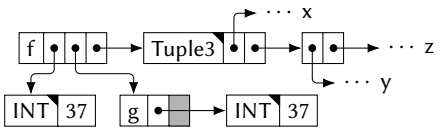


Figure 1: The runtime memory layout of a Clean program.

entry points are used to facilitate currying and lazy evaluation. When a function application is not guaranteed to be needed for the outcome of the program, a thunk is created which contains the arguments and points to the *lazy* entry of the function. When the thunk needs to be evaluated, the program jumps to this entry, which moves the arguments from the node to the stack, evaluates the strict arguments, and proceeds to the strict entry. Afterwards the original node is filled with the result. When a closure is applied to a new argument and this satisfies the arity requirement for evaluation of that node, the *apply* entry is used. This situation is similar to that for the lazy entry, but in this case one or more arguments are already on the stack and a new node is created for the result.

2.2 Concrete implementations

There are several code generators which generate actual machine code for various common instruction sets from the intermediate ABC code. To these target platforms, we have added an ABC interpreter which can execute a bytecode very close to ABC code. The main differences with actual ABC code are a number of straightforward optimizations and the addition of specialized instructions for frequent instruction sequences. This is needed because the ABC language is primarily designed to be an easy compilation target rather than an efficiently interpretable bytecode.

The implementation of the instructions in the interpreter is generated from a small DSL embedded in Clean, with views to produce both C and WebAssembly versions. The supporting code consists of standard bootstrap code (to parse and load the program and print the result) and a copying garbage collector. Only standard techniques were used to implement the interpreter; see [13] for an overview. The C interpreter uses direct threaded code when the compiler supports it (as is the case for GCC and Clang, but not MSVC). The WebAssembly interpreter uses a giant switch approach, which is also the fallback option for the C interpreter.

The runtime layout of the heap is the same whether a program is run natively or in the ABC interpreter. The heap contains two types of nodes: thunks and head normal forms. For both types, the first node points to a *descriptor* containing information about the object, such as the arity and the different entry points. A thunk then contains a contiguous block of pointers to its arguments. A head normal form has the same structure, but is split over two memory blocks if it has more than two arguments. This is exemplified in Figure 1, which shows the memory layout for the expression f 37 (g 37) (x, y, z): it is a thunk with three arguments, and (g 37) a thunk with one argument and a reserved spot. The INT descriptor is for head normal forms (indicated with the triangle in the upper right corner) with one unboxed argument; Tuple3 is a descriptor for head normal forms with three (boxed) arguments. By reserving three words for all thunks (as seen for g) and splitting large head normal form nodes up (as seen for Tuple3), a thunk can always be overwritten by a head normal form (as is done in the lazy entry point).

[1]Here and elsewhere we use the term "closure" for unsaturated function applications. Internally these are represented as head normal form nodes. The head refers to the function that is to be applied when all arguments have been "curried" into it; the node arguments are the (possibly zero) arguments that have already been bound.

3 (DE)SERIALIZATION AND INTERWORKING

Given the memory layout described above, it is easy to see how graphs can be serialized: the graph is traversed in a depth-first manner, marking visited nodes so that shared pointers remain shared after serialization. This technique is commonly referred to as *graph packing* [8, p. 41 and references therein]. It is also implemented for Clean under the name *GraphCopy* [20], with a wrapper, *GraphCopy-with-names*, that translates descriptor addresses to descriptor names. This wrapper can be used to work around the well-known issue that graphs serialized this way can normally only be deserialized by the same executable, because they contain hard-coded addresses (e.g. [12, p. 126]), but does not allow one to extend program functionality yet: all code required to evaluate the expression must be present in the deserializing executable.

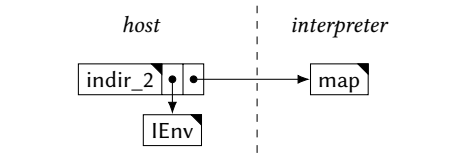
The new serialization library is essentially a wrapper around GraphCopy-with-names. A serialized expression now contains the graph, the descriptor names, and the bytecode. The necessary bytecode is extracted using reachability analysis starting from the descriptors that are used in the graph.

To deserialize an expression using the ABC interpreter, the following steps are performed. First, new heap and stack spaces are allocated, and the bytecode is parsed and loaded into memory. The symbol table of the bytecode program is matched against the symbol table of the executable, to facilitate interworking later on (see section 3.2). Second, the descriptor names in the serialized expression are replaced with the actual addresses of the instantiated bytecode program. After this, the graph can be unpacked with the straightforward inverse operation of the graph packing algorithm described above. The interpreter can now start to evaluate the graph.

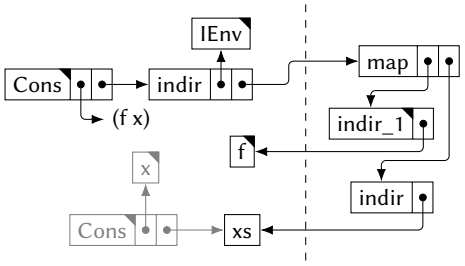
3.1 Interworking with compiled code

To make sure deserialization is lazy, the interpreter does not evaluate the whole graph to a normal form at once. Instead, it evaluates it to head normal form. This head normal form is then copied to the native Clean program, called the host.

Two representative stages of this process are shown in Figure 2, where the well-known function map is interpreted and applied to two native arguments. The native heap is shown on the left of these figures; the interpreter's heap on the right. Figure 2a shows the initial state in which a map closure has been created in the interpreter and an indirection has been made in the host. Figure 2b reflects the state after the function is applied to some native operator f and a native list (Cons x xs), and after the first Cons node of the result has been copied to the native heap. This is the top left Cons node; the other Cons node is the head of the original list (which can now be garbage collected if the host has no other references to it). The first result element (f x) is not shown for brevity. The tail of the resulting list is a reference to a map thunk in the interpreter. The IEnv nodes are required arguments for indirections from the host



(a) Before application.



(b) During evaluation.

Figure 2: Snapshots of the heaps when interpreting map.

to the interpreter; this will be discussed below. We now describe each case of the copying process in detail.

Node arguments that are unboxed values or are in head normal form (and are not closures) are copied directly. This could for example be the case for the first argument of the result Cons node in Figure 2b (the evaluation of f x), if we are dealing with a head-strict list. Although theoretically this can lead to large values being copied even though they are not needed in the host (because they just happen to have been evaluated in the interpreter), this did not appear to be a problem in practice. We expect that creating indirections for nodes that are already in head normal form would lead to additional overhead, so we have decided to copy these nodes directly instead. However, we currently do not have data to confirm or reject this hypothesis.

It is in principle possible to also copy thunks for which the relevant code is present in the host program (for example, the first argument of the result Cons node in case of a lazy list). However, this would impose a security risk: if this lazy value causes a runtime fault, this must be caught by the interpreter, so the lazy value cannot be copied to the host.[2] Instead, an indirection thunk is created in the host for every thunk that is encountered, as can be seen in the indir node on the host side in Figure 2b. This indirection node references the node in the interpreter heap. When evaluated, the interpreter is used to evaluate the node (with the *lazy* entry point), after which a new copy cycle begins.

During the copying of a node, visited nodes are marked, so that cyclical data structures can be copied without problems and shared pointers remain intact. However, this matching between interpreter and host nodes is not retained after the copy cycle has finished (this would require a significant amount of extra work to keep the matching up to date during garbage collection).

So far we have covered situations in which a serialized expression copied to the host may not be fully evaluated but will still eventually be rewritten to a graph that solely consists of data. It is however also possible that the interpreted expression contains closures. These can then be applied to native arguments. As with thunks, we cannot copy these to the host because of security risks and because the relevant evaluation code may not be present in the host. Instead, also in this case an indirection is created in the host. This is the case shown in Figure 2a. This indirection is a closure which still requires as many arguments as the underlying interpreter node; hence, indir_2 is used here instead of indir. The arguments are therefore collected in the native Clean program; only when enough arguments are present they are copied to the interpreter and is the function evaluated. This is done using the *apply* entry of the node.

In the case of closures in deserialized expressions it is therefore necessary to copy values from the host to the interpreter. This works in much the same way as in the other direction. Hence the interpreter heap may also contain indirections to the host heap, as is seen in Figure 2b. The indir node here references the tail of the original list to which map is applied; it is a thunk, because xs has not been evaluated in the host yet. The indir_1 node refers to the operator argument; it is a closure which still requires one argument, mirroring the node in the host.

As can be seen, the relationship is much more symmetrical than the term "host" suggests, and indirections in both directions are required to facilitate laziness. One important difference is, however, that there may be several interpreters while there is always only one host. Hence, indirections from the host to the interpreter have an extra argument, the *interpretation environment* (IEnv in Figure 2), which contains such information as the locations of the bytecode, heap and stack of the interpreter. These are needed to start the interpreter to evaluate the referenced node. Indirections from the interpreter to the host do not need this extra argument.

It is possible that the host applies an interpreted function to a value that is deserialized using another interpreter. This can become quite complex, if there are multiple interpreters and higher order functions involved. This case is automatically caught by the setup presented here, but not very efficiently: indirections from one interpreter to another will in fact be indirections to a host indirection to that other interpreter. This can in principle be optimized by allowing interpreter indirections in the interpreter itself.

3.2 Translation of descriptors

The host program and the interpreter do not share their descriptors, because the descriptor layout is different for different binary platforms and contains references to program code.[3] The consequence is that descriptor addresses must be translated when copying a node. To do this efficiently, the descriptor blocks in the interpreter are expanded with an extra field which is a pointer to the corresponding descriptor in the host. Hence, translation of the descriptor from the interpreter to the host can be done in constant time.

Because a host descriptor may have to be mapped to several interpreter descriptors (when multiple expressions are being deserialized simultaneously), we cannot translate host descriptors to interpreter descriptors in the same way. Instead, each interpreter keeps a separate binary search tree for this purpose.

[2] For the same reason, interpreter instances do not share common libraries with each other or with the host.

[3] To be precise, descriptors contain pointers to entry points to evaluate the node or add arguments to a closure. Depending on the type of node other information can be included as well, e.g. the arity (for garbage collection), the name (of constructors, for printing), the types of unboxed values (for printing), etc.

It is possible that a descriptor of the interpreter does not exist in the host or vice versa. This can for example occur when the interpreted expression uses types unknown to the host or when the interpreted expression is a polymorphic function and is called with a native argument of a type unknown to the interpreter. In this case we allocate a new block of memory and copy the descriptor. This is needed because the garbage collector uses information from the descriptors. Since only fully saturated head normal forms are copied, we do not need all elements from the copied descriptor. For instance, descriptors contain a curry table which is used to curry new arguments into closures. This curry table is not copied, because it will never be used. Effectively this means that we only need to copy data in a descriptor, not references to code.

3.3 Impact on garbage collection

As described above, both the host heap and the interpreter heap(s) contain indirections to each other. When garbage collection runs, these indirections must be taken into account so that, for example, interpreter nodes that are referenced from the host (but not from the interpreter heap itself) are kept in memory. Furthermore, references in both directions must be updated when a garbage collector moves objects. As has been mentioned above, the interworking setup is much more symmetric than one may expect, and hence ideas to solve this problem were taken from implementations of distributed systems [18, pp. 102–111]. Nevertheless we are able to present a solution that relies on only one special feature of the native runtime system. This feature can be emulated in other runtime systems with more common features (see note 4).

To retain host nodes referenced from the interpreter, the interpretation environment (in the host) contains an array of nodes shared with the interpreter. Because this array lives in the host, the references are updated by the host garbage collector — and because the interpretation environment is referenced from every indirection from the host to the interpreter, the array is kept alive as long as necessary. An indirection from the interpreter to the host is simply an index into this array and therefore does not need to be updated after garbage collection has taken place in the host. When garbage collection runs in the interpreter, the array is cleaned up.

To retain interpreter nodes referenced from the host, all references from the host to the interpreter are kept in a weakly linked list in the host. The Clean garbage collector ensures that values in the linked list that are not reachable from elsewhere in the heap are removed from the list.[4] The interpreter's garbage collector makes sure that objects referenced from the weakly linked list are kept in memory and have their references updated.

Finally, the interpretation environment is wrapped in a finalizer, so that a C function is called when it is removed by the garbage collector. This C function then frees all related memory (program code, heap, stack, and descriptor matching); only descriptors that have been copied to the host remain (because they may still be in use in the host).

[4]This is the only uncommon feature of the Clean runtime system that we use. A similar solution can be implemented in runtime systems with hooks at strategic places in the garbage collector, to ensure that the references from the weakly linked list are not considered to determine whether an object can be freed, but *are* updated when objects are moved. This solution can also be emulated in runtime systems with support for finalizers (the ability to execute code after a referenced object has been removed from memory), but with a larger memory footprint.

3.4 Sandboxed deserialization

While it is possible for the ABC interpreter to implement the ABC instructions to deal with file I/O, pointers, and the C foreign function interface, this has not been done so far. The reason behind this is that interpreted programs should in principle not have side effects, because deserialization is a pure function. Should a host program wish to offer interpreted programs access to such features, it can do so through an ad hoc interface.

However, interpreted programs can still crash, for example by running out of heap or stack space or dividing by zero. In some contexts it is required that expressions are interpreted in such a way that it cannot crash the host program. This is for example the case in plugin systems where the plugins would be interpreted. However, this turns out to be tedious in a lazy language that is furthermore strongly typed and must therefore reflect "exceptions" (like when interpretation failed) in types.

3.4.1 Safety and laziness. Where the type of our deserialization function was previously simply (String → a),[5] it would now have to be (String → MaybeDeserialized a), with MaybeDeserialized recognizing different kinds of exceptions:

```
:: MaybeDeserialized a
   = HeapFull | StackOverflow | IllegalInstruction // various exceptions
   | Ok a                                          // success
```

However, it cannot be decided after the first interpretation step (i.e., when a head normal form has been reached) which constructor must be created. If no fault was detected, we still cannot create an (Ok *val*), because *val* may contain indirections to the interpreter, which, when evaluated, may cause a fault.

For evaluations further down the evaluation tree we even cannot wrap the result in a type like MaybeDeserialized. Suppose we are deserializing a value of type List Int:

```
:: List a = Cons a (List a) | Nil
```

Because both arguments of the Cons constructor may trigger a fault, a naive deserializer would now generate values like (Ok (Cons (Ok 1) (Ok Nil))), which are ill-typed. Indeed, faults further down the evaluation tree *must* be reflected in a top-level wrapper type. This requires fully evaluating the expression to a normal form to decide which constructor must be used. We have nevertheless implemented this functionality; the programmer can now choose between using a lazy deserialization variant which may crash the host program, and a hyperstrict variant which catches faults. Our library therefore provides two functions: deserialize :: String → a and safe_deserialize :: String → MaybeDeserialized a.

With GHC's exceptions (which can be thrown in pure expressions and caught in the IO monad) it would not be necessary to reflect possible failure in the type, and it may be possible to combine safety and laziness.

3.4.2 Function types. There are still situations in which the normal form that is the result of the full evaluation of the serialized value can cause more interpretation steps in the future. This is the case

[5]In reality, the type is a little different, because the function needs access to the executable's symbol table to be able to translate descriptors. The function also has an extra argument for a settings record with which the programmer can for example set the heap size given to the interpreter. These additional arguments are orthogonal to the discussion here and are omitted for clarity.

when the normal form contains closures. If the host curries more arguments into these nodes, this may trigger a new interpretation step. In this context, we do not explicitly cause the interpretation step, so the programmer has no way to choose between deserialize and safe_deserialize.

The only sensible choice here is to assume that such evaluation steps must be performed in a safe context as well. Imagine a plugin system where plugins are records of a number of functions and metadata. The initial deserialization step only evaluates the metadata but cannot apply the functions yet, while actually these functions are what must be sandboxed. When the library creates an unsaturated indirection from the host to the interpreter as the result of a safe interpretation, this indirection is therefore also safe. This has the unfortunate consequence that the type of the interpreted value changes. If the serialized value was of type (Int, a → b), the deserialized value will now be of type (MaybeDeserialized (Int, a → MaybeDeserialized b)). Unfortunately, this means that programs using sandboxed deserialization may need unsafe type casts to handle these values later on.

3.4.3 Notes on the implementation. Some faults, like illegal or prohibited instructions, are caught quite easily as an additional instruction which exits the interpreter loop. To check for stack overflows, a segmentation fault handler is installed, and the C functions (sig)setjmp/(sig)longjmp to save and restore application state are used to jump back to the start of interpretation after a fault. The stack pointers are then reset to their original value, but the heap pointer remains untouched, because part of the original heap may now have references to nodes added in the last interpretation step.

Because an interpretation may trigger evaluation in the host (when the host has shared unevaluated function arguments), which may in turn trigger new interpretations, there can be a theoretically endless chain of interpretation steps. This happens, for example, when applying a lazy, interpreted variant of foldr to native arguments, where the interpreted foldr and the native operator argument become intertwined on the stack. It is important to distinguish the different interpretation steps when resetting the application state in case of a fault. Consider the following example:

```
x = deserialize "f" (g (deserialize "bad"))
```

Assume that (deserialize "f") yields a function in the interpreter, that g is a normal native function, and that (deserialize "bad") results in a fault in the interpreter. If g handles the fault (for example, by plugging in a default value in case a fault has occurred), x must evaluate to a "good" deserialized value. This must be distinguished from the case that x = deserialize "bad". Hence, the chain of interpretation steps must be reflected in the data structure that is used to restore the application state. We therefore use a stack of restore points so that each interpretation step can be recovered from.

However, we also cannot create a restore point for each individual interpretation step, but only for those which were requested to catch faults by the programmer (i.e., those that were created for safe_deserialize, and closures in the results of such safely deserialized values). If nodes are copied from the interpreter to the host as the argument of a (host) function, these must *not* be wrapped in the MaybeDeserialized type, and hence we must not create a restore point for these values. Should evaluation of these nodes fail, this will cause the interpretation that required application of that host function to fail instead, as expected. The purity of the host function that is called in combination with the strictness of the safe deserialization step guarantees that the unsafe indirection created this way cannot appear elsewhere after the overarching safe interpretation step has finished. In our implementation, we use one bit in each indirection from the host to the interpreter to signal whether lazy and seamless or strict and safe interpretation is to be used for a particular node.

3.5 Type-safe deserialization

Even with the sandbox environment provided by separate heap and stack spaces and the segmentation fault handler described in the previous subsection, it turns out to be relatively easy for *malicious* interpreted expressions to crash the host program. Serialized expressions contain no information about their type. The type of the deserialization function is simply (String → a) (or String → MaybeDeserialized a), so the context in which the deserialization function is called determines a, the type that the interpreted expression is assumed to have. Since this information is not available at runtime, it cannot be checked that the expected type matches that of the interpreted value when it is copied into the host heap. In other words, the interpreter may copy an integer into the host heap, and the host may interpret it as a pointer, etc.

Another issue is that descriptors are matched by looking at symbol names only. Therefore, if the host and the interpreter contain a constructor with the same name (and from the same module), these are matched regardless of whether their arity or the types of their arguments match. This is not only a security concern: the same situation can occur when the host and the interpreted program share source code from some common library, but have used different versions of that library between which types have been changed.

The latter issue can be solved relatively easily by storing more type information in descriptors. Currently, descriptors contain such information as their arity and name (to print them), and for nodes with unboxed values a type string to identify the types of the arguments. However, this type string only distinguishes the various basic values (integers, booleans, etc.) and node values, which are boxed; there is no distinction between nodes of different types. If information about the types of node arguments were added to descriptors, runtime checks could be added to verify type coherence.

This does not solve the first issue yet, since it would still be possible, for example, to deserialize an integer and use it as a pointer. To resolve this issue we can use Clean dynamic types on top of our serialization library. With dynamic types, almost any[6] Clean expression can be packed into an opaque type Dynamic, which internally consists of the expression and a representation of its type [1, 2, 23]. Dynamics can be unpacked using a custom pattern match against types, which internally uses a unification algorithm to match the expected type against the actual one.

We could thus write a deserialization function with type (String → MaybeDeserialized Dynamic). For the reasons outlined in section 3.4.1, it does not make sense to combine this approach with lazy, non-sandboxed deserialization. When the expression is in normal form, it must be checked that (1) the generated graph is well-typed

[6]Some values, such as files, cannot be packed into dynamics. However, these are typically also the kind of values for which (de)serialization does not make sense.

(i.e., conforms to the type specifications of the referenced descriptors), (2) the types of the used descriptors match the types of the descriptors in the host, and (3) the type of the root node is Dynamic. If this is the case, the value can be copied safely to the host.

Unfortunately, the types of closures will still change in this setup, in the same way as described in section 3.4.2. A way around this would be to not use MaybeDeserialized, but instead define a new type DeserializationException:

```
:: DeserializationException
   = HeapFull | StackOverflow | IllegalInstruction
```

The type of the deserialization function now becomes (String → Dynamic). The dynamic contains either a value of type DeserializationException, or a value of any other type which has been correctly deserialized. If the types of the closures are chosen well and all return a Dynamic, we no longer need to change these types, so that unsafe type casts can be avoided in the host program. This advantage outweighs the obvious drawback that in this setup there is no way to distinguish a faulty interpreted expression from an interpreted expression that correctly evaluates to a value of type DeserializationException.

4 INTERWORKING ON THE WEB

The previous section remarked at several stages how symmetrical the interworking between a native Clean program and our interpreter is: it is, essentially, a specific type of distributed system. Recognizing this, it becomes easy to see that we can also apply a similar technique in web applications, by plugging in a communication channel to transmit values back and forth. We use the WebAssembly version of the interpreter currently in *iTasks*, a task oriented programming framework for developing web applications [24, 25].

Commonly, web applications use at least two different programming languages: one on the server, and one on the client. This is because only few languages can be used in web browsers: most notably JavaScript and WebAssembly [14]. When the server is written in another language, the developers need to make sure that the client-server interface is correct: when one side changes, the other must as well. Even with standards like JSON this remains tedious.

In iTasks, this problem was solved by Sapl [10]: a minimal functional programming language that can be targeted by the Clean compiler and compiled to JavaScript. This allowed part of the iTasks application to be compiled to JavaScript and run in the browser. Communication was done using generic functions (targeting JSON, although this is an implementation detail). However, there were several issues with this approach. First, due to the way Clean was compiled to Sapl, not all Clean values could be sent to the browser: they had to be evaluated to a normal form first. Second, because Sapl branched off relatively early in the compiler toolchain, it was time-consuming to maintain. Third, the compiled JavaScript had a bad worst-case performance, in part due to tricks required to work around the low stack recursion limit in JavaScript engines. This led programmers to write two versions of some functions: one optimized for native Clean, and one to be run in JavaScript in the browser. This is however at odds with the overall goal of the system: to write the entire application in one source language.

To overcome these issues, the latest version of the iTasks framework now uses our new ABC interpreter in the browser. Because the ABC language is on a much lower level than Sapl, this simplified the compiler toolchain significantly. More importantly, the memory model in the interpreter is the same as that of native Clean, so it is now straightforward to copy any value from the server to the client (using the GraphCopy module described earlier). For the same reason, the performance penalty has become much more predictable and there is no need for different implementations of the same function any more.

4.1 Interworking with the iTasks server

Communication between the iTasks server and the interpreted Clean program running in the browser can be set up in several ways. The iTasks framework already provides a web socket with which messages can be passed efficiently without polling the server. It would hence be possible to apply the same method of interworking as in the deserialization setting here, with the difference that nodes would be copied over the network rather than in memory. However, this would lead to a lot of network traffic due to lazy evaluation of the result. Furthermore, we are usually not interested in the actual result of the function that is sent to the client. For instance, it may only be used to interface with JavaScript to set up part of the user interface, such as using a third-party JavaScript library to add a date picker to an input field.

For this reason, the iTasks framework currently only allows sending a limited set of functions to the client: functions of type (JSVal *JSWorld → *JSWorld).[7] Such a function is executed when an iTasks component is instantiated. The *JSWorld is an opaque type, realized internally as a boxed integer but inaccessible to the programmer. Because all functions in the JavaScript foreign function interface require a value of *JSWorld, and uniqueness typing ensures that this value cannot be duplicated, the programmer can enforce the execution order. The JSVal contains a reference to the underlying iTasks component, which is a JavaScript object. This value can be used to set up the initial state of the component, including installing Clean functions as callbacks for JavaScript events.

In some cases, initialisation of a form field is all that needs to be done. For instance, when a date field is created, we call a third-party JavaScript library to initialise a date picker, but the changes to the value of the input field are communicated with the server through the core iTasks framework. Other cases are more complex, and do require that we can send a message to the server from within the WebAssembly interpreter. For instance, in interactive SVG images [4], both a model and a view of the image is kept. When an event is handled, these can be changed. Some changes can be handled entirely locally, while in other cases the changes must be synchronised with the server. To this end, the iTasks component has a doEditEvent property: a JavaScript function which is used to send events to the server. It would be possible to set up lazy communication in this case: we could use the GraphCopy functionality to copy the graph from the WebAssembly interpreter heap to the server. However, this imposes a security risk, since the doEditEvent method can be called by the user of the web application as well,

[7] Here, the asterisk in *JSWorld indicates that the type is *unique*: the value may only be used once [6] (cf. *linear types* for Haskell [7]). This is the way file I/O and destructive updates are handled in Clean: through unique types like *File and unique arrays [26, §9]. Also, recall that Clean function types can have an arity larger than 1. The Clean type (a b → c) corresponds to the Haskell type (a → b → c).

who can then evaluate arbitrary code on the server. For this reason, we currently use JSON encoding to transmit edit events, and hence require the values to be fully evaluated.

Communication from the server to the client (besides the initial initialisation step) is also handled through common iTasks functionality. In this case, the server sends an *attribute change* to a specific iTasks component. The component can have a listener installed on attribute changes, which is called when this happens. In the initialisation function of the component, the programmer may thus set up a Clean function as an attribute change listener. Because attribute values are simply strings, the programmer is free to use any encoding. This includes serializing a possibly lazy Clean value on the server and sending the serialized value to the client. A dedicated function in the JavaScript interface, jsDeserializeGraph, can be used to deserialize this value into the WebAssembly heap.

4.2 Interworking with JavaScript and the DOM

While the new client runtime integrates much more neatly with the server, our approach also has some downsides compared to the previous system. One of the main goals of the integration on the browser is to make the application interactive. For this, we need access to the Document Object Model (DOM) of the webpage to create and remove elements or change their attributes.

In the Sapl setup, Clean programs could contain a dedicated set of functions which had a hard-coded implementation in JavaScript and thus allowed calling arbitrary JavaScript functions and handling DOM elements. This was interwoven with the JavaScript code generated for the actual Clean program. Hence, there was no concept of a foreign function interface between Clean and JavaScript; in the compiled code, these were identical.

Interaction with the DOM is currently impossible in WebAssembly, so we now need to call JavaScript functions from the WebAssembly interpreter to interact with JavaScript objects and the DOM. Therefore, the current system does include a foreign function interface, with a small number of traps: dedicated ABC instructions which bail out to JavaScript to evaluate JavaScript expressions. Because we do not know in general which functions can be called, this interface must be general: it is essentially a wrapper around JavaScript's `Function` constructor. Given a string, this constructor parses it as a JavaScript function body, and returns the corresponding function. Because this is a client-side operation and the string is controlled by the iTasks application, this does not introduce security issues. The result of the evaluation of this function is then passed back to Clean. The string that is sent to JavaScript is immediately freed from the Clean heap to reduce heap usage. However, the string must still be built and parsed, so the interface with JavaScript has become slower in the new setup.

4.3 Impact on garbage collection

Not all JavaScript values can be copied to Clean. When a JavaScript evaluation results in an object or function, it is stored in a common array on the JavaScript side and Clean receives a reference to that array instead. This way, a Clean program can still create JavaScript objects and call functions on them. It also turns out to be useful to be able to store an arbitrary Clean value on the JavaScript side. For instance, a Clean function that is set as a callback for a JavaScript event may need to access Clean values that are modified from elsewhere. These references are also stored in an array on the JavaScript side.

To clean the array of shared JavaScript values up, the Clean garbage collector keeps track of the references it finds to that array. After a garbage collection cycle, the array is cleaned up. This is analogous to cleaning shared host nodes in the deserialization setting.

The other direction is trickier, because we do not have access to JavaScript's garbage collector. Hence there is no way to determine whether a Clean value referenced from JavaScript must be kept in memory or not. There is a proposal under consideration for inclusion in the JavaScript standard (properly termed ECMAScript) to add finalizers to the language [29]. However, this new feature will be of little use here (or elsewhere), because finalizers will not be *guaranteed* to run after an object is garbage-collected.

We must therefore implement our own small garbage collector on the JavaScript side as well. We do this by explicitly linking every Clean value shared with JavaScript explicitly to an iTasks component. This component can be as small as a single text field or can be a larger collection of different views. As opposed to arbitrary JavaScript values and DOM nodes, iTasks components do have proper finalizers (provided by the iTasks framework). The components now keep track of the linked Clean values, and remove them from the shared array when they are destroyed.

To this basic scheme we add one optimisation, which is useful when components live so long that we cannot wait for them to be destroyed: when the Clean program sets a JavaScript variable to some value, it is first checked whether the variable contained a reference to Clean. If this is the case, the reference is freed directly. This is useful in situations where the same variable is continuously updated with new Clean values.

However, in some cases, even this is not enough. In a library used to create interactive SVG images [4], for every update of the SVG image new Clean functions could be attached to DOM nodes as event callbacks. Because these are not overwritten with a new value but simply disappear from the DOM, the previously described optimisation does not target this case. Currently, the only solution in this case is to keep track of the installed callbacks in the SVG library itself and remove them when the corresponding object has been removed.

A current proposal to add garbage collector integration to WebAssembly [27] may allow us to implement parts of the JavaScript interface in a neater way, but it is not yet advanced enough to determine whether it will be helpful here or not.

4.4 The JavaScript foreign function interface

To illustrate the above, this section describes the foreign function interface that can be used to access the JavaScript world from Clean. Other than the interworking described in section 3, this is a proper foreign function interface where copying of values has to be explicit. This is necessary because the execution models of JavaScript and Clean are too different.

A representative excerpt of the interface is given in Figure 3. The server-side iTasks application contains code using this foreign function interface, which is sent to the browser as described in

```
:: *JSWorld
:: JSVal
:: JSFun :== JSVal

(.?) infixl 1 !JSVal !*JSWorld → (!JSVal, !*JSWorld)

jsValToInt    :: !JSVal → Maybe Int // and similar for other types
jsIsUndefined :: !JSVal → Bool // check for JS value 'undefined'

class (.#) infixl 3 attr :: !JSVal !attr → JSVal
instance .# String, Int

generic gToJS a :: !a → JSVal
derive gToJS Int, Bool, String, JSVal // etc.

(.=) infixl 1 !JSVal !a !*JSWorld → *JSWorld | gToJS{|*|} a

class toJSArgs a :: !a → {!JSVal}
instance toJSArgs Int, Bool, String, JSVal, () // etc.
instance toJSArgs (a,b) | gToJS{|*|} a & gToJS{|*|} b // and for (a,b,c), etc.

(.$)  infixl 2 :: !JSFun !a !*JSWorld → (!JSVal, !*JSWorld) | toJSArgs a
(.$!) infixl 2 :: !JSFun !a !*JSWorld →          *JSWorld  | toJSArgs a

jsGlobal :: !String → JSVal
jsNew :: !String !a !*JSWorld → (!JSVal, !*JSWorld) | toJSArgs a
addJSFromURL :: !String !(Maybe JSFun) !*JSWorld → *JSWorld

jsMakeCleanReference :: a !JSVal !*JSWorld → (!JSVal,   !*JSWorld)
jsGetCleanReference  ::   !JSVal !*JSWorld → (!Maybe a, !*JSWorld)
jsFreeCleanReference ::   !JSVal !*JSWorld →            *JSWorld

jsWrapFun :: !({!JSVal} *JSWorld → *JSWorld) !JSVal !*JSWorld
    → (!JSFun, !*JSWorld)

jsDeserializeGraph :: !String !*JSWorld → (!a, !*JSWorld)
```

Figure 3: The JavaScript foreign function interface.

section 4.1. The programmer is responsible for making sure this interface is only used on the client-side, as it uses trap instructions which are not defined in the native Clean runtime system. This is trivial, because the JSWorld can only be obtained on the client-side since it is provided by the supporting JavaScript code.

As can be seen, JSVal is an opaque type. Internally, it is an algebraic data type representing different kinds of JavaScript expressions, including references to the arrays of shared JavaScript and Clean values. The JSFun synonym type is merely provided to improve readability in function types.

To evaluate a Clean expression, the .? operator can be used. Note that the use of *JSWorld here and elsewhere enforces evaluation order, as has been mentioned above. Because not all JavaScript values can be copied to Clean, the result of this function is still a JSVal. Other functions, like jsValToInt and jsIsUndefined, can be used to move the result into the Clean world.

The .# class is used to get properties of JavaScript objects. For instance, obj .# "key" corresponds to the JavaScript expression obj["key"], while obj .# 0 corresponds to the expression obj[0].

The generic function gToJS converts Clean values to JavaScript values. This is possible for basic types and records (for which JavaScript objects are created), but not for algebraic data types, which have no direct equivalent in JavaScript. The .= operator sets a certain JavaScript reference to some value using gToJS.

To call JavaScript functions, the .$ and .$! operators can be used, the difference being only that the first returns the result. The type of the arguments of the JavaScript function must instantiate the toJSArgs class. This class is essentially the same as the generic function gToJS, with the exception that it allows the void type () for function applications without arguments, and tuples of convertible types for functions with more than one argument.

To use global JavaScript objects (e.g., window or document), jsGlobal can be used. Similarly one can use jsNew to create new JavaScript objects with the new keyword. It is also possible to load external JavaScript files using addJSFromURL. The second argument to this function is an optional callback, which will be executed after the file has loaded. This is for example used in iTasks to load the external *Pikaday* library for date pickers [9], after which jsNew "Pikaday" (…) is used to initialize the input field.

The functions jsMakeCleanReference and jsGetCleanReference are used to store and retrieve references to Clean values in the JavaScript world. The second argument to jsMakeCleanReference must be an iTasks component. This component is used for garbage collection, as explained above: the shared Clean value is kept in memory as long as the iTasks component exists. Alternatively, one may use jsFreeCleanReference to free the value earlier.

To install Clean functions as callbacks for JavaScript events, jsWrapFun is used. This is nothing more than a wrapper around jsMakeCleanReference, so this function also has a JSVal argument which must be a reference to an iTasks component and is used for garbage collection. Once called, the function arguments are passed as a single array of JSVal values. This is needed because JavaScript functions can be called with any number of arguments.

Finally, jsDeserializeGraph is the WebAssembly version of the graph unpacking function in GraphCopy. It is provided so that the iTasks server can send lazy serialized expressions to the client.

5 BENCHMARKS

We have benchmarked the different implementations of the interpreter and compared its performance against that of native Clean and Sapl. The results are in Table 1, with time relative to native Clean plotted in Figure 4. The benchmarks come from a standard set of small programs,[8] for easy comparison with more minimal runtime systems, and some somewhat larger programs from [15][9] to more accurately estimate performance in real use cases. Additionally, we have benchmarked a real-world program, namely the Cloogle search engine [28].[10]

The runtimes benchmarked are: native Clean; the C interpreter with direct threaded code (compiled with gcc 6.3 and -Ofast -fno-unsafe-math-optimizations); the C interpreter with a giant switch

[8] The benchmarks are implemented as follows: nfib naively computes the 43rd Fibonacci number, adding 1 for each function call; tak computes $\tau(36, 24, 12)$, where τ is the Takeuchi function; lqueen solves the N-queens problem for a 13×13 chess board; reverse reverses a list of 30,000 elements 30,000 times; fsieve computes the 10,000th prime number using the sieve of Eratosthenes 200 times.

[9] complab is an image processing program; event simulates 1,500,000 transitions in a flip-flop; ida is an implementation of the Iterative Deepening A* algorithm; sched is a job scheduler; parstof is a parser; transform performs an AST transformation.

[10] The implementation of the benchmark can be found on https://gitlab.science.ru.nl/cloogle/benchmark.

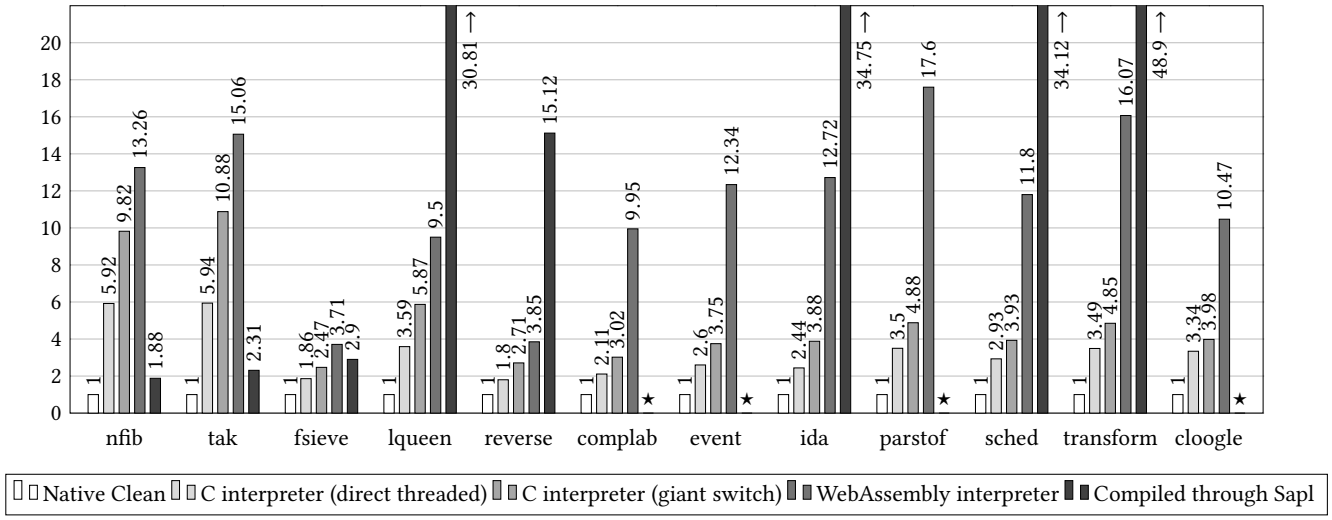


Figure 4: Performance of the ABC interpreter and Sapl, relative to native Clean (lower is better).

Benchmark	Native Clean	C (direct threaded)	C (giant switch)	WebAssembly	Sapl
nfib 43	2.38	14.08	23.36	31.56	4.47
tak 36 24 12	5.77	34.26	62.76	86.87	13.32
lqueen 13	2.26	8.11	13.26	21.47	69.62
reverse 30,000	4.05	7.28	10.96	15.60	61.23
200× fsieve 10,000	1.53	2.85	3.78	5.68	4.43
30× complab	2.44	5.15	7.38	24.27	★
10× event	1.62	4.21	6.08	19.99	★
10× ida	3.13	7.64	12.16	39.81	108.8
1,000× parstof	1.80	6.30	8.79	31.68	★
sched	1.08	3.16	4.24	12.74	36.85
3,000× transform	0.86	3.00	4.17	13.82	42.05
cloogle	181.5	605.4	722.9	1900	★

Table 1: Wall-clock time in seconds for native Clean, the ABC interpreter, and Sapl.

(idem); the WebAssembly interpreter (in SpiderMonkey 68.0); Sapl (idem). The benchmarks were run on an i7-5500U processor at 2.4GHz. Native Clean and the ABC interpreter received 200MB of heap for each program. Because Sapl uses JavaScript expressions to emulate the Clean graph, we cannot easily limit the heap size in this setup. A number of benchmarks failed to run with Sapl, for a variety of reasons (stack depth limits; run-time exceptions; lack of support for unique array selections). The lqueen benchmark was adapted to solve a smaller problem (on a 12×12 chess board); the results were then scaled accordingly. The other failing benchmarks are indicated with a star: "★". Lastly, the reader should keep in mind that the native Clean compiler generates highly performant code.

As can be expected, programs that rely heavily on the heap (such as reverse) have a much smaller performance penalty than those that use more basic values on the stack (such as nfib and tak). This is because the former programs spend more time in garbage collection, which is nearly as fast as in native Clean, and because the latter are overall simpler programs, with smaller instructions and hence a larger dispatch overhead in the interpreter.

On most practical benchmarks for which Sapl does not crash, the WebAssembly interpreter is faster by a factor of around 3. Only simple programs that deal with a lot of basic values (like nfib, tak and fsieve) clearly benefit from the fact that the Sapl code is just-in-time compiled instead of interpreted. However, because JavaScript has a small recursion limit, much too small to implement a functional programming language, Sapl uses continuation passing style (i.e., "trampolined code"), causing an overhead even on these small programs. Sapl's performance costs become much clearer for larger programs which use heap structures. To achieve lazy evaluation, Sapl represents thunks and head normal forms as JavaScript arrays, which are evaluated using a central "trampoline" function. How efficient that is, is difficult to predict, as is reflected in the wide range of performance penalties compared to native Clean, between 15 times slower on the reverse benchmark and 49 times slower on transform. Because the ABC interpreter remains much closer to

the native Clean runtime environment, its overhead is much more predictable (taking into account the fact that it is faster when a program requires more garbage collection, as mentioned above).

The difference between the native interpreter (with a giant switch) and the WebAssembly interpreter is higher than expected, since WebAssembly is expected to have near-native performance [14, p. 197]. Looking at different JavaScript engines does not help much: Google's V8 engine (version 7.3), for instance, is about 30% slower on our interpreter than Mozilla's SpiderMonkey (68.0). Upon inspection of the code generated by the just-in-time compiler, this turns out to be a register allocator issue, due to which SpiderMonkey's register allocator performs particularly bad on large interpreter loops like ours.[11] The Clean program counter and heap and stack pointers are therefore continuously stored and reloaded from the WebAssembly stack. To work around this issue, we currently use global variables instead of locals to keep this state of the interpreter, which causes the variables to be stored in memory rather than on the stack and thus reduces the number of spills needed. This is about 40% faster on our benchmarks (included in the results presented above). However, a proper solution would be to store the pointers in registers without spilling them to the stack. The difference also *varies* more than one would expect — at the moment it is unclear why. Unfortunately, it is difficult to investigate this issue without first fixing the aforementioned problem with the register allocator.

Using Emscripten [32] it would have been possible to compile the C version of the ABC interpreter to WebAssembly, instead of creating a separate WebAssembly version. However, initial tests showed that this approach would be rather slow, this WebAssembly interpreter being around seven times slower than the C version, i.e., between 10 and 70 times slower than native Clean. Similar, though not quite as bad, results were obtained in other projects [30]. Generating the WebAssembly interpreter ourselves furthermore keeps our toolchain lightweight and gives more freedom to improve the generated WebAssembly with domain knowledge.

We do not provide benchmarks for the node copying overhead (in the deserialization setting) or the JavaScript interface (in the browser context) here, because we have not yet encountered realistic programs that spend a significant amount of time on these steps. The serialization library has been tested with the *Soccer-Fun* project [3] in which a soccer match is simulated. In our setup, the AIs of the players were interpreted and the gameplay was handled natively. Even though there is a lot of communication to make AIs aware of the state of the game, the copying of nodes back and forth was negligible in comparison to the time spent to calculate the next move, even with simple AIs. In the browser setting, we have tested the JavaScript interface with the interactive SVG editors described in [4]. Here too, only little time is spent in the interface, and most time is spent in the browser's rendering engine and the actual computations done inside the interpreter.

6 RELATED WORK

The ABC machine which our interpreter simulates was originally described by Koopman [19]. Since then, many additions and improvements have been made, most of which are not documented in the literature. Also a comprehensive description of the current native runtime system used by Clean remains a desideratum, although earlier versions have been described [31].

As mentioned above, our serialization library is a wrapper around the GraphCopy library, which was briefly described by Oortgiese et al. [20]. It is similar to solutions in other languages (see e.g. [8] for Haskell); Epstein et al. [12] use a different, class-based approach. While these solutions facilitate sharing of lazy expressions between executables containing *the same code* on *different platforms*, there have also been projects to allow the sharing of *unknown code* on *the same platform*. An example of this is Clean's dynamics system, which on 32-bit Windows systems can write and read values of type Dynamic to and from the disk [22]. The relevant object code is stored centrally and linked dynamically by what we called here the host program. Pil [22, p. 244] already anticipated the goal of sharing these dynamic values between different platforms, but noted: "We would like to stress here that we are only interested in efficient solutions. [...] The representation of a function by its source code, which can be interpreted at run-time, is not a satisfactory solution." Clearly, our knowledge of interpreters as well as processor speed has improved since 1997, and at least for some applications, interpretation has now become a feasible strategy — especially because in some contexts, like the web, remain unsuitable targets for the compilation of functional languages.

Du Bois and Da Rocha Costa [11] experimented with a small functional language that compiles to the JVM and uses Java remote method invocation to parallelise execution over several virtual machines. In this setup, as in ours, nodes in the distributed system can execute code it has not seen before. However, this comes at the cost of *all* code being interpreted or just-in-time compiled; there is no interworking of compiled and interpreted code.

In Erlang, compiled and interpreted code do work together. Code can at runtime be replaced by newer versions to fix bugs [5, pp. 78–80]. While originally Erlang was run in a virtual machine, it can now be just-in-time compiled as well [21]. In this setup, the compiled and the interpreted code work together on the same heap and stack. This clearly avoids the node copying overhead of our serialization library, but is less tailored to sandboxed execution.

In the *iTasks* framework [24, 25], our work has replaced that of Domoszlai et al. [10] as a method to bring Clean code to the browser and have a native Clean server communicate with the simulated Clean program on the client. In the previous setup, Clean was compiled to the minimal functional programming language Sapl [17], which was then compiled to JavaScript. The previous setup was faster on small benchmarks because the overhead of interpretation weighs heavier in this case. For larger programs, interpretation on a lower level (i.e., ABC instead of Clean or Sapl) turns out to be preferable. This also simplified communication between the server and the client. Due to limitations of WebAssembly, the interface with JavaScript has become slower than in the previous setting, but this difference has not been recognizable in actual usage.

7 CONCLUSIONS

We began our paper with the statement that functional languages are theoretically ideally suited for the execution of code unknown at compile-time: because functions are first-class citizens, this is just

[11] https://bugzilla.mozilla.org/show_bug.cgi?id=1167445.

a special case of deserialization. We then described the implementation of a serialization library which supports this feature, to the best of our knowledge the first to do so in a platform-independent way. Internally, our library relies on an interpreter for Clean's intermediate language ABC, and copies nodes back and forth between the native and the interpreter heaps. This setup works well, and in the practical applications tested the overhead of copying nodes was not felt to be a major drawback.

We have described the impact on garbage collection and highlighted parallels with distributed systems that can aid implementers of similar systems. Since we rely on only one special feature of Clean's native runtime system, our implementation can be an example for similar libraries in other functional programming languages.

Unfortunately we have not been able to reconcile our two aims of getting the interpreter to interwork lazily and seamlessly with the host on the one hand, and evaluate serialized expressions safely on the other. Exception propagation, as supported by the GHC runtime system for Haskell, may be able to resolve this issue.

As outlined above, our sandboxed interpretation scheme is currently not entirely safe yet. We have described methods that can be used to ensure type coherence. In conjunction with dynamic typing, our serialization library would then become type-safe.

Lastly, we have shown that an interpreter for the stack-based intermediate language ABC is a suitable way to bring functional programming to the browser. Browser languages currently are a sub-ideal compilation target for functional languages due to the lack of recursion depth and non-local goto's. We have seen that the performance of our interpreter is more predictable and in most cases better than other solutions which work around this issue using continuation-passing style. Nevertheless, we intend to investigate whether a synthesis of these approaches is useful to further improve performance, for example by only using the interpreter in parts of the program where deep recursion or non-local goto's are required.

8 ACKNOWLEDGEMENTS

We thank Erin van der Veen for his previous work on the interpreter. We are also grateful to the IFL reviewers for their thoughtful comments.

REFERENCES

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems* 13, 2 (1991), 237–268.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. 1995. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming* 5, 1 (1995), 92–103.

[3] Peter Achten. 2011. The Soccer-Fun project. *Journal of Functional Programming* 21 (2011), 1–19. Issue 1.

[4] Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics. In *Proceedings of the 26nd International Symposium on Implementation and Application of Functional Languages, IFL '14*. ACM, New York.

[5] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph.D. Dissertation. The Royal Institute of Technology, Stockholm.

[6] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6 (1996), 579–612. Issue 6.

[7] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. 2 (2018). Issue POPL.

[8] Jost Berthold. 2011. Orthogonal Serialisation for Haskell. In *Implementation and Application of Functional Languages. 22nd International Symposium, IFL 2010 (Lecture Notes in Computer Science)*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer, Heidelberg, 38–53.

[9] David Bushell and Ramiro Rikkert. [n.d.]. Pikaday/Pikaday: A refreshing JavaScript Datepicker. https://github.com/Pikaday/Pikaday. Retrieved June 15th, 2019.

[10] László Domoszlai, Eddy Bruël, and Jan Martin Jansen. 2011. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae* 3 (2011), 76–98. Issue 1.

[11] André Rauber Du Bois and Antônio Carlos Da Rocha Costa. 2001. Distributed Execution of Functional Programs Using the JVM. In *Computer Aided Systems Theory — EUROCAST 2001 (Lecture Notes in Computer Science)*, Roberto Moreno-Díaz, Bruno Buchberger, and José-Luis Freire (Eds.). Springer, Heidelberg.

[12] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Haskell '11: Proceedings of the 4th ACM symposium on Haskell*. ACM, New York, 118–129.

[13] M. Anton Ertl and David Gregg. 2003. The Structure and Performance of *Efficient* Interpreters. *Journal of Instruction-Level Parallelism* 3 (2003), 1–25.

[14] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2017*. ACM, New York, 185–200.

[15] Pieter H. Hartel and Koen G. Langendoen. 1993. Benchmarking implementations of lazy functional languages. In *FPCA '93 Proceedings of the conference on Functional programming languages and computer architecture*. ACM, New York, 341–349.

[16] Tweag I/O. [n.d.]. WebAssembly as a Haskell compilation target - Asterius. https://tweag.github.io/asterius/webassembly/. Retrieved June 15th, 2019.

[17] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. 2006. Efficient interpretation by transforming data types and patterns to functions. In *Trends in Functional Programming*, Henrik Nilsson (Ed.), Vol. 7. 73–90.

[18] Marco Kesseler. 1996. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* Ph.D. Dissertation. Radboud University Nijmegen.

[19] Pieter Koopman. 1990. *Functional programs as executable specifications.* Ph.D. Dissertation. Katholieke Universiteit Nijmegen.

[20] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages*. ACM, New York.

[21] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. 2002. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Functional and Logic Programming. FLOPS 2002 (Lecture Notes in Computer Science)*, Zhenjiang Hu and Mario Rodríguez-Artalejo (Eds.). Springer, Heidelberg, 228–244.

[22] Marco Pil. 1997. First Class File I/O. In *Implementation of Functional Languages. 8th International Workshop, IFL '96 (Lecture Notes in Computer Science)*, Werner Kluge (Ed.). Springer, Heidelberg, 233–246.

[23] Marco Pil. 1998. Dynamic Types and Type Dependent Functions. In *Implementation of Functional Languages. 10th International Workshop, IFL '98 (Lecture Notes in Computer Science)*, Kevin Hammond, Tony Davie, and Chris Clack (Eds.). Springer, Heidelberg, 169–185.

[24] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*, N. Ramsey (Ed.). ACM, New York, 141–152.

[25] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. PPDP '12*. ACM, New York, 195–206.

[26] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2011. Clean 2.2 Language Report. http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf

[27] Andreas Rossberg (champion). [n.d.]. GC Proposal for WebAssembly. https://github.com/WebAssembly/gc/. Retrieved June 15th, 2019.

[28] Camil Staps and Mart Lubbers. 2016–2019. Cloogle: a search engine for the Clean programming language. https://cloogle.org. Retrieved June 15th, 2019.

[29] Dean Tribble, Mark Miller, and Till Schneidereit (champions). [n.d.]. WeakReferences TC39 proposal. https://github.com/tc39/proposal-weakrefs. Retrieved June 15th, 2019.

[30] Noah van Es, Quentin Stievenart, Jens Nicolay, Theo D'Hondt, and Coen De Roover. 2017. Implementing a performant scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures* 49 (2017), 62–81.

[31] John van Groningen. 1990. *Implementing the ABC-machine on MC680x0 based architectures.* Master's thesis. University of Nijmegen.

[32] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM Press, Portland, Oregon, USA, 301–312.

Attribute Grammars Fly First-class... Safer!

Dealing with DSL errors in type-level programming

Juan García Garland
jpgarcia@fing.edu.uy
Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Alberto Pardo
pardo@fing.edu.uy
Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Marcos Viera
mviera@fing.edu.uy
Instituto de Computación
Universidad de la República
Montevideo, Uruguay

ABSTRACT

AspectAG is a domain specific language embedded in Haskell to represent modular Attribute Grammars. In AspectAG attribute grammar fragments can be defined independently (even in separate modules) and then combined in a safe way. This flexibility is achieved through the use of extensible records, which are implemented using type-level programming techniques.

Type-level programming support has remarkably evolved in Haskell since the first version of AspectAG was designed; having incorporated new features like data promotion and kind polymorphism, among others, which allows to program in a "strongly typed" way at the level of types in a similar way as when programming at the level of values.

In this paper we redefine AspectAG applying the new type-level programming techniques. As a consequence, we obtain a more robust system with better error messages.

KEYWORDS

Attribute Grammars, EDSL, Type Level Programming, Haskell

ACM Reference Format:
Juan García Garland, Alberto Pardo, and Marcos Viera. 2020. Attribute Grammars Fly First-class... Safer!: Dealing with DSL errors in type-level programming. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Attribute Grammars (AGs) were originally introduced to describe semantics for context free languages [10]. Given a context-free grammar, attributes are associated to each of its productions. Attribute values are computed in every node of the abstract syntax tree, according to semantic rules that are expressed in terms of the attribute values of the children and the parent. Attributes are classified in at least two sets: synthesized attributes (where information flows bottom up) and inherited attibutes (where it flows top down). AGs have not only proven useful to implement programming language semantics, but as a general purpose programming paradigm.

`AspectAG` is a Haskell EDSL (Embedded Domain Specific Language), introduced by Viera *et al* [27], that implements first class AGs. It uses extensible polymorphic records and predicates encoded using old fashioned type level programming features, such as Multi Parameter Typeclasses [14] and Functional Dependencies [8], to ensure well-formedness of AGs at compile time.

An important drawback of EDSLs in general, and of `AspectAG` in particular, is that they are simply embedded libraries and when type errors occur they usually do not deliver error messages that refer to domain terms, leaking in addition implementation details in those messages. This breaks all abstraction mechanisms that may have taken into account in building the library. The problem is even worse if we use type-level programming techniques to implement the DSL. In the specific case of the original `AspectAG` library, the type-level programming techniques that were used were really ad-hoc, exploiting extensions originally introduced for other uses. In particular, at type level, programming was essentialy untyped.

More recent versions of GHC provide extensions to the type system to support a more robust and trustworthy programming at the type level. Notably `TypeFamilies` [3, 23], to define functions at type level, `DataKinds` [28], implementing data promotion, `PolyKinds` providing kind polymorphism, `KindSignatures` to document and deambiguate kinds, or `TypeApplications` [5] to provide visible application at type level. By using such extensions, we propose a reworked version of `AspectAG`[1] that tackles the previously mentioned drawbacks. These type-level programming techniques allowed us to do that in a strongly typed fashion at type level (we say, strongly kinded). We also define a framework to manipulate type errors, keeping track of the context of the possible sources of errors, to show precise (DSL oriented) messages when they occur.

The structure of the rest of the paper is as follows: In Section 2 we present the EDSL using an example, including a set of example error cases with their corresponding messages. In Section 3 we make a summary of the techniques we used, introducing polymorphic extensible records and proposing a methodology to manage type errors. In Section 4 we present some implementation details. Finally, we discuss some related work and conclude.

2 OVERVIEW OF THE LIBRARY

In this section we explain the main features of the library by means of an example. We start with a simple expression language formed

[1] http://hackage.haskell.org/package/AspectAG

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
type Nt_Expr = 'NT "Expr"
expr = Label @ Nt_Expr
type P_Add = 'Prd "Add" Nt_Expr
add = Label @ P_Add
type P_Val = 'Prd "Val" Nt_Expr
val = Label @ P_Val
type P_Var = 'Prd "Var" Nt_Expr
var = Label @ P_Var
leftAdd  = Label @ ('Chi "leftAdd"  P_Add ('Left Nt_Expr))
rightAdd = Label @ ('Chi "rightAdd" P_Add ('Left Nt_Expr))
ival     = Label @ ('Chi "ival"     P_Val ('Right ('T Int)))
vname    = Label @ ('Chi "vname"    P_Var ('Right ('T String)))
```

Figure 1: Grammar declaration

by integer values, variables and addition, which we then extend syntactically and semantically in order to show the different operations supported by the library.

Grammar declaration. The abstract syntax of the expression language is given by the following grammar:

$$expr \longrightarrow ival$$
$$expr \longrightarrow vname$$
$$expr \longrightarrow expr_l + expr_r$$

where *ival* and *vname* are terminals corresponding to integer values and variable names (given by strings), respectively. Both are said to be *children* in their productions. The third production has two children of type *expr*, which we name with an index to refer them unambiguously.

In our EDSL this grammar is declared as shown in Figure 1. First, we declare the non-terminal *expr* for our language. Non-terminals are types. They are built by a promoted algebraic datatype constructor *'NT* (we say, an algebraic datakind) applied to a name (`"Expr"`). Names are types of kind *Symbol*, the kind of promoted string symbols in modern Haskell. At term level we define an expression using the *Label* constructor. *Label* is a proxy type, with an appropiate name according to our domain instead of the usual *Proxy*. We use the `TypeApplications` extension of GHC to associate the type information to the label. The "@" symbol above is a visible type application.

Productions *add*, *val* and *var*, are also identified by a name, and are related to a non-terminal, once again using algebraic datakinds.

The last ingredient of the grammar declaration is given by the introduction of the children that occur in the productions (*leftAdd*, *rightAdd*, *ival* and *vname*). Each child has a name, is tied to a production and can be either a non-terminal or a terminal, in the latter case we include the type of values it denotes (e.g. *('T Int)*).

Summing up the information just provided, we can see that our grammar declaration indirectly contains all the ingredients that take part in the datatype representation of its ASTs:

```
data Expr = Val Int | Var String | Add Expr Expr
```

plus a set of names used to name explicitly each component. An important thing to notice is that the grammar representation is independent of this datatype.

We provide Template Haskell [19] functions that can be used to generate all the *boilerplate* defined in Figure 1, as follows:

```
$ (addNont "Expr")
$ (addProd "Val" 'NT_Expr [("val", Ter 'Int)])
$ (addProd "Add" 'NT_Expr [("leftAdd",  NonTer 'NT_Expr),
                           ("rightAdd", NonTer 'NT_Expr)])
```

Semantics definition. With the aim to provide semantics, AGs decorate the productions of context-free grammars with *attribute* computations. In an expression language as the one defined, the usual semantics are given by the evaluation semantics. This can be defined by using two attributes: *eval*, to represent the result of the evaluation, and *env*, to distribute the environment containing the values for variables. In the rest of this subsection we explain how the evaluation semantics can be implemented using our library.

The complete definition is shown in Figure 2. In lines 1 and 2 we declare the attributes, specifying their types.

The attribute *eval* denotes the value of an expression. Attributes like this, where the information computed flows from the children to their parent productions, are called *synthesized attributes*.

On the *add* production (Line 4) we compute *eval* as the sum of the denotation of subexpressions. On each subexpression there is a proper attribute *eval* that contains its value. Attribute *eval* is defined using function *syndefM*, which is the library operation to define synthesized attributes. It takes an attribute (the one for which the semantics is being defined), a production (where it is being defined), and the respective computation rule for the attribute.

Using an applicative interface, we take the values of *eval* at children *leftAdd* and *rightAdd*, and combine them with the operator (+). By means of the operation *at leftAdd eval* we pick up the attribute *eval* from the collection of synthesized attributes of the child *leftAdd*. We refer to these collections of attributes as *attributions*.

At the *val* production the value of the terminal corresponds to the semantics of the expression. In terms of our implementation (Line 5) the attribute *eval* is defined as the value of the terminal *ival*. *ter* is simply a reserved keyword in our EDSL.

Finally, on the *var* production (Line 6), the expression denotes the value of the variable on the given environment. We lookup up the variable, with name given by the terminal *vname*, in the environment provided by the attribute *env*. The name *lhs* indicates that we receive the *env* attribute from the parent. Attributes like *env*, that flow in a top-down way, are called *inherited attributes*. The use of *fromJust* is of course unsafe. We assume that the environment has an entry for each variable used in the expression evaluated. Error handling can be treated orthogonally with new attributes.

We combine all these rules on an *aspect* in Line 9. The operator (◃) is a combinator that adds a rule to an aspect (it associates to the right). An aspect is a collection of rules. Here we build an aspect with all the rules for a given attribute, but the user can combine them in the way she wants (for example, by production). Aspects can be orthogonal among them, or not.

Aspects keep all the information of which attributes are computed, and which children are used, taken from the rules used to build them. All the structure of the grammar is known at compile time, which is used to compute precise errors.

```
1  eval = Label @ ('Att "eval" Int)
2  env  = Label @ ('Att "env" (Map String Int))
3
4  add_eval = syndefM eval add $ (+) <$> at leftAdd eval ⊛ at rightAdd eval
5  val_eval = syndefM eval val $ ter ival
6  var_eval = syndefM eval var $ slookup <$> ter vname ⊛ at lhs env
7    where slookup nm = fromJust ∘ lookup nm
8
9  asp_eval = traceAspect (Proxy @ ('Text "eval")) $ add_eval ◃ val_eval ◃ var_eval ◃ emptyAspect
10
11 add_leftAdd_env  = inhdefM env add leftAdd  $ at lhs env
12 add_rightAdd_env = inhdefM env add rightAdd $ at lhs env
13
14 asp_env = traceAspect (Proxy @ ('Text "env")) $ add_leftAdd_env ◃ add_rightAdd_env ◃ emptyAspect
15
16 asp = asp_eval ⋈ asp_env
17
18 evalExpr e m = sem_Expr asp e rootAtt #. eval
19               where rootAtt = env =. m .*. emptyAtt
```

Figure 2: Evaluation Semantics

The function *traceAspect* and also -implicitly- each application of *syndefM* tag definitions to show more information on type errors. This is useful to have a hint where the error was actually introduced. For instance, note that that asp_{eval} clearly depends on an attribute *env* with no rules attached to it at this point, so it is not -yet- useful at all. We cannot decide locally that the definition is wrong since the rules for *env* could be defined later (as we will do), or perhaps in another module! If we actually use asp_{eval} calling it on a semantic function there will be a type error but it will be raised on the semantic function application. Showing the trace is helpful in those scenarios as we will see in Section 2.3.

For the definition of the inherited attribute *env* we use the *inhdefM* combinator, which takes an attribute name, a production (where the rule is being defined), and a child to which the information is being distributed. In our example, *env* is copied to both children on the *add* production, so we build one rule for each child (lines 11 and 12), and combine them on an aspect in Line 14.

We can combine aspects with the (⋈) operator. In Line 16 we combine asp_{eval} and asp_{env}, to get the aspect with all the attributes needed for the evaluation semantics. Note that this time we decided not to add a new tag.

Finally, given an implementation of the abstract syntax tree, like the *Expr* datatype, we can encode a generic *semantic function*:

```
sem_Expr asp (Add l r) = knitAspect add asp
    $ leftAdd  .=. sem_Expr asp l
    .*. rightAdd .=. sem_Expr asp r .*. EmptyRec
sem_Expr asp (Val i)   = knitAspect val asp
    $ ival     .=. sem_Lit i        .*. EmptyRec
sem_Expr asp (Var v)   = knitAspect var asp
    $ vname .=. sem_Lit v           .*. EmptyRec
```

Again, this code can be derived automatically using Template Haskell:

```
$ (mkSemFunc ''NT_Expr)
```

The semantic function sem_{Expr} takes the aspect, the AST of an expression, and an initial attribution (with the inherited attributes of the root) and computes the synthesized attibutes.

In the particular case of the evaluation semantics of our example we can define an evaluator as the one in Line 18. using the aspect (*asp*), the AST of an expression (*e*) and an initial attribution (*rootAtt*). It finally returns the value of the expression by performing a lookup operation (#.) of the attribute *eval* in the resulting attribution.

2.1 Semantic Extension: Adding and Modifying attributes

Our approach allows the users to define alternative semantics or extending already defined ones in a simple and modular way. For instance suppose that we want to collect the integral literals occurring in an expression. We define an attribute *lits*:

```
lits = Label @ ('Att "lits" [Int])
```

and the rules to compute it. This time we combine them on the fly:

```
asp_lits = syndefM lits add ((+)  <$> at leftAdd lits
                                ⊛  at rightAdd lits)
      ◃ syndefM lits val  ((:[]) <$> ter ival)
      ◃ syndefM lits var  (pure [])
      ◃ emptyAspect


litsExpr e = sem_Expr asp_lits e emptyAtt #. lits
```

The function *litsExpr* returns a list with the literals occurring in an expression from left to right.

It is also possible to modify a semantics in a modular way. For instance, to get the literals in the reverse order we extend the original aspect asp_{lits} with a rule that redefines the computation of *lits* for the production *add* in this way.

$$asp_{litsRev} = synmodM\ lits\ add\ ((+) <\$> at\ rightAdd\ lits \circledast at\ leftAdd\ lits) \triangleleft asp_{lits}$$

Notice that in this case we used *synmodM* instead of *syndefM*. The *mod* variants of the combinators *syndefM* and *inhdefM* modify an existing attribute instead of defining a new one, overriding the previous semantic function definition.

2.2 Grammar Extension: Adding Productions

Now let us expand the grammar with a new production to bind local definitions:

$$expr \rightarrow \textbf{let}\ vname = expr_d\ \textbf{in}\ expr_i$$

We implement it with these definition:

$$\textbf{type}\ P_{Let} = `Prd\ \texttt{"Let"}\ Nt_{Expr}$$
$$elet = Label\ @\ P_{Let}$$
$$exprLet = Label\ @\ (`Chi\ \texttt{"exprLet"}\ P_{Let}\ (`Left\ Nt_{Expr}))$$
$$bodyLet = Label\ @\ (`Chi\ \texttt{"bodyLet"}\ P_{Let}\ (`Left\ Nt_{Expr}))$$
$$vlet = Label\ @\ (`Chi\ \texttt{"vlet"}\ P_{Let}\ (`Right\ (`T\ String)))$$

We extend the aspects we had with the definition of the attributes for the new production:

$$aspEval2 = traceAspect\ (Proxy\ @\ (`Text\ \texttt{"eval2"})) \$\ syndefM\ eval\ elet\ (at\ bodyLet\ eval) \triangleleft asp_{eval}$$
$$aspEnv2 = traceAspect\ (Proxy\ @\ (`Text\ \texttt{"env2"})) \$\ inhdefM\ env\ elet\ exprLet\ (at\ lhs\ env) \triangleleft inhdefM\ env\ elet\ bodyLet\ (insert <\$> ter\ vlet \circledast at\ exprLet\ eval \circledast at\ lhs\ env) \triangleleft asp_{env}$$

and again combine them:

$$asp2 = aspEval2 \bowtie aspEnv2$$

Since we are not tied to any particular datatype implementation, we can now define the semantic functions for another datatype definition that includes the new production.

2.3 Error Messages

In a EDSL implemented using type-level programming type error messages are hard to understand and they often leak implementation details. Our library is designed to provide good, DSL-oriented error messages for the mistakes an AG programmer may make. We identify four categories of static errors. In this section we list them and show examples of each one.

2.3.1 Type errors in attribute expressions. When defining an attribute we can have type errors in the expressions that define the computation. For example if in Line 4 of Figure 2 we use an attribute with a different type than the one expected (*env* instead of *eval*):

$$add_{eval} = syndefM\ eval\ add\ \$\ (+) <\$> at\ leftAdd\ eval \circledast at\ rightAdd\ env$$

we obtain a *GHC* type error, pointing at this position in the code, with the message:

```
Couldn't match type 'Map String Int' with 'Int'
```

Similar messages are obtained if the expression has other internal type errors, like writing *pure* (2 + *False*). This is acomplished "for free" since `AspectAG` is embedded in Haskell, and this kind of error was well reported in old versions of the library.

2.3.2 Defining a computation that returns a value of a different type than the type of the attribute. For example, if in Figure 2 instead of Line 6 we have the following declaration that uses *lookup* instead of *slookup*:

$$var_{eval} = syndefM\ eval\ var \$\ lookup <\$> ter\ vname \circledast at\ lhs\ env$$

we get the error:

```
Error: Int /= Maybe Int
trace: syndef( Attribute eval:Int
             , Non-Terminal Expr::Production Var)
```

expressing that we provided a *Maybe Int* where an *Int* was expected. There is also some `trace` information, showing the context where the error appears. In this case it is not really necessary since the source of the error and the place where it is detected are the same. But we will see later in this section some examples where this information will guide us to the possible source of an error that is produced in a later stage. This kind of error looks similar to the previous one from the user's perspective, but it is very different to us. It requires implementation work. In previous versions of `AspectAG` this kind of error could not be detected, since attributes had no information about their type. Probably the program would fail anyway, possibly with an error like the one in 2.3.1 somewhere else. Then the programmer should track where the error was actually introduced.

2.3.3 References to lacking fields. This kind of errors are related to the well-formedness of the AG, like trying to access to a child that does not belong to the production where we are defining the attribute, trying to lookup attributes that are not defined, etc.

For example, if we modify Line 4 with the following code:

$$add_{eval} = syndefM\ eval\ add\ \$\ ter\ ival$$

where we use the child *ival* that does not belong to the production *add*. In this case the error points to *ter ival*, and says:

```
Error: Non-Terminal Expr::Production Val
       /=
       Non-Terminal Expr::Production Add
trace: syndef( Attribute eval:Int
             , Non-Terminal Expr::Production Add)
```

expressing that the production of type *Val* (of the non-terminal *Expr*) is not equal to the expected production of type *Add* (of the non-terminal *Expr*).

Another example similar to the previous one, is to treat a non-terminal as a terminal, or the other way around. Then, if we use *ter* to get a value out of the child *leftAdd*:

$$add_{eval} = syndefM\ eval\ add\ \$\ ter\ leftAdd$$

We obtain a message indicating that the child (belonging to the production *Add* of the non-terminal *Expr*) is a non-terminal (*Expr*), and not a terminal of type *Int* as it was expected:

```
Error: Non-Terminal Expr::Production Add
       ::Child leftAdd:Non-Terminal Expr
       /=
```

```
       Non-Terminal Expr::Production Add
       ::Child leftAdd:Terminal Int
trace: syndef( Attribute eval:Int
             , Non-Terminal Expr::Production Add)
```

Now suppose we have an attribute *foo*, of type *Int*, but without any rules defining its computation, and we use it in the definition of the rule add_{eval}:

$$add_{eval} = syndefM\ eval\ add\ \$\ (+) <\$> at\ leftAdd\ eval \circledast at\ rightAdd\ foo$$

At this point this is not an error, because this rule can be combined into an aspect where this attribute is defined. However, it becomes an error at Line 18, if we finally try to evaluate the incomplete aspect *asp*.

```
Error: Field not Found on Attribution
       looking up Attribute foo:Int
trace: syndef( Attribute eval:Int
             , Non-Terminal Expr::Production Add)
       aspect eval
```

Notice that in this case the trace guides us to the place where the unsatisfied rule is defined: the synthesized attribute *eval* at the production *Add* (Line 4), into the aspect *eval* (Line 9). In previous versions of `AspectAG` this kind of error was detected, but generating a long and noisy message referred to a missing instance of the class used to implement look up in records. The message was not related to our domain an leaked implementation information.

2.3.4 Duplication of Fields. An attribute should not have more than one rule to compute it in a given production. Also, children are unique. For instance this kind of error could be introduced if add_{eval} is defined twice when defining asp_{eval} at Line 9.

$$asp_{eval} = traceAspect\ (Proxy\ @\ ('Text\ \texttt{"eval"})) \$\ add_{eval} \triangleleft add_{eval} \triangleleft val_{eval} \triangleleft var_{eval} \triangleleft emptyAspect$$

Due to some flexibility matters that will become more clear in the next section, we do not detect this error at this point.

The error appears again at Line 18, when we close the AG:

```
Error: Duplicated Labels on Attribution
       of Attribute eval:Int
trace: aspect eval
```

This is another case where the trace is helpful in the task of finding the source of an error. The trace information says that the duplication was generated when we defined the aspect *eval*; i.e. Line 9. As in 2.3.3 previous `AspectAG` implementations produced a type error in this scenario, but it was difficult to read, and it leaked implementation details.

3 RECORDS AND REQUIREMENTS

In order to provide flexibility, safety, and achieve error messages as the ones shown in the previous section, `AspectAG` internals are built from strongly typed extensible records. Mistakes due to references to lacking fields (Section 2.3.3) are detected at compile time as an incorrect look up in a given record. Also, the definition of duplicated attributes (Section 2.3.4) results in a type error, due to an incorrect record extension. Then using user-defined type errors, a tool introduced in GHC to help improving the quality of type-level programming error messages, custom error messages are printed out using the type family *GHC.TypeLits.TypeError*.

In this section we show an implementation of extensible records and introduce a framework to encode EDSL type errors that keeps track of the possible sources of errors.

3.1 Polymorphic Heterogeneous Records

The implementation of the library is strongly based on the use of extensible records. They are used in the representation of:

- *Attribution*s, which are mappings from attribute names to values. In our example, *eval* and *env*, defined in Figure 2, are labels of such records.
- For each production, there is a set of children, each one with an associated attribution. In this case each field is not a flat value, but a full record by itself. The labels in the example are: *leftAdd*, *rightAdd*, *ival* and *vname*.
- *Aspects*, which are records of rules indexed by productions. In this case the labels are: *add*, *val* and *var*.
- Semantic functions, which are kept in a record (not visible by the user).

Extensible records coded using type-level programming are already well known in the Haskell community. The `HList` library [9] popularized them. One common way to implement a record is by using a GADT indexed by the list of types of the values stored in its fields. These types are usually of kind *Type*, what makes sense since *Type* is the kind of inhabited types, and records store values. However, in cases such as our children records, where we store attributions that are also represented by an indexed GADT, we would like to be able to reflect some of this structural information at the indexes of the record. This can be achieved if we are polymorphic in the kind of the types corresponding to the record fields. Based on this approach we designed a new, polykinded, extensible record structure:

$$\begin{aligned}&\textbf{data}\ Rec\ (c :: k)\ (r :: [(k', k'')]) :: Type\ \textbf{where}\\ &\quad EmptyRec\ ::\ Rec\ c\ '[]\\ &\quad ConsRec\ \ ::\ LabelSet\ ('(l, v)\ ' : r)\\ &\qquad\qquad \Rightarrow TagField\ c\ l\ v \rightarrow Rec\ c\ r \rightarrow Rec\ c\ ('(l, v)\ ' : r)\end{aligned}$$

A record is indexed by a parameter *c*, pointing out which instance of record we are defining (e.g. attribution, set of children, aspect, etc.), and a promoted list of pairs *r* representing the fields. The first component of each pair is the label. The kind k' is polymorphic, since it is not mandatory to the type of labels to be inhabited; they need to live only at type level. The second component is also polymorphic and it can have an elaborate kind k''. By doing this, in the cases where the type of a stored value is an indexed GADT, we can use its index as the index that represents the field. *Tagfield* is the type of the fields of our records.

$$\begin{aligned}&\textbf{data}\ TagField\ (c :: k)\ (l :: k')\ (v :: k'')\ \textbf{where}\\ &\quad TagField\ ::\ Label\ c \rightarrow Label\ l \rightarrow WrapField\ c\ v\\ &\qquad\qquad \rightarrow TagField\ c\ l\ v\\ &\textbf{data}\ Label\ (l :: k) = Label\end{aligned}$$

Labels are proxies, notice that all the labels in Section 2 are defined using the constructor *Label*, varying the phantom type. The third argument -of type *WrapField c v*- should be the value that we want to tag. Note that *v* is kind polymorphic. For instance, a concrete

instance of v can be something like $[(Att, Type)]$ in the case of children. When actually creating a field to append in a record, an actual value must be stored; this information must be Wrapped by a type constructor. The type family

```
type family WrapField (c :: k) (v :: k') :: Type
```

computes, depending on the index c, the wrapping of v under a suitable type constructor. Note that if v is already inhabitated then *WrapField c* can be the identity (type) function.

LabelSet is a predicate that encodes the fact that there are no repeated labels. A relevant design decision is the implementation of the *LabelSet* constraint. A type class with similar semantics is introduced on the `HList` library, where the property of non duplication of labels is implemented by the (recursive) instances of the class. By using a type class to encode a predicate, new instances can be defined anytime since type classes are open. When an instance is not found, one could argue that this does not mean that the predicate is *False*, but that the typechecker did not find a proof. In the case of *LabelSet*, given a set of labels (if we know how to compare them) the property is closed and decidable. Also, to use our unified way to process type errors we need to manipulate the truth value of the result. For both arguments a Boolean type family seems the way to go.

```
type family LabelSetF (r :: [(k, k')]) :: Bool where
  LabelSetF '[]              = True
  LabelSetF ('(l, v) ': '[]) = True
  LabelSetF ('(l, v) ': '(l', v') ': r)
    = And3 (Not (l ≡ l'))
           (LabelSetF ('(l, v) ': r))
           (LabelSetF ('(l', v') ': r))
```

where $(\equiv)$ is the type level equality operator (*Data.Type.Equality*), and *And3* a type family computing the conjunction of three boolean arguments. Then we can encode the predicate as the constraint:

```
type LabelSet r = LabelSetF r ~ True
```

3.2 Record Instances

In our library most functions over records are implemented over the generic implementation. After that we implement our record-like data structures as particular instances of the general datatype. To introduce a new record structure we must give an index acting as a name (the "c" parameter), and code the family instance *WrapField*. As we shall see later, in order to print out domain specific error messages we also need to provide instances for the type families *ShowField* and *ShowRec*.

To code in a strongly kinded way, it is also useful to provide specific datatypes for labels, where *Att* is used for attributions, *Prod* for productions, and *Child* for children.

```
data Att   = Att Symbol Type
data Prod  = Prd Symbol NT
data Child = Chi Symbol Prod (Either NT T)
data NT    = NT Symbol
data T     = T Type
```

We give some examples of record instances, that we use in our implementation of `AspectAG`.

Example: Attribution. Attributions are mappings from attribute names to values. We instance *Rec* with an empty datatype *AttReco* as index. We define a descriptive name and fix the polymorphic kinds, since Attribution labels are of kind *Att*, and fields of kind *Type*.

```
data AttReco
type Attribution (attr :: [(Att, Type)]) = Rec AttReco attr
```

The label *add*, with type $Label\ (`Prd\ \texttt{"Add"}\ Nt_{Expr})$, of Section 2 is an example of a valid *Attribution* label.
We do not need to wrap the fields since they are simply values:

```
type instance WrapField AttReco (v :: Type) = v
```

We also use an specific name for fields:

```
type Attribute (l :: Att) (v :: Type) = TagField AttReco l v
```

We can define specialized versions of the constructors *TagField*, *EmptyRec* and *ConsAtt* by using GHC's pattern synonyms [15].

```
pattern Attribute     :: v → TagField AttReco l v
pattern Attribute v   =  TagField Label Label v
pattern EmptyAtt      :: Attribution '[]
pattern EmptyAtt      =  EmptyRec
pattern ConsAtt       :: LabelSet ('(att, val) ': atts)
                       ⇒ Attribute att val → Attribution atts
                       → Attribution ('(att, val) ': atts)
pattern ConsAtt a as  =  ConsRec att atts
```

Example: Children Records. A child is associated to a production. Our instance has itself an index with this information. Recall that *Prod* has a name for the production but also a name for the nonterminal that it rewrites, so all this information is contained on a child and used to check well formedness where it is used. In this case the labels are of kind *Child*, and the values have a kind that represents the list of associations attribute-value; i.e. an attribution.

```
data ChiReco (prd :: Prod)
type ChAttsRec prd (chs :: [(Child, [(Att, Type)])])
  = Rec (ChiReco prd) chs
```

WrapField takes the type information of the field, which is not inhabited, and puts the *Attribution* wrapper.

```
type instance WrapField (ChiReco prd) (v :: [(Att, Type)])
  = Attribution v
```

Examples of types of Attribution labels of the example are:

```
rightAdd :: Label ('Chi "rightAdd" P_Add ('Left Nt_Expr))
ival     :: Label ('Chi "ival"     P_Val ('Right ('T Int)))
```

3.3 Requirements

As a framework to encode type errors we introduce the concept of requirements.

```
class Require (op :: Type) (ctx :: [ErrorMessage]) where
  type ReqR op :: Type
  req :: Proxy ctx → op → ReqR op
```

Some functions require that specific non trivial conditions are met on types, or a type error must occur otherwise. For instance, each time we look up in a record we require that some label actually

belongs to the record. Given an operation represented by a datatype *op*, that takes all the arguments needed for the current operation to be performed, *req* extracts the tangible results, whose return type depends on the operation. The function *req* also uses some context information *ctx* (i.e. the *trace* of the error) to provide more useful information in the error message.

We collect the constraints imposed to a *Require* instance in *RequireR*:

```
type RequireR (op :: Type) (ctx :: [ErrorMessage]) (res :: Type)
  = (Require op ctx, ReqR op ~ res)
```

Some requirements such as label equality are only about types, which means that *req* is not used. It is still useful to keep type errors in this framework, and in that case we use only the *Require* constraint.

To pretty print type errors, we define a special operation:

```
data OpError (m :: ErrorMessage)
```

OpError is a phantom type containing some useful information to print. A call to *OpError* happens when some requirement is not fulfilled. A non satisfied requirement means that there will be no regular instance of this class and it produces a *OpError* requirement. When we call *req* with this operator the type checker reports an error since the instance is:

```
instance (TypeError (Text "Error: " :<>: m :$$:
                     Text "trace: " :<>: ShowCTX ctx))
  => Require (OpError m) ctx
```

The type family *GHC.TypeLits.TypeError* works like the value-level function *error*. When it is evaluated at compile time a type error is raised, with a given error message including a description *m* of the error and the context *ctx* where it occurred. *TypeError* is completely polymorphic so it can be used with kind *ErrorMessage* → *Constraint* as we do here. This generic instance is used to print all sort of type errors in a unified way, otherwise catching type errors case by case would be difficult and error prone. Note that specific information of what happened is on *OpError* which is built from a specific instance of *Require*, from a given operator and where every type relevant to show the error message was in scope.

3.3.1 Record Requirements: Lookup. As an example of record requirements, we define the lookup operation.

```
data OpLookup (c :: Type) (l :: k) (r :: [(k, k')]) :: Type where
  OpLookup :: Label l → Rec c r → OpLookup c l r
```

The operation is specified by an algebraic datatype, parametric on: the record class (*c*), the index we are looking for (*l*), and the proper record (*r*). The head label is inspected and depending on the types the head value is returned or a recursive call is performed. To make decisions over types, and set return types depending on arguments, we implement a dependent type function, which is encoded in Haskell with the usual idioms of type level programming. For instance, the proof of equality must be made explicit using a proxy to help GHC to carry this type level information. We introduce a new *OpLookup′* with an auxiliary *Bool*ean at type level:

```
data OpLookup' (b :: Bool)
               (c :: Type)
               (l :: k)
               (r :: [(k, k')]) :: Type where
  OpLookup' :: Proxy b → Label l → Rec c r
            → OpLookup' b c l r
```

For *OpLookup* we take the head label l' -there must be a head, otherwise we are on the error instance that we introduce later- and use the auxiliary function with the value equal to the predicate of equality of l' and the argument label *l*:

```
instance (Require (OpLookup' (l ≡ l') c l ('(l', v) ' : r)) ctx)
  => Require (OpLookup c l ('(l', v) ' : r)) ctx where
  type ReqR (OpLookup c l ('(l', v) ' : r))
    = ReqR (OpLookup' (l ≡ l') c l ('(l', v) ' : r))
  req ctx (OpLookup l r)
    = req ctx (OpLookup' (Proxy @ (l ≡ l')) l r)
```

The instance of *OpLookup′* when $b \equiv True$ corresponds to a *head* function, since we know that the searched label is on the first position:

```
instance Require (OpLookup' 'True c l ('(l, v) ' : r)) ctx where
  type ReqR (OpLookup' 'True c l ('(l, v) ' : r))
    = WrapField c v
  req Proxy (OpLookup' Proxy Label (ConsRec f _))
    = untagField f
```

Note that we set the return type to be *WrapField c v*, which is completely generic.

When $b \equiv False$ a call to *OpLookup* on the tail of the record is performed:

```
instance (Require (OpLookup c l r) ctx)
  => Require (OpLookup' False c l ('(l', v) ' : r)) ctx where
  type ReqR (OpLookup' False c l ('(l', v) ' : r))
    = ReqR (OpLookup c l r)
  req ctx (OpLookup' Proxy l (ConsRec _ r))
    = req ctx (OpLookup l r)
```

When we try to look up in an empty record an error happens: we went over all the record and there was no value tagged with the searched label. Here we require the instance of *OpError* informing the actual error:

```
instance Require (OpError
    ( Text "field not Found on " :<>: Text (ShowRec c)
  :$$: Text "looking up the "    :<>: Text (ShowField c)
  :<>: ShowT l)) ctx
  => Require (OpLookup c l ('[] :: [(k, k')])) ctx where
  type ReqR (OpLookup c l ('[] :: [(k, k')])) = ()
  req = ⊥
```

Notice that this error message is the one we obtain in Section 2.3.3, when trying to access to an undefined attribute *foo*.

This procedure is recurrent in all our development:

- Use requirements for operations that can introduce DSL errors, keeping track of the call trace in the *ctx* index.
- Encode the bad cases, requiring an *OpError*.

Then, when the user does not fulfill a requirement, the type checker triggers a type error since type class resolution comes upon this singleton instance of *OpError*.

In the previous definition *ShowRec* and *ShowField* type families were used to pretty print the type information depending on which instance of records we are working on.

```
type family ShowRec   (c :: k) :: Symbol
type family ShowField (c :: k) :: Symbol
```

For instance, for attributions we implement:

```
type instance ShowRec   AttReco = "Attribution"
type instance ShowField AttReco = "attribute named "
```

For children:

```
type instance ShowRec   (ChiReco a) = "Children Map"
type instance ShowField (ChiReco a) = "child labelled "
```

The fact that type families can be open is very convenient in this context; new records can be defined in a customized and modular way. We think that *Require* and *Rec* transcend the status of internal modules for `AspectAG` and are useful as a standalone library on their own.

The *ShowT* family is also interesting:

```
type family ShowT (t :: k) :: ErrorMessage
```

When we print labels (with kinds such as *Att*, *Prd* or *Chi*) to show clear type errors we should print different information depending on the kind. This imposes some sort of ad-hoc polymorphism at the kind level. Haskell does not provide such *kind classes*, but they are not necessary since polykinded type families are actually not parametric. When programming at type level we can actually inspect kinds and pattern match on them as using *typeOf* in languages where types are avaiable at run time. Therefore, the following definitions are completely legal:

```
type instance ShowT ('Att l t)
   = Text "Attribute " :<>: Text l :<>: Text     ":" :<>: ShowT t
type instance ShowT ('Prd l nt)
   = ShowT nt :<>: Text "::Production " :<>: Text l
```

and so on. Also, we can build an instance for any type of kind *Type*, so inhabited types are printed with their standard name:

```
type instance ShowT (t :: Type) = ShowType t
```

Then, for example in Section 2.3.3 the error messages say "`Attribute eval:Int`" when referring to the attribute *eval* and "`Non-Terminal Expr::Production Add`" when referring to the production *add*.

3.3.2 Label Equality Requirements. We use *RequireEq* to require label (and thus type) equality, which is actually a sugar for a couple of constraints:

```
type RequireEq (t1 :: k) (t2 :: k) (ctx :: [ErrorMessage])
   = (Require (OpEq t1 t2) ctx, t1 ~ t2)
```

The first constraint is a requirement, using the following operator:

```
data OpEq t1 t2
instance RequireEqRes t1 t2 ctx
   ⇒ Require (OpEq t1 t2) ctx where
  type ReqR (OpEq t1 t2) = ()
  req = ⊥
```

Notice that in this case we do not define an implementation for the function *req*, since this requirement is used only at type level. The type family *RequireEqRes* is a type level function computing a constraint. It reduces to the requirement of an *OpError* if $t1 \neq t2$, building a readable error message, or to the trivial (Top) constraint otherwise.

```
type family RequireEqRes (t1 :: k) (t2 :: k)
                (ctx :: [ErrorMessage]) :: Constraint where
  RequireEqRes t1 t2 ctx
    = If (t1 `Equal` t2)
         (() :: Constraint)
         (Require (OpError (Text "" :<>: ShowT t1
                                :<>: Text "  /=  " :<>: ShowT t2)) ctx)
```

This is the kind of error message that appears in 2.3.2 and in most of the examples of 2.3.3, where the actual type is not the expected.

4 IMPLEMENTATION

In this section we show how we put records and requirements to work in the implementation of `AspectAG`.

4.1 Families and Rules

We use the concept of *families* as input and output for attribute computations. A family for a given production contains an attribution for the parent, and a collection of attributions, one for each of the children. A family *Fam* is a product of *Attribution* and *ChAttsRec*, and it is indexed with the production to which it belongs:

```
data Fam (prd :: Prod)
         (c   :: [(Child, [(Att, Type)])])
         (p   :: [(Att, Type)]) :: Type where
  Fam :: ChAttsRec prd c → Attribution p → Fam prd c p
```

Attribute computations, or rules, are actually functions from an *input family* (inherited attributes from the parent and synthesized of the children) to an *output family* (synthesized attributes for the parent, inherited to children). We implement them with an extra arity to make them composable; a well-known trick[13].

```
type Rule (prd :: Prod)
          (sc  :: [(Child, [(Att, Type)])])
          (ip  :: [(Att, Type)])
          (ic  :: [(Child, [(Att, Type)])])
          (sp  :: [(Att, Type)])
          (ic' :: [(Child, [(Att, Type)])])
          (sp' :: [(Att, Type)])
  = Fam prd sc ip → Fam prd ic sp → Fam prd ic' sp'
```

Given an input family we build a function that updates the output family constructed thus far. Note that the rules are indexed by a production.

To carry context information printable on type errors, most of the time we actually manipulate *tagged* rules:

```
newtype CRule (ctx :: [ErrorMessage]) prd sc ip ic sp ic' sp'
  = CRule { mkRule :: Proxy ctx → Rule prd sc ip ic sp ic' sp' }
```

4.2 Defining Attributes

The function *syndef* introduces a new synthesized attribute; i.e. a rule that extends the attribution for the parent in the output family, provided that some requirements are fullfiled.

```
syndef :: (ctx' ~ ((Text "syndef(" :◇: ShowT ('Att att t) :◇:
                    Text ", " :◇: ShowT prd :◇: Text ")")
                   ' : ctx)
          , RequireEq t t' ctx'
          , RequireR (OpExtend AttReco ('Att att t) t sp) ctx
                     (Attribution sp'))
       ⇒ Label ('Att att t) → Label prd
       → (Proxy ctx' → Fam prd sc ip → t')
       → CRule ctx prd sc ip ic sp ic sp'
syndef att prd f
  = CRule $ λctx inp (Fam ic sp)
    → Fam ic $ req ctx (OpExtend att (f Proxy inp) sp)
```

It takes an attribute name *att*, a production *prd*, and a function *f* that computes the value of the attribute in terms of the input family. The function *f* takes an extra *proxy* argument to carry context information. In this case, we extend the current context *ctx* to a new one (*ctx'*) including information about the *syndef* definition where it was called. We require the type *t'* of the value returned by *f* to be equal to the type *t* of the attribute, using a *RequireEq* requirement. Notice that we could have implemented this restriction by using the same type variable *t* for *t* and *t'*. But in this case we would not have a customized error message. The last requirement (*OpExtend*) states that we have to be able to extend the attribution indexed by *sp* with the attribute *att* and the result is an attribution with index sp'. This requirement imposes the constraint that assures that this insertion does not duplicate the attribute *att*. Since this requirement is not internal to the computation defined in *syndef*, we use the current context *ctx* instead of *ctx'*.

Using *syndef* we can define rules like add_{eval} of Section 2:

```
add_eval = syndef eval add $ λProxy (Fam sc ip) →
  (+) (sc .#. leftAdd .#. eval) (sc .#. rightAdd .#. eval)
```

where (.#.) is the lookup operator.

By looking at the type of *syndef*, it becomes more clear how, for example, the error message of 2.3.2 is generated. The error is produced by failing the equality requirement of *t* and *t'* (*Int* and *Maybe Int*), and the trace information is given by the context *ctx'*.

In practice it is useful to use a monadic version of *syndef*, which is the one we used in Section 2:

```
syndefM att prd = syndef att prd ∘ def

def :: Reader (Proxy ctx, Fam prd chi par) a
  → (Proxy ctx → (Fam prd chi par) → a)
def = curry ∘ runReader
```

With a *Reader* we avoid to explicitly manipulate the input family. We defined the monadic function *at* used to sugarize definitions:

```
class At pos att m where
  type ResAt pos att m
  at :: Label pos → Label att → m (ResAt pos att m)
```

with instances for looking up attributes into the attribution of the parent (*lhs*) or the attribution of a given child. The following is the instance for the case of children, where are two lookups involved, because we have to find the child in the record of children and then the attribute in its attribution. We also require some equalities, including the fact that the child has to be a non-terminal ('*Left* ('*NT n*)).

```
instance
    (RequireR (OpLookup (ChiReco prd) ('Chi ch prd nt) chi)
              ctx (Attribution r)
    , RequireR (OpLookup AttReco ('Att att t) r) ctx t'
    , RequireEq prd prd' ctx
    , RequireEq t t' ctx
    , RequireEq ('Chi ch prd nt) ('Chi ch prd ('Left ('NT n))) ctx)
  ⇒ At ('Chi ch prd nt) ('Att att t)
       (Reader (Proxy ctx, Fam prd' chi par)) where
  type ResAt ('Chi ch prd nt) ('Att att t)
             (Reader (Proxy ctx, Fam prd' chi par))
    = t
  at ch att
    = liftM (λ(ctx, Fam chi _)
                 → let atts = req ctx (OpLookup ch chi)
                   in  req ctx (OpLookup att atts))
            ask
```

The function *inhdef* defines an inherited attribute. For simplicity reasons we omit the constraints.

```
inhdef :: (...) ⇒ Label ('Att att t)
                → Label prd
                → Label ('Chi chi prd ntch)
                → (Proxy ctx' → Fam prd sc ip → t')
                → CRule ctx prd sc ip ic sp ic' sp
inhdef att prd chi f
  = CRule $ λctx inp (Fam ic sp)
    → let catts  = req ctx (OpLookup chi ic)
          catts' = req ctx (OpExtend att (f Proxy inp) catts)
          ic'    = req ctx (OpUpdate chi catts' ic)
      in  Fam ic' sp
```

An inherited attribute *att* is defined in a production *prd* for a given child *chi*. In this case we have to lookup the attribution of the child into the inherited attributes of the children *ic*, then extend it and update *ic*.

Again, the monadic alternative *inhdefM* is provided. Functions to define synthesized and inherited attributes are neccesary to compose nontrivial attribute grammars. More constructs are useful in practice. In Section 2, *synmod* (that modifies an attribute instead of adding it) was used and proved to be useful to change semantics. Its inherited counterpart *inhmod* is also provided. On top of this we can implement some common patterns that generate rules from higher level specifications.

4.3 Combining Rules.

Functions as *syndef* or *inhdef* build rules from scratch, defining how to compute one single new attribute from a given family using functions of the host language. A full rule is usually more complex, since it builds a full output family, where usually many attributes are computed in many ways. To build such rules we compose from smaller rules. Composing rules is easy, given the extra arity trick:

```
ext :: RequireEq prd prd' (Text "ext" : ctx)
    ⇒ CRule ctx prd sc ip ic sp ic' sp'
    → CRule ctx prd' sc ip a b ic sp
    → CRule ctx prd sc ip a b ic' sp'
(CRule f) `ext` (CRule g)
  = CRule $ λctx input → f ctx input ∘ g ctx input
```

Note that we require the rules to be tagged from the same production. If instead of using *RequireEq* we unify *prd* and *prd′* to the same type variable, and try to combine two rules from different productions, we get a huge type error where the type mismatch is obfuscated on hundreds of lines of error code, printing every record, such as *ic* or *sp*, and every class constraint, such as *Require*.

4.4 Aspects

Aspects are collections of rules, indexed by productions. They are an instance of *Rec*, defined as:

```
data PrdReco
type instance WrapField PrdReco (rule :: Type) = rule
type Aspect (asp :: [(Prod, Type)]) = Rec PrdReco asp
type instance ShowRec PrdReco    = "Aspect"
type instance ShowField PrdReco  = "production named "
```

As done in Section 4.1 with rules, to keep track on contexts we introduce the concept of a tagged aspect:

```
newtype CAspect (ctx :: [ErrorMessage]) (asp :: [(Prod, Type)])
   = CAspect { mkAspect :: Proxy ctx → Aspect asp }
```

In Section 4.2 we saw how that context is extended when an attribute is defined using *syndef* or *inhdef*. In the example of Section 2 the function *traceAspect* was introduced as a tool for the user to place marks, visible in the trace when a type error occurs. We implement *traceAspect* using a sort of *map* function, traversing the record.

```
traceAspect (_ :: Proxy (e :: ErrorMessage))
   = mapCAspect $ λ(_ :: Proxy ctx)
                → Proxy @ ((Text "aspect " :◇: e) : ctx)
mapCAspect fctx (CAspect fasp)
   = CAspect $ mapCtxRec fctx ∘ fasp ∘ fctx
```

where *mapCtxRec* is a dependent function:

```
class MapCtxAsp (r    :: [(Prod, Type)])
                (ctx  :: [ErrorMessage])
                (ctx' :: [ErrorMessage]) where
  type ResMapCtx r ctx ctx' :: [(Prod, Type)]
  mapCtxRec :: (Proxy ctx → Proxy ctx')
            → Aspect r → Aspect (ResMapCtx r ctx ctx')
```

whose implementation does not offer new insights.

4.5 Combining Aspects

An aspect models a piece of semantics of a grammar. To make semantics extensible it is enough to implement an algorithm to merge two aspects, and a way to make an aspect from one single rule. Since our most basic primitives *syndef* and *inhdef* build a single rule, adding rules one by one to an aspect is a common operation.

4.5.1 Adding a Rule. We define the *aspect extension* function, that adds a tagged rule to a tagged Aspect.

```
rule ◃ (CAspect fasp)
   = CAspect $ λctx → req ctx (OpComRA rule (fasp ctx))
```

To implement this function we define an operation *OpComRA*, that combines a rule with an aspect.

```
data OpComRA (ctx :: [ErrorMessage])
             (prd :: Prod)
             (sc  :: [(Child, [(Att, Type)])])
             (ip  :: [(Att, Type)])
             (ic  :: [(Child, [(Att, Type)])])
             (sp  :: [(Att, Type)])
             (ic' :: [(Child, [(Att, Type)])])
             (sp' :: [(Att, Type)])
             (a   :: [(Prod, Type)]) where
  OpComRA :: CRule ctx prd sc ip ic sp ic' sp'
          → Aspect a → OpComRA ctx prd sc ip ic sp ic' sp' a
```

This operation has two cases. If the rule is indexed by a production not appearing on the aspect, the combination is simply an append. Otherwise we must lookup the current rule an update it, combining the inserted rule. We implement these cases in a lower level operation *OpComRA′*, where the truth value of the production membership is explicit, in a similar way as we have done for the lookup operation in Section 3.3.1. In this case the predicate is the type-level function *HasLabel*:

```
type family HasLabel (l :: k) (r :: [(k, k')]) :: Bool where
  HasLabel l '[]            = False
  HasLabel l ('(l', v) ': r) = Or (l ≡ l') (HasLabel l r)
```

The *Require* instance for *OpComRA′* in the case where the first parameter is *'False* implements an append. The *'True* case is a little bit more verbose, but anyway inmediate: we lookup the rule at the original aspect, extend the rule with the one as argument, and update the aspect with the resulting rule.

4.5.2 Combining two aspects. To combine two aspects we define the operation *OpComAsp*, which takes two aspects as parameters:

```
data OpComAsp (al :: [(Prod, Type)]) (ar :: [(Prod, Type)]) where
  OpComAsp :: Aspect al → Aspect ar → OpComAsp al ar
```

We chose arbitrarly to do the recursion on the second argument. The empty aspect is a neutral element:

```
instance Require (OpComAsp al '[]) ctx where
  type ReqR (OpComAsp al '[]) = Aspect al
  req ctx (OpComAsp al _) = al
```

In the recursive case, we take the tail *ar* of the recursive argument, and call the recursive function with *al* and *ar*. The resulting aspect is combined with the head rule using the operation *OpComRA*.

```
instance
    (RequireR (OpComAsp al ar) ctx (Aspect ar')
    , Require  (OpComRA ctx prd sc ip ic sp ic' sp' ar') ctx)
  ⇒ Require (OpComAsp al
    ('(prd, CRule ctx prd sc ip ic sp ic' sp') ': ar)) ctx where
  type ReqR (OpComAsp al
              ('(prd, CRule ctx prd sc ip ic sp ic' sp') ': ar))
    = ReqR (OpComRA ctx prd sc ip ic sp ic' sp'
              (UnWrap (ReqR (OpComAsp al ar))))
  req ctx (OpComAsp al (ConsRec prdrule ar))
    = req ctx (OpComRA (untagField prdrule)
                       (req ctx (OpComAsp al ar)))
```

Thus, the function that combines two tagged aspects is:

```
al ⋈ ar
  = CAspect $ λctx → req ctx (OpComAsp (mkAspect al ctx)
                                       (mkAspect ar ctx))
```

4.6 Semantic functions

In Section 2 we show how sem_{Expr} is defined. It takes an aspect, an AST, and builds a function from the synthesized attributes to the inherited attributes. More in general, for the domain associated with each non-terminal we take the function mapping its inherited to its synthesized attributes. The function *knitAspect* is a wrapper to add context

```
knitAspect (prd :: Label prd) asp fc ip
  = let ctx  = Proxy @ '[]
        ctx' = Proxy @ '[Text "knit" :◇: ShowT prd]
    in  knit ctx
        (req ctx' (OpLookup prd ((mkAspect asp) ctx))) fc ip
```

and the real work is done by the circular funtion *knit*, wich takes the combined rules for a node and the semantic functions of the children, and builds a function from the inherited attributes of the parent to its synthesized attributes.

```
knit (ctx  :: Proxy ctx)
     (rule :: CRule ctx prd (SCh fc) ip (EmptiesR fc) '[] (ICh fc) sp)
     (fc   :: SemFunRec fc)
     (ip   :: Attribution ip)
  = let (Fam ic sp) = mkRule rule ctx (Fam sc ip)
                                      (Fam ec emptyAtt)
        sc          = kn fc ic
        ec          = empties fc
    in  sp
```

where the function *kn* is a dependent *zipWith* ($).

```
class Kn (fcr    :: [(Child, Type)]) (prd :: Prod) where
  type ICh fcr :: [(Child, [(Att, Type)])]
  type SCh fcr :: [(Child, [(Att, Type)])]
  kn :: SemFunRec fcr → ChAttsRec prd (ICh fcr)
     → ChAttsRec prd (SCh fcr)
```

and *empties* builds an empty attribution for each child.

```
class Empties (fc :: [(Child, Type)]) (prd :: Prod) where
  type EmptiesR fc :: [(Child, [(Att, Type)])]
  empties :: Record fc → ChAttsRec prd (EmptiesR fc)
```

While they are nice examples of type-level programming, we left the implementation out of this paper, since this technique is well documented in the literature [4, 13, 27].

4.7 Terminals

A production specifies how a nonterminal symbol can be rewritten. It can rewrite to a mix of terminal and nonterminal symbols. Usually, in attribute grammar systems a terminal has only one attribute: itself. In `AspectAG` all children are put in a record, each position containing an attribution. In previous versions of `AspectAG` terminals where directly put as a children instead of an attributions. This was possible since at type-level this records were essentialy untyped. We decided to lift the shape of the structure to kinds, adding up static guarantees, but losing this flexibility. There are at least two approaches to treat terminals, of which we choose the second for simplicity:

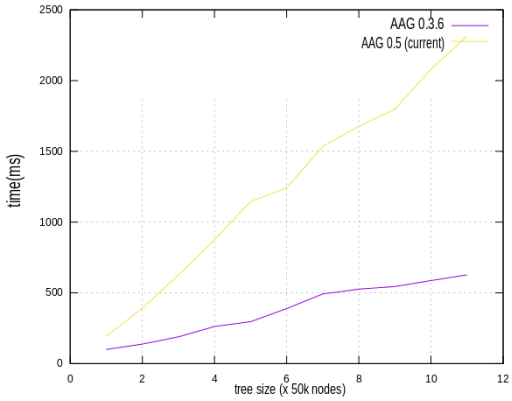


Figure 3: Performance Comparison

- *ChAttsRec* could be a record containing a promoted sum type, each child is either a terminal, or a nonterminal with an index attribution of kind $[(Att, Type)]$. At term level, constructors for inhabitants can be build using a GADT.
- Model all children as a record of attributions, with a trivial attribution containing only one attribute for the terminal case.

As seen before, to introduce an attribute the user defines a unique name (a label). As we say, there is a trivial attribute for each terminal. To chose a name is not a problem since it is isolated behind a children. Accordingly, semantic functions of the children can be coded in a polymorphic way.

```
class SemLit a where
  sem_Lit :: a → Attribution ('[] :: [(Att, Type)])
     → Attribution ['(Att "term" a, a)]
  lit :: Label ('Att "term" a)

instance SemLit a where
  sem_Lit a _ = (Label .=. a) ∗. emptyAtt
  lit = Label @ ('Att "term" a)
```

All of them are labelled with the attribute named `"Term"`, accesible using the *lit* expression, and the semantic function simply wraps a value in an attribution.

5 RELATED WORK

There is a significant number of AG implementations. Some of them are implemented as standalone compilers or generators like LRC [16], UUAGC [24], LISA [12], JastAdd [6] and Silver [25], and others are embbeded in languages like Scala (e.g. Kiama [20]) or Haskell ([1, 4, 11, 13, 26, 27]). This work is based on `AspectAG` [27], where extensible records are used to implement a strongly typed first class AG DSL embedded in Haskell. By using new Haskell type level programming techniques we obtain a more clear design and a safer implementation while preserving its main characteristics.

However, the current version of the library introduces a performance penalty compared to previous versions. Figure 3 shows a comparison of execution times with `AspectAG` 0.3.1 and 0.5, of the *repmin problem* [2], linearly increasing the size of the tree. We

detected a big linear overhead, due to the runtime penalty generated by all the new structures generate a runtime penalty. An overhead in compilation time is also expected, since all type computations must be performed. This has not been tangibe to us as library users. Nevertheless, this overhead is difficult to quantify, since old versions of `AspectAG` do not compile in modern GHC.

Error messages were a big downside that we solved. Managing type errors on EDSLs is an old problem and an active research area. The idea of transforming a typing problem into a constraint problem is not new [21, 22]. Other embedded implementations of AGs [26] solve the type diagnose problem at the cost of making it staged.

Compiler support added with the *TypeError* type family was essential, but further support would be desirable, in particular to control class constraint solving and to avoid leaks and non readable messages. Research by Heeren [7] was implemented for the Helium compiler. Recently, Serrano Mena and Hage [17, 18] developed a set of techniques for customizing type error diagnosis for GHC. We think this can complement our more ad-hoc approach.

6 CONCLUSION AND FUTURE WORK

In this paper we presented a library of first class strongly kinded attribute grammars. Using type level programming we achieved to get precise domain specific type errors.

Grammars do not need to be tied to a datatype. Reusing an AG in a new datatype when a language is extended is nice, but the semantic function must be implemented (or derived) twice. This is not a problem of our implementation, but of Haskell's expresiveness. To explore how to integrate our library with extensible datatypes is left as an open problem.

We think the library is useful and easy to use. Having the DSL embedded in Haskell allows to develop furher abstractions, such as common patterns, or macros, or to use the power of higher order to generate grammars. In addition to the examples we have coded during the development, the library is being tested with success in the implementation of a real compiler of a non trivial functional language. Working on performance optimization, to alleviate the performance overhead, is left for future work.

To get clear error messages we had to deal with some tradeoffs. It requires careful management of context information annotated in types, and explicit term level proxy arguments to carry type information during type checking. Nevertheless, this implementation details are transparent to the user. By strongly typing we have lost some flexibilities. For example, rules are related to a production, this was not designed this way in previous versions of `AspectAG`, which allowed us to reuse some rules out of the box. Anyway, this can be shallowed since the host language provides type -and kind- polymorphism.

We developed a methodology to manage error message generation using *Requirements*. We think this idea can be applied similarly in other EDSL implementations and it deserves to be explored.

REFERENCES

[1] Florent Balestrieri. 2015. *The productivity of polymorphic stream equations and the composition of circular traversals.* Ph.D. Dissertation. University of Nottingham, UK. http://eprints.nottingham.ac.uk/29745/

[2] Richard Bird. 1984. Using circular programs to eliminate multiple traversals of data. Acta Informática. 21 (10 1984), 239–250.

[3] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. *SIGPLAN Not.* 40, 9 (Sept. 2005), 241–253.

[4] Oege de Moor, Simon L. Peyton Jones, and Eric Van Wyk. 1999. Aspect-Oriented Compilers. In *Generative and Component-Based Software Engineering, First International Symposium, GCSE'99, Erfurt, Germany, September 28-30, 1999, Revised Papers.* 121–133.

[5] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application.. In *ESOP (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 229–254.

[6] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.* 69, 1-3 (2007), 14–26.

[7] Bastiaan Heeren. 2005. *Top quality type error Messages.* Ph.D. Dissertation. Utrecht University, Netherlands. http://dspace.library.uu.nl:8080/handle/1874/7297

[8] Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *ESOP (Lecture Notes in Computer Science)*, Vol. 1782. Springer, 230–244.

[9] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04).* ACM, New York, NY, USA, 96–107.

[10] Donald E. Knuth. 1968. Semantics of Context-Free Languages. In *In Mathematical Systems Theory.* 127–145.

[11] Pedro Martins, João Paulo Fernandes, and João Saraiva. 2013. Zipper-based Attribute Grammars and their Extensions. In *XVII Simpósio Brasileiro de Linguagens de Programação (LNCS).*

[12] Marjan Mernik and Viljem Žumer. 2005. Incremental programming language development. *Computer languages, Systems and Structures* 31 (2005), 1–16.

[13] Oege De Moor. 1999. First-class Attribute Grammars. *Informatica* 24 (1999), 2000.

[14] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell workshop* (haskell workshop ed.).

[15] Matthew Pickering, Gergo Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. https://www.microsoft.com/en-us/research/publication/pattern-synonyms/

[16] João Saraiva. 2002. Component-based Programming for Higher-Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GCSE 2002, Held as Part of the Confederation of Conferences on Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002, Pittsburgh, PA, USA, October 3-8, 2002 (LNCS)*, Don Batory, Charles Consel, and Walid Taha (Eds.), Vol. 2487. 268–282.

[17] Alejandro Serrano. 2018. *Type Error Customization for Embedded Domain-Specific Languages.* Ph.D. Dissertation. Utrecht University, Netherlands.

[18] Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling expression-level type errors by type-level programming. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017, Bristol, UK, August 30 - September 01, 2017.* 2:1–2:15.

[19] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75.

[20] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. 2009. A Pure Object-Oriented Embedding of Attribute Grammars. In *of the Ninth Workshop on Language Descriptions, Tools, and Applications.*

[21] P.J. Stuckey, M. Sulzmann, and J. Wazny. 2003. Interactive Type Debugging in Haskell.

[22] Peter Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving type error diagnosis. *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, Haskell'04.*

[23] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07).* ACM, New York, NY, USA, 53–66.

[24] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João A. Saraiva. 1999. Designing and Implementing Combinator Languages. In *Advanced Functional Programming, Third International School, AFP'98 (LNCS)*, S. Doaitse Swierstra, Pedro Henriques, and José Oliveira (Eds.), Vol. 1608. Springer-Verlag, 150–206.

[25] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54.

[26] Marcos Viera, Florent Balestrieri, and Alberto Pardo. 2018. A Staged Embedding of Attribute Grammars in Haskell. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA, September 5-7, 2018.* 95–106.

[27] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. 2009. Attribute Grammars Fly First-class: How to Do Aspect Oriented Programming in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09).* ACM, New York, NY, USA, 245–256.

[28] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12).* ACM, New York, NY, USA, 53–66.

The Type Errors Of Our Ways

A Quantitative Approach

Olivier Danvy
Yale-NUS College & School of Computing
National University of Singapore
danvy@acm.org

Risa Shindo
Yale-NUS College
National University of Singapore
risa.shindo@u.yale-nus.edu.sg

Quang-Trung Ta
School of Computing
National University of Singapore
taqt@comp.nus.edu.sg

ABSTRACT

In principle, a type system such as ML's specifies an extensional property – the most general type of any given term, if it has one. In theory, a type system does not convey anything meaningful for ill-typed terms. And in practice, the type system is operationalized into an implementation: given an ill-typed term, this implementation emits an error message in a context. As helpful as this error message may be, we are not the first to observe that this message only reflects the particular traversal strategy of the implementation: another strategy to traverse a given term is likely to elicit another error message in another context, be it for the same type error or for another one.

We propose to parameterize the implementation of a type inferencer with its traversal strategy so that more than one strategy can be used over ill-typed terms and a wider variety of error messages can be emitted on demand. In practice, if some of these messages are more immediately understandable than the first ones, the developer can fix the corresponding terms quicker and move forward faster. In theory, the more sophisticated the type system, the more useful this parameterization should be for the developer. And in principle, the more traversal strategies, the less committed to one and so the more faithful the (intensional) implementation to the (extensional) type system.

KEYWORDS

type systems, type errors, traversal strategies

ACM Reference Format:
Olivier Danvy, Risa Shindo, and Quang-Trung Ta. 2019. The Type Errors Of Our Ways: A Quantitative Approach. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

As often pointed out by Bob Harper, a type system is a declarative logical construct that says nothing about type errors. And indeed a type system such as OCaml's has the extensional property that inferring the type of a well-typed term yields the most general type accounting for this term, but it says nothing about ill-typed terms, except that they are ill typed.

To infer the type of a term, the type system is operationalized into an algorithm. This algorithm implements a strategy and traverses subterms, e.g., from left to right or from right to left, etc. It is sound if the type it infers is the most general one.

More often than not, however, our programs are ill typed, and this is where the algorithm can be useful in that it can say something about an ill-typed term in the form of an error message. If there are several errors in a program, the one that is encountered first depends on the traversal strategy implemented by the algorithm.

And that is what Bob Harper points out when attending talks about finding the source of type errors, no matter how appreciated the findings in practice: the finding process has nothing to do with the type system in its declarative abstraction and everything to do with the traversal strategy implemented by the algorithm in its operational concretization. Indeed another algorithm that is just as sound, but that traverses the given term in another order is likely to find another type error, should there be several in the given term [McAdam 2002]. Errors are out of the scope of logic – if a property doesn't hold, it doesn't hold – and error messages are the province of implementations.

What about the traversal orders of the dynamic semantics (for evaluating a term) and of the static semantics (for inferring the type of a term), say, for ML? Should they be the same? For a well-typed term, it does not matter because of the extensional property of the type system and of the soundness its implementation. But what about an ill-typed term? Of course, the question is somewhat moot since, for example, both branches of a conditional expression are traversed by the type inferencer and only one is traversed by the evaluator, but still, the answer is no in practice. For example, OCaml evaluates from right to left but infers types from left to right:

```
# (Array.get [||] 0, 1/0);;
Exception: Division_by_zero.

# (succ 'x', pred "y");;
Characters 6-9:
  (succ 'x', pred "y");;
        ^^^
Error: This expression has type char
       but an expression was expected of type int
```

In the first interaction, evaluating either component of a pair raises an error, and the right-most error is raised, since OCaml evaluates from right to left. In the second interaction, inferring the type of either component of a pair raises an error, and the left-most error is raised, indicating that OCaml infers types from left to right, a

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

mismatch that does not seem to have bothered anyone so far. Not that it should: as long as the type inferencer is sound, its traversal order is inconsequential, in principle.

Against this backdrop, we propose to parameterize the type-inference algorithm with a traversal strategy and to offer the user another round of type inference with another strategy in case of type error. Also, in our implementation, we extend the default behaviour of OCaml from stopping at the first type error to continue the type inference in order to emit more error messages instead of just the first one, as in the implementations of Standard ML (see Section 5).

So for example, revisiting the type-incorrect example just above, the user can receive a first round of error messages corresponding to left-to-right traversal and pointing out that (a) the expression `'x'` has type char but was expected to have type `int`, and (b) the expression `"y"` has type string but was expected to have type `int`, and then, on demand, a second round corresponding to right-to-left traversal and pointing out that (a) the expression `"y"` has type string but was expected to have type `int`, and (b) the expression `'x'` has type char but was expected to have type `int`.

Here is another pathological term:

```
fun x -> (x 0, x ^ "", not x)
```

Both a left-to-right and a right-to-left traversal concur that `x ^ ""` is ill-typed, but for different reasons: one because `x` has type `int -> 'a` and the other because `x` has type `bool`. The conjunction of these two error messages indicates that there is more than one type error.

Here is a last pathological term:

```
let f x = (succ x z, z)
```

A left-to-right traversal yields a type error to the effect that `succ x` is of type `int` and therefore cannot be applied, and a right-to-left traversal indicates that `z` is undeclared. This rattling second error message is easier to address than the first.

Overall, our hope is that

- one of the error messages will be simpler to understand than the first (or than the previous ones), and/or
- the error messages will synergize towards understanding not just what is wrong but how to fix it.

As reviewed in Section 5, a lot of implementation work has gone into issuing understandable error messages: issuing more error messages taps into this quality work.

The rest of this article is organized as follows. Section 2 presents some motivational examples. Section 3 describes our implementation and its methodology. Section 4 illustrates the implementation. Section 5 reviews related work. Section 6 concludes.

2 MOTIVATION

This section presents three examples where multiple type errors pay off: unit-test functions (Section 2.1), continuation-passing style (Section 2.2), and defunctionalization (Section 2.3). The second author's BSc dissertation [Shindo 2019] contains more examples that illustrate her user interface.

2.1 Unit-test functions

A beginning OCaml programmer easily writes the following unit-test function for the reverse function, which is polymorphic:

```
let test_reverse candidate =
  let b0 = (candidate [] = [])
  and b1 = (candidate ["foo"] = ["foo"])
  and b2 = (candidate ['a'; 'b'] = ['b'; 'a'])
  and b3 = (candidate [1; 2; 3] = [3; 2; 1])
  (* etc. *)
  in b0 && b1 && b2 && b3 (* etc. *);;
```

They are quite rebuffed by the error message to the effect that `'a'` has type char but an expression was expected of type string.

Given multiple error messages, however, they are reminded (i.e., made aware concretely) that OCaml only offers let-polymorphism. Henceforth they mindfully write monomorphic unit-test functions, in informed readiness for rank-2 polymorphism down the road:

```
let test_reverse_string candidate =
  let b0 = (candidate [] = [])
  and b1 = (candidate ["foo"] = ["foo"])
  (* etc. *)
  in b0 && b1 (* etc. *);;

let test_reverse_char candidate =
  let b0 = (candidate [] = [])
  and b2 = (candidate ['a'; 'b'] = ['b'; 'a'])
  (* etc. *)
  in b0 && b2 (* etc. *);;

let test_reverse_int candidate =
  let b0 = (candidate [] = [])
  and b3 = (candidate [1; 2; 3] = [3; 2; 1])
  (* etc. *)
  in b0 && b3 (* etc. *);;
```

All told, the multiple error messages have assisted the beginning developer and sped up the development.

2.2 Continuation-passing style

Let us consider the Fibonacci function, given a reference implementation `fib`.

Ostensibly, `fib_cps1` and `fib_cps2` implement two continuation-passing versions of the canonical (and inefficient) Fibonacci function:

- one that, given $n + 2$, first recurses on $n + 1$ and then on n, and
- one that, given $n + 2$, first recurses on n and then on $n + 1$.

To wit:

```
let rec fib_cps1 n k =
  if n < 2
  then n
  else fib_cps1 (n - 1) (fun n1 ->
         fib_cps1 (n - 2) (fun n2 ->
           k (n1 + n2)));;
```

```
let rec fib_cps2 n k =
  if n < 2
  then k n
  else fib_cps2 (n - 2) (fun n2 ->
         fib_cps2 (n - 1) (fun n1 ->
           n1 + n2));;
```

Each of these implementations contains a standard bug when writing CPS programs: the continuation is not applied,

- in the base case, for `fib_cps1`, and
- in the induction step, for `fib_cps2`.

We are also given a unit-test function to verify whether the three implementations agree. This unit-test function uses an initial continuation that reveals the standard bug:

```
let test_fib_cps n =
  let r0 = [fib_lin n]
  and r1 = fib_cps1 n (fun a -> [a])
  and r2 = fib_cps2 n (fun a -> [a])
  in r0 = r1 && r0 = r2;;
```

Due to the standard bug in `fib_cps1` and `fib_cps2`, there are two type errors in the second and third definienses of the unit-test function. Both type errors occur because of the type of `fib_cps1` and of `fib_cps2`, which is `int -> (int -> int) -> int` instead of `int -> (int -> 'a) -> 'a`, an undetected type error.

Now how does the user spot this error?

Answer: by looking at the base case for `fib_cps1`, and by looking at the induction step for `fib_cps2`, which is a purely subjective take. Either way, once the easiest error is spotted in one of the definitions, the user realizes that the same error occurs in the other definition, and so the second error is easy to fix.

All told, the multiple error messages have assisted the developer and sped up the development.

2.3 Defunctionalization

Let us consider the abstract data type of environments:

```
type name = string;;

module type ENV =
sig
  type 'a env
  val empty : 'a env
  val extend : name -> 'a -> 'a env -> 'a env
  val lookup : name -> 'a env -> 'a option
end;;
```

Here is an implementation where the environment is a function:

```
module Env_fun : ENV =
struct
  type 'a env = name -> 'a option

  let empty =
    fun y -> None

  let lookup x e =
    e x

  let extend x d e =
    fun y -> if x = y then Some d else e y
end;;
```

Defunctionalization [Reynolds 1998a] is a program transformation where a function type is replaced by a data type representing each of the function abstractions that will give rise to its inhabitants, together with an apply function that dispatches on the data-type constructors. In a defunctionalized program, each function abstraction is replaced by a data-type construction, and each function application is replaced by a call to the apply function.

In the present case,

- the function space is `name -> 'a option` and
- the two function abstractions giving rise to the inhabitants of the function space are
  - `fun y -> None`, where no variables occur free, and
  - `fun y -> if x = y then Some d else e y`, where `x` (of type `name`), `d` (of type `'a`), and `e` (of type `'a env`) occur free.

Let us represent the function type with the following data type together with its apply function (which, oops, contains a bug):

```
type 'a env =
| E0
| E1 of name * 'a * 'a env

let apply_env e =
  match e with
  | E0 ->
     (fun y -> None)
  | E1 (x, d, e) ->
     (fun y -> if x = y then Some d else e y)
```

The type constructors `E0` and `E1` are parameterized with the free variables of the function abstractions they represent.

More colloquially, rather than having the **`match`**-expression yield a function, `apply_env` is defined with two parameters, making it manifest that a defunctionalized program is first order:

```
let apply_env e y =
  match e with
  | E0 ->
     None
  | E1 (x, d, e) ->
     if x = y then Some d else e y
```

Thus equipped, the defunctionalized implementation of the environment reads as follows, replacing each function abstraction by the corresponding data-type constructor and forgetting to replace

each application by a call to apply_env, which will give rise to a type error:

```
module Env_def : ENV =
struct
  type 'a env = ...

  let apply_env e y = ...

  let empty =
    E0

  let lookup x e =
    e x

  let extend x d e =
    E1 (x, d, e)
end;;
```

To sum up,

- the function type is replaced with a data type,
- an apply function dispatching on the data-type constructors is added,
- each function abstraction is replaced with a data-type construction involving (a) the corresponding data-type constructor and (b) all its free variables, and
- each function application is replaced with a call to the apply function.

The present case study features the common error of forgetting to call the apply function (a) in the definition of apply_env and (b) in the definition of lookup.

Two identical error messages are issued, alerting the developer that they should call the apply function in two places. That fixed, another error message is issued, alerting the developer that apply_env should be defined recursively, since the defunctionalized environment is recursive.

All told, the multiple error messages have assisted the developer and sped up the development.

As analyzed elsewhere [Danvy and Millikin 2009], this example is of independent interest in that the defunctionalized environment is isomorphic to the standard representation of environments as association lists.

3 IMPLEMENTATION

We have implemented our parameterized type inference approach on top of the official compiler of the OCaml programming language [INRIA 2019]. Our customized compiler can be downloaded from the GitHub repository:

https://github.com/type-errors/ocaml,

at the branch 4.06. We mainly customized the module typecore.ml (located at ocaml/typing/typemore.ml) to enable the OCaml compiler to (i) report multiple type errors at the same time and (ii) to apply different traversal orders when inferring the types of sub-expressions.

More specifically, to support the display of multiple type errors, we re-programmed the OCaml compiler to make it continue the type inference of an input program after it discovers a type error. Note that we return an *ill-typed* expression for each incorrectly typed expression discovered to maintain the original structure of the input program.

For supporting different traversal orders when inferring the type of sub-expressions, we identify typical scenarios where these sub-expressions can by independently traversed. For example, the sub-expressions can occur in a tuple, or as the test, consequent, and alternative of a conditional expression, or in the definienses of a **let** expression. For such sub-expressions, we can change the sequencing order of their type inference from left to right, from right to left, or for a random order.

3.1 Background of type inference by the OCaml compiler

In the official OCaml compiler [INRIA 2019], most of the tasks pertaining to type inference are implemented in the module typecore.ml. In this module, the function type_expect is the main function that infers the type of an expression with a given type environment and an expected type. The (simplified) interface of type_expect is as follows, where env is the type environment, and sexp expected_ty are respectively the untyped input expression and its expected type.

```
let type_expect env sexp expected_ty =
  ...
```

The function type_expect above frequently calls auxiliary functions to unify the current type with the expected type of the input expression. If an error occurs during this type unification, these auxiliary functions immediately report it by raising an exception like Error (loc, env, error), where env is the current type environment, loc is the source code location of the considered expression.

For example, the auxiliary function unify_exp_types in the module typecore.ml can raise an exception to report a clashing error (Expr_type_clash) between the current type and the expected type.

```
let unify_exp_types loc env ty expected_ty =
  try
    unify env ty expected_ty
  with
  | Unify trace ->
     raise (Error(loc, env, Expr_type_clash trace))
  | ...
```

If the original OCaml compiler can capture one of these exceptions, it immediately displays the error message and exits the type-inference process.

3.2 Overview of our implementation

Firstly, to prevent the OCaml compiler from exiting at the first type error, we customized the module typecore.ml to manually catch and handle all Error exceptions raised by the type-inference function type_expect. In particular, we delayed all the function calls to type_expect by turning them into thunks (a thunk is a function whose domain is the unit type). Then, we wrote an auxiliary function to force this thunk and catch and report all type errors reported from this evaluation.

For example, given the following function call to type_expect,

```
let exp = type_expect env sexp expected_ty in
...
```

we can wrap the call to type_expect by defining the function thunk. Then, we can call an auxiliary function apply to force this thunk and assign the result to the target expression exp:

```
let thunk () = type_expect env sexp expected_ty in
let exp = apply thunk in
...
```

Our auxiliary function apply is simply implemented as follows:

```
let apply thunk =
  try
    thunk ()
  with
  | Error (loc, env, error) ->
     Location.print_error std_formatter loc;
     report_error env std_formatter error;
     mk_ill_typed_exp env loc;;
```

In this apply function, we wrap the application thunk (), which performs the type inference, in a **try** ... **with** ... expression. Here, apply can catch all the exceptions Error (loc, env, error) raised by the application thunk () and display the corresponding error location and message. Since this exception is only handled here and will not be raised again, our customized OCaml compiler will not exit after detecting the first type error, like the original compiler. Furthermore, apply can also return a special *ill-typed* expression to replace the incorrectly typed one so that our compiler can continue to infer the type of other expressions of the input program and report as many type errors as possible in one go.

Secondly, by delaying and wrapping the type inference, we can easily alter the traversal order of the type-inference algorithm for sub-expressions. In particular, instead of traversing sub-expressions from left to right like the default OCaml type inferencer, now we can flexibly enable the type inferencer to traverse sub-expressions from right to left or in a random order.

In the following, we introduce auxiliary functions that infer the type of a pair or a list of expressions with different traversal order. These auxiliary functions are extensively used in our customized compiler to infer the type of all OCaml expressions.

3.2.1 Type-checking a pair of expressions. When inferring the types of two sub-expressions sexp1 and sexp2 in the pair (sexp1, sexp2), the original OCaml compiler first infers the left-most sub-expression (sexp1), then infer the right-most sub-expression (sexp2).

```
let exp1 = type_expect env sexp1 expected_ty1 in
let exp2 = type_expect env sexp2 expected_ty2 in
...
```

Using our previously mentioned approach, we can easily delay the call to type_expect by introducing the thunks thunk1, thunk2 corresponding to the type inference of sexp1 and sexp2. Then, these two thunks are evaluated by an auxiliary function apply_pair to discover the typed counterparts of sexp1 and sexp2.

```
let thunk1() = type_expect env sexp1 expected_ty1 in
let thunk2() = type_expect env sexp2 expected_ty2 in
let (exp1, exp2) = apply_pair thunk1 thunk2 in
...
```

where the function apply_pair is implemented as follows:

```
let apply_pair thunk1 thunk2 =
  let apply_left_to_right () =
    let exp1 = apply thunk1 in
    let exp2 = apply thunk2 in
    (exp1, exp2) in
  let apply_right_to_left () =
    let exp2 = apply thunk2 in
    let exp1 = apply thunk1 in
    (exp1, exp2) in
  match traversal_order with
  | LeftToRight -> apply_left_to_right ()
  | RightToLeft -> apply_right_to_left ()
  | RandomOrder ->
     if Random.bool () then apply_left_to_right ()
     else apply_right_to_left ();;
```

In the auxiliary function , we can flexibly alter the traversal order when type-checking the sub-expressions of the pair. Here, we capture the traversal order of the type inferencer in the global variable traversal_order, whose value can be LeftToRight, RightToLeft, or RandomOrder. They respectively represent the three strategies to evaluate all sub-expressions of an expression: from left to right, from right to left, or in random order.

A user can initialize the value of this variable by invoking our customized OCaml compiler with one of the three options:

- -type-infer-left-to-right
- -type-infer-right-to-left
- -type-infer-random-order

Based on the traversal order specified by the user, the function apply_pair can flexibly evaluate first the left-most sub-expression (by applying thunk1 first) or the right-most sub-expression (by applying thunk2 first), or randomly any of them (by randomly applying thunk1 or thunk2 first).

3.2.2 Type-checking a list of expressions. Similarly, when inferring the types of each element in a list of expressions, the original OCaml compiler infers the type of the expressions from the head to the tail (left to right) of the list. The illustrated code is as follows, where sexpl and expected_tyl are the list of the candidate expressions and their expected types.

```
let expl = List.map2 (fun e t ->
      type_expect env e t) sexpl expected_tyl in
...
```

Here, we can also easily delay the type inference by using thunks. Then, these thunks are forced by an auxiliary function apply_list to discover the type of their element expressions.

```
let thunkl = List.map2 (fun e t ->
  (fun () ->
     type_expect env e t)) sexpl expected_tyl in
let expl = apply_list thunkl in
...
```

The auxiliary function `apply_list` is implemented as follows, where `shuffle` is a function that randomly reorders elements of a list, and `unshuffle` is the inverse function of `shuffle`, to restore the original order of a shuffled list. These two functions can be generated by a shuffle generator (via the application `generate ()`). For brevity, we do not present the implementation of the shuffle generator here. It can also be referred to in our GitHub repository.

```
let apply_list thunkl =
  match traversal_order with
  | LeftToRight ->
     List.map apply thunkl
  | RightToLeft ->
     List.rev_map apply (List.rev thunkl)
  | RandomOrder ->
     let shuffle, unshuffle = generate () in
     unshuffle (List.map apply (shuffle thunkl));;
```

In the following sections 3.3 and 3.4, we will illustrate the details of our implementation on how to customize the OCaml compiler to infer the type of the `if-then-else` and `tuple` expressions.

3.3 Illustration 1: inferring the type of the `if-then-else` expression

In this section, we illustrate our approach to modify the OCaml compiler to infer the type of conditional expressions. The original code fragment is presented in Figure 1. The new code fragment is presented in Figure 2.

```
match sexp.pexp_desc with
...
| Pexp_ifthenelse(scond, sifso, sifnot) ->
  let cond = type_expect env scond type_bool in
  (match sifnot with
   | None ->
     let ifso = type_expect env sifso type_unit in
     rue {
       exp_desc = Texp_ifthenelse(cond, ifso, None);
       exp_loc = loc; exp_extra = [];
       exp_type = ifso.exp_type;
       exp_attributes = sexp.pexp_attributes;
       exp_env = env
     }
   | Some sifnot ->
     let ifso = type_expect env sifso ty_expected in
     let ifnot = type_expect env sifnot ty_expected in
     unify_exp env ifnot ifso.exp_type;
     re {
       exp_desc = Texp_ifthenelse(cond, ifso, Some ifnot);
       exp_loc = loc; exp_extra = [];
       exp_type = ifso.exp_type;
       exp_attributes = sexp.pexp_attributes;
       exp_env = env
     })
| ...
```

Figure 1: Inferring the type of an if-then-else (original)

Compared to the original code fragment, in our modified version, we turn all the type unifications of the conditional expression, the consequent, and the alternative into thunks (`thunk1`, `thunk2`, `thunk3`). When inferring the type of the conditional expression, the consequent and the alternative, we invoke the function `apply_pair` to evaluate them on a particular sequencing order specified by the user (left-to-right, right-to-left, or random order).

```
match sexp.pexp_desc with
...
| Pexp_ifthenelse(scond, sifso, sifnot) ->
  let cond = type_expect env scond type_bool in
  (match sifnot with
   | None ->
     let ifso = type_expect env sifso type_unit in
     rue {
       exp_desc = Texp_ifthenelse(cond, ifso, None);
       exp_loc = loc; exp_extra = [];
       exp_type = ifso.exp_type;
       exp_attributes = sexp.pexp_attributes;
       exp_env = env
     }
   | Some sifnot ->
     let thunk1 () = type_expect env sifso ty_expected in
     let thunk2 () = type_expect env sifnot ty_expected in
     let (ifso, ifnot) = apply_pair thunk1 thunk2 in
     unify_exp env ifnot ifso.exp_type;
     re {
       exp_desc = Texp_ifthenelse(cond, ifso, Some ifnot);
       exp_loc = loc; exp_extra = [];
       exp_type = ifso.exp_type;
       exp_attributes = sexp.pexp_attributes;
       exp_env = env
     })
| ...
```

Figure 2: Inferring the type of an if-then-else (our new version)

3.4 Illustration 2: inferring the type of the `tuple` expression

In this section, we illustrate our approach to modify the OCaml compiler to infer the type of tuples. The original fragment code is presented in Figure 3. The new code is presented in Figure 4.

```
match sexp.pexp_desc with
...
| Pexp_tuple sexpl ->
  assert (List.length sexpl >= 2);
  let subtypes = List.map (fun _ -> newgenvar ()) sexpl in
  let to_unify = newgenty (Ttuple subtypes) in
  unify_exp_types loc env to_unify ty_expected;
  let expl = List.map2 (fun body ty ->
               type_expect env body ty) sexpl subtypes in
  re {
    exp_desc = Texp_tuple expl;
    exp_loc = loc; exp_extra = [];
    exp_type = newty (Ttuple (List.map (fun e ->
                                        e.exp_type) expl));
    exp_attributes = sexp.pexp_attributes;
    exp_env = env
  }
| ...
```

Figure 3: Inferring the type of a tuple (original)

Compared to the original code fragment, in our modified version, we turn all the type unifications of the sub-expressions into a list of thunks (`thunkl`). Then, we invoke the function `apply_list` to force these thunks on a particular sequencing order specified by the user (left-to-right, right-to-left, or random order).

```
match sexp.pexp_desc with
...
| Pexp_tuple sexpl ->
  assert (List.length sexpl >= 2);
  let subtypes = List.map (fun _ -> newgenvar ()) sexpl in
  let to_unify = newgenty (Ttuple subtypes) in
  unify_exp_types loc env to_unify ty_expected;
  let thunkl = List.map2 (fun body ty ->
    (fun () -> type_expect env body ty)) sexpl subtypes in
  let expl = apply_list thunkl in
  re {
    exp_desc = Texp_tuple expl;
    exp_loc = loc; exp_extra = [];
    exp_type = newty (Ttuple (List.map (fun e ->
                                        e.exp_type) expl));
    exp_attributes = sexp.pexp_attributes;
    exp_env = env
  }
| ...
```

Figure 4: Inferring the type of a tuple (our new version)

4 EXPERIMENT

In this section, we illustrate our customized compiler with two OCaml programs that contain tuple and conditional expressions. Similar to the original compiler, the command to run our version is still `ocamlc`. More specifically, this `ocamlc` compiler behaves the same as the original OCaml compiler if it is run without any option, or with existing options supported by original compiler. We also add the new option

`-multi-type-errors`

to enable the compiler to report multiple type errors. Moreover, the evaluation order of sub-expressions can be specified by the following three flags:

- `-type-infer-left-to-right`
- `-type-infer-right-to-left`
- `-type-infer-random-order`

For the time being, the two experiments are meant to illustrate the options of the compiler more than their practical usefulness to accelerate the software cycle of getting a program to be accepted by the type inferencer.

4.1 Experiment with the `tuple` expression

In this experiment, we test our compiler with a program named `tuple.ml` in Figure 5 that is meant to return a tuple:

```
1 let tuple x =
2   (x 1, x + 2, x && true,  x ^ "");;
```

Figure 5: Content of the file tuple.ml

4.1.1 Type error reported by the original compiler. The original OCaml compiler reports only one type error:

```
$ ocamlc tuple.ml

File "tuple.ml", line 2, characters 8-9:
Error: This expression has type int -> 'a
       but an expression was expected of type int
```

4.1.2 Multiple type errors reported by our customized compiler, from left to right. With the two options `-multi-type-errors` and `-type-infer-left-to-right`, our compiler reports 3 type errors:

```
$ ocamlc tuple.ml -multi-type-errors -type-infer-left-to-right

File "tuple.ml", line 2, characters 8-9:
Error: This expression has type int -> 'a
       but an expression was expected of type int

File "tuple.ml", line 2, characters 15-16:
Error: This expression has type int -> 'a
       but an expression was expected of type bool

File "tuple.ml", line 2, characters 27-28:
Error: This expression has type int -> 'a
```

4.1.3 Multiple type errors reported by our customized compiler, from right to left. With the two options `-multi-type-errors` and `-type-infer-right-to-left`, our compiler reports 3 type errors:

```
$ ocamlc tuple.ml -multi-type-errors -type-infer-right-to-left

File "tuple.ml", line 2, characters 15-16:
Error: This expression has type string
       but an expression was expected of type bool

File "tuple.ml", line 2, characters 8-9:
Error: This expression has type string
       but an expression was expected of type int

File "tuple.ml", line 2, characters 3-4:
Error: This expression has type string
       This is not a function; it cannot be applied.
```

4.1.4 Multiple type errors reported by our customized compiler, in random order. We illustrate several runs of our customized compiler in random traversal order with the options `-multi-type-errors` and `-type-infer-random-order` as follows.

– Random order, first run:

```
$ ocamlc tuple.ml -multi-type-errors -type-infer-random-order

File "tuple.ml", line 2, characters 27-28:
Error: This expression has type int -> 'a
       but an expression was expected of type string

File "tuple.ml", line 2, characters 8-9:
Error: This expression has type int -> 'a
       but an expression was expected of type int

File "tuple.ml", line 2, characters 15-16:
Error: This expression has type int -> 'a
       but an expression was expected of type bool
```

– *Random order, second run:*

```
$ ocamlc tuple.ml -multi-type-errors -type-infer-random-order

File "tuple.ml", line 2, characters 3-4:
Error: This expression has type bool
       This is not a function; it cannot be applied.

File "tuple.ml", line 2, characters 8-9:
Error: This expression has type bool
       but an expression was expected of type int

File "tuple.ml", line 2, characters 27-28:
Error: This expression has type bool
       but an expression was expected of type string
```

– *Random order, third run:*

```
$ ocamlc tuple.ml -multi-type-errors -type-infer-random-order

File "tuple.ml", line 2, characters 15-16:
Error: This expression has type int -> 'a
       but an expression was expected of type bool

File "tuple.ml", line 2, characters 8-9:
Error: This expression has type int -> 'a
       but an expression was expected of type int

File "tuple.ml", line 2, characters 27-28:
Error: This expression has type int -> 'a
       but an expression was expected of type string
```

4.2 Experiment with the `if-then-else` expression

Now, we test our compiler with a program named `max3.ml` in Figure 6 that is meant to return the maximum of three values:

```
1 let max3 a b c =
2   let a = a + 0
3   and b = b ^ ""
4   and c = c && true in
5   if a > b then
6     if a > c then a
7     else c
8   else if b > c then b
9   else c;;
```

Figure 6: Content of the file max3.ml

4.2.1 Type error reported by the original compiler. The original OCaml compiler reports only one type error:

```
$ ocamlc max3.ml

File "max3.ml", line 5, characters 9-10:
Error: This expression has type string
       but an expression was expected of type int
```

4.2.2 Multiple type errors reported by our customized compiler, from left to right. With the two options `-multi-type-errors` and `-type-infer-left-to-right`, our compiler reports 6 type errors:

```
$ ocamlc max3.ml -multi-type-errors -type-infer-left-to-right

File "max3.ml", line 5, characters 9-10:
Error: This expression has type string
       but an expression was expected of type int

File "max3.ml", line 6, characters 11-12:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 7, characters 9-10:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 8, characters 14-15:
Error: This expression has type bool
       but an expression was expected of type string

File "max3.ml", line 8, characters 21-22:
Error: This expression has type string
       but an expression was expected of type int

File "max3.ml", line 9, characters 7-8:
Error: This expression has type bool
       but an expression was expected of type int
```

4.2.3 Multiple type errors reported by our customized compiler, from right to left. With the two options `-multi-type-errors` and `-type-infer-right-to-left`, our compiler reports 5 type errors:

```
$ ocamlc max3.ml -multi-type-errors -type-infer-right-to-left

File "max3.ml", line 8, characters 21-22:
Error: This expression has type string
       but an expression was expected of type bool

File "max3.ml", line 8, characters 14-15:
Error: This expression has type bool
       but an expression was expected of type string

File "max3.ml", line 6, characters 18-19:
Error: This expression has type int
       but an expression was expected of type bool

File "max3.ml", line 6, characters 11-12:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 5, characters 9-10:
Error: This expression has type string
       but an expression was expected of type int
```

4.2.4 Multiple type errors reported by our customized compiler, in random order. We illustrate several runs of our customized compiler in random traversal order with the options `-multi-type-errors` and `-type-infer-random-order` as follows.

– *Random order, first run:*

```
$ ocamlc max3.ml -multi-type-errors -type-infer-random-order

File "max3.ml", line 8, characters 14-15:
Error: This expression has type bool
       but an expression was expected of type string

File "max3.ml", line 9, characters 7-8:
Error: This expression has type bool
       but an expression was expected of type string

File "max3.ml", line 5, characters 9-10:
Error: This expression has type string
       but an expression was expected of type int

File "max3.ml", line 6, characters 18-19:
Error: This expression has type int
       but an expression was expected of type string

File "max3.ml", line 6, characters 11-12:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 7, characters 9-10:
Error: This expression has type bool
       but an expression was expected of type string
```

– *Random order, second run:*

```
$ ocamlc max3.ml -multi-type-errors -type-infer-random-order

File "max3.ml", line 5, characters 9-10:
Error: This expression has type string
       but an expression was expected of type int

File "max3.ml", line 8, characters 14-15:
Error: This expression has type bool
       but an expression was expected of type string

File "max3.ml", line 8, characters 21-22:
Error: This expression has type string
       but an expression was expected of type bool

File "max3.ml", line 6, characters 11-12:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 6, characters 18-19:
Error: This expression has type int
       but an expression was expected of type bool
```

– *Random order, third run:*

```
$ ocamlc max3.ml -multi-type-errors  -type-infer-random-order

File "max3.ml", line 7, characters 9-10:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 6, characters 11-12:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 8, characters 21-22:
Error: This expression has type string
       but an expression was expected of type int

File "max3.ml", line 9, characters 7-8:
Error: This expression has type bool
       but an expression was expected of type int

File "max3.ml", line 8, characters 14-15:
Error: This expression has type bool
       but an expression was expected of type string

File "max3.ml", line 5, characters 9-10:
Error: This expression has type string
       but an expression was expected of type int
```

5 RELATED WORK

Reporting syntax errors meaningfully appears to be a solved problem, witness the sparsity of research papers about finding the source of these errors. That is however not the case for type errors because a type inferencer is not solely structural over an abstract-syntax tree: it also entails solving constraints (here, type equations), which is a non-local activity. However, since inferring the type of an ML term always gives the most general type, it does not matter in which order type inference proceeds. Regarding type errors, it was Bruce McAdam's thesis that some orders elicit more meaningful error messages than others [2002]. Perhaps the most visible way this freedom has been exploited is in Pottier and Rémy's work on staging type inference into (1) generating constraints and (2) solving these constraints [2005]. Reflecting this freedom, Hage and Heeren have designed a language for specifying strategies for solving constraints [2009], which they use to diagnose type errors. So finding the source of type errors requires one to keep track of dependencies as the constraint solver proceeds [Wand 1986], to give rise to an explanation space [Beaven and Stansifer 1993]. Alternatively, one can extend the type system to capture all the type errors in one derivation [Neubauer and Thiemann 2003].

There are many ways a constraint solver can proceed and therefore there are many ways errors can be reported. How can these reports be understood more easily? We do not believe that one solving strategy is better than another in this respect in general and therefore we propose to unleash several strategies on demand in order to emit several error messages in several contexts, on the off chance that some will be clearer than others and/or that their conjunction will be more informative. If not, the user is no worse off, though they might need an error assistant to order and classify error messages. In that sense, our approach should synergize with previous work, e.g., Haack and Wells's slice-based approach [2003], Lerner, Flower, Grossman, and Chambers's oracle-based approach [2007], and Pavlinovic, King, and Wies's ranking approach [2014]. Still our closest related work is Chen, Erwig, and Smeltzer's [2017]: they combine strategies that are known to work well in order to improve the accuracy of type error messages. Their approach is qualitative and builds on much Haskell expertise, whereas ours is quantitative and is applicable to any type system.

Conceptually, one learns from one's own mistakes, and understanding the errors of one's ways is often said to be A Good Thing. Still, some programmers care more about getting the type inferencer to accept their program than about understanding why the current version of their program is ill-typed. These programmers appreciate Lerner et al.'s take on suggesting fixes rather than explaining mistakes. We are curious to see whether alternative strategies will give rise to substantively different suggestions for fixes.

On another note, and in passing, we cannot help noticing that Lerner et al.'s and Pavlinovic et al.'s starting point is OCaml, which stops at the first type error and motivated them to continue in order to catch multiple type errors in the same go. Implementations of Standard ML such as Standard ML of New Jersey [Appel and MacQueen 1991] could have been a better starting point in that they do not stop at the first type error, but keep processing the remaining subterm:

```
- fn (x, y) => (((not x, x 1), y "1"), not y);

stdIn:1.17-3.12 Error: operator is not a function
  [tycon mismatch]
  operator: bool
  in expression:
    x 1

stdIn:1.15-3.28 Error: operator and operand don't agree
  [tycon mismatch]
  operator domain: bool
  operand: string -> 'Z
  in expression:
    not y
```

The type inferencer proceeds from left to right. It first infers that `x` has a boolean type, and subsequently reports that it cannot be applied. It then proceeds with inferring that `y` has a function type, and subsequently reports that it cannot be negated. In contrast, OCaml bails out at the first type error:

```
# fun (x, y) -> (((not x, x 1), y "1"), not y);;

Characters 24-25:
  fun (x, y) -> (((not x, x 1), y "1"), not y);;
                          ^
Error: This expression has type bool
       This is not a function; it cannot be applied.
```

Our implementation is not only parameterized by the traversal strategy, it also continues beyond the first type error, which is more practical.

6 CONCLUSION AND PERSPECTIVES

Mathematical logic tells us what it means for a term to have a type, but it tells us nothing about what it means for a term to not have a type. And indeed errors are out of the scope of mathematical logic: if a property doesn't hold, it doesn't hold. As programming-language users, we expect to be given meaningful error messages, but that in itself is problematic:

- is our error a bona-fide error, or are we bumping our nose on a limitation of the type system, as in Section 2.1?
- newcomers are in a Catch-22: they do not understand the language yet so how can they understand its error messages?
- the more sophisticated the type system, the more complicated the error messages.

Error messages are the province of implementations, and have been the topic of a considerable amount of work. To this end, implementers equip their data structures with extra information about the source term and display this information in type-error messages. Each implementation, however, has a specific strategy to traverse the source term and to solve type equations, and therefore distinct implementations are likely to emit distinct error messages in distinct contexts.

In this article, we have proposed to parameterize the implementation of a type inferencer with its traversal strategy so that more than one strategy can be used over ill-typed terms and a wider variety of error messages can be emitted on demand. This way, the user can be given not one but several error messages in several contexts that correspond to several strategies to infer the type of their program, reflecting the declarative nature of the type system where order does not matter. Our thesis is simple: if some of these error messages are more immediately understandable than the others, the developer can straighten out (parts of) their program and move forward faster, which matters if they (and if not them, their boss) are more interested to have a final version that works than to understand why intermediate versions did not work. As for the other developers, having several views of their type errors might help them out of their quagmire. Accordingly, the more sophisticated the type system, the more useful this parameterization should be, both for the implementer and for the developer: if the developer is presented with multiple error messages, it should be less critical for the implementer to bend over backwards in order to deliver single error messages that are understandable. And last but not least: the idea of parameterizing the implementation of a type inferencer with its traversal strategy to emit more error messages on demand opens the door to revisiting previous work about type errors to tap their power.

Acknowledgments: Thanks are due to the anonymous reviewers for perceptive comments.

REFERENCES

Andrew W. Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming (Lecture Notes in Computer Science)*, Jan Małuszyński and Martin Wirsing (Eds.). Springer-Verlag, Passau, Germany, 1–13.

Mike Beaven and Ryan Stansifer. 1993. Explaining Type Errors in Polymorphic Languages. *ACM Letters on Programming Languages and Systems* 2, 1 (1993), 17–30.

Sheng Chen, Martin Erwig, and Karl Smeltzer. 2017. Exploiting Diversity in Type Checkers for Better Error Messages. *Journal of Visual Languages and Computing* 39 (2017), 10–21.

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at Work. *Science of Computer Programming* 74, 8 (2009), 534–549.

Christian Haack and Joe B. Wells. 2003. Type Error Slicing in Implicitly Typed Higher-Order Languages. In *European Symposium on Programming (ESOP)*. 284–301.

Jurriaan Hage and Heeren Bastiaan. 2009. Strategies for Solving Constraints in Type and Effect Systems. In *Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008) (Electronic Notes in Theoretical Computer Science)*, Alessandro Aldini, Maurice H. ter Beek, and Fabio Gadducci (Eds.), Vol. 236. Bertinoro, Italy, 163–183.

INRIA. 2019. The core OCaml system: compilers, runtime system, base libraries. https://github.com/ocaml/ocaml Accessed: 2019-06-11.

Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *Conference on Programming Language Design and Implementation (PLDI)*. 425–434.

Bruce McAdam. 2002. *Repairing Type Errors in Functional Programs.* Ph.D. Dissertation. University of Edinburgh, Edinburgh, Scotland. ECS-LFCS-02-427.

Matthias Neubauer and Peter Thiemann. 2003. Discriminative sum types locate the source of type errors. In *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming (ICFP'03) (SIGPLAN Notices, Vol. 38, No. 9)*, Colin Runciman and Olin Shivers (Eds.). ACM Press, Uppsala, Sweden, 15–26.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding minimum type error sources. In *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2014*. 525–542.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, Chapter 10, 389–489.

John C. Reynolds. 1998a. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword [Reynolds 1998b].

John C. Reynolds. 1998b. Definitional Interpreters Revisited. *Higher-Order and Symbolic Computation* 11, 4 (1998), 355–361.

Risa Shindo. 2019. *A New Interface for Reporting Type Errors in OCaml.* BSc thesis. Yale-NUS College, Singapore.

Mitchell Wand. 1986. Finding the Source of Type Errors. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, Mark Scott Johnson and Ravi Sethi (Eds.). ACM Press, St. Petersburg, Florida, 38–43.

Deriving Compositional Random Generators

Agustín Mista
Chalmers University of Technology
Gothenburg, Sweden
mista@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

ABSTRACT

Generating good random values described by algebraic data types is often quite intricate. State-of-the-art tools for synthesizing random generators serve the valuable purpose of helping with this task, while providing different levels of invariants imposed over the generated values. However, they are often not built for composability nor extensibility, a useful feature when the shape of our random data needs to be adapted while testing different properties or sub-systems.

In this work, we develop an extensible framework for deriving compositional generators, which can be easily combined in different ways in order to fit developers' demands using a simple type-level description language. Our framework relies on familiar ideas from the à la Carte technique for writing composable interpreters in Haskell. In particular, we adapt this technique with the machinery required in the scope of random generation, showing how concepts like generation frequency or terminal constructions can also be expresed in the same type-level fashion. We provide an implementation of our ideas, and evaluate its performance using real-world examples.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; *Software maintenance tools.*

KEYWORDS

random testing, type-level programming, Haskell

ACM Reference Format:
Agustín Mista and Alejandro Russo. 2020. Deriving Compositional Random Generators. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Random property-based testing is a powerful technique for finding bugs [1, 10, 11, 16]. In Haskell, *QuickCheck* is the predominant tool for this task [2]. The developers specify (i) the testing properties their systems must fulfill, and (ii) random data generators (or generators for short) for the data types involved at their properties. Then, *QuickCheck* generates random values, and uses them to evaluate the testing properties in search of possible counterexamples, which always indicate the presence of bugs, either in the program or in the specification of our properties.

Although *QuickCheck* provides default generators for the common base types, like `Int` or `String`, it requires implementing generators for any user-defined data type we want to generate. This process is cumbersome and error prone, and commonly follows closely the shape of our data types. Fortunately, there exists a variety of tools helping with this task, providing different levels of invariants on the generated values as well as automation [6, 8, 14, 18]. We divide the different approaches in two kinds: those which are *manual*, where generators are often able to enforce a wide-range of invariants on the generated data, and those which are *automatic* where the generators can only guarantee lightweight invariants like generating well-typed values.

On the manual side, *Luck* [14] is a domain-specific language for manually writing testing properties and random generators in tandem. It allows obtaining generators specialized to produce random data which is proven to satisfy the preconditions of their corresponding properties. In contrast, on the automatic side, tools like *MegaDeTH* [8, 9], *DRAGEN* [18] and *Feat* [6] allow obtaining random generators automatically at compile time. *MegaDeTH* and *DRAGEN* derive random generators following a simple recipe: to generate a value, they simply pick a random data constructor from our data type with a given probability, and proceed to generate the required sub-terms recursively. *MegaDeTH* pays no attention to the generation frequencies, nor the distribution induced by the derived generator—it just picks among data constructors with uniform probability. Differently, *DRAGEN* analyzes type definitions, and tunes the generation frequencies to match the desired distribution of random values specified by developers. Finally, *Feat* relies on functional enumerations, deriving random generators which sample random values uniformly across the whole search space of values of up to a given size of the data type under consideration. In this work, we focus on automatic approaches to derive generators.

While *MegaDeTH*, *DRAGEN*, and *Feat* provide a useful mechanism for automating the task of writing random generators by hand, they implement a derivation procedure which is often too generic to synthesize useful generators in common scenarios, mostly because *they only consider the structural information encoded in type definitions.* To illustrate this point, consider the following type definition encoding basic HTML pages—inspired by the widely used *html* package:[1]

```
data Html =
    Text String
  | Sing String
  | Tag  String Html
  | Html :+: Html
```

This type allows building HTML pages via four possible data constructors: `Text` is used for plain text values; `Sing` and `Tag` represent singular and paired HTML tags, respectively; whereas the infix `(:+:)` constructor simply concatenates two HTML pages one

IFL'19, September 2019, Singapore
2020.

[1] http://hackage.haskell.org/package/html

after another. Note that the constructors `Tag` and `(:+:)` are recursive, as they have at least one field of type `Html`. Then, the example page `<html>hi<br><b>bye</b></html>` can be encoded with the following `Html` value:

```
Tag "html" (Text "hi" :+: Sing "br" :+: Tag "b" (Text "bye"))
```

In this work, we focus on two scenarios where deriving generators following only the information extracted from type definitions does not work well. The first case is when type definitions are too general (like the case of `Html`) where, as consequence, the generation process leaves a large room for ill-formed values, e.g., invalid HTML pages. For instance, when generating an `Html` value using the `Sing` constructor, it is very likely that an automatically derived generator will choose a random string not corresponding to any valid HTML singular tag. In such situations, a common practice is to rely on existing abstract interfaces to generate random values—such interfaces are often designed to preserve our desired invariants. As an example, consider that our `Html` data type comes equipped with the following abstract interface:

```
br :: Html
bold :: Html → Html
list :: [Html] → Html
(⟨+⟩) :: Html → Html → Html
```

These high-level combinators let us represent structured HTML constructions like line breaks (`br`), bold blocks (`bold`), unordered lists (`list`) and concatenation of values one below another (`⟨+⟩`). This methodology of generating random data employing high-level combinators has shown to be particularly useful in the presence of monadic code [3, 9].

The second scenario that we consider is that where derived generators fails at producing very specific patterns of values which might be needed to trigger bugs. For instance, a function for simplifying `Html` values might be defined to branch differently over complex sequences of `Text` and `(:+:)` constructors:

```
simplify :: Html → Html
simplify (Text t1 :+: Text t2) = ···
simplify (Text t  :+: x :+: y) = ···
simplify ···                   = ···
```

(Symbol ··· denote code that is not relevant for the point being made.) Generating values that match, for instance, the pattern `Text t :+: x :+: y` using *DRAGEN* under an uniform distribution will only occur 6% of the times! Clearly, these input pattern matchings should be included as well into our generators, allowing them to produce random values satisfying such inputs. This structural information can help increasing the chances of reaching portions of our code which otherwise would be very difficult to test. Functions pattern matchings often expose interesting relationships between multiple data constructors, a valuable asset for testing complex systems expecting highly structured inputs [13].

Our previous work [17] focuses on extending *DRAGEN*'s generators as well as its predictive approach to include all these extra sources of structural information, namely high-level combinators and functions' input patterns, while allowing tuning the generation parameters based on the developers' demands. In turn, this work focuses on an orthogonal problem: that of *modularity*. In essence, all

(a) Machinery derivation

```
derive [constructors ''Html
       ,interface    ''Html
       ,patterns     'simplify]
```

(b) Generators specification

```
type Html_valid =
     Con "Text" ⊗ 2
   ⊕ Con ":+:"  ⊗ 4
   ⊕ Fun "hr"   ⊗ 3
   ⊕ Fun "bold" ⊗ 2
   ⊕ Fun "list" ⊗ 3
   ⊕ Fun "<+>"  ⊗ 5

type Html_simplify =
     Con "Text"       ⊗ 2
   ⊕ Con "Sing"       ⊗ 1
   ⊕ Con "Tag"        ⊗ 3
   ⊕ Con ":+:"        ⊗ 4
   ⊕ Pat "simplify" 1 ⊗ 3
   ⊕ Pat "simplify" 2 ⊗ 5

genHtml_valid    = genRep @Html_valid
genHtml_simplify = genRep @Html_simplify
```

Figure 1: Usage example of our framework. Two random generators obtained from the same underlying machinery.

the automatic tools cited above work by synthesizing *rigid* monolithic generator definitions. Once derived, these generators have almost no parameters available for adjusting the shape of our random data. Sadly, this is something we might want to do if we need to test different properties or sub-systems using random values generated in slightly different ways. As the reader might appreciate, it can become handy to cherry pick, for each situation, which data constructors, abstract interfaces functions, or functions' input patterns to consider when generating random values.

The contribution of this work is an automated framework for synthesizing compositional random generators, which can be naturally extended to include the extra sources of structural information mentioned above. Using our approach, a user can obtain random generators following different *generation specifications* whenever necessary, all of them built upon the *same* underlying machinery which only needs to be derived *once*.

Figure 1 illustrates a possible usage scenario of our approach. We first invoke a derivation procedure (1a) to extract the structural information of the type `Html` encoded on (i) its data constructors, (ii) its abstract interface, and (iii) the patterns from the function `simplify`. Then, two different generation specifications, namely $\texttt{Html}_{\texttt{valid}}$ and $\texttt{Html}_{\texttt{simplify}}$ can be defined using a simple type-level idiom (1b). Each specification mentions the different sources of structural information to consider, along with (perhaps) their respective generation frequency. Intuitively, $\texttt{Html}_{\texttt{valid}}$ chooses among the constructors `Text` and `:+:`, as well as functions from `Html`'s abstract interface; while $\texttt{Html}_{\texttt{simplify}}$ chooses among all `Html`'s constructors and the patterns of the first and second clauses in the function `simplify`. The syntax used there will be addressed in detail in Sections 3 to 5. Finally, we obtain two concrete random generators following such specifications by writing `genRep @`$\texttt{Html}_{\texttt{valid}}$ and `genRep @`$\texttt{Html}_{\texttt{simplify}}$, respectively.

The main contribution of this paper are:

- We present an extensible mechanism for representing random values built upon different sources of structural information, adopting ideas from *Data Types à la Carte* [24] (Section 3).

- We develop a modular generation scheme, extending our representation to encode information relevant to the generation process at the type level (Section 4).
- We propose a simple type-level idiom for describing extensible generators, based on the types used to represent the desired shape of our random data (Section 5).
- We provide a Template Haskell tool[2] for automatically deriving all the required machinery presented throughout this paper, and evaluate its generation performance with three real-world case studies and a type-level runtime optimization (Section 6).

Overall, we present a novel technique for reusing automatically derived generators in a composable fashion, in contrast to the usual paradigm of synthesizing rigid, monolithic generators.

2 RANDOM GENERATORS IN HASKELL

In this section, we introduce the common approach for writing random generators in Haskell using *QuickCheck*, along with the motivation for including extra information into our generators, discussing how this could be naively implemented in practice.

In order to provide a common interface for writing generators, *QuickCheck* uses Haskell's overloading mechanism known as *type classes* [26], defining the `Arbitrary` class for random generators as:

```
class Arbitrary a where
  arbitrary :: Gen a
```

where the overloaded symbol `arbitrary::Gen a` denotes a monadic generator for values of type `a`. Using this mechanism, a user can define a sensible random generator for our `Html` data type as follows:

```
instance Arbitrary Html where
  arbitrary = sized gen
    where gen 0 = frequency
                  [(2, Text ⟨$⟩ arbitrary)
                  ,(1, Sing ⟨$⟩ arbitrary)]
          gen d = frequency
                  [(2, Text ⟨$⟩ arbitrary)
                  ,(1, Sing ⟨$⟩ arbitrary)
                  ,(4, Tag  ⟨$⟩ arbitrary ⟨*⟩ gen (d-1))
                  ,(3, (:+:) ⟨$⟩ gen (d-1)  ⟨*⟩ gen (d-1))]
```

At the top level, this definition parameterizes the generation process using *QuickCheck*'s `sized` combinator, which lets us build our generator via an auxiliary, locally defined function `gen :: Int → Gen Html`. The `Int` passed to `gen` is known as the *generation size*, and is threaded seamlessly by *QuickCheck* on each call to `arbitrary`. We use this parameter to limit the maximum amount of recursive calls that our generator can perform, and thus the maximum depth of the generated values. If the generation size is positive (case `gen d`), our generator picks a random `Html` constructor with a given generation frequency (denoted here by the arbitrarily chosen numbers 2, 1, 4 and 3) using *QuickCheck*'s `frequency` combinator. Then, our generator proceeds to fill its fields using randomly generated subterms—here using Haskell's applicative notation [15] and the default `Arbitrary` instance for `String`s. For the case of the recursive sub-terms, this generator simply calls the function `gen` recursively with a smaller depth limit (`gen (d-1)`). This process repeats until we reach the base case (`gen 0`) on each recursive sub-term. At this point, our generator is limited to pick only among terminal `Html` constructors, hence ending the generation process.

As one can observe, the previous definition is quite mechanical, and depends only on the generation frequencies we choose for each constructor. This simple generation procedure is the one used by tools like *MegaDeTH* or *DRAGEN* when synthesizing generators.

2.1 Abstract Interfaces

A common choice when implementing abstract data types is to transfer the responsibility of preserving their invariants to the functions on their abstract interface. Take for example our `Html` data type. Instead of defining a different constructor for each possible HTML construction, we opted for a small generic representation that can be extended with a set of high-level combinators:

```
br :: Html
br = Sing "br"

bold :: Html → Html
bold = Tag "b"

list :: [Html] → Html
list [] = Text "empty list"
list xs = Tag "ul" (foldl1 (:+:) (Tag "li" ⟨$⟩ xs))

(⟨+⟩) :: Html → Html → Html
(⟨+⟩) x y = x :+: br :+: y
```

Note how difficult it would be to generate random values containing, for example, structurally valid HTML lists, if we only consider the structural information encoded in our `Html` type definition. After all, much of the valid structure of HTML has been encoded on its abstract interface.

A synthesized generator could easily contemplate this structural information by creating random values arising from applying such functions to randomly generated inputs:

```
instance Arbitrary Html where
  arbitrary = ···
    frequency
    [...
    ,(1, pure br)
    ,(5, bold ⟨$⟩ gen (d-1))
    ,(2, list ⟨$⟩ listOf (gen (d-1)))
    ,(3, (⟨+⟩) ⟨$⟩ gen (d-1) ⟨*⟩ gen (d-1))]
```

where `(...)` represents the rest of the code of the random generator introduced before. From now on, we will refer to each choice given to the `frequency` combinator as a different *random construction*, since we are not considering generating only single data constructors anymore, but more general value fragments.

2.2 Functions' Pattern Matchings

A different challenge appears when we try to test functions involving complex pattern matchings. Consider, for instance, the full definition of the function `simplify` introduced in Section 1:

[2] Available at https://github.com/OctopiChalmers/dragen2

```
simplify :: Html → Html
simplify (Text t1 :+: Text t2) = Text (t1 ++ t2)
simplify (Text t  :+: x :+: y) =
  simplify (Text t :+: simplify (x :+: y))
simplify (x :+: y) = simplify x :+: simplify y
simplify (Tag t x) = Tag t (simplify x)
simplify x = x
```

This function traverses `Html` values, joining together every contiguous pair of `Text` constructors. Ideally, we would like to put approximately the same testing effort into each clause of `simplify`, or perhaps even more to the first two ones, since those are the ones performing actual simplifications. However, these two clauses are the most difficult ones to test in practice! The probability of generating a random value satisfying nested patterns decreases multiplicatively with the number of constructors we simultaneously pattern match against. In our tests, we were not able to exercise any of these two patterns more than 6% of the overall testing time, using random generators derived using both *MegaDeTH* and *DRAGEN*. As expected, most of the random test cases were exercising the simplest (and rather uninteresting) patterns of this function.

To solve this issue, we could opt to consider each complex pattern as a new kind of random construction. In this light, we can simply generate values satisfying patterns directly by returning their corresponding expressions, where each variable or wildcard pattern is filled using a random sub-expression:

```
instance Arbitrary Html where
  arbitrary = ···
    frequency
    [ ...
    , (2, do t1 ← arbitrary;  t2 ← arbitrary;
             return (Text t1 :+: Text t2))
    , (4, do t ← arbitrary;  x ← gen (d-1);  y ← gen (d-1);
             return (Text t :+: x :+: y))]
```

While the ideas presented in this section are plausible, accumulating cruft from different sources of structural information into a single, global `Arbitrary` instance is unwieldy, especially if we consider that some random constructions might not be relevant or desired in many cases, e.g., generating the patterns of the function `simplify` might only be useful when testing properties involving such function, and nowhere else.

In contrast, the following sections of this paper present our extensible approach for deriving generators, where the required machinery is derived once, and each variant of our random generators is expressed on a per-case basis.

3 MODULAR RANDOM CONSTRUCTIONS

This section introduces a unified representation for the different constructions we might want to consider when generating random values. The key idea of this work is to lift each different source of structural information to the type level. In this light, the shape of our random data is determined entirely by the types we use to represent it during the generation process.

For this purpose, we will use a set of simple "open" *representation types*, each one encoding a single random construction from our *target* data type, i.e., the actual data type we want to randomly generate. These types can be (i) combined in several ways depending on the desired shape of our test data (applying the familiar à la Carte technique); (ii) randomly generated (see Section 4); and finally, (iii) transformed to the corresponding values of our target data type automatically. This representation can be automatically derived from our source code at compile time, relieving programmers of the burden of manually implementing the required machinery.

3.1 Representing Data Constructors

When generating values of algebraic data types, the simplest piece of meaningful information we ought to consider is the one given by each one of its constructors. In this light, each constructor of our target type can be represented using a single-constructor data type. Recalling our `Html` example, its constructors can be represented as:

```
data Con_Text r = Mk_Text String
data Con_Sing r = Mk_Sing String
data Con_Tag  r = Mk_Tag  String r
data Con_(:+:) r = Mk_(:+:) r r
```

Each representation type has the same fields as its corresponding constructor, except for the recursive ones which are abstracted away using a type parameter `r`. This parametricity lets us leave the type of recursive sub-terms unspecified until we have decided the final shape of our random data. Then, for instance, the value `Mk_Tag "div" x :: Con_Tag r` represents the `Html` value `Tag "div" x`, for some sub-term `x :: r` that can be transformed to `Html` as well. Note how these representations types encode the minimum amount of information they need, leaving everything else unspecified.

An important property of these parametric representations is that, in most cases, they form a functor over its type parameter, thus we can use Haskell's **deriving** mechanism to obtain suitable `Functor` instances for free—this will be useful for the next steps.

The next building block of our approach consists of providing a mapping from each constructor representation to its corresponding target value, provided that each recursive sub-term has already been translated to its corresponding target value. This notion is often referred as an *F-Algebra* over the functor used to represent each different construction. Accordingly, to represent this mapping, we will define a type class `Algebra` with a single method `alg` as follows:

```
class Functor f ⇒ Algebra f a | f → a where
  alg :: f a → a
```

where `f` is the functor type used to represent a construction of the target type `a`. The functional dependency `f → a` helps the type system to solve type of the type variable `a`, which appears free on the right hand side of the ⇒. This means that, every representation type `f` will uniquely determine its target type `a`. Then, we need to instantiate this type class for each data constructor representation we are considering, providing an appropriate implementation for the overloaded `alg` function:

```
instance Algebra Con_Text Html where
  alg (Mk_Text x) = Text x
instance Algebra Con_Sing Html where
  alg (Mk_Sing x) = Sing x
```

```
instance Algebra Con_Tag Html where
  alg (Mk_Tag t x) = Tag t x
instance Algebra Con_(:+:) Html where
  alg (Mk_(:+:) x y) = x :+: y
```

There, we simply transform each constructor representation into its corresponding data constructor, piping its fields unchanged.

3.2 Composing Representations

So far we have seen how to represent each data constructor of our `Html` data type independently. In order to represent interesting values, we need to be able to combine single representations into (possibly complex) composite ones. For this purpose, we will define a functor type $\oplus$ to encode the choice between two given representations:

```
data ((f :: * → *) ⊕ (g :: * → *)) r = In_L (f r) | In_R (g r)
```

This infix type-level operator let us combine two representations `f` and `g` into a composite one `f ⊕ g`, encoding either a value drawn from `f` (via the In_L constructor) or a value drawn from `g` (via the In_R constructor). This operator works pretty much in the same way as Haskell's `Either` data type, except that, instead of combining two base types, it works combining two *parametric type constructors*, hence the kind signature $* \to *$ in both `f` and `g`. For instance, the type $\text{Con}_{\text{Text}} \oplus \text{Con}_{\text{Tag}}$ encodes values representing either plain text HTMLs or paired tags. Such values can be constructed using the injections In_L and In_R on each case, respectively.

The next step consists of providing a mapping from composite representations to target types, provided that each component of can be translated to the same target type:

```
instance (Algebra f a, Algebra g a) ⇒ Algebra (f ⊕ g) a where
  alg (In_L fa) = alg fa
  alg (In_R ga) = alg ga
```

There, we use the appropriate `Algebra` instance of the inner representation, based on the injection used to create the composite value.

Worth remarking, the order in which we associate each operand of $\oplus$ results semantically irrelevant. However, in practice, associativity takes as dramatic role when it comes to generation speed. This phenomenon is addressed in detail in Section 6.

3.3 Tying the Knot

Even though we have already seen how to encode single and composite representations for our target data types, there is a piece of machinery still missing: our representations are not recursive, but parametric on its recursive fields. We can think of them as a encoding a *single layer* of our target data. In order to represent recursive values, we need to close them *tying the knot* recursively, i.e., once we have fixed a suitable representation type for our target data, each one of its recursive fields has to be instantiated with itself. This can be easily achieved by using a type-level fixed point operator:

```
data Fix (f :: * → *) = Fix {unFix :: f (Fix f)}
```

Given a representation type `f` of kind $* \to *$, the type `Fix f` instantiates each recursive field of `f` with `Fix f`, closing the definition of `f` into itself—thus the kind of `Fix f` results $*$.

In general, if a type `f` is used to represent a given target type, we will refer to `Fix f` as a *final representation*, since it cannot be further combined or extended—the $\oplus$ operator has to be applied *within* the `Fix` type constructor.

The effect of a fixed point combinator is easier to interpret with an example. Let us imagine we want to represent our `Html` data type using all of its data constructors, employing the following type:

```
type Html' = Con_Text ⊕ Con_Sing ⊕ Con_Tag ⊕ Con_(:+:)
```

Then, for instance, the value `x = Text "hi" :+: Sing "hr" :: Html` can be represented with a value `x' :: Fix Html'` as:

```
x' = Fix (In_R (In_R (In_R (Mk_(:+:)
       (Fix (In_L (Mk_Text "hi")))
       (Fix (In_R (In_L (Mk_Sing "hr"))))))))
```

where the sequences of In_L and In_R data constructors *inject* each value from an individual representation into the appropriate position of our composite representation `Html'`.

Finally, we can define a generic function `eval` to evaluate any value of a final representation type `Fix f` into its corresponding value of the target type `a` as follows:

```
eval :: Algebra f a ⇒ Fix f → a
eval = alg ∘ fmap eval ∘ unFix
```

This function exploits the `Functor` structure of our representations, unwrapping the fixed points and mapping their algebras to the result of evaluating recursively each recursive sub-term.

In our particular example, this function satisfies `eval x' == x`. More specifically, the types `Html` and `Fix Html'` are in fact isomorphic, with `eval` as the witness of one side of this isomorphism—though this is not the case for any arbitrary representation.

3.4 Representing Additional Constructions

The representation mechanism we have developed so far let us determine the shape of our target data based on the type we use to represent its constructors. However, it is hardly useful for random testing, as the values we can represent are still quite unstructured. It is not until we start considering more complex constructions that this approach becomes particularly appealing.

3.4.1 Abstract Interfaces. Let us consider the case of generating values obtained by abstract interface functions. If we recall our `Html` example, the functions on its abstract interface can be used to obtain `Html` values based on different input arguments. Fortunately, it is easy to extend our approach to incorporate the interesting structure arising from these functions into our framework. As before, we start by defining a set of open data types to encode each function as a random construction:

```
data Fun_br    r = Mk_br
data Fun_bold  r = Mk_bold r
data Fun_list  r = Mk_list [r]
data Fun_⟨+⟩   r = Mk_⟨+⟩ r r
```

Each data type represents a value resulting from evaluating its corresponding function, using the values encoded on its fields as input arguments. Once again, we replace each recursive field (representing a recursive input argument) with a type parameter `r` in order to leave the type of the recursive sub-terms unspecified until we have decided the final shape of our data.

By representing values obtained from function application this way, we are not performing any actual computation—we simply store the functions' input arguments. Instead, these functions are evaluated when transforming each representation into its target type, by the means of an `Algebra`:

```
instance Algebra Fun_br Html where
  alg Mk_br = br
instance Algebra Fun_bold Html where
  alg (Mk_bold x) = bold x
instance Algebra Fun_list Html where
  alg (Mk_list xs) = list xs
instance Algebra Fun_⟨+⟩ Html where
  alg (Mk_⟨+⟩ x y) = x⟨+⟩y
```

Where we simply return the result of evaluating each corresponding function, using its representation fields as an input arguments.

It is important to remark that this approach inherits any possible downside from the functions we use to represent our target data. In particular, representing non-terminating functions might produce a non-terminating behavior when calling to the `eval` function.

3.4.2 Functions' Pattern Matchings. The second source of structural information that we consider in this work is the one present in functions' pattern matchings. If we recall to our `simplify` function, we can observe it has two complex, non-trivial patterns that we might want to satisfy when generating random values. We can extend our approach in order to represent these patterns as well. We start by defining data types for each one of them, this time using the fields of each single data constructor to encode the free pattern variables (or wildcards) appearing on its corresponding pattern:

```
data Pat_simplify#1 r = Mk_simplify#1 String String
data Pat_simplify#2 r = Mk_simplify#2 String r r
```

where the number after the `#` distinguishes the different patterns from the function `simplify` by the index of the clause they belong to. As before, we abstract away every recursive field (corresponding to a recursive pattern variable or wildcard) with a type variable `r`.

Then, the `Algebra` instance of each pattern will expand each representation into the corresponding target value resembling such pattern, where each pattern variable gets instantiated using the values stored in its representation field:

```
instance Algebra Pat_simplify#1 Html where
  alg (Mk_simplify#1 t1 t2) = Text t1 :+: Text t2
instance Algebra Pat_simplify#2 Html where
  alg (Mk_simplify#1 t x y) = Text t :+: x :+: y
```

3.5 Lightweight Invariants for Free!

Using the machinery presented so far, we can represent values of our target data coming from different sources of structural information in a compositional way.

Using this simple mechanism we can obtain values exposing lightweight invariants very easily. For instance, a value of type `Html` might encode invalid HTML pages if we construct them using invalid tags in the process (via the `Sing` or `Tag` constructors). To avoid this, we can explicitly disallow the direct use of the `Sing` and `Tag` constructors, replacing them with safe constructions from its abstract interface. In this light, a value of type:

```
Con_Text ⊕ Con_(:+:) ⊕ Fun_br ⊕ Fun_bold ⊕ Fun_list ⊕ Fun_⟨+⟩
```

always represents a valid HTML page.

Similarly, we can enforce that every `Text` constructor within a value will always appear in pairs of two, by using the following type:

```
Con_Sing ⊕ Con_Tag ⊕ Con_(:+:) ⊕ Pat_simplify#1
```

Since the only way to place a `Text` constructor within a value of this type is via the construction `Pat_simplify#1`, which always contains two consecutive `Text`s.

As a consequence, generating random data exposing such invariants will simply become using an appropriate representation type while generating random values, without having to rely on runtime reinforcements of any sort. The next section introduces a generic way to generate random values from our different representations, extending them with a set of combinators to encode information relevant to the generation process directly at the type level.

4 GENERATING RANDOM CONSTRUCTIONS

So far we have seen how to encode different random constructions representing interesting values from our target data types. Such representations follow a modular approach, where each construction is independent from the rest. This modularity allows us to derive each different construction representation individually, as well to specify the shape of our target data in simple and extensible manner.

In this section, we introduce the machinery required to randomly generate the values encoded using our representations. This step also follows the modular fashion, resulting in a random generation process entirely compositional. In this light, our generators are built from simpler ones (each one representing a single random construction), and are solely based on the types we use to represent the shape of our random data.

Ideally, our aim is to be able to obtain random generators with a behavior similar to the one presented for `Html` in Section 2. If we take a closer look at its definition, there we can observe three factors happening simultaneously:

- We use *QuickCheck*'s generation size to limit the depth of the generated values, reducing it by one on each recursive call of the local auxiliary function `gen`.
- We differentiate between *terminal* and *non-terminal (i.e. recursive) constructors*, picking only among terminal ones when we have reached the maximum depth (case `gen 0`).
- We generate different constructions in a different frequency.

For the rest of this section, we will focus on modeling these aspects in our modular framework, in such a way that does not compromise the compositionality obtained so far.

4.1 Depth-Bounded Modular Generators

The first obstacle that arises when trying to generate random values with a limited depth using our approach is related to modularity. If we recall the random generator for `Html` from Section 2 we can observe that the depth parameter `d` is threaded to the different recursive calls of our generator, always within the scope of the local function `gen`. Since each construction will have an specialized

random generator, we cannot group them as we did before using an internal `gen` function. Instead, we will define a new type for depth-bounded generators, wrapping *QuickCheck*'s `Gen` type with an external parameter representing the maximum recursive depth:

```
type BGen a = Int → Gen a
```

A `BGen` is, essentially, a normal *QuickCheck* `Gen` with the maximum recursive depth as an input parameter. Using this definition, we can generalize *QuickCheck*'s `Arbitrary` class to work with depth-bounded generators simply as follows:

```
class BArbitrary (a :: ∗) where
  barbitrary :: BGen a
```

From now on, we will use this type class as a more flexible substitute of `Arbitrary`, given that now we have two parameters to tune: the maximum recursive depth, and the *QuickCheck* generation size. The former is useful for tuning the overall size of our random data, whereas the latter can be used for tuning the values of the *leaf types*, such as the maximum length of the random strings or the biggest/smallest random integers.

Here we want to remark that, even though we could have used *QuickCheck*'s generation size to simultaneously model the maximum recursive depth and the maximum size of the leaf types, doing so would imply generating random values with a decreasing size as we move deeper within a random value, obtaining for instance, random trees with all zeroes on its leaves, or random lists skewed to be ordered in decreasing order. In addition, one can always obtain a trivial `Arbitrary` instance from a `BArbitrary` one, by setting the maximum depth to be equal to *QuickCheck*'s generation size:

```
instance BArbitrary a ⇒ Arbitrary a where
  arbitrary = sized barbitrary
```

Even though this extension allows *QuickCheck* generators to be depth-aware, here we also need to consider the parametric nature of our representations. In the previous section, we defined each construction representation as being parametric on the type of its recursive sub-terms, as a way to defer this choice until we have specified the final shape of our target data. Hence, each construction representation is of kind $∗ \longrightarrow ∗$. If we want to define our generators in a modular way, we also need to parameterize somehow the generation of the recursive sub-terms! If we look at *QuickCheck*, this library already defines a type class `Arbitrary1` for parametric types of kind $∗ \longrightarrow ∗$, which solves this issue by receiving the generator for the parametric sub-terms as an argument:

```
class Arbitrary1 (f :: ∗ → ∗) where
  liftArbitrary :: Gen a → Gen (f a)
```

Then, we can use this same mechanism for our modular generators, extending `Arbitrary1` to be depth-aware as follows:

```
class BArbitrary1 (f :: ∗ → ∗) where
  liftBGen :: BGen a → BGen (f a)
```

Note the similarities between `Arbitrary1` and `BArbitrary1`. We will use this type class to implement random generators for each construction we are automatically deriving. Recalling our `Html` example, we can define modular random generators for the constructions representing its data constructors as follows:

```
instance BArbitrary1 Con_Text where
  liftBGen bgen d = Mk_Text ⟨$⟩ arbitrary
instance BArbitrary1 Con_Sing where
  liftBGen bgen d = Mk_Sing ⟨$⟩ arbitrary
instance BArbitrary1 Con_Tag where
  liftBGen bgen d = Mk_Tag ⟨$⟩ arbitrary ⟨∗⟩ bgen (d-1)
instance BArbitrary1 Con_(:+:) where
  liftBGen bgen d = Mk_(:+:) ⟨$⟩ bgen (d-1) ⟨∗⟩ bgen (d-1)
```

Note how each instance is defined to be parametric of the maximum depth (using the input integer `d`) and of the random generator used for the recursive sub-terms (using the input generator `bgen`). Every other non-recursive sub-term can be generated using a normal `Arbitrary` instance—we use this to generate random `String`s in the previous definitions.

The rest of our representations can be generated analogously. For example, the `BArbitrary1` instances for $\texttt{Fun}_{\texttt{bold}}$ and $\texttt{Pat}_{\texttt{simplify\#2}}$ are as follows:

```
instance BArbitrary1 Fun_bold where
  liftBGen bgen d = Mk_bold ⟨$⟩ bgen (d-1)
instance BArbitrary1 Pat_simplify#2 where
  liftBGen bgen d =
    Mk_simplify#2 ⟨$⟩ arbitrary ⟨∗⟩ bgen (d-1) ⟨∗⟩ bgen (d-1)
```

Then, having the modular generators for each random construction in place, we can obtain a concrete depth-aware generator (of kind ∗) for any final representation `Fix f` as follows:

```
instance BArbitrary1 f ⇒ BArbitrary (Fix f) where
  barbitrary d = Fix ⟨$⟩ liftBGen barbitrary d
```

There, we use the `BArbitrary1` instance of our representation `f` to generate sub-terms recursively by lifting itself as the parameterized input generator (`liftBGen barbitrary`), wrapping each recursive sub-term with a `Fix` data constructor.

The machinery developed so far lets us generate single random constructions in a modular fashion. However, we still need to develop our generation mechanism a bit further in order to generate composite representations built using the ⊕ operator. This is the objective of the next sub-section.

4.2 Encoding Generation Behavior Using Types

As we have seen so far, generating each representation is rather straightforward: there is only one data constructor to pick, and every field is generated using a mechanical recipe. In our approach, most of the generation complexity is encoded in the random generator for composite representations, built upon the ⊕ operator. Before introducing it, we need to define some additional machinery to encode the notions of terminal construction and generation frequency.

Recalling the random generator for `Html` presented in Section 2, we can observe that the last generation level (see `gen 0`) is constrained to generate values only from the subset of terminal constructions. In order to model this behavior, we will first define a data type `Term` to tag every terminal construction explicitly:

```
data Term (f :: ∗ → ∗) r = Term (f r)
```

Then, if `f` is a terminal construction, the type `Term f ⊕ g` can be interpreted as representing data generated using values drawn both from `f` and `g`, but closed using only values from `f`. Since this data type will not add any semantic information to the represented values, we can define suitable `Algebra` and `BArbitrary1` instances for it simply by delegating the work to the inner type:

```
instance Algebra f a ⇒ Algebra (Term f) a where
  alg (Term f) = alg f
instance BArbitrary1 f ⇒ BArbitrary1 (Term f) where
  liftBGen bgen d = Term ⟨$⟩ liftBGen bgen d
```

Worth mentioning, our approach does not requires the final user to manually specify terminal constructions—a repetitive task which might lead to obscure non-termination errors if a recursive construction is wrongly tagged as terminal. In turn, this information can be easily extracted at derivation time and included implicitly in our refined type-level idiom, described in detail in Section 5.

The next building block of our framework consists in a way of specifying the generation frequency of each construction. For this purpose, we can follow the same reasoning as before, defining a type-level operator ⊗ to explicitly tag the generation frequency of a given representation:

```
data ((f :: ∗ → ∗) ⊗ (n :: Nat)) r = Freq (f r)
```

This operator is parameterized by a type-level natural number `n` (of kind `Nat`) representing the desired generation frequency. In this light, the type `(f⊗3)⊕(g⊗1)` represents data generated using values from both `f` and `g`, where `f` is randomly chosen three times more frequently than `g`. In practice, we defined ⊗ such that it associates more strongly than ⊕, thus avoiding the need of parenthesis in types like the previous one. Analogously as `Term`, the operator ⊗ does not add any semantic information to the values it represents, so we can define its `Algebra` and `BAbitrary1` instance by delegating the work to the inner type as before:

```
instance Algebra f a ⇒ Algebra (f ⊗ n) a where
  alg (Freq f) = alg f
instance BArbitrary1 f ⇒ BArbitrary1 (f ⊗ n) where
  liftBGen bgen d = Freq ⟨$⟩ liftBGen bgen d
```

With these two new type level combinators, `Term` and ⊗, we are now able to express the behavior of our entire generation process based solely on the type we are generating.

In addition to these combinators, we will need to perform some type-level computations based on them in order to define our random generator for composite representations. Consider for instance the following type—expressed using parenthesis for clarity:

```
(f ⊗ 2) ⊕ ((g ⊗ 3) ⊕ (Term h ⊗ 5))
```

Our generation process will traverse this type one combinator at a time, processing each occurrence of ⊕ independently. This means that, in order to select the appropriate generation frequency of each operand we need to calculate the overall sum of frequencies on each side of the ⊕. For this purpose, we rely on Haskell's type-level programming feature known as *type families* [23]. In this light, we can implement a type-level function `FreqOf` to compute the overall sum of frequencies of a given representation type:

```
type family FreqOf (f :: ∗ → ∗) :: Nat where
  FreqOf (f ⊕ g)    = FreqOf f + FreqOf g
  FreqOf (f ⊗ n)    = n ∗ FreqOf f
  FreqOf (Term f) = FreqOf f
  FreqOf _          = 1
```

This type-level function takes a representation type as an input and traverses it recursively, adding up each frequency tag found in the process, and returning a type-level natural number. Note how in the second equation we multiply the frequency encoded in the ⊗ tag with the frequency of the type it is wrapping. This way, the type `((f ⊗ 2) ⊕ g) ⊗ 3` is equivalent to `(f ⊗ 6) ⊕ (g ⊗ 3)`, following the natural intuition for the addition and multiplication operations over natural numbers. Moreover, if a type does not have an explicit frequency, then its generation frequency is defaulted to one.

Furthermore, the last step of our generation process, which only generates terminal constructions, could be seen as considering the non-terminal ones as having generation frequency zero. This way, we can introduce another type-level computation to calculate the *terminal generation frequency* $\texttt{FreqOf}^{\texttt{T}}$ of a given representation:

```
type family FreqOf^T (f :: ∗ → ∗) :: Nat where
  FreqOf^T (f ⊕ g)    = FreqOf^T f + FreqOf^T g
  FreqOf^T (f ⊗ n)    = n ∗ FreqOf^T f
  FreqOf^T (Term f) = FreqOf f
  FreqOf^T _          = 0
```

Similar to `FreqOf`, the type family above traverses its input type adding the terminal frequency of each sub-type. However, $\texttt{FreqOf}^{\texttt{T}}$ only considers the frequency of those representation sub-types that are explicitly tagged as terminal, returning zero in any other case.

Then, using the `Term` and ⊗ combinators introduced at the beginning of this sub-section, along with the previous type-level computations over frequencies, we are finally in position of defining our random generator for composite representations:

```
instance (BArbitrary1 f, BArbitrary1 g)
  ⇒ BArbitrary1 (f ⊕ g) where
  liftBGen bgen d =
    if d > 0
    then frequency
      [(freqVal @(FreqOf f), In_L ⟨$⟩ liftBGen bgen d)
      ,(freqVal @(FreqOf g), In_R ⟨$⟩ liftBGen bgen d)]
    else frequency
      [(freqVal @(FreqOf^T f), In_L ⟨$⟩ liftBGen bgen d)
      ,(freqVal @(FreqOf^T g), In_R ⟨$⟩ liftBGen bgen d)]
```

Like the generator for `Html` introduced in Section 2, this generator branches over the current depth `d`. In the case we can still generate values from any construction (`d > 0`), we will use *QuickCheck*'s `frequency` operation to randomly choose between generating a value of each side of the ⊕, i.e., either a value of `f` or a value of `g`, following the generation frequencies specified for both of them, and wrapping the values with the appropriate injection $\texttt{In}_\texttt{L}$ or $\texttt{In}_\texttt{R}$ on each case. Such frequencies are obtained by *reflecting* the type-level natural values obtained from applying `FreqOf` to both `f` and `g`, using a type-dependent function `freqVal` that returns the number corresponding to the type-level natural value we apply to it:

```
freqVal :: ∀n . KnownNat n ⇒ Int
```

Note that the type of `freqVal` is ambiguous, since it quantifies over every possible known type-level natural value `n`. We use a *visible type application* [7] (employing the `@(...)` syntax) to disambiguate to which natural value we are actually referring to. Then, for instance, the value `freqVal @(FreqOf (f ⊗ 5 ⊕ g ⊗ 4))` evaluates to the concrete value `9 :: Int`.

The **else** clause of our random generator works analogously, except that, this time we only want to generate terminal constructions, hence we use the $\texttt{FreqOf}^{\texttt{T}}$ type family to compute the terminal generation frequency of each operand. If any of $\texttt{FreqOf}^{\texttt{T}}\ \texttt{f}$ or $\texttt{FreqOf}^{\texttt{T}}\ \texttt{g}$ evaluates to zero, it means that such operand does not contain any terminal constructions, and `frequency` will not consider it when generating terminal values.

Moreover, if it happens that both $\texttt{FreqOf}^{\texttt{T}}\ \texttt{f}$ and $\texttt{FreqOf}^{\texttt{T}}\ \texttt{g}$ compute to zero simultaneously, then this will produce a runtime error triggered by the function `frequency`, as it does not have anything with a positive frequency to generate. This kind of exceptions will arise, for example, if we forget to include at least one terminal construction in our final representation—thus leaving the door open for potential infinite generation loops. Fortunately, such runtime exceptions can be caught at compile time. We can define a type constraint `Safe` that ensures we are trying to generate values using a representation with a strictly positive terminal generation frequency—thus containing at least a single terminal construction:

```
type family Safe (f :: ∗ → ∗) :: Constraint where
  Safe f = IsPositive (FreqOf^T f)
type family IsPositive (n :: Nat) :: Constraint where
  IsPositive 0 = TypeError "No terminals"
  IsPositive _ = ()
```

These type families compute the terminal generation frequency of a representation type `f`, returning either a type error, if its result is zero; or, alternatively, an empty constraint `()` that is always trivially satisfied. Finally, we can use this constraint to define a safe generation primitive `genRep` to obtain a concrete depth-bounded generator for every target type `a`, specified using a "safe" representation `f`:

```
genRep :: ∀f a . (BArbitrary1 f, Safe f, Algebra f a) ⇒ BGen a
genRep d = eval ⟨$⟩ barbitrary @(Fix f) d
```

Note how this primitive is also ambiguous in the type used for the representation. This allows us to use a visible type application to obtain values from the same target type but generated using different underlying representations. For instance, we can obtain two different concrete generators of our `Html` type simply by changing its generation representation type as follows:

```
genHtml_valid :: BGen Html
genHtml_valid = genRep @Html_valid

genHtml_simplify :: BGen Html
genHtml_simplify = genRep @Html_simplify
```

where $\texttt{Html}_{\texttt{valid}}$ and $\texttt{Html}_{\texttt{simplify}}$ are the representations types introduced in Figure 1b—the syntax used to define them is completed in the next section.

So far we have seen how to represent and generate values for our target data type by combining different random constructions, as well as a series of type-level combinators to encode the desired generation behavior. The next section refines our type-level machinery in order to provide a simple idiom for defining composable random generators.

5 TYPE-LEVEL GENERATION SPECIFICATIONS

This section introduces refinements to our basic language for describing random generators, making it more flexible and robust in order to fit real-world usage scenarios.

The first problem we face is that of naming conventions. In practice, the actual name used when deriving the representation for each random construction needs to be generated such that it complies with Haskell's syntax, and also that it is *unique within our namespace*. This means that, type names like $\texttt{Fun}_{\langle+\rangle}$ or $\texttt{Pat}_{\texttt{simplify\#1}}$ are, technically, not valid Haskell data type names, thus they will have to be synthesized as something like `Fun_lt_plus_gt_543` and `Pat_simplify_1_325`, where the last sequence of numbers is inserted by Template Haskell to ensure uniqueness.

This naming convention results hard to use, specially if we consider that we do not know the actual type names until they are synthesized during compilation, due to their unique suffixes. Fortunately, it is easy to solve this problem using some type-level machinery. Instead of imposing a naming convention in our derivation tool, we define a set of open type families to hide each kind of construction behind meaningful names:

```
type family Con (c :: Symbol)
type family Fun (f :: Symbol)
type family Pat (p :: Symbol) (n :: Nat)
```

where `Symbol` is the kind of type-level strings in Haskell. Then, our derivation process will synthesize each representation using unique names, along with a type instance of the corresponding type family, i.e., `Con` for data constructors, `Fun` for interface functions, and `Pat` for functions' patterns. For instance, along with the constructions representations $\texttt{Con}_{\texttt{Text}}$, $\texttt{Fun}_{\langle+\rangle}$ and $\texttt{Pat}_{\texttt{simplify\#1}}$, we will automatically derive the following type instances:

```
type instance Con "Text"       = Term Con_Text_123
type instance Fun "<+>"        = Fun_lt_plus_gt_543
type instance Pat "simplify" 1 = Term Pat_simplify_1_325
```

As a result, the end user can simply refer to each particular construction by using these synonyms, e.g., with representation types like `Con "Text" ⊕ Fun "<+>"`. The additional `Nat` type parameter on `Pat` simply identifies each pattern number uniquely.

Moreover, notice how we include the appropriate `Term` tags for each terminal construction automatically—namely `Con "Text"` and `Pat "simplify" 1` in the example above. Since this information is statically available, we can easily extract it during derivation time. This relieves us of the burden of manually identifying and declaring the terminal constructions for every generation specification Additionally, it helps ensuring the static termination guarantees provided by our `Safe` constraint mechanism.

Using the type-level extension presented so far, we are now able to write the generation specifications presented in Figure 1b in a clear and concise way.

5.1 Parametric Target Data Types

So far we have seen how to specify random generators for our simple self-contained `Html` data type. In practice, however, we are often required to write random generators for parametric target data types as well. Consider, for example, the following `Tree` data type definition encoding binary trees with generic information of type `a` in the leaves:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

In order to represent its data constructors, we can follow the same recipe presented in Section 3, but also parameterizing our representations over the type variable `a` as well:

```
data Con_Leaf a r = Mk_Leaf a
data Con_Node a r = Mk_Node r r
```

The rest of the machinery can be derived in the same way as before, carrying this type parameter and including the appropriate `Arbitrary` constraints all along the way:

```
instance Algebra (Con_Leaf a) (Tree a) where ···
instance Algebra (Con_Node a) (Tree a) where ···

instance Arbitrary a ⇒ BArbitrary1 (Con_Leaf a) where ···
instance Arbitrary a ⇒ BArbitrary1 (Con_Node a) where ···
```

Then, instead of carrying this type parameter in our generation specifications, we can avoid it by hiding it behind an existential type:

```
data Some (f :: ∗ → ∗ → ∗) (r :: ∗) = ∀ (a :: ∗) . Some (f a r)
```

The type constructor `Some` is a wrapper for a 2-parametric type that hides the first type variable using an explicit existential quantifier. Note thus that the type parameter `a` does not appears at the left hand side of `Some` on its definition. In this light, when deriving any `Con`, `Fun` or `Pat` type instance, we can use this type wrapper it to hide the additional type parameters of each construction representation:

```
type instance Con "Leaf" = Term (Some Con_Leaf)
type instance Con "Node" = Some Con_Node
```

As a consequence, we can write generation specifications for our `Tree` data type without having to refer to its type parameter anywhere. For instance:

```
type Tree_Spec = Con "Leaf" ⊗ 2
               ⊕ Con "Node" ⊗ 3
```

Instead, we defer handling this type parameter until we actually use it to define a concrete generator. For instance, we can write a concrete generator of `Tree Int` as follows:

```
genIntTree :: BGen (Tree Int)
genIntTree = genRep @(Tree_Spec ◁ Int)
```

Where ◁ is a type family that simply traverses our generation specification, applying the `Int` type to each occurrence of `Some`, thus eliminating the existential type:

```
type family (f :: ∗ → ∗) ◁ (a :: ∗) :: ∗ → ∗ where
  (Some f) ◁ a = f a
  (f ⊕ g)  ◁ a = (f ◁ a) ⊕ (g ◁ a)
  (f ⊗ n)  ◁ a = (f ◁ a) ⊗ n
  (Term f) ◁ a = Term (t ◁ a)
  f        ◁ a = f
```

As a result, in `genIntTree`, the ◁ operator will reduce the type (`Tree_Spec ◁ Int`) to the following concrete type:

```
(Term (Con_Leaf Int) ⊗ 2)  ⊕  ((Con_Node Int) ⊗ 3)
```

Worth mentioning, this approach for handling parametric types can be extended to multi-parametric data types with minor effort.

Along with our automated constructions derivation mechanism, the machinery introduced in this section allows us to specify random generators using a simple type-level specification language.

The next section evaluates our approach in terms of performance using a set of case studies extracted from real-world Haskell implementations, along with an interesting runtime optimization.

6 BENCHMARKS AND OPTIMIZATIONS

The random generation framework presented throughout this paper allows us to write extensible generators in a very concise way. However, this expressiveness comes attached to a perceptible runtime overhead, primarily inherited from the use of Data Types à la Carte—a technique which is not often scrutinized for performance. In this section, we evaluate the implicit cost of composing generators using three real-world case studies, along with a type-level optimization that helps avoiding much of the runtime bureaucracy.

Balanced Representations. As we have shown in Section 4, the random generation process we propose in this paper can be seen as having two phases. First, we generate random values from the representation types used to specify the shape of our data; and then we use their algebras to translate them to the corresponding values of our target data types. In particular, this last step is expected to pattern match repeatedly against the $\mathtt{In_L}$ and $\mathtt{In_R}$ constructors of the ⊕ operators when traversing each construction injection. Because of this, in general, we expect a performance impact with respect to manually-written concrete generators.

As recently analyzed by Kiriyama et al., this slowdown is expected to be linear in the depth of our representation type [12]. In this light, one can drastically reduce the runtime overhead by associating each ⊕ operator in a balanced fashion. So, for instance, instead of writing (`f⊕g⊕h⊕i`), which is implicitly parsed as (`f⊕(g⊕(h⊕i))`); we can associate constructions as ((`f ⊕ g`) ⊕ (`h ⊕ i`)), thus reducing the depth of our representation from four to three levels and, in general, from a $O(n)$ to a $O(log(n))$ complexity in the runtime overhead, where n is the amount of constructions under consideration.

Worth mentioning, this balancing optimization cannot be applied to the original fashion of Data Types à la Carte by Swierstra. This limitation comes from that the linearity of the representation types is required in order to define *smart injections*, allowing users to construct values of such types in an easy way, injecting the appropriate sequences of $\mathtt{In_L}$ and $\mathtt{In_R}$ constructors automatically. There, a naïve attempt to use smart injections in a balanced representation may fail due to the nature of Haskell's type checker, and in particular on the lack of backtracking when solving typeclass constraints. Fortunately, smart injections are not required for our purposes, as users are not expected to construct values by hand at any point—they are randomly constructed by our generators.

Benchmarks. We analyzed the performance of generating random values using three case studies: (i) Red-Black Trees (RBT), inspired by Okasaki's formulation [19], (ii) Lisp S-expressions (SExp),

Case Study	#Con	#Fun	#Pat	Total Constructions
RBT	2	5	6	13
SExp	6	-	9	15
HTML	4	132	-	136

Table 1: Overview of the size of our case studies.

inspired by the package *hs-zuramaru*[3], and (iii) HTML expressions (HTML), inspired by the *html* package, which follows the same structure as our motivating `Html` example. The magnitude of each case study can be outlined as shown in Table 1.

These case studies provide a good combination of data constructors, interface functions and patterns, and cover from smaller to larger numbers of constructions.

Then, we benchmarked the execution time required to generate and fully evaluate 10000 random values corresponding to each case study, comparing both manually-written concrete generators, and those obtained using our modular approach. For this purpose, we used the *Criterion* [20] benchmarking tool for Haskell, and limited the maximum depth of the generated values to five levels. Additionally, our modular generators were tested using both linear and balanced generation specifications. Figure 2 illustrates the relative execution time of each case study, normalized to their corresponding manually-written counterpart—we encourage the reader to obtain a colored version of this work.

As it can be observed, our approach suffers from a noticeable runtime overhead when using linearly defined representations, specially when considering the HTML case study, involving a large number of constructions in the generation process. However, we found that, by balancing our representation types, the generation performance improves dramatically. At the light of these improvements, *our tool includes an additional type-level computation that automatically balances our representations* in order to reduce the generation overhead as much as possible.

On the other hand, it has been argued that the generation time is often not substantial with respect to the rest of the testing process, especially when testing complex properties over monadic code, as well as using random values for penetration testing [9, 18].

[3] http://hackage.haskell.org/package/zuramaru

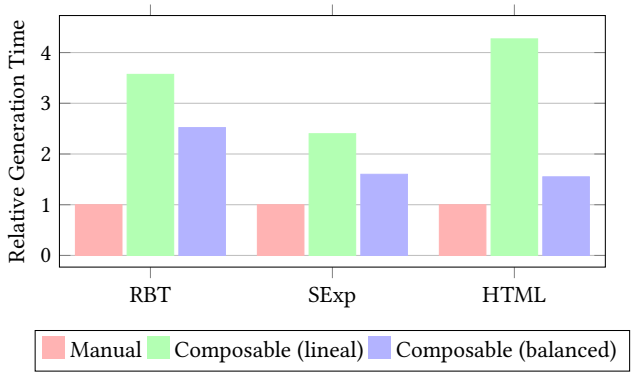


Figure 2: Generation time comparison between manually written and automatically derived composable generators.

All in all, we consider that these results are fairly encouraging, given that the flexibility obtained from using our compositional approach does not produce severe slowdowns when generating random values in practice.

7 RELATED WORK

Extensible Data Types. Swierstra proposed Data Types à la Carte [24], a technique for building extensible data types, as a solution for the *expression problem* coined by Wadler [25]. This technique has been successfully applied in a variety of scenarios, from extensible compilers, to composable machine-mechanized proofs [4, 5, 21, 27]. In this work, we take ideas from this approach and extend them to work in the scope of random data generation, where other parameters come into play apart from just combining constructions, e.g., generation frequency and terminal constructions.

From the practical point of view, Kiriyama et al. propose an optimization mechanism for Data Types à la Carte, where a concrete data type has to be derived for each different composition of constructions defined by the user [12]. This solution avoids much of the runtime overhead introduced when internally pattern matching against sequences of $\texttt{In}_\texttt{L}$ and $\texttt{In}_\texttt{R}$ data constructors. However, this approach is not entirely compositional, as we still need to rely on Template Haskell to derive the machinery for *each* specialized instance of our data type. In our particular setting, we found that our solution has a fairly acceptable overhead, achieved by automatically balancing our representation types.

Domain Specific Languages. Testing properties using small values first is a good practice, both for performance and for obtaining small counterexamples. In this light, *SmallCheck* [22] is a library for defining *exhaustive* generators inspired by *QuickCheck*. Such generators can be used to test properties against *all* possible values of a data type of up to a given depth. The authors also present *Lazy SmallCheck*, a variation of *SmallCheck* prepared to use partially defined inputs to explore large parts of the search space at once.

Luck [14] is a domain-specific language for describing testing properties and random generators in parallel. It allows obtaining random generators producing highly constrained random data by using a mixture of backtracking and constraint solving while generating values. While this approach can lead to quite good testing results, it still requires users to manually think about how to generate their random data. Moreover, the generators obtained are not compiled, but interpreted. In consequence, *Luck*'s generators are rather slow, typically around 20 times slower than compiled ones.

In contrast to these tools, this work lies on the automated side, where we are able to provide lightweight invariants over our random data by following the structural information extracted from the users' codebase.

Automatic Derivation Tools. In the past few years, there has been a bloom of automated tools for helping the process of writing random generators.

MegaDeTH [8, 9] is a simple derivation tool that synthesizes generators solely based on their types, paying no attention whatsoever to the generation frequency of each data constructor. As a result, it has been shown that its synthesized generators are biased towards generating very small values [18].

Feat [6] provides a mechanism to uniformly generating values from a given data type of up to a given size. It works by enumerating all the possible values of such type, so that sampling uniformly from it simply becomes sampling uniformly from a finite prefix of natural numbers—something easy to do. This tool has been shown to be useful for generating unbiased random values, as they are drawn uniformly from their value space. However, sampling uniformly may not be ideal in some scenarios, specially when our data types are too general, e.g., using *Feat* to generate valid HTML values as in our previous examples would be quite ineffective, as values drawn uniformly from the value space of our `Html` data type represent, in most cases, invalid HTML values.

On the other hand, *DRAGEN* is a tool that synthesizes optimized generators, tuning their generation frequencies using a simulation-based optimization process, which is parameterized by the distribution of values desired by the user [18]. This simulation is based on the theory of *branching processes*, which models the growth and extinction of populations across successive generations. In this setting, populations consist of randomly generated data constructors, where generations correspond to each level of the generated values. This tool has shown to improve the code coverage over complex systems, when compared to other automated generators derivation tools.

In a recent work, we extended this approach to generate random values considering also the other sources of structural information covered here, namely abstract interfaces and function pattern matchings [17]. There, we focus on the generation model problem, extending the theory of branching processes to obtain sound predictions about distributions of random values considering these new kinds of constructions. Using this extension, we shown that using extra information when generating random values can be extremely valuable, in particular under situations like the ones described in Section 2, where the usual derivation approaches fail to synthesize useful generators due to a lack of structural information. In turn, this paper tackles the representation problem, exploring how a compositional generation process can be effectively implemented and automated in Haskell using advanced type-level features.

In the light of that none of the aforementioned automated derivation tools are designed for composability, we consider that the ideas presented in this paper could perhaps be applied to improve the state-of-the-art in automatic derivation of random generators in the future.

8 CONCLUSIONS

We presented a novel approach for automatically deriving flexible composable random generators inspired by the seminal work on Data Types à la Carte. In addition, we incorporate valuable structural information into our generation process by considering not only data constructors, but also the structural information statically available in abstract interfaces and functions' pattern matchings.

In the future, we aim to extend our mechanism for obtaining random generators with the ability of performing stateful generation. In this light, a user could indicate which random constructions interact with their environment, obtaining random generators ensuring strong invariants like well scopedness or type correctness, all this while keeping the derivation process as automatic as possible.

ACKNOWLEDGMENTS

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet.

REFERENCES

[1] T. Arts, J. Hughes, U. Norell, and H. Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*.
[2] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.
[3] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. *SIGPLAN Not.* 37, 12 (2002), 47–59. https://doi.org/10.1145/636517.636527
[4] L. E. Day and G. Hutton. 2014. Compilation À La Carte. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL '13)*.
[5] Benjamin Delaware, Bruno C d S Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 207–218.
[6] J. Duregård, P. Jansson, and M. Wang. 2012. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*.
[7] Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming*. Springer, 229–254.
[8] G. Grieco, M. Ceresa, and P. Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. In *Proc. of the ACM SIGPLAN International Symposium on Haskell*.
[9] G. Grieco, M. Ceresa, A. Mista, and P. Buiras. 2017. QuickFuzz testing for fun and profit. *Journal of Systems and Software* 134 (2017).
[10] J. Hughes, C. Pierce B, T. Arts, and U. Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation*.
[11] J. Hughes, U. Norell, N. Smallbone, and T. Arts. 2016. Find more bugs with QuickCheck!. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*.
[12] H. Kiriyama, H. Aotani, and H. Masuhara. 2016. A Lightweight Optimization Technique for Data Types a La Carte. In *Companion Proceedings of the 15th Int. Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA.
[13] Casey Klein and Robert Bruce Findler. 2009. Randomized testing in PLT Redex. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*.
[14] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia. 2017. Beginner's luck: a language for property-based generators. In *Proc. of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*.
[15] C. McBride and R. Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (Jan. 2008).
[16] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson. 2017. Effect-driven QuickChecking of compilers. *In Proceedings of the ACM on Programming Languages, Volume 1* ICFP (2017).
[17] Agustín Mista and Alejandro Russo. 2019. Generating Random Structurally Rich Algebraic Data Type Values. In *Proceedings of the 14th International Workshop on Automation of Software Test*.
[18] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*.
[19] Chris Okasaki. 1999. Red-black trees in a functional setting. *Journal of functional programming* 9, 4 (1999), 471–477.
[20] Bryan O'Sullivan. 2014. Criterion: a Haskell microbenchmarking library. http://www.serpentine.com/criterion/
[21] Anders Persson, Emil Axelsson, and Josef Svenningsson. 2011. Generic monadic constructs for embedded languages. In *International Symposium on Implementation and Application of Functional Languages*. Springer, 85–99.
[22] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, Vol. 44. ACM, 37–48.
[23] T. Schrijvers, M. Sulzmann, S. P. Jones, and M. Chakravarty. 2007. Towards open type functions for Haskell. In *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*.
[24] Wouter Swierstra. 2008. Data Types à la Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436. https://doi.org/10.1017/S0956796808006758
[25] Philip Wadler. 1998. The expression problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
[26] P. Wadler and S. Blott. 1989. 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)*. 60–76.
[27] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. (2014).

Friot: Functional Reactive Abstraction for IoT Programming

(Draft Papers for Presentations)

Yahui Song
National University of Singapore
Singapore
yahuis@comp.nus.edu.sg

Andreea Costea
National University of Singapore
Singapore
andreeac@comp.nus.edu.sg

ABSTRACT

Friot (Functional Reactive IoT) is a domain-specific language (DSL) which features a dependent-refinement type-and-effect system for composing interactive cyber-physical devices. The complicated interactions between software and hardware modules together with the large scale of generated data lead to considerable complexity. This growing complexity makes it harder to maintain a system using an imperative programming style due to the long-running control flow and the high reliance on global values. Instead of relying on global values, functional reactive languages aim to construct the relationships between system states through asynchronous data streams. Higher-order functions lead to more concise and reusable code, thus promoting maintainable IoT environments. The type-based checking mechanism ensures that well-typed programs satisfy given properties to minimize software errors. In this paper, we focus on engaging a type system for reasoning about the temporal properties of higher-order and infinite-data programs. We specify the temporal effects using conditional value-dependent ω-regular expressions to describe event sequences for each procedure, then modularly verify the effects of larger programs. We show how this technique can be used as oracles to provide safety and liveness guarantees, estimate resource usage bounds, and WCET information. In summary, combining the abstractions of FRP with the benefits of type-and-effect systems pushes the IoT programming style into a declarative and verifiable era, which is not prevalent so far. By doing so, we can benefit from a responsive and scalable functional IoT system without sacrificing its efficiency nor functionality.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; Software prototyping; • **Computing methodologies** → *Model development and analysis.*

KEYWORDS

Logic and Verification, Temporal Property, Domain Specific Language, Functional Reactive Programming, Type-and-Effect System

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Yahui Song and Andreea Costea. 2020. Friot: Functional Reactive Abstraction for IoT Programming: (Draft Papers for Presentations). In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Functional Reactive Programming (FRP) is a declarative way to model and build reactive systems. Though the original formulation of FRP by Conal Elliott and Paul Hudak[9] was designed for animation, nowadays it is an expressive formalism employed in a diversity of domains: graphical user interfaces[6], robotics[14, 26], computer vision[27] and so on. These domains can be regarded as *reactive* because they require interactions with a wide range of external inputs such as keyboards and sensors. These reactive systems benefit from being built using FRP instead of the traditional imperative programming style for many reasons: (i) to avoid the *callback hell* [10] without sacrificing the asynchrony, (ii) to provide a straightforward way to compose various operations with the usage of higher-order functions, and (iii) to make it easier to understand, extend and reason about the code. Besides, immutability and static-checking of purely functional programs are great for reliability and maintainability [15], which allows one to enforce a variety of invariants at compile time, thereby nipping in the bud a large swathe of run-time errors.

Generally speaking, IoT aims to seamlessly interconnect different embedded systems using just the network. These connected systems are actually collaborating nodes equipped with low-power and low-memory sensors that collect data from the surroundings and communicate it to the system. To facilitate the integration of such systems, a common approach is to abstract the provided hardware and software resources. To this purpose, different programming models have been proposed for the various types of IoT applications [3, 13, 24]. However, most of these models rely on the traditional imperative programming style, hence they are prone to hard-to-manage callbacks, long-running control flows, and synchronization issues due to shared global values [4].

Much of these difficulties, we believe, stem from the lack of sufficiently high- level abstractions, and in particular from failing to clearly distinguish between *modelling* and *presentation*, or in other words, between *what* an IoT system is and *how* it should be presented. To tackle this existing general problem of IoT programs, this work focuses on extending the usage of FRP architecture to the IoT domain. Not only because the continues time-varying values in FRP can be neatly mapped to the data streams in IoT systems, but also because the abstracted architecture frees the developers

from handling many of the trivial details while designing a system. Furthermore, on top of the functional abstraction, we can also take advantage of the various formal methods to help programmers write better quality software, to guarantee specific temporal properties for safety-critical IoT systems such as air-crafts, autonomous car or medical treatment.

There are two main choices for verifying IoT systems: (i) *ensure the logic is correct* - extract the logical design from the code to prove the core design using model checkers which guarantee specific safety, security and functionality requirements [16, 22, 25]. However, this approach may lead to unsound results due to simplified assumptions, and, more importantly, to unreliable implementation of the checked design due to engineering bugs; (ii) *ensure the logic and the implementation are both correct* - prove that the implementation of the core design is correct by investigating the actual code with respect to the user-provided specifications. This approach produces a conservative approximation of the execution result returned at run-time and checks it against the design's specification. In this work, we choose the latter approach where the function specifications are given as types, taking the advantage of the automated and modular type checking process, which requires minimal user effort. More particularly, we focus on verifying temporal properties of IoT programs by decorating the type system with temporal specifications in the form of *effects*.

The temporal effects in previous works are specified using simple sets of event traces either via ω-regular sets[12] or by Büchi automata[19]. However, these works tend to loose precision along the coarse over-approximation process. To overcome that, [23] captures value-dependent temporal effects as fixpoint logic predicates on event sequences and program values. Conventionally, their types have the form $(\tau \& (\Phi_u, \Phi_v))$ where τ refers to dependent-refinement types and, the effects pair correspond to a finite and an infinite effect respectively. Inspired by prior work, but different from their approach, instead of separating these (finite and infinite) effects, we use disjunctions of conditional value-dependent effects to construct a more precise summary of terminating/non-terminating behaviors for each procedure. We can express, for example, that the effect of a function *delay t* is given by Φ_{delay} defined as:

$$\Phi_{delay} \triangleq (t \geq 0 \wedge \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}}) \vee (t < 0 \wedge \underline{\mathbf{Tick}}^\omega)$$

The effect predicate denotes the fact that for non-negative values of the parameter t the *delay* function generates a finite traces comprising a sequence of length t of **Tick** events, followed by a final **LightUp** event. For the case when the parameter is less than 0, it generates an infinite **Tick** event trace. Note that t is a parameter to *delay*, making the effect *dependent* with respected to the value of *delay*'s argument.

To show the benefits of capturing the temporal properties of an IoT system with our proposal, we discuss various program analysis: (i) conditional safety and liveness behavior, (ii) high water mark of resource usages including energy and memory consumption, and (iii) the worst-case execution time (WCET) analysis. (Sec. 5)

In this paper, we propose and prototype the DSL Friot for IoT development. It separates the *modelling* and *presentation*, which guides IoT developers to explicitly state complex devices-control dependencies in a higher-order declarative way. Based on this language abstraction, we provide the dependent type-and-effect system to algorithmically check the temporal properties of each procedure, which enables the compositional reasoning of the whole IoT program.

Contributions. To summarize, we make the following main contributions:

(1) **Language Abstraction:** By enforcing a purely functional programming style, Friot provides the infrastructure for IoT development. It contains libraries and syntactic sugar that simplify the creation of rich and responsive IoT systems. (Sec. 3)
(2) **Dependent Temporal Effect Specifications:** We define the syntax and semantics of the effect specifications expressed using conditional value-dependent ω-regular expressions, describing all the possible finite and infinite event sequences. (Sec. 4.1)
(3) **Type System:** We specify a dependent-refinement type-and-effect system supporting programs written in Friot with higher-order features and ranging over infinite data. We define the type checking and composition rules to reason about given temporal properties. (Sec. 4)

Organization. Sec. 2 gives examples and uses them to highlight our main contribution. Sec. 3 specifies the syntax and architecture of our language infrastructure, Friot. Sec. 4 presents the syntax and semantics of our dependent temporal effects, then shows the type checking rules and effects composition rules according to the refinement types and effect predicates. Sec. 5 displays more examples to show the applicability of our work, demonstrating how our effect specifications can be used to conduct more critical analysis. We discuss related works in Sec. 6 and conclude in Sec. 7.

2 OVERVIEW

We give a summary of our techniques using three simple examples. Two of them are used to present the key features of Friot: (i) purely functional IO control; and (ii) higher-order declarative dependencies modeling. The last example is used to demonstrate the type-and-effect reasoning process.

2.1 Language Features

Friot clearly distinguishes between IO presentation and logical dependency modeling, leading to a congregation of pure functions and data streams, pushing the side-effects to the edge of the system. As shown in the first example, function *io* is dedicated to IO opertions in Friot: the definition of this function is used to receive input *signals* (cf. Sec. 3.2) (including sensor readings, user operations, HTTP requests, etc.), generate responding messages and control output devices.

Example 1. This program demonstrates a sensor-controlled light. If the motion sensor gets activated, the system generates a message *Passby*, which triggers a new *model* responding to this motion signal. Based on the value of the light state in the current *model*, the system controls the LED device correspondingly.

```
io model = raspberry_pi
             [led 0 access (model, light_state)]
             [motion onActivate Passby]
```

Note that, *raspberry_pi*, *led*, *motion*, *onActivate* are all predefined functions in the libraries supporting the deployment of the Raspberry Pi board. *access*(*model*, *light_state*) is syntactic sugar to quickly access to the *light_state* field inside of the *model* argument. *model* is a collection of relevant states for all devices needed to be tracked in the application. *raspberry_pi* takes two arguments: the first argument defines a set of output devices, here, it specifies an LED light with a port number 0 (i.e., this device is plugged into the port position 0) and an output state; the second argument defines a set of inputs devices along with the messages they may generate, here, it specifies that there is a motion sensor, and whenever it activated, it generates a message *Passby* to indicate that someone has just passed by.

As shown above, the abstraction of the IO control from Friot architecture makes it easier to create complex IoT applications as it is easy to read, refactor, and add more requirements. So far, Friot provides libraries for raspberry pi 3 model B+ [28] including IoT primitives such as LED, LCD, fan, buzzer, speaker, various sensors: temperature, pressure, infrared, motion, humidity, optical and so on. *(Note that, besides these basic primitives, specific libraries are needed for specific hardware.)* The next example demonstrates how Friot's IoT primitives use higher-order functions to produce rich and responsive IoT systems.

Example 2. This example targets an LCD screen which is used to display a pair of data: the temperature and time. Although simple to describe, this is often relatively difficult to implement in today's IoT development, since the content is dynamically updated. However, in Friot, it is just a one-line program:

```
lcd = lift2 (,) Env.temprature Time.everySec
```

This code relies on signals, which are the fundamental abstraction of FRP. A signal is a value that changes over time. In this example, the temperature from the environment and every second's timing are primitive signals. By composing them together, the data pair displayed on the LCD screen becomes a new signal as well. Note that all the signals in Friot are discrete, which means that instead of continuously sampling those changing values (i.e., the system *pulls* values from the external environment), the system waits to be notified when the values are changed (i.e., values are *pushed* to the system only when they change) [6]. To combine these signals, higher-order compositional functions are generally used in Friot such as the $lifting$ in the above example (act on the current values) and the $folding$ in the following example (remember the past):

```
passer_by = fold (\x acc -> x + acc) 0 motion
```

This example shows a past-dependent computation where function *passer_by* is defined to compute the total number of times that the motion sensor got activated, which indicates the number of passers-by. As $fold$ receives a list of values from the motion input signals, it combines them with an accumulator (here, the accumulator is initialized to 0). The output signal is the most recent value of the accumulator.

Table 1: (a) Source code for Light_control; (b) Recursive functions' typing

(a) Source Code

```
until_ready () =
   if * then ev (Ready)
   else ev (Wait);
         until_ready ();

delay t =
   if t == 0 then ev (LightUp)
   else ev (Tick);
         delay (t - 1);

light_control t =
      until_ready ();
      delay t;
      light_control t;
```

(b) Types & effects for recursive functions

$\tau_{until\_ready} = Unit \rightarrow (Unit \,\&\, \Phi_{until\_ready})$

$\Phi_{until\_ready} = (\underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}}) \vee (\underline{\mathbf{Wait}}^\omega)$

$\tau_{delay} = t : Int \rightarrow (Unit \,\&\, \Phi_{delay})$

$\Phi_{delay} = (t \geq 0 \wedge \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}}) \vee (t < 0 \wedge \underline{\mathbf{Tick}}^\omega)$

$\tau_{light\_control} = (t : \{u : Int \mid u \geq 0\}) \rightarrow (Unit \,\&\, \Phi_{light\_control})$

$\Phi_{light\_control} = (t \geq 0 \wedge (\underline{\mathbf{Ready}} \cdot \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}} \vee \underline{\mathbf{Wait}})^\omega)$

2.2 Type-and-Effect Reasoning

Our type system is extended with effect specifications which are described using conditional dependent ω-regular expressions. We now illustrate how our approach can prove the specifications attached to the IoT program in Table 1, highlighting the contributions along the way.

Example 3. The three-parts program code in Table 1 **(a)** implements an automatic light-controlled device. Function *light_control* calls *until_ready* which makes a non-deterministic choice between raising a **Ready** event and raising a **Wait** event followed immediately by a recursive call to itself. If the **Ready** event ever occurs, function *until_ready* returns, and the program control is passed to *delay* which generates t instances of **Tick**. Finally, *light_control* recurs, hence it exhibits no finite behavior. Its infinite trace effect is described by the following expression:

$$(t{\geq}0 \wedge (\underline{\mathbf{Ready}} \cdot \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}} \vee \underline{\mathbf{Wait}}^\omega)) \vee (t{<}0 \wedge (\underline{\mathbf{Wait}}^\omega \vee \underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}} \cdot \underline{\mathbf{Tick}}^\omega)).$$

Notice that this effect depends on the input to the program t. This example illustrates that the effect specification in our technique is more precise than previous work and escapes classical LTL or the μ-calculus.

The typings for the recursive functions in this example can be found in Table 1 **(b)**. The type specified for *light_control* is: $\tau_{light\_control} = (t{:}\,\{u{:}Int \mid u{\geq}0\}) \rightarrow (Unit \,\&\, \Phi_{light\_control})$. We assume that the reader is already familiar with dependent-refinement

Table 2: (a) Type checking trees for until_ready; I: Effects computation for until_ready; (b) Type checking trees for light_control; A,B: Sub trees for light_control's type checking; II: Effects computation for light_control; C,D: Sub trees for light_control's effects computation;

(a) until_ready

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Phi = \underline{\mathbf{Ready}}}{\Gamma' \vdash ev\ (Ready) : Unit\ \&\ \underline{\mathbf{Ready}}}\ (T\text{-}Event) \quad \dfrac{\Phi = \underline{\mathbf{Wait}}}{\Gamma' \vdash ev\ (Wait) : Unit\ \&\ \underline{\mathbf{Wait}}}\ (T\text{-}Event)}{\Gamma' \vdash (ev\ (Wait);\ until\_ready\ ();) : Unit\ \&\ (\underline{\mathbf{Wait}} \cdot \Phi_{until\_ready})} \quad \dfrac{\dfrac{sty(\Gamma'(until\_ready)) \in \rightarrow}{\Gamma' \vdash\ until\_ready : \tau_{until\_ready}}\ (T\text{-}VaF) \quad \Gamma' \vdash () : Unit}{\Gamma' \vdash until\_ready\ () : (Unit\ \&\ \Phi_{until\_ready})}\ (T\text{-}App)}{\Gamma' \vdash (if * then\ ev\ (Ready)\ else\ ev\ (Wait);\ until\_ready\ ();) : Unit\ \&\ (\underline{\mathbf{Ready}} \vee \underline{\mathbf{Wait}} \cdot \Phi_{until\_ready})}\ (T\text{-}Let)}{\dfrac{\dots\ (Effects\ Computation\ \textbf{\textit{I}})\ \dots}{\Gamma \vdash (until\_ready\ () = if * then \dots else \dots) : Unit \rightarrow (Unit\ \&\ (\underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}}) \vee (\underline{\mathbf{Wait}}^\omega))}\ (T\text{-}Fun)}}\ (T\text{-}If)$$

$\Gamma' = \Gamma,\ until\_ready : \tau_{until\_ready}$

$$\mathbf{I}_{\text{norm}}:\ \underline{\mathbf{Ready}} \vee (\underline{\mathbf{Wait}} \cdot \Phi_{until\_ready}) \equiv \underline{\mathbf{Ready}} \vee (\underline{\mathbf{Wait}} \cdot (\underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}} \vee \underline{\mathbf{Wait}}^\omega)) \equiv \underline{\mathbf{Ready}} \vee (\underline{\mathbf{Wait}} \cdot \underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}} \vee \underline{\mathbf{Wait}} \cdot \underline{\mathbf{Wait}}^\omega)$$
$$\equiv \underline{\mathbf{Ready}} \vee (\underline{\mathbf{Wait}}^{1+*} \cdot \underline{\mathbf{Ready}} \vee \underline{\mathbf{Wait}}^{1+\omega}) \equiv \underline{\mathbf{Ready}} \vee (\underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}} \vee \underline{\mathbf{Wait}}^\omega) \equiv \underline{\mathbf{Ready}} \vee \Phi_{until\_ready}$$

$$\mathbf{I}_{\leq -check}:\ \dfrac{\dfrac{\dfrac{true}{\underline{\mathbf{Ready}} \preceq \underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}}}\ (\text{SUB-SEQ})}{\underline{\mathbf{Ready}} \preceq \underline{\mathbf{Wait}}^* \cdot \underline{\mathbf{Ready}} \vee \underline{\mathbf{Wait}}^\omega}\ (\text{SUB-RHS-OR}) \quad \dfrac{true}{\Phi_{until\_ready} \preceq \Phi_{until\_ready}}}{\underline{\mathbf{Ready}} \vee \Phi_{until\_ready} \preceq \Phi_{until\_ready}}\ (\text{SUB-LHS-OR})$$

(b) light_control

$$\dfrac{\dfrac{\dfrac{\dfrac{sty(\Gamma'(until\_ready)) \in \rightarrow}{\Gamma' \vdash\ until\_ready : \tau_{until\_ready}}\ (T\text{-}VaF) \quad \Gamma' \vdash () : Unit}{\Gamma' \vdash until\_ready\ () : (Unit\ \&\ \Phi_{until\_ready})}\ (T\text{-}App) \quad \mathbf{A} \quad \mathbf{B}}{\Gamma' \vdash (until\_ready\ ();\ delay\ t;\ light\_control\ t;) : Unit\ \&\ (\Phi_{until\_ready} \cdot \Phi_{delay} \cdot \Phi_{light\_control})}\ (T\text{-}Let) \quad \dfrac{sty(\Gamma'(t)) = Int}{\Gamma' \vdash t : \{u : Int \mid u \geq 0\}}\ (T\text{-}Var)}{\dfrac{\dots\ (Effects\ Computation\ \textbf{\textit{II}})\ \dots}{\Gamma \vdash (light\_control\ t\ =\ \dots) : (t : \{u : Int \mid u \geq 0\}) \rightarrow (Unit\ \&\ (t \geq 0 \wedge (\underline{\mathbf{Ready}} \cdot \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}} \vee \underline{\mathbf{Wait}})^\omega))}\ (T\text{-}Fun)}$$

$\Gamma' = \Gamma,\ light\_control : \tau_{light\_control},\ t : \{u : Int \mid u \geq 0\}$

A:
$$\dfrac{\dfrac{sty(\Gamma'(delay)) \in \rightarrow}{\Gamma' \vdash delay : (t : Int) \rightarrow (Unit\ \&\ (t \geq 0 \wedge \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}}) \vee (t < 0 \wedge \underline{\mathbf{Tick}}^\omega))}\ (T\text{-}VaF) \quad \dfrac{sty(\Gamma'(t)) = Int}{\Gamma' \vdash t : \{u : Int \mid u \geq 0\}}\ (T\text{-}Var)}{\Gamma' \vdash delay\ t : (Unit\ \&\ (t \geq 0 \wedge \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}}))}\ (T\text{-}App, S\text{-}Base)$$

B:
$$\dfrac{\dfrac{sty(\Gamma'(light\_control)) \in \rightarrow}{\Gamma' \vdash light\_control : (t : \{u : Int \mid u \geq 0\}) \rightarrow (Unit\ \&\ \Phi_{light\_control})}\ (T\text{-}VaF) \quad \dfrac{sty(\Gamma'(t)) = Int}{\Gamma' \vdash t : \{u : Int \mid u \geq 0\}}\ (T\text{-}Var)}{\Gamma' \vdash light\_control\ t : Unit\ \&\ (\Phi_{light\_control})}\ (T\text{-}App)$$

$$\mathbf{II}_{\text{norm}}:\ \Phi_{until\_ready} \cdot \Phi_{delay} \cdot \Phi_{light\_control} \equiv \overset{(\text{norm})}{\dots} \equiv Q^* \cdot P \cdot (P \vee Q)^\omega \vee Q^\omega \quad \text{where} \quad Q \triangleq \underline{\mathbf{Wait}},\ \text{and}\ P \triangleq \underline{\mathbf{Ready}} \cdot \underline{\mathbf{Tick}}^t \cdot \underline{\mathbf{LightUp}}$$

$$\mathbf{II}_{\leq -check}:\ \dfrac{\dfrac{\mathbf{C}}{Q^* \cdot P \cdot (P \vee Q)^\omega \preceq (P \vee Q)^\omega}\ (\text{I}_1) \quad \dfrac{\mathbf{D}}{Q^\omega \preceq (P \vee Q)^\omega}\ (\text{L}_4)}{Q^* \cdot P \cdot (P \vee Q)^\omega \vee Q^\omega \preceq (P \vee Q)^\omega}\ (\text{SUB-LHS-OR})$$

C:
$$\dfrac{\dfrac{\dfrac{\dfrac{true}{Q^* \preceq \exists v.Q^v}\ (*)}{Q^* \preceq \exists v.P^v \vee Q^v}\ (\text{SUB-RHS-OR})}{Q^* \preceq \exists v.(P \vee Q)^v}\ (\text{L}_4) \quad \dfrac{\dfrac{true}{P \preceq \exists v.(P \vee Q)^v} \quad \dfrac{true}{(P \vee Q)^\omega \preceq (P \vee Q)^\omega}\ (\text{SUB-SEQ})}{P \cdot (P \vee Q)^\omega \preceq \exists v.(P \vee Q)^v \cdot (P \vee Q)^\omega}\ (I_3)}{Q^* \cdot P \cdot (P \vee Q)^\omega \preceq \exists v.(P \vee Q)^v \cdot (P \vee Q)^\omega}\ (\text{SUB-SEQ})$$

D:
$$\dfrac{\dfrac{true}{Q^\omega \preceq P^\omega \vee Q^\omega}}{Q^\omega \preceq (P \vee Q)^\omega}\ (\text{L}_4)$$

types such as above, which states that *light_control* is a function from non-negative integers to $Unit$, having effects described by $\Phi_{light\_control}$. We write $t : Int$ as an abbreviation for $t : \{u : Int \mid true\}$.

Our type-and-effect system deploys the standard type checking algorithm which uses declared types to check agreement - i.e., to examine the body of each function. We define our type judgements (Figure 9.) and effect composition rules (Figure 6.) in Sec. 4. The full type checking trees and effect computations for *until_ready* and *light_control* are given in Table 2. Notice that, the type rule for function F requires that the effect of a total application of F be compatible with the effect of the body of F, which is itself derived from the typing rules (*T-Fun*). Since the normal let expression $let\ x = e_1\ in\ e_2$ becomes a simple expression sequence $(e_1;\ e_2)$ when x is not used in e_2, the expression sequences and let expressions are sharing a common type rule (*T-Let*). $sty(\sigma)$ returns the simple type corresponding to the effect qualified type σ, for example, $sty(\{u{:}Int \mid u{\geq}0\}\ \&\ \Phi) = Int$. We assume that the type environment Γ in Table 2 **(b)** already contains $until\_ready : \tau_{until\_ready}$ and $delay : \tau_{delay}$.

The effects pass through a number of phases during the type checking. First, they need to be composed to cope with choices and sequential composition. Next, they are normalized to a form which supports subsumption check. The normalization and subsumption checks are highlighted in Table 1 and formalized in Sec. 4.1. We skip the explanation of these steps for now, and resume it after having introduce their corresponding rules. In the following sections, we turn to the formal presentation of our work.

3 LANGUAGE INFRASTRUCTURE

The syntax of Friot is ML-like (i.e., typed, higher-order, and call-by-value). The full Friot language contains various libraries and syntactic sugars, which are meant to simplify the creation of rich and responsive IoT applications. In this section, we introduce the core design of Friot presenting the semantics of Friot's fundamental abstractions.

$$
\begin{array}{rrcl}
(Constants) & c & ::= & n \mid true,\ false \mid string \\
(Pattern) & p & ::= & Any \mid x \mid c \\
(Expressions) & e & ::= & () \mid c \mid x \mid F \mid ev(\underline{\mathbf{a}}) \mid \lambda x.e \\
 & & & \mid e_1 \oplus e_2 \mid e\ e_1 \mid D\ \overline{e} \mid \textbf{access}\ (e, x) \\
 & & & \mid \textbf{let}\ (e_1; x.e_2) \mid \textbf{case}\ x = e\ \textbf{of}\ \{D\ \overline{p} \rightarrow e_p\} \\
 & & & \mid \textbf{if}\ (e_1; e_2; e_3) \mid \textbf{update}\ e\ \{D\ \overline{x} \rightarrow e_x\} \\
 & & & \mid \textbf{lift\_n}\ e\ e_1\ \ldots\ e_n \mid \textbf{fold}\ e_1\ e_2\ e_3 \\
(Program) & \mathcal{P} & ::= & \mathcal{P} \cup \{F\ \overline{x}.e\} \mid \varnothing
\end{array}
$$

$$
n ::\in \mathbb{Z} \quad string ::\in \textbf{String} \quad x, F ::\in \textbf{Var} \quad \underline{\mathbf{a}} ::\in \Sigma
$$

Figure 1: Syntax of Friot.

3.1 Syntax

The syntax of Friot is given in Figure 1. Here, we regard c, x, and $\underline{\mathbf{a}}$ are meta-variables. The atomic constants c includes numbers, boolean values, and strings. **Var** represents the countably infinite set of arbitrary identifiers. $\underline{\mathbf{a}}$ refers to a certain event. Σ donates a finite set of events. Each event raising expression ev is parametrized by an event $\underline{\mathbf{a}}$ (we treat only the unary case here which can be easily extended to n-ary events). For pattern matching, we define the actual arguments of a constructor D as a list of Pattern p, which can be an Anything (written as "_" in the code), an identifier or a constant. Besides, we write $\overline{p}$, $\overline{x}$, $\overline{e}$ for a finite sequence (possibly empty) of patterns, variables and expressions respectively. A Program $\mathcal{P}$ is a finite set of mutually recursive function definitions. $F\ \overline{x}.e$ uniquely defines a recursive function named F with the formal parameters $\overline{x}$ and a closed expression e as the body. Note that nested recursive functions can be supported via λ lifting[17].

Expressions comprise variables x; constants c; function names F; event raises $ev(\underline{\mathbf{a}})$; lambda abstractions $\lambda x.e$; operator $\oplus$ ranges over total constant binary operations on basic types; function applications $e\ e_1$; fully applied data structures D that wrap expressions; elimination form for tuples, **access** (e, x), evaluates to the field which identified by x of the tuple e; let expressions **let** $(e_1; x.e_2)$ refer to $let\ x = e_1\ in\ e_2$, (Note that expression sequences are let expressions when x is not used in e_2); and **if** $(e_1; e_2; e_3)$ represents the condition branches $if\ e_1\ then\ e_2\ else\ e_3$, which evaluates to e_2 when e_1 evaluates to non-zero values, e_3 otherwise. A case expression, **case** $x = e$ **of** $\{D\ \overline{p} \rightarrow e_p\}$ matches the expression e with at least one alternative $D\ \overline{p}$ and each alternative must have at least one execution body e_p. The update expression is syntactic sugar to concisely update more than one field of a tuple, for example, the expression $\{model \mid count = count + 1\}$ means the *count* field of the tuple *model* will be *updated* into its increment. (Note that the update expressions here still maintain the functional purity, because it allocates memory for a new model with some modification on specific fields based on the previous model.)

In Friot, we make use of $lift\_n$ and $fold$ higher-order primitives to accommodate three types of signal manipulation: (i) transformation of a signal, (ii) combination of multiple signals, and (iii) past-dependent transformation of a signal.

3.2 Architecture

3.2.1 FRP Background. FRP reconciles purely functional paradigm and mutable values by recasting them as *time-varying* values, capturing the temporal aspect of mutability. Friot adopts the isomorphism abstraction from the original FRP, called *signal*, to present both continuous changing values and discrete occurring values.

Signal. reactive time-varying modelling from users or processes. Signals abstract time-varying values, represented as a function taking a time and producing a value. At time t, the event has a value v conforming to type α.

$$Signal\ \alpha\ =\ Time\ \rightarrow\ \alpha$$

This abstraction initially focused on animation [9] and strongly influenced the semantics and goals of future FRP work. In this work, each of these time-varying signal functions mostly represent the state of an IoT device at a particular time point (cf. ***Example 2*** Sec. 2). Temperature, light intensity, motion, etc. can all be represented naturally with signals.

3.2.2 Model-Update-IO Abstraction. Inspired by the language Elm [6], every IoT program in Friot can be broken up into three cleanly separated parts: (i) Model: the state of one application; (ii) Update: predefined dependencies to update the state, - i.e., the dependencies among various signals; and (iii) IO: a way to interact with the real world, - i.e., to receive input signals and control output devices correspondingly. For example, a system which controls a temperature-sensitive fan, can be constructed using the following data types:

```
data Model = Home { temperature:: Int }
```

We create a new *Model* type as above, which means that each instance of such general type *Model* contains an *Int* value indicating the temperature in the current environment. The *Model* type needs to be customized based on different system demands. Then, to bridge the connection between IO and Update, we enumerate the possibly used messages, reacting to different signals, typed as *Msg* as follows:

```
data Msg = TurnOn | TurnOff
```

This code shows there are two kinds of messages to control the fan, - i.e., turn on the fan whenever the *TurnOn* message gets triggered, otherwise, turn it off. Next, we show the simple types of the update and io function:

```
update : Msg -> Model -> Model
io : Model -> IO
```

As we can see, *update* function takes two arguments, a message, and a model, then based on the dependencies required by the application, it returns a new *model* responding to the newly generated message. The *io* function takes a *model* and projects it into real IoT devices (cf. ***Example 1*** Sec. 2).

The purpose and the benefit of this *Model-Update-IO* abstraction is as follows: it starts with reading from the world and end up with writing to the world. Inside of this functional reactive IoT system, we only have pure functions and data streams pushing the side-effects to the edge of the system, which makes formal verification more accessible.

4 TYPE SYSTEM

We now formalize the type-and-effect system. Intuitively, our type-and-effect system is an orchestrating language, that glues together information from the temporal property verifier oracles. We adopt and extend the refinement type-based verification with value-dependent effect, that has garnered recent popularity.

4.1 The Effect System

Figure 2. depicts the language used for describing the program's effects. An effect is a conditioned sequence of events ($\pi \wedge es$), where es^n is just a syntax sugar for concatenating an event with itself n times, $es \cdot \ldots \cdot es$, and ϵ is the empty event or es^0. An event sequence may be of finite fixed length es^c, finite arbitrary length es^*, or infinite length es^ω. $A(\overline{n})$ represents atomic formulas such as the equality or inequality on integers and sequences. We often use just es to describe the effect, which is a shortcut for $true \wedge es$. Events

$$
\begin{array}{ll}
(\textit{Effects}) & \Phi ::= \pi \wedge es \mid \Phi \vee \Phi \mid \exists\, v.\Phi \mid \forall\, v.\Phi \\
(\textit{Event Sequence}) & es ::= \epsilon \mid \underline{\mathbf{a}} \mid es^n \mid es \cdot es \mid es \vee es \\
(\textit{Pure Formula}) & \pi ::= \beta \mid \pi \wedge \pi \mid \pi \vee \pi \mid \neg(\pi) \\
(\textit{Boolean Formula}) & \beta ::= true \mid false \mid A(\overline{n}) \\
 & n ::= k \mid v \mid n{+}n \mid n{-}n \mid k{\times}n \\
(\textit{Constants}) & k ::= c \mid \omega \mid * \\
\multicolumn{2}{l}{(\textit{First Order Variable})\ v \qquad (\textit{Numerical Constant})\ c} \\
\multicolumn{2}{l}{(\textit{Event})\ \underline{\mathbf{a}} \in \Sigma \qquad (\textit{Klenee Star})\ * \qquad (\textit{Infinity})\ \omega}
\end{array}
$$

Figure 2: System of Effects.

form a monoid with the sequential composition $(\Sigma, \cdot, \epsilon)$, while the choice relation forms a commutative semigroup $(\Sigma, \vee)$ as per the axioms in Figure 3. During our formal reasoning we always assume that the formulae describing the effects are in a normal form tailored accordingly using the lemmas in Figure 4.

We support effects subsumption check, denoted by $\Phi_1 \preceq \Phi_2$, using the rules in Figure 5. We abuse the $\preceq$ notation to also denote the subsumption relation between two sequences of events:

$$
\frac{t{\geq}0 \Rightarrow t{>}0}{t{\geq}0 \wedge \underline{\mathbf{a}} \not\preceq t{>}0 \wedge \underline{\mathbf{a}}}\ (\textsc{sub-eff}) \qquad \frac{true}{\underline{\mathbf{a}} \preceq \underline{\mathbf{a}}_0^* \cdot \underline{\mathbf{a}}}\ (\textsc{sub-seq})
$$

$$
\begin{array}{rcll}
(es_1 \cdot es_2) \cdot es_3 & \equiv & es_1 \cdot (es_2 \cdot es_3) & (A_1) \\
(es_1 \vee es_2) \vee es_3 & \equiv & es_1 \vee (es_2 \vee es_3) & (A_2) \\
es_1 \cdot (es_2 \vee es_3) & \equiv & es_1 \cdot es_2 \vee es_1 \cdot es_3 & (D_1) \\
(es_1 \vee es_2) \cdot es_3 & \equiv & es_1 \cdot es_3 \vee es_2 \cdot es_3 & (D_2) \\
es_1 \vee es_2 & \equiv & es_2 \vee es_1 & (C_1) \\
\epsilon \cdot es & \equiv & es \cdot \epsilon \ \equiv\ es & (I_1) \\
\epsilon \vee es & \equiv & es & (I_2) \\
es^\omega & \equiv & \exists v.v{\geq}0 \wedge es^v \cdot es^\omega & (I_3)
\end{array}
$$

Figure 3: Structural Equivalence for Sequence of Events.

$$
\begin{array}{rcl}
es^{n_1} \cdot es^{n_2} \Leftrightarrow es^{n_1+n_2} & & (L_1) \\
es^\omega \cdot es \Rightarrow es^\omega & & (L_2) \\
(es^* \cdot es') \vee (es^n \cdot es') \Rightarrow es^* \cdot es' \text{ when } n{\neq}\omega & & (L_3) \\
es_1^n \vee es_2^n \Leftarrow (es_1 \vee es_2)^n & & (L_4)
\end{array}
$$

Figure 4: Events Normalization Rules.

The subsumption on the left in the above example does not hold because the conditions of the effects are not fully compatible. The right subsumption holds by applying the (SUB-SEQ) rule with $n_1{=}0$ and $n_2{=}*$. The effects can be composed according to the rules in Figure 6. According to the first composition rule, if the conditions of two effects are mutually disjoint the effect is ignored since its condition becomes $false$, or in other words the corresponding events never take place.

Example 3 - revisited. Before checking the effects subsumptions, the effects formulae go through some normalization steps to prepare them for applying the rules from Figure 5. The effect of executing the body of *until_ready* in Table 1 is derived to be $\underline{\mathbf{Ready}} \vee (\underline{\mathbf{Wait}} \cdot \Phi_{until\_ready})$. By successively applying $D_1, L_1(\Rightarrow)$ twice, and using the properties of $*$ and ω the resulted effect is reduced to $\underline{\mathbf{Ready}} \vee \Phi_{until\_ready}$. The subsumption check is straightforward in this form. The *light_control* reaches a normal form after successively applying the following

$$\frac{\pi_1 \Rightarrow \pi_2 \quad es_1 \preceq es_2}{\pi_1 \wedge es_1 \preceq \pi_2 \wedge es_2}\ [\text{SUB-EFF}] \qquad \frac{\underline{\mathbf{a}}_1 \equiv \underline{\mathbf{a}}_2}{\underline{\mathbf{a}}_1 \preceq \underline{\mathbf{a}}_2}\ [\text{SUB-EV}]$$

$$\frac{es_1 \equiv es_2 \quad (n_1{=}n_2 \text{ or } n_2{=}*) \quad es_1' \preceq es_2'}{es_1^{n_1} \cdot es_1' \preceq es_2^{n_2} \cdot es_2'}\ [\text{SUB-SEQ}]$$

$$\frac{es \preceq es_1 \quad \text{or} \quad es \preceq es_2}{es \preceq es_1 \vee es_2}\ [\text{SUB-RHS-OR}] \qquad \frac{es_1 \preceq es \quad es_2 \preceq es}{es_1 \vee es_2 \preceq es}\ [\text{SUB-LHS-OR}]$$

$$\frac{\Phi \preceq \Phi_1 \quad \text{or} \quad \Phi \preceq \Phi_2}{\Phi \preceq \Phi_1 \vee \Phi_2}\ [\text{SUB-RHS-OR}] \qquad \frac{\Phi_1 \preceq \Phi \quad \Phi_2 \preceq \Phi}{\Phi_1 \vee \Phi_2 \preceq \Phi}\ [\text{SUB-LHS-OR}]$$

Figure 5: Events Subsumption.

$$\begin{aligned}(\pi_1 \wedge es_1) \cdot (\pi_2 \wedge es_2) &\stackrel{\text{def}}{=} ((\pi_1 \wedge \pi_2) \wedge es_1 \cdot es_2)\\ \Phi \cdot (\Phi_1 \vee \Phi_2) &\stackrel{\text{def}}{=} (\Phi \cdot \Phi_1) \vee (\Phi \cdot \Phi_2)\\ (\Phi_1 \vee \Phi_2) \cdot \Phi &\stackrel{\text{def}}{=} (\Phi_1 \cdot \Phi) \vee (\Phi_2 \cdot \Phi)\end{aligned}$$

Figure 6: Composition of Effects.

rules $D_2, L_2, D_2, L_2, D_2, L_2$. For the subsumption check it is sound to strengthen the RHS of the subsumption relation (e.g. applying L_4 in the proof subtree tagged with D). Intuitively, this strengthening is permitted since it is sound to derive a type which is stronger than the annotation type. The rest of the derivations are a straightforward application of the rules in Figure 5.

$$\begin{aligned}(\textit{Basic Types}) \quad & B ::= \mathit{Unit} \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{String}\\ (\textit{Effect Types}) \quad & \sigma ::= (\tau \ \& \ \Phi)\\ (\textit{Refinement Types}) \quad & \tau ::= \{u : B \mid \pi\} \mid x : \tau \to \sigma\\ (\textit{Type Environment}) \quad & \Gamma ::= \emptyset \mid \Gamma,\ k : \tau\\ (\textit{Dom}(\Gamma)) \quad & k ::\in \{x, F\}\end{aligned}$$

Figure 7: Syntax of Types and Effects.

4.2 Syntax of types and effects

The syntax of types and effects is shown in Figure 7. Besides *Int*, *Bool*, and *String* are basic types, we take () as an *Unit* type. An effect qualified type is of the form of $(\tau \ \& \ \Phi)$ where τ is a dependent refinement type and Φ is an effect. $\{u : B \mid \pi\}$ is a refinement type where given that B is a basic type and π is some pure logic predicates on B. A dependent function type $x : \tau \to \sigma$ is the type of functions which take an argument x of the type τ and behave according to the return type σ. Since the scope of x is within σ, the effect in σ can depend on the argument x. Often, we write $x : B$ as an abbreviation of $x : \{u : B \mid \mathit{true}\}$. A type environment Γ is a sequence of variable bindings range from identifiers x and function names F. Notice that type bindings in type environments and arguments of function types are of the form $(k : \tau)$ instead of $(k : \sigma)$. This is because Friot is a call-by-value language where variables are always bound to values whose evaluation never exhibits temporal effects. The auxiliary function $sty(\sigma)$ represents the simple type corresponding to the qualified type σ. We next formally define the semantics.

$$\begin{aligned}[\![(\tau \ \& \ \Phi)]\!] \triangleq \{e \in sty(\tau)\\ \mid (\forall \Phi_f, w.(e \Downarrow w \ \& \ \Phi_f) \Rightarrow (w \in [\![\tau]\!]) \wedge (\models \lfloor \Phi_f \rfloor \in \Sigma^*))\\ \wedge (\forall \Phi_{inf}.(e \Uparrow \mathit{false} \ \& \ \Phi_{inf}) \Rightarrow (\models \lfloor \Phi_{inf} \rfloor \in \Sigma^\omega))\}\\ [\![\{u : B \mid \pi\}]\!] \triangleq \{c \in B \mid \models [c/u]\,\pi\}\\ [\![(x : \tau) \to \sigma]\!] \triangleq \{w \in sty(\tau \to \sigma) \mid \forall w' \in [\![\tau]\!].ww' \in [\![[w'/x]\sigma]\!]\}\end{aligned}$$

Figure 8: Semantics of Type-and-Effect.

4.3 Semantic Typing

Figure 8. defines the semantics of qualified types. Here, w is a meta-variable ranging over closed values (i.e., w does not include free variables). We write $e \in B$ if the expression e has the simple type B, $w \in B$ if the closed value w has the simple type B. Φ_f donates a finite effect while Φ_{inf} donates an infinite effect. $\lfloor \Phi \rfloor$ extracts all the possible sequences (finite or infinite) of the Φ discarding the conditions. $[\![(\tau \ \& \ \Phi)]\!]$ represents the set of expressions e such that: (i) for all terminating runs $e \Downarrow w \ \& \ \Phi_f$, w conforms to the type τ and $\lfloor \Phi_f \rfloor \subseteq \lfloor \Phi \rfloor$ ranges from Σ^*; and (ii) for all non-terminating runs $e \Uparrow \mathit{false} \ \& \ \Phi_{inf}$, $\lfloor \Phi_{inf} \rfloor \subseteq \lfloor \Phi \rfloor$ ranges from Σ^ω. $[\![\{u : B \mid \pi\}]\!]$ are the set of closed values have the simple type B and satisfy the pure logic predicate π. Similarly, $[\![(x : \tau) \to \sigma]\!]$ are the set of closed functions from τ to σ. For example, $\tau = (x : \{u : \mathit{Int} \mid u \geq 0\}) \to (\{v : \mathit{Int} \mid v > x\} \ \& \ (x \geq 0 \wedge \underline{\mathbf{a}}^x))$ refers to a function who takes a non-negative integer value x as an argument, returns an integer greater than x and raises the event $\underline{\mathbf{a}}\ x$ many times.

4.4 Typing Rules

Figure 9. presents some essential typing and subtyping rules. These rules derive judgments of the form $\Gamma \vdash e : \sigma$, saying that e behaves according to σ under a type environment conforming to Γ. The rules *T-Const* types constants, *T-Var* types basic type variables and *T-VaF* types function type variables, here, $\to$ donates the simple types of the set of function-types. *T-Event* types event raising operations and is self-explanatory. *T-Op* types constant operator applications where $\oplus$ is a binary operator from the arguments τ_1 and τ_2 to the return value of type τ_3.

T-Fun types a recursive function definition. It uses the declared types to examine the agreement of the body. *T-Lam* types function definitions, and *T-App* types function applications. Since Friot is a call-by-value language, the effect of an application is a composition of effects of the arguments and the effects of the applied expression. *T-If* types conditional expressions, and the final effect is a union of effects from both branches. Similarly, *T-Case* types pattern matching expressions, it matches a data structure with all possible constructors where $Ty(D)$ is the type of a constructor D. The effect of a case expression is a union of effects from all the possible cases.

The rules *S-Base* and *S-Fun* for subtyping refinement basic types and dependent function types, they are equivalent to those from the previous work on dependent-refinement type systems. The rule *S-Qual* subtypes effect qualified types. It asserts that the refinement type part of the qualified types τ_1 and τ_2 are in the subtyping relationship. Further, it checks that the left effect Φ_1 is a subeffect of the

$$\frac{c \in B}{\Gamma \vdash c : (\{u : B \mid u = c\})}(T\text{-}Const) \quad \frac{sty(\Gamma(x)) = B}{\Gamma \vdash x : (\{u : B \mid u = x\})}(T\text{-}Var) \quad \frac{sty(\Gamma(x)) \in \rightarrow}{\Gamma \vdash x : (\Gamma(x))}(T\text{-}VaF) \quad \frac{\Phi = \underline{\mathbf{a}}}{\Gamma \vdash ev(a) : (Unit \ \& \ \Phi)}(T\text{-}Event)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma(\oplus) \vdash x_1 : \tau_1 \rightarrow x_2 : \tau_2 \rightarrow \tau_3}{\Gamma \vdash e_1 \oplus e_2 : \tau_3[e_1/x_1][e_2/x_2]}(T\text{-}Op) \qquad \frac{\tau_F = (\overline{x} : \overline{\tau}) \rightarrow (\tau \ \& \ \Phi_F) \quad \Gamma,\ F : \tau_F,\ \overline{x} : \overline{\tau} \vdash e : \tau \ \& \ \Phi_F}{\Gamma \vdash (F\ \overline{x}\ =\ e) : \tau_F \ \& \ \Phi_F}(T\text{-}Fun)$$

$$\frac{\Gamma, p : \tau_p \vdash e : (\tau \ \& \ \Phi)}{\Gamma \vdash \lambda p.e : (p : \tau_p) \rightarrow \tau}(T\text{-}Lam) \ \frac{\Gamma \vdash e : (x_1 : \tau_1) \rightarrow \tau \ \& \ \Phi \quad \Gamma \vdash e_1 : \tau_1 \ \& \ \Phi_1}{\Gamma \vdash e\ e_1 : \tau[e_1/x_1] \ \& \ \Phi_1 \cdot \Phi}(T\text{-}App) \ \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \Phi_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \ \& \ \Phi_2}{\Gamma \vdash \textbf{let}\ (e_1; x.e_2) : \tau_2 \ \& \ \Phi_1 \cdot \Phi_2}(T\text{-}Let)$$

$$\frac{\Gamma,\ e \Downarrow true \vdash e_1 : \tau \ \& \ \Phi_1 \quad \Gamma,\ e \Downarrow false \vdash e_2 : \tau \ \& \ \Phi_2}{\Gamma \vdash \textbf{if}\ (e; e_1; e_2) : \tau \ \& \ \Phi_1 \vee \Phi_2}(T\text{-}If) \quad \frac{\Gamma \vdash e : \tau_e \quad \Gamma,\ Ty(D_i) = \overline{p_j : \tau_j} \rightarrow \tau_e,\ x : \tau_e \vdash e_i : \tau \ \& \ \Phi_i}{\Gamma \vdash \textbf{case}\ x = e\ \textbf{of}\ \{D_i\ \overline{p_j} \rightarrow e_i\} : \tau \ \& \ (\vee_{i \in I}\ \Phi_i)}(T\text{-}Case)$$

$$\frac{T \in B \quad \models \Gamma \vdash \pi_1 \Rightarrow \pi_2}{\Gamma \vdash \{u : T \mid \pi_1\} <: \{u : T \mid \pi_2\}}(S\text{-}Base) \qquad \frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma, x : \tau_2 \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash (x : \tau_1) \rightarrow \sigma_1 <: (x : \tau_2) \rightarrow \sigma_2}(S\text{-}Fun) \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Phi_1 \leq \Phi_2}{\Gamma \vdash (\tau_1 \ \& \ \Phi_1) <: (\tau_2 \ \& \ \Phi_2)}(S\text{-}Qual)$$

Figure 9: Type Judgements.

right effect Φ_2. The sub effecting relation checks that the finite(resp. infinite) parts of Φ_1 logically implies the finite (resp. infinite) parts of Φ_2, under the assertions implied by the type environment Γ.

5 APPLICATIONS

As explained above, we can check the given type-and- effect temporal properties for an IoT program. This is aligned with the goal of minimizing software errors via the type-based checking mechanism which ensures that well-typed programs cannot go wrong with respect to certain classes of properties. In this section, we show the applicabilities of this technique using more examples (Table 3). Safety and liveness properties are of primary importance and have been extensively investigated for reactive systems [2]. The high water mark of memory usage [32] and the worst-case execution time [34] analyses are particularly beneficial and critical for IoT systems. Thus, we will discuss each of them one by one.

5.1 Safety and Liveness:

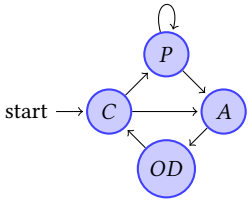
Informally, safety properties assert that *something bad never happens*, while liveness properties assert that *something good will eventually happen*. To formalize these two properties, we abuse the following notations: Σ^ω and Σ^* are the sets of infinite and finite event sequences respectively. We call elements of Σ^ω executions and elements of Σ^* partial executions. For a certain property Φ, we write $\alpha \models \Phi$ when execution α is in property Φ [1]. Here, we define the notion of safety:

Safety. $\forall x : x \in \Sigma^* : x \not\models \Phi \Rightarrow (\forall y : y \in \Sigma^\omega : x \cdot y \not\models \Phi)$
Let Φ be a safety property. $x \not\models \Phi$ refers to the fact that the partial execution x is not in Φ (also called *bad prefix*). The property Φ holds as a safety specification iff: for all *bad prefix* finite sequence x, for all infinite sequences y it holds that the concatenation $x \cdot y$ is not in Φ. In other words, no matter how x is extended, the result will never be shown in Φ. The bad prefix x also can be interpreted as a finite counterexample showing that the safety property is violated. Next, we formalize liveness:

Liveness. $\forall x : x \in \Sigma^* : (\exists y : y \in \Sigma^\omega : x \cdot y \models \Phi)$
The definition does not restrict what a *good thing* can be. y is an infinite collection of discrete events stipulating that some *good thing* eventually happens. Thus property Φ holds as a liveness specification iff: there is a single execution y that can be appended to every partial execution x so that the resulting sequence is in Φ.

We demonstrate that the specifications checked in our technique can be used to capture safety and liveness properties using the example shown in Table 3 **(a)**. Suppose we want to control an initially closed door based on the motion sensor. Once the sensor is activated, the door will be opened for a *delay* period of time. The verified effects for function *motion_sensor* is: $\underline{\textbf{Passive}}^* \cdot \underline{\textbf{Active}} \vee \underline{\textbf{Passive}}^\omega$, and the effects for function *door_control* is: $(\underline{\textbf{Close}} \cdot (\underline{\textbf{Passive}}^* \cdot \underline{\textbf{Active}} \vee \underline{\textbf{Passive}}^\omega) \cdot \underline{\textbf{Open}} \cdot \underline{\textbf{Delay}})^\omega$. Figure 10. shows the infinite transition graph to visualize this specification:
(C = **Close**; P = **Passive**; A = **Active**; OD = **Open** · **Delay**;)

Figure 10: A Transition Graph.



Now, say there is a safety demand which states that the door cannot be opened if the sensor has not been activated - i.e., the bad prefix x = **Passive** · **Open**. Analyzing $\Phi_{door\_control}$ we can easily conclude that the counterexample x will never occur in any of the possible event sequences. Thus the safety property holds. Next, we consider a liveness demand which states that whenever the sensor has been activated, the door should be immediately opened - i.e., the infinite trace $y = \cdots$ **Active** · **Open** $\cdots$ should eventually hold. Again, inspecting $\Phi_{door\_control}$, we can easily conclude that this trace indeed eventually occurs. Thus the liveness property holds too.

5.2 High-Water Mark (HWM) of Resource Usage

To enable reliable IoT applications for small, low power, memory-limited devices, the peak in the utilization of resources need to be priorly considered. Therefore the HWM analyses are necessary. We demonstrate that the specifications checked with our technique can be used to provide HWM information using the example in Table 3 **(b)**. Suppose there is a temperature-controlled system aiming to adjust the temperature using a combination of fan and AC - i.e., if the temperature is too low none of the devices will be operated, while if the temperature is too high both of the devices would be put into operation. The verified effects for function *temperature_control* are: $((t{<}20 \wedge \underline{\textbf{CloseBoth}}) \vee (20{\leq}t{<}30 \wedge \underline{\textbf{Fan}}) \vee (30{\leq}t{<}40 \wedge \underline{\textbf{Alternate}}) \vee (40{\leq}t{<}50 \wedge \underline{\textbf{AC}}) \vee (50{\leq}t \wedge \underline{\textbf{OpenBoth}}))^\omega$. Though the summary

Table 3: Further examples of IoT programs showing the applicabilities of our approach.

(a) Safety and Liveness

```
motion_sensor () =
    if * then
        ev (Active)
    else
        ev (Passive);
        motion_sensor ();

door_control ()=
    ev (Close);
    motion_sensor ();
    ev (Open);
    ev (Delay);
    door_control ();
```

(b) High Water Mark of Resource Usage

```
device_control t =
    if t < 20 then ev (CloseBoth)
    else if 20 <= t < 30
           then ev (Fan)
    else if 30 <= t < 40
           then ev (Alternate)
    else if 40 <= t < 50
           then ev (AC)
    else ev (OpenBoth);

temperature_control () =
    device_control (get_temp ());
    temperature_control ();
```

(c) Worst-Case Execution Time (Higher-order)

```
camera_on = ev (10Mins)
permission t f =
    if * then return ev (Legal)
    else if t == 0 then camera_on ()
    else
        ev (Tick);
        permission (f t) f

surveillance t =
    motion_sensor ();
    permission t (\x -> x - 1);
    ev (CameraOff);
    surveillance t;
```

of the effects in this example is trivial, it actually provides more information when composed with multiple such functions. Based on the analysis in energy/memory consumption for different devices, in the end, we could conclude what is the highest usage of energy/memory and under which circumstances.

5.3 Worst-Case Execution Time (WCET)

WCET is typically used in real-time systems, where understanding the worst case timing behavior of software is crucial for reliability and correct functionality. As per the example (higher-order functions) in Table 3 **(c)**, suppose there is a surveillance system which, whenever it detects a motion activity from the sensor, it sends a permission request for turning on the camera. If a manager proved the legality of the activity, the program simply returns keeping the camera off; otherwise, the system turns on the camera after t occurrences of event **Tick**. The effect of surveillance is: $((\underline{\textbf{Passive}}^* \cdot \underline{\textbf{Active}} \vee \underline{\textbf{Passive}}^\omega) \cdot (\underline{\textbf{Legal}} \vee ((t{<}0 \wedge \underline{\textbf{Tick}}^\omega) \vee (t{\geq}0 \wedge \underline{\textbf{Tick}}^t \cdot \underline{\textbf{10Mins}}))) \cdot \underline{\textbf{CameraOff}})^\omega$. We demonstrate that the specification verified in our technique can be used to estimate WCET. For example, an instance of WCET requires for the estimated time from the point when an intruder entered until the camera is turned on - i.e., the time from **Active** to **10Mins**. A straightforward analysis of the effect concludes that WCET depends on the input value t: if t is negative, the camera will never be turned on; otherwise, it is equivalent to the time needed for t occurrences of **Tick** to be fired.

6 RELATED WORK

Recent years have seen many proposals for abstracting low-level IoT development into different frameworks aiming to reduce the complexity of creating and maintaining such fast-growing systems. Some of these developed reactive languages, such as Céu [31] (2015) and Juniper [11] (2016), are designed for small-scale IoT systems such as micro-controllers. However, their designs are based on imperative(Céu) and non-purely-functional paradigms (Juniper). Thus it is not easy to have the benefits of testability and verifiability obtained from the purity of functional programming.

More recent works are making use of the functional reactive programming paradigm, taking model-based approaches as a mechanism to abstract low-level programming details and processes. For example, the work [5] (2017) allows programmers to write functional reactive IoT programs and generates C code from their DSL embedded in Scala. However, such designs [21][8][7][29] are mostly first-order and still on the stage of exploring more general and suitable abstractions. Besides, none of them provided formal methods for verification based on their proposed approaches.

To formally verify IoT systems, Ruchkin Ivan et al. [30] introduces the *Integration Property Language (IPL)* which primarily reasons and verifies specifications for a robot system. Similarly, [22][25][16] work on extracting the logical specifications from the code, prove the core design using various model checking tools to guarantee specific safety, security and functionality requirements. In contrast, we focus on verifying the IoT code itself using our type-and-effect system which aims to provide tighter guarantees for a run-time IoT system.

As mentioned above, there is always a gap between the functional reactive modeling and the local verification for IoT systems. Bridging this gap could mean exploiting the best of both worlds. Therefore, this work not only designs a functional DSL for IoT development but also aims to verify the temporal properties for our framed higher-order programs locally.

The verification of higher-order programs is getting more and more popular. Many approaches have been proposed for automatically/semi-automatically verifying a wide range of temporal properties employing different techniques. For example, [18] generates a tree representing all the possible event sequences of the transformed higher-order recursion schemes (HORS's), and then the HORS is model-checked, whereas, the termination verification from [20] is based on a reduction to binary reachability analysis via program transformation. The dependent effects of our work is inspired by another community who works on supporting temporal specifications via the type-and-effect systems [12][19][23].

The temporal effects in prior work are simply sets of event traces that coarsely over-approximate the actual temporal behavior of the program terms either via ω-regular sets [12] or by allowing

recursive functions to have infinite effects [19]. However, some of these works preclude value-dependent temporal properties as effects, while others, lost significant precision by performing over-approximation for infinite-state programs. We take inspiration from the benefits of [23], and advance the latter to capture the branching properties of value-dependent effects leading thus to a modular compositional verification strategy without losing precision.

7 CONCLUSION AND FUTURE WORK

In this paper, we design and prototype a domain-specific language, Friot, for functional reactive IoT development. We conclude that using the FRP abstraction reduces the code size and the complexity of an interactive IoT application in general. Besides, we extend the dependent type-and-effect system with disjunctions, leading to a compositional and more precise local reasoning. By making use of the techniques for temporal properties verification, we are able to automatically verify non-trivial properties of higher-order IoT programs. To the best of our knowledge, our work is the first technique that bridges the gap between the FRP IoT system and its local reasoning on event sequences. In the end, we foresee two main future works. (i) We plan to build the effect system on top of Liquid-haskell [33] based on the theoretical guidance from this paper; (ii) We plan to extend this work with concurrency optimization, as our technique currently targets only strict sequential programs. Capturing temporal properties for a concurrent FRP system is considered to be challenging.

REFERENCES

[1] Bowen Alpern and Fred B Schneider. 1985. Defining liveness. *Information processing letters* 21, 4 (1985), 181–185.

[2] Bowen Alpern and Fred B Schneider. 1987. Recognizing safety and liveness. *Distributed computing* 2, 3 (1987), 117–126.

[3] Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate programming for the internet of things. *Computer* 48, 9 (2015), 22–30.

[4] MA Burhanuddin, Ali Abdul-Jabbar Mohammed, Ronizam Ismail, and Halizah Basiron. 2017. Internet of things architecture: Current challenges and future direction of research. *International Journal of Applied Engineering Research* 12, 21 (2017), 11055–11061.

[5] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. 2017. FRP IoT modules as a Scala DSL. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. ACM, 15–20.

[6] Evan Czaplicki. 2012. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University* (2012).

[7] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. 2018. Building IoT Systems Using Distributed First-Class Reactive Programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 185–192.

[8] João Pedro Dias, João Pascoal Faria, and Hugo Sereno Ferreira. 2018. A Reactive and Model-Based Approach for Developing Internet-of-Things Systems. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 276–281.

[9] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 263–273.

[10] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[11] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, 8–16.

[12] Martin Hofmann and Wei Chen. 2014. Abstract interpretation from Büchi automata. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 51.

[13] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. 2013. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*. ACM, 15–20.

[14] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2002. Arrows, robots, and functional reactive programming. In *International School on Advanced Functional Programming*. Springer, 159–187.

[15] John Hughes. 1989. Why functional programming matters. *The computer journal* 32, 2 (1989), 98–107.

[16] Koray İnçki, İsmail Arı, and Hasan Sözer. 2017. Runtime verification of Iot systems using complex event processing. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*. IEEE, 625–630.

[17] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*. Springer, 190–203.

[18] Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 416–428.

[19] Eric Koskinen and Tachio Terauchi. 2014. Local temporal reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 59.

[20] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic termination verification for higher-order functional programs. In *European Symposium on Programming Languages and Systems*. Springer, 392–411.

[21] Haidong Lv, Xiaolong Ge, Hongzhi Zhu, Zhiwei Yuan, Zhen Wang, and Yongkang Zhu. 2018. Designing of IoT Platform Based on Functional Reactive Pattern. In *2018 International Conference on Computer Science, Electronics and Communication Engineering (CSECE 2018)*. Atlantis Press.

[22] Kulani Tharaka Mahadewa, Kailong Wang, Guangdong Bai, Ling Shi, Jin Song Dong, and Zhenkai Liang. 2018. HOMESCAN: Scrutinizing Implementations of Smart Home Integrations. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 21–30.

[23] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A fixpoint logic and dependent effects for temporal property verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 759–768.

[24] Stefan Nastic, Sanjin Sehic, Michael Vögler, Hong-Linh Truong, and Schahram Dustdar. 2013. PatRICIA–a novel programming model for iot applications on cloud platforms. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE, 53–60.

[25] Shinpei Ogata, Hiroyuki Nakagawa, Yoshitaka Aoki, Kazuki Kobayashi, and Yuko Fukushima. 2017. A Tool to Edit and Verify IoT System Architecture Model.. In *MODELS (Satellite Events)*. 571–575.

[26] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. 2002. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, 168–179.

[27] John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. 2001. FVision: A declarative language for visual tracking. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 304–321.

[28] Raspberry Pi. 2015. Raspberry Pi Model B.

[29] Evgeny L Romanov and Galina V Troshina. 2017. The IoT-architecture on the principles of reactive programming. In *2017 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*. IEEE, 317–322.

[30] Ivan Ruchkin, Joshua Sunshine, Grant Iraci, Bradley Schmerl, and David Garlan. 2018. IPL: An integration property language for multi-model cyber-physical systems. In *International Symposium on Formal Methods*. Springer, 165–184.

[31] Francisco Sant'Anna, Roberto Ierusalimschy, and Noemi Rodriguez. 2015. Structured synchronous reactive programming with Céu. In *Proceedings of the 14th International Conference on Modularity*. ACM, 29–40.

[32] Muhammad Syafrudin, Ganjar Alfian, Norma Fitriyani, and Jongtae Rhee. 2018. Performance Analysis of IoT-Based Sensor, Big Data Processing, and Machine Learning Model for Real-Time Monitoring System in Automotive Manufacturing. *Sensors* 18, 9 (2018), 2946.

[33] Niki Vazou, Eric L Seidel, and Ranjit Jhala. 2014. Liquidhaskell: Experience with refinement types in the real world. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 39–51.

[34] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 36.

A Trustworthy Framework for Resource-Aware Embedded Programming

Adam D. Barwell
adb23@st-andrews.ac.uk
University of St Andrews
Scotland, UK

Christopher Brown
cmb21@st-andrews.ac.uk
University of St Andrews
Scotland, UK

ABSTRACT

Systems with *non-functional* requirements, e.g. energy consumption or execution time, are of increasing importance due to the proliferation of devices with limited resources such as drones, wireless sensors, and tablet computers. *Drive*, developed collaboratively between the University of St Andrews and Inria, Rennes, is a framework for forming contracts to support the developer with the tools and techniques needed to meet non-functional specifications. In this paper, we will investigate the formal specification of the annotations and assertions that are included as part of *Drive* for a representative subset of C, $C_{\subset}$. We define both an improved abstract interpretation that automatically derives proofs of assertions, and will define inference algorithms for the derivation of both abstract interpretations and the context over which the interpretation is indexed. We use the dependently-typed programming language, Idris, to ensure the soundness of our abstract interpretation, and will prove the soundness of our context inference algorithm with respect to the operational semantics of $C_{\subset}$.

KEYWORDS

Dependent Types, Idris, Lightweight Verification, Non-Functional Properties, Abstract Interpretation, Proof-Carrying Code, Embedded Systems

ACM Reference Format:
Adam D. Barwell and Christopher Brown. 2020. A Trustworthy Framework for Resource-Aware Embedded Programming. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Programs that consider non-functional properties, such as energy consumption or maximum execution time, are of increasing importance due to the proliferation of devices with limited resources; e.g. embedded medical devices, drones, wireless sensors, mobile phones and tablets. Whilst conventional understanding of software correctness pertains to the *functional* properties of a program, such as the absence of errors and bugs, resource-limited embedded devices prompt additional conformance to *non-functional* requirements [24]. A system that does not conform to its non-functional specification may ultimately render the system useless, or worse, a potential danger to others; e.g. a drone depleting its battery before it can land safely. It is therefore necessary to develop such systems with an awareness of, and a demonstration of conformity to, their (non-functional) specification.

Drive [7] is a framework for capturing, and reasoning about, non-functional properties in C programs. It includes the Contract Specification Language (CSL), an Embedded Domain Specific Language that defines C-statement annotations in order to capture non-functional properties of the statements they annotate, including energy usage, worst-case execution time (WCET), and the degree of vulnerability to side-channel attacks. This non-functional information is captured using existing tools and/or approaches, such as OTAWA [4], TimeWeaver [20], and Chronos [22] for WCET, and those provided by Eder *et al.* [25] for energy consumption. The information derived from such techniques is made available to the programmer as a value assigned to a given variable, thus making captured non-functional information *first-class citizens. Drive* also facilitates reasoning about non-functional properties via assertion annotations; e.g. whether a statement can be executed within an energy budget. These contracts are automatically verified using an abstract interpretation implemented in the dependently-typed programming language, Idris [6]. This lightweight approach to verification is a form of *proof-carrying code* [26] since the abstract interpretation automatically derives a proof of whether each assertion holds true for a given context.

In order for the proofs produced by *Drive* to be meaningful and trustworthy, the framework must itself be demonstrably sound. In the full paper, we will investigate the soundness of the *Drive* system for a subset of the C language in terms of the construction of abstract interpretation and inference of contexts. We will introduce a novel abstract interpretation capable of reasoning about non-ground assertions, providing a comparison with the original defined by Brown *et al.* [7]. We use Idris to implement our parser and abstract interpretation in type theory. In line with the Curry-Howard correspondence [29, 31], we will formulate our definitions of language, properties, rewrites, and logical and arithmetic formulæ as types, and transformations over types that enact rewrites or determine proofs of properties as total functions. Type-checking ensures the soundness of these functions relative to the definitions given as types, thus ensuring soundness of abstract interpretation and context inference.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1.1 Contributions

The final paper will have the following contributions.

(1) A formal definition of annotations within the Contract Specification Language [7] for a representative subset of C, denoted $C_{\subset}$.
(2) A novel abstract interpretation of well-formed annotated programs in $C_{\subset}$ that facilitates the lightweight verification of compliance to non-functional specifications, expressed via assertion annotations.
(3) An approach to automatic inference of contexts and abstract interpretations of well-formed annotated $C_{\subset}$ programs that is sound with respect to the definitions of $C_{\subset}$, well-formedness, contexts, and our abstract interpretation, demonstrating that the principles behind the *Drive* framework are trustworthy.

2 THE *DRIVE* SYSTEM

The *Drive* system [7], developed collaboratively between the Unveristy of St Andrews and Inria, Rennes, is a framework for forming contracts to support both the experienced and inexperienced software developer with the tools and techniques needed to reason about non-functional properties of their source code. *Drive* is made up of two components: the *Contract Specification Language*, and an abstract interpretation for the production of *certificates*.

The Contract Specification Language (CSL) [7] is an embedded domain-specific language that extends C with special annotations describing non-functional properties of source code and assertions (or contracts) around those properties. Listing 1 shows an extract from the Levenshtein Distance algorithm, implemented in C and taken from the BEEBs benchmarks [28], to which the programmer has introduced CSL annotations. For example, at Line 16, the programmer annotates the source code with a call to `__csl_time_worst()` passing in a reference to the variable `loop_time` (declared on Line 5). This annotation uses a trusted external tool to infer the worst-case execution time of the following loop (starting on Line 17) and assigns the value to `loop_time`. Since the value of the captured non-functional property is assigned to `loop_time`, the programmer can subsequently use this information within the program itself as a first-class citizen, both as part of standard C statements but also within CSL assertion annotations.

CSL allows the programmer to specify assertions in source code, expressing specifications that can be proved to hold true or false. An example of such an assertion is given on Line 30 of Listing 1. Here, the programmer wishes to assert that the worst-case execution time of the loop at Line 17 (assigned to the variable `loop_time` on Line 16) does not exceed some upper bound of execution time (i.e `tl * sl * OP_Time`). Brown *et al.* define assertion expressions as being logical and arithmetic expressions comprising: conjunction and disjunction over logical expressions; and equality and inequality over arithmetic expressions. Arithmetic expressions are defined to comprise: literal natural numbers, variables, logarithms, addition, subtraction, multiplication, division, and the modulo operator. The proof of whether an assertion holds true is inferred by an underlying dependently-typed implementation; the resulting proof, i.e. the certificate, is passed back to the programmer.

Listing 1: An extract of the Levenshtein algorithm from the BEEBs benchmarks suite for C

```
1  int levenshtein_distance(const char *s, const char *t) {
2    int i, j;
3    int sl = strlen(s);
4    int tl = strlen(t);
5    unsigned int loop_time;
6    int d[sl + 1][tl + 1];
7
8    for (i = 0; i <= sl; i++) {
9      d[i][0] = i;
10   }
11
12   for (j = 0; j <= tl; j++) {
13     d[0][j] = j;
14   }
15
16   __csl_time_worst(&loop_time);
17   for (j = 1; j <= tl; j++) {
18     for (i = 1; i <= sl; i++) {
19       if (s[i - 1] == t[j - 1]) {
20         d[i][j] = d[i - 1][j - 1];
21       }
22       else {
23         d[i][j] = min(d[i - 1][j] + 1, // deletion
24                       min(d[i][j - 1] + 1, // insertion
25                           d[i - 1][j - 1] + 1)); // substitution
26       }
27     }
28   }
29
30   __csl_assert(loop_time <= tl*sl * OP_TIME);
31   return d[sl][tl];
32 }
```

3 PROOF-CARRYING CODE AND DEPENDENT TYPES

Proof-carrying code [26] is the concept that source code must be supplied with an automatically checkable proof that attests to the code's adherence to some specification. By providing proofs of both functional and non-functional properties that demonstrate compliance to some specification, both systems and programmers alike can have confidence that a program will be behave in a certain way. Accordingly, programs become more *trustworthy*.

To support proof-carrying code, a language must support the robust mechanisms that allow properties and proofs of properties to be expressed. The well-known Curry-Howard correspondence tells us that, given a suitably rich type system, (certain kinds of) proofs can be represented as programs [31]. Dependently typed languages take advantage of this correspondence and represent a convenient vehicle for proof-carrying code [29]. For languages with insufficiently rich type systems, such as C, dependently-typed languages can be used to produce an *abstract interpretation* [12] of a given program in those languages. Such abstract interpretations can be used to derive proofs of the desired properties [2].

In the case of dependently-typed languages, under the propositions as types view, dependent types are used to represent predicates [32]. For example, `(Even : (n : Nat) -> Type)` defines the type of evidence (or proofs) that a natural number, `n`, is even. In cases where the property does *not* hold true, e.g. `Even 1`, and assuming a suitably restricted definition of that property, the type is uninhabited. An uninhabited type represents falsity. Evidence that a

predicate does not hold true can be represented by the type function, `(Not a = a -> Void)`, where `a` is a type variable and `Void` is the empty type; i.e. it has no constructors. Using dependent types in this way, properties that represent a (non-)functional specification can be encoded as predicates (i.e. types). Accordingly, total functions, `f : A -> B`, allow for the derivation of evidence that the predicate `B` can be constructed given evidence of `A`. Type-checking ensures the soundness of these functions [29].

We take advantage of the above features by implementing our system in the dependently-typed language Idris [6]. The syntax of Idris is similar to Haskell [19], and like Haskell, Idris supports algebraic data types with pattern matching, type classes, and do-notation. Unlike Haskell, Idris evaluates its terms eagerly. Definitions, e.g. of languages and well-formedness, are defined by giving their definitions as types in Idris. For example, the aforementioned `Even` predicate can be defined:

```
1  data Even : (n : Nat) -> Type where
2    Zero : Even Z
3    Succ : (ek : Even k) -> Even (S (S k))
```

where, `Even` is the name of the type being defined, `Nat` is the type of natural numbers, `(n : Nat)` is a (named) argument to the type, and both `Zero` and `Succ` are constructors. Constructors may have (named) arguments; e.g. `(ek : Even k)`. Constructors may also restrict the values of their arguments; e.g. `Zero` explicitly states that `n = 0`, and `Succ` states that `n = S (S k)` (i.e. `k + 2`), given an inhabitant of `Even k`. As is desired, under this definition, there is no way to construct an element of, e.g., `(Even 1)` using either constructor.

In order to determine whether `(Elem n)` is inhabited for a given `n`, we define the function `isEven`.

```
1  isEven : (n : Nat) -> Dec (Even n)
2  isEven Z = Yes Zero
3  isEven (S k) with (k)
4    isEven (S k) | Z = No absurd
5    isEven (S k) | (S j) with (isEven j)
6      isEven (S k) | (S j) | (Yes prf) = Yes (Succ prf)
7      isEven (S k) | (S j) | (No contra) =
8        No (\(Succ x) => contra x)
```

Here, `Dec` is the type for a decidable property, where `Yes` holds a proof of the property and `No` holds a proof of its contradiction. The `with` rule is used to pattern matche on intermediate values, similar to a `case` expression in Haskell. The function `(absurd : Uninhabited t => t -> a)` is a convenience function for contradictions where a type, here `(Even 1)`, is uninhabited. The argument to `No` on Line 8 is a function of type `(Even (S (S j)) -> Void)` and represents a contradiction of `(Even (S (S j)))` when `(Even j)` is uninhabited. Since this is a total definition, as guaranteed by the type-checker, `isEven` is a decision procedure for the type (predicate) `(Elem n)` for all values of `n`.

4 C-SUBSET LANGUAGE, $C_\subset$

Although CSL is targeted at C, our formalism is based on a representative subset. Our C-subset, denoted $C_\subset$, can be seen as an extension of a standard While language [27], adding functions and arrays. By embedding CSL in $C_\subset$, we are able to keep our definitions and proofs simple and manageable whilst retaining the ability to reflect the fundamental characteristics of the system.

The definition of $C_\subset$ is given in Figure 1. A program $p : C_\subset$ comprises one or more function declarations. A function declaration

$$
\begin{array}{rcl}
a, a_1, a_2, a_n : \text{AExp} & ::= & n \mid x \mid y[a_1]\cdots[a_n] \\
& \mid & a_1 + a_2 \mid a_1 \times a_2 \mid a_1 - a_2 \\
b, b_1, b_2 : \text{BExp} & ::= & \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \\
& \mid & \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
s, s_1, s_2 : \text{Stmt} & ::= & x := a \\
& \mid & x := f(x_1, \ldots, x_i)(xs_1, \ldots, xs_j) \\
& \mid & xs[a_1]\cdots[a_k] \\
& \mid & xs[a_1]\cdots[a_k] := a \\
& \mid & \text{skip} \\
& \mid & s_1 \,;\, s_2 \\
& \mid & \text{if } b \text{ then } s_1 \text{ else } s_2 \\
& \mid & \text{while } b \text{ do } s \\
p, p_1, p_2 : C_\subset & ::= & \text{fun } f(x_1, \ldots, x_i, xs_1 \ldots, xs_j) \text{ ret } x \\
& & \text{do } s \\
& \mid & w1 \,;\, w2
\end{array}
$$

Figure 1: Definition of $C_\subset$.

comprises: a name, f : FVar; zero or more numeric parameters, $x_1, \ldots, x_i$: AVar; zero or more array parameters, $xs_1, \ldots, xs_j$: LVar; a numeric variable whose value will be returned by the function, x : AVar; and the body of the function, s : Stmt. One consequence of the above is that higher-order functions are not expressible. A function body is a statement, defined as being: an assignment, $x := a$; a function call assigning the returned value, $x := f(x_1, \ldots, x_i)(xs_1, \ldots, xs_j)$; an array declaration, $xs[a_1]\cdots[a_n]$; an array assignment, $xs[a_1]\cdots[a_n] := a$; the empty statement, skip; a pair of statements, $s_1; s_2$; an if-statement,if b then s_1 else s_2; or a while-statement, while b do s. Numeric parameters to function calls are arithmetic expressions, but array parameters are array variables. If- and while-statements take a boolean condition expression, b : BExp. Boolean expressions comprise: true and false; definitional equality, $a_1 = a_2$; inequalities, $a_1 \leq a_2$; negation, $\neg b$; conjunction, $b_1 \wedge b_2$; and disjunction, $b_1 \vee b_2$. Finally, arithmetic expressions, a : AExp, comprise: natural numbers, n; numeric variables, x; array accesses $xs[a_1]\cdots[a_n]$; and addition, multiplication, and subtraction over natural numbers.

Meta-variables include: $i, j, k, n : \mathbb{N}$, ranging over natural numbers; $x, x_1, \ldots, x_i$: AVar, ranging over variables that are substituted for natural numbers; $xs, xs_1, \ldots, xs_j$: LVar, ranging over variables that are substituted for arrays; and f, g, h : FVar ranging over function names. Where additional variables are used, we will follow the standard convention of singular variable names for numeric variables and plural variable names for array variables.

Example 4.1. Given the C definition of the Levenshtein Distance function in Listing 1, Listing 2 represents an equivalent definition in $C_\subset$. Here, the parameters `s` and `t` are assumed to be arrays of natural numbers (potentially encoding characters) and are denoted `ss` and `ts`. The lengths of the input strings are also assumed to be provided as the arguments `sl` and `tl`. For-statements are replaced by equivalent combinations of assignment and while-statements, and the return statement on Line 31 of Listing 1 is replaced by an assignment to the designated return variable, `dist`. For compound statements, $s_1; s_2$, where $s1$ is an if- or while-statement, indentation is used to disambiguate between s_2 and the if- or while-statement

Listing 2: Reformulation of Listing 1 in $C_\subset$

```
1  fun ld(sl,tl)(ss,ts) ret dist do
2    ds[sl + 1][tl + 1];
3
4    i := 0;
5    while i <= sl do
6      d[i][0] := i;
7      i := i + 1;
8
9    j := 0;
10   while j <= tl do
11     d[0][j] := j;
12     j := j + 1;
13
14   i := 1;
15   j := 1;
16   while j <= tl do
17     while i <= sl do
18       if ss[i - 1] = ts[j - 1]
19       then
20         ds[i][j] := ds[i - 1][j - 1]
21       else
22         x := ds[i - 1][j] + 1;
23         y := ds[i][j - 1] + 1;
24         z := ds[i - 1][j - 1] + 1;
25         r := min(x,y,z);
26         ds[i][j] := r;
27       i := i + 1;
28       j := j + 1;
29   dist := ds[sl][tl]
```

body. As in the C definition, we omit the `min` definition, but this can be defined in the obvious way: a combination of if-statements using `<=` to test which of `x`, `y`, and `z` has the smallest value.

4.1 Implementation and Well-Formedness

We assume that abstract interpretations are derived only from well-formed code. In order to derive contexts and abstract interpretations of a $C_\subset$ program, we encode $C_\subset$ as the type `CSbst`, defined in Listing 3. `CSbst` is indexed by three vectors, `vs`, `ls`, and `fs`, representing sets of numeric variables, array variables, and function symbols; i.e. AVar, LVar, and FVar, respectively. Variables are referenced by a proof of their existence in `vs`, `ls`, or `fs` using the `Elem` type; e.g. `(Elem NumVar vs)` on Line 5 of Listing 3 for numeric variables. This ensures that variable symbols are consistently typed. Moreover, array variables are indexed by their dimensions in order to ensure that, e.g., a program does not access an element three dimensions deep, `xs[i][j][k]`, when `xs` is declared to be a two-dimensional array. Function symbols are similarly indexed by their number of both numeric and array arguments to ensure correct application statements. Where `CSbst` encodes $C_\subset$, the concomitant types `Stmt`, `BExp`, and `AExp` encode Stmt, BExp, and AExp, respectively.

Example 4.2. In the final paper we will show an encoding of the Levenshtein Distance function in Listing 2 in `CSbst`.

Although type-safety is encoded in `CSbst`, `Stmt`, `BExp`, and `AExp`, correct scoping is not. In $C_\subset$, numeric variables must be assigned before use, array variables must be declared prior to assignment or access expressions, and functions must be defined before they can be called in subsequent function definitions. We note that this restriction disallows both recursive and mutually recursive functions. In order to enforce well-formedness, we define additional types,

Listing 3: Encoding of $C_\subset$ in Idris.

```
1  data AExp : (vs : Vect vn (Var Numerical))
2           -> (ls : Vect ln (Var Array))
3           -> (fs : Vect fn (Var Function)) -> Type where
4    Val : (n : Nat) -> AExp vs ls fs
5    Var : (x : Elem NumVar vs) -> AExp vs ls fs
6    Acc : (xs : Elem (LstVar dims) ls)
7       -> (idcs : Vect dims (AExp vs ls fs)) -> AExp vs ls fs
8    Add : (a1 : AExp vs ls fs)
9       -> (a2 : AExp vs ls fs) -> AExp vs ls fs
10   Mul : (a1 : AExp vs ls fs)
11      -> (a2 : AExp vs ls fs) -> AExp vs ls fs
12   Sub : (a1 : AExp vs ls fs)
13      -> (a2 : AExp vs ls fs) -> AExp vs ls fs
14
15 data BExp : (vs : Vect vn (Var Numerical))
16          -> (ls : Vect ln (Var Array))
17          -> (fs : Vect fn (Var Function)) -> Type where
18   TT : BExp vs ls fs
19   FF : BExp vs ls fs
20   Eq : (a1 : AExp vs ls fs)
21     -> (a2 : AExp vs ls fs) -> BExp vs ls fs
22   LTE : (a1 : AExp vs ls fs)
23      -> (a2 : AExp vs ls fs) -> BExp vs ls fs
24   Neg : (b : BExp vs ls fs) -> BExp vs ls fs
25   And : (b1 : BExp vs ls fs)
26      -> (b2 : BExp vs ls fs) -> BExp vs ls fs
27   Or  : (b1 : BExp vs ls fs)
28      -> (b2 : BExp vs ls fs) -> BExp vs ls fs
29
30 data Stmt : (vs : Vect vn (Var Numerical))
31          -> (ls : Vect ln (Var Array))
32          -> (fs : Vect fn (Var Function)) -> Type where
33   VAsn : (x : Elem NumVar vs)
34       -> (a : AExp vs ls fs) -> Stmt vs ls fs
35   FAsn : (x : Elem NumVar vs)
36       -> (f : Elem (FunVar vargs largs) fs)
37       -> (vparams : Vect vargs (AExp vs ls fs))
38       -> (lparams : Vect largs (Elem (LstVar dims) ls))
39       -> Stmt vs ls fs
40   LDec : (xs : Elem (LstVar d) ls)
41       -> (dims : Vect d (AExp vs ls fs))
42       -> Stmt vs ls fs
43   LAsn : (xs : Elem (LstVar dims) ls)
44       -> (idcs : Vect dims (AExp vs ls fs))
45       -> (a : AExp vs ls fs)
46       -> Stmt vs ls fs
47   Skip : Stmt vs ls fs
48   Comp : (s1 : Stmt vs ls fs)
49       -> (s2 : Stmt vs ls fs)
50       -> Stmt vs ls fs
51   If   : (b : BExp vs ls fs)
52       -> (s1 : Stmt vs ls fs)
53       -> (s2 : Stmt vs ls fs)
54       -> Stmt vs ls fs
55   Iter : (b : BExp vs ls fs)
56       -> (s1 : Stmt vs ls fs)
57       -> Stmt vs ls fs
58
59 data CSbst : (vs : Vect vn (Var Numerical))
60           -> (ls : Vect ln (Var Array))
61           -> (fs : Vect fn (Var Function)) -> Type where
62   Fun   : (f : Elem (FunVar nvps nlps) fs)
63        -> (vps : Vect nvps (Elem NumVar vs))
64        -> (lps : Vect nlps (Elem (LstVar dims) ls))
65        -> (ret : Elem NumVar vs)
66        -> (body : Stmt vs ls fs)
67        -> CSbst vs ls fs
68   WComp : (w1 : While vs ls fs)
69        -> (w2 : While vs ls fs)
70        -> CSbst vs ls fs
```

Listing 4: Well-Formedness Predicate Indexed by `Stmt`.

```
1  data WFStmt : (s : Stmt vs ls fs)
2             -> (ws : Vect nws (Elem NumVar vs))
3             -> (ms : Vect nms (Elem m ls))
4             -> (gs : Vect ngs (Elem g fs)) -> Type where
5    VAsn : (wfa : WFAExp a ws ms gs)
6         -> (prop : Not (Elem x ws))
7         -> WFStmt (VAsn x a) (x::ws) ms gs
8    VReAsn : (wfa : WFAExp a ws ms gs)
9           -> (prop : Elem x ws)
10          -> WFStmt (VAsn x a) ws ms gs
11   LDec : (wfdims : AllWFAExp dims ws ms gs)
12        -> (prop : Not (Elem xs ms))
13        -> WFStmt (LDec xs dims) ws (xs::ms) gs
14   LReDec : (wfdims : AllWFAExp dims ws ms gs)
15        -> (prop : Elem xs ms)
16        -> WFStmt (LDec xs dims) ws (xs::(dropElem ms prop)) gs
17   LAsn : Elem xs ms
18        -> (wfdims : AllWFAExp dims ws ms gs)
19        -> (wfa : WFAExp a ws ms gs)
20        -> WFStmt (LAsn xs idcs a) ws ms gs
21   Skip : WFStmt Skip ws ms gs
22   Comp : (wfs1 : WFStmt s1 ws1 ms1 gs)
23        -> (wfs1 : WFStmt s2 ws2 ms2 gs)
24        -> (prop1 : SuffixOf ws1 ws2)
25        -> (prop2 : SuffixOf ms1 ms2)
26        -> WFStmt (Comp s1 s2) ws ms gs
27   If   : (wfb : WFBExp b ws ms gs)
28        -> (wfs1 : WFStmt s1 ws ms gs)
29        -> (wfs2 : WFStmt s2 ws ms gs)
30        -> WFStmt (If b s1 s2) ws ms gs
31   Iter : (wfb : WFBExp b ws ms gs)
32        -> (wfs1 : WFStmt s1 ws ms gs)
33        -> WFStmt (Iter b s1) ws ms gs
```

`WFCSbst`, `WFStmt`, `WFBExp`, and `WFAExp`, that are indexed over `CSbst`, `Stmt`, `BExp`, and `AExp`, respectively. Listing 4 gives the definition of `WFStmt`. Here, `ws`, `ms`, and `gs` are vectors denoting the numeric variables, array variables, and function symbols that are in scope within the context of the given statement, `s`. We refer to the set of variables that are in scope for `s` its *environment*, in order to avoid confusion with *contexts* defined in Section 6. Numeric variables are added to the environment upon assignment, via the `(x::ws)` term in the `VAsn` constructor on Lines 5–6 when `x` is fresh. Reassignment of numeric variables do not update the environment (`VReAsn`). Array element assignments (`LAsn`) require that the referenced array, `xs`, is in the environment via the `(Elem xs ms)` argument to the constructor. Fresh array variables are added to the environment upon declaration (`LDec`); when re-declaring an array variable, `xs`, the old declaration is replaced by the new declaration in order to ensure that the recorded dimensions of `xs` in the environment are correct (`LReDec`). All elements of a declared array are defined to be 0. Both the specified return variable and variables defined as parameters to functions are deemed to be in the environment in function bodies; global variables are not permitted. `WFCSbst`, `WFBExp`, and `WFAExp` are defined in a similar manner, with numeric variable and array access `WFAExp` constructors requiring proofs that the variable or array being accessed is in the environment.

Example 4.3. In the final paper we will show the Levenshtein Distance function in Example 4.2 as an inhabitant of `WFCSbst`.

In the final paper, we will define a decidable covering function for well-formedness, deriving an instance of `Dec (WFSbst s ws ms gs)` for a given statement `s`, and similar instances for `WFCSbst`, `WFBExp`, and `WFAExp`. Additionally, we will define an operational semantics for $C_{\subset}$ as a set of rewrite rules. For practicality reasons, the last defined function will be treated as the main function and evaluated first. Of immediate semantic interest are the semantics compound statements, $s_1; s_2$, where s_1 is an if- or while-statement whose evaluated sub-statement contain one or more assignments, the variables that are assigned will maintain their assigned values in s_2. For example, in Listing 2, the value of `i` will be incremented due to the assignment on Line 7 until the value of `i` causes the boolean condition on Line 5 evaluates to false, i.e. `sl + 1`; `i` will be assigned to `sl + 1` in Lines 8–13, and `1` at Line 14 where it is reassigned. We intend to demonstrate that the rewrite rules that make up our semantics are both sound and confluent.

5 EXPRESSING NON-FUNCTIONAL PROPERTIES

CSL defines a number of annotations that capture non-functional information and enable the programmer to express assertions. Capture annotations assign a corresponding non-functional measurement to the variable that is passed to the annotation. Assertion annotations contain an expression that may use one or more variables. We define two types indexed by `AWFCSbst` and `WFStmt`, respectively: `AWFCSbst`, for well-formed programs; and `AWFStmt`, for well-formed statements. Intuitively, `AWFStmt` extends `WFStmt` by three constructors: `Cert`, defining assertion annotations; and `NFPCAsn` and `NFPCReAsn`, both defining non-functional property capture annotations that assign to fresh or previously assigned variables. Otherwise, constructors make no additional requirements or restrictions on the respective well-formedness definitions. For example, `AWFCSbst` is defined:

```
1  data AWFCSbst : (wfp : WFCSbst p gs)
2               -> (anns : WFNFPCAnnotations as rfs) -> Type where
3    Fun  : (awfs : AWFStmt wfBody anns)
4        -> AWFCSbst (Fun wfBody) anns
5    FComp : (awfp1 : AWFCSbst wfp1 anns)
6         -> (awfp2 : AWFCSbst wfp2 anns)
7         -> AWFCSbst (FComp wfp1 wfp2 prop) anns
```

where, `Fun` defines an annotated function declaration to be a well-defined function declaration, taking a proof of its body being an annotated statement. Similarly, `FComp` defines an annotated compound function declaration such that the comprising declarations are annotated function declarations.

Example 5.1. In the final paper we will extend the well-formed Levenshtein Distance program in Example 4.3 with example annotations, reflecting those in Listing 1.

Listing 5 gives a partial definition of `AWFStmt`, focussing on the constructors that correspond to annotations. Assertion annotations (Lines 5–10) are straightforward, comprising a boolean expression (`b`, Line 8) and the statement being annotated (`awfs`, Line 9). Assertion annotations do not affect the semantic meaning of `awfs`, and exist to be automatically proven by our abstract interpretation (defined in Section 6). The boolean expression, `(b : LimWFBExp ws)`, represents a subset of well-formed boolean (and concomitant arithmetic) expressions, `WFBExp`. `LimWFBExp` excludes `TT` and `FF` constructors; the subset of arithmetic expressions, `LimWFAExp`, exclude array accesses. Both `LimWFBExp` and `LimWFAExp` represent a simplification in automatically determining whether assertions hold; we intend to incorporate the full `WFBExp` and `WFAExp` in the final version.

Listing 5: Partial definition of Annotated Statements.

```
1  data AWFStmt : (wfs : WFStmt s ws ms gs)
2               -> (anns : WFNFPCAnnotations as rfs) -> Type where
3    {- Other constructors correspond to WFStmt constructors. -}
4
5    Cert : (b : LimWFBExp ws)
6        -> (awfs : AWFStmt wfs anns)
7        -> AWFStmt wfs anns
8
9    NFPCAsn  : (x : Elem NumVar vs)
10           -> (rfPtr : Elem (NFPCAnn (NFPCFunSymb rf)) as)
11           -> (wfs : WFStmt (Comp (VAsn x (Val (rf wfs0))) s) ws
         ms gs)
12           -> (propws : SuffixOf ws0 ws)
13           -> (propx : Not (Elem x ws0))
14           -> AWFStmt wfs anns
15
16   NFPCReAsn  : (x : Elem NumVar vs)
17              -> (rfPtr : Elem (NFPCAnn (NFPCFunSymb rf)) as)
18              -> (wfs : WFStmt (Comp (VAsn x (Val (rf wfs0))) s)
         ws ms gs)
19              -> (propx : Elem x ws)
20              -> AWFStmt wfs anns
```

Capture annotation statements that assign to a fresh numeric variable are defined by `NFPCAsn` (Lines 12–22). These statements are indexed by a non-functional property (NFP) capture function (`rf`, Line 18), which is used to derive the value that is assigned to the given variable (`x`, Line 16). A NFP function, `(nfpf : WFStmt s ws ms gs -> Nat)`, represents some tool or analysis that is performed over a given program and returns a quantification of some non-functional property pertaining to the annotated statement. For example, as in Line 16 of Listing 1, an annotation might measure the worst-case execution time of a following for-loop. Alternative examples include how much memory a statement uses, or how much energy is likely to be consumed by the statement. In *Drive*, these resource analyses are derived from external tools or models [7]. These can either be *trusted*, or may be themselves proof-carrying.

Example 5.2. In the final paper, we will define a simple abstract resource analysis function in Idris, measuring either time or memory usage, in order to demonstrate our approach.

Example 5.3. In the final paper, we will demonstrate the use of our exemplar resource analysis function given in Example 5.2 on the annotated well-formed program in Example 5.1

Irrespective of a NFP function's concrete definition, we represent one such function by the type `NFPCFunctionSymbol`.

```
1  data NFPCFunctionSymbol : Type where
2    NFPCFunSymb : (nfpf : WFStmt s ws ms gs -> Nat)
3                -> NFPCFunctionSymbol
4
5  data NFPCAnnotation : (rf : NFPCFunctionSymbol) -> Type where
6    NFPCAnn : (rf : NFPCFunctionSymbol) -> NFPCAnnotation rf
7
8  data WFNFPCAnnotations : (as : Vect nas (NFPCAnnotation rf))
9                         -> (rfs : Vect nas NFPCFunctionSymbol)
10                        -> Type where
11   Nil : WFNFPCAnnotations [] []
12   Cons : (a : NFPCAnnotation rf)
13       -> (prop1 : Elem rf rfs)
14       -> (prop2 : WFNFPCAnnotations as (dropElem rfs prop1))
15       -> WFNFPCAnnotations (a::as) rfs
```

A NFP capture annotation is defined for each NFP function, such that there is a bijective mapping between capture annotations and capture functions. We define `NFPCAnnotation` to represent NFP capture annotations. Finally, `WFNFPCAnnotations` specifies that for each capture annotation, `a`, in a given set of annotations, `as`, the NFP function is in the context, `prop1`, and no other capture annotation is mapped to that same capture function `prop2`. As with numeric and array variables, capture annotations are referred to by the proof of their inclusion in the context (vector, `as`).

Listing 6: Assertion Annotation Certificate Definition.

```
1  data Assertion : {ws : Vect nws (Elem NumVar vs)}
2                -> {wfs : WFStmt s ws ms gs}
3                -> (awfs : AWFStmt wfs anns)
4                -> (lwfb : LimWFBExp ws)
5                -> Type where
6    MkAssertion : Assertion (Cert lwfb k) lwfb
7
8  isVerifLimWFBExp : (ctx : Vect nws (Assignment awfs))
9                   -> (lwfb : LimWFBExp ws)
10                  -> Dec (VerifLimWFBExp ctx lwfb)
11
12 data Cert  : (ctx : Vect nws (Assignment awfs))
13           -> (awfs : AWFStmt wfs anns)
14           -> {prop : Assertion awfs lwfb}
15           -> (prf : Dec (VerifLimWFBExp ctx lwfb))
16           -> Type where
17   MkCert : (ctx : Vect nws (Assignment awfs))
18         -> Cert ctx (Cert lwfb k) (isVerifLimWFBExp ctx lwfb)
```

An annotated statement, `AWFStmt`, is indexed by the set of capture annotations, `as`, that adheres to the aforementioned bijection. An annotation statement, i.e. `NFPCAsn` or `NFPCReAsn`, is similarly indexed by the specific capture annotation, `rfPtr`, that it represents. Since annotation statements are defined as an assignment statement, `NFPCAsn` and `NFPCReAsn` are indexed by a compound statement, `wfs`, that first assigns the result of the NFP capture function to the given variable, then continues with the annotated statement, `s`. Since `NFPCAsn` defines the case when `x` is a fresh numeric variable, `propws` is a proof that `x` has been added to the context, `ws`, of `s`.

6 ABSTRACT INTERPRETATION

Given an annotated program, we certify whether each assertion holds automatically via an abstract interpretation. This corresponds directly to the original abstract interpretation defined by Brown *et al.* [7]. Here, we define a *novel* annotation interpretation. Our motivation for defining a novel interpretation is threefold.

(1) We make fewer assumptions in the novel interpretation; e.g. that variables are within the environment and/or context, and that we are given normalisation functions for arithmetic and boolean expressions. This has the additional benefit of generally requiring fewer separate mechanical parts; e.g. we no longer require separate normalisation functions since arithmetic expressions are indexed by the value that they represent, and boolean values are expressed by type inhabitance.
(2) Improved variable reification; i.e. we avoid silent failure of string-lookup in lists of string-value pairs through the use of proofs of membership. Failure to find a value is now explicitly represented in the derived certificate; we intend to use this as a base for investigating in the final paper type-safe (in)equation solving for non-ground terms.

(3) Derived certificates are now defined as proofs over the *assertion expressions themselves*, as opposed to their normalisation in the original interpretation. For example, the proof of the simple assertion expression `0 <= 1` is given as the term `(LTE Val Val (Yes LTEZero): VerifLimWFBExp ctx (LTE (Val 0) (Val 1))` in our novel interpretation definition, and as `(Yes LTEZero)` in our original interpretation definition.

We note that whilst we currently have fewer arithmetic operations in our novel interpretation definitions, the set of operations can be extended as desired. Additional operations may include additional appropriate predicates; e.g. in the case of division (`div`) in order to ensure, by definition, that `div` will be applied only to those values for which it is defined. In the final paper, we will provide a detailed comparison of the two abstract interpretation definitions.

We define a model of an annotated program, `awfp : AWFCSbst`, as the vector of all assertion annotations, `Cert : LimWFBExp ws -> AWFStmt wfs anns -> AWFStmt wfs anns`, in `awfp`. In the final paper, we will provide this definition as a type and its associated coverage function. Since certificates are indexed by a *context*, we will also define the inference function for contexts, and give a soundness proof with respect to the operational semantics of $C_{\subset}$ that will be defined in Section 4.

Example 6.1. In the final paper, we will demonstrate the derivation of an abstract interpretation for the Levenshtein Distance annotated representation in Example 5.1.

6.1 Assertions as Proof Terms

Each assertion is represented by a certificate, `Cert`, defined in Listing 6. A certificate is indexed by: a context, `ctx`, mapping numeric variables to values; the assertion annotation, `awfs = (Cert lwfb k)`; and a proof, `prf`, of whether the assertion holds true under the given context. The `Assertion` type is used to relate an assertion, `lwfb`, and `awfs`, since `AWFStmt` is not indexed by its assertions. A certificate is constructed using the constructor, `MkCert`, which takes a context, and derives `prf` automatically via the decision procedure `isVerifLimWFBExp` (not defined in Listing 6, but a definition will be provided in the final paper).

An assertion is deemed to hold true for a given context when `VerifLimWFBExp` is *inhabited* for a given assertion and context. Conversely, if `VerifLimWFBExp` is *uninhabited* for a given assertion and context, then the assertion is deemed to not hold true for that context. The definition of `VerifLimWFBExp` is given in Listing 7.

`VerifLimWFBExp` is indexed by a context, denoted `ctx`, and an assertion, denoted `lwfb`. Its constructors can be seen as an extension of `LimWFBExp`. Since inhabitance represents truth, negation (`Neg lwfb`, Lines 30–31) requires a proof that `VerifLimWFBExp ctx lwfb` is uninhabited for its subexpression, `lwfb`. Similarly, the conjunction (`And lwfb1 lwfb2`, Lines 32–34) constructor requires proofs that the predicate `VerifLimWFBExp ctx` holds for both subexpressions, `lwfb1` and `lwfb2`. Disjunction (`Or lwfb1 lwfb2`, Lines 35–38) is similar to conjunction, but requires a proof that the predicate `VerifLimWFBExp ctx` holds true for at least one of its subexpressions; accordingly, disjunction is represented by two constructors: `OrL` and `OrR`. Both assertions of equality (`Eq lwfa1 lwfa2`, Lines 22–25) and of inequality (`LTE lwfa1 lfwa2`, Lines 26–29) require proofs that their subexpressions are valid, via inhabitance of `VerifLimWFAExp ctx`, and proofs that their respective properties hold; i.e. the values represented by `lwfa1` and `lwfa2`, denoted `v1` and `v2`, are propositionally equal for equality and `v1` $\leq$ `v2` for inequality.

Listing 7: Type of Assertions that Hold True.

```
1  data VerifLimWFAExp  : (ctx : Vect nws (Assignment awfs))
2                       -> (lwfa : LimWFAExp ws)
3                       -> (v : Nat)
4                       -> Type where
5    Val  : VerifLimWFAExp ctx (Val v) v
6    Var  : (ctxPtr : Elem (Asn ptr v) ctx)
7        -> VerifLimWFAExp ctx (Var ptr) v
8    Add  : (vlwfa1 : VerifLimWFAExp ctx lwfa1 v1)
9        -> (vlwfa2 : VerifLimWFAExp ctx lwfa2 v2)
10       -> VerifLimWFAExp ctx (Add lwfa1 lwfa2) (v1 + v2)
11   Mul  : (vlwfa1 : VerifLimWFAExp ctx lwfa1 v1)
12       -> (vlwfa2 : VerifLimWFAExp ctx lwfa2 v2)
13       -> VerifLimWFAExp ctx (Mul lwfa1 lwfa2) (v1 * v2)
14   Sub  : (vlwfa1 : VerifLimWFAExp ctx lwfa1 v1)
15       -> (vlwfa2 : VerifLimWFAExp ctx lwfa2 v2)
16       -> (smaller : LTE v2 v1)
17       -> VerifLimWFAExp ctx (Sub lwfa1 lwfa2) (v1 - v2)
18
19 data VerifLimWFBExp  : (ctx : Vect nws (Assignment awfs))
20                      -> (lwfb : LimWFBExp ws)
21                      -> Type where
22   Eq   : (vlwfa1 : VerifLimWFAExp ctx lwfa1 v1)
23       -> (vlwfa2 : VerifLimWFAExp ctx lwfa2 v2)
24       -> (prop : v1 = v2)
25       -> VerifLimWFBExp ctx (Eq lwfa1 lwfa2)
26   LTE  : (vlwfa1 : VerifLimWFAExp ctx lwfa1 v1)
27       -> (vlwfa2 : VerifLimWFAExp ctx lwfa2 v2)
28       -> (prop : LTE v1 v2)
29       -> VerifLimWFBExp ctx (LTE lwfa1 lwfa2)
30   Neg  : (vlwfb : Not (VerifLimWFBExp ctx lwfb))
31       -> VerifLimWFBExp ctx (Neg lwfb)
32   And  : (vlwfb1 : VerifLimWFBExp ctx lwfb1)
33       -> (vlwfb2 : VerifLimWFBExp ctx lwfb2)
34       -> VerifLimWFBExp ctx (And lwfb1 lwfb2)
35   OrL  : (vlwfb1 : VerifLimWFBExp ctx lwfb1)
36       -> VerifLimWFBExp ctx (Or lwfb1 lwfb2)
37   OrR  : (vlwfb2 : VerifLimWFBExp ctx lwfb2)
38       -> VerifLimWFBExp ctx (Or lwfb1 lwfb2)
```

The type `VerifLimWFAExp` defines valid algebraic expressions under some context; it is indexed by a context, `ctx`, an algebraic assertion expression, `lwfa`, and the value `lwfa` represents, `v`. A literal value (`Val v, Line~5`) is inherently valid; a numeric variable (`Var ptr`, Lines 6–8) is considered valid when there is a mapping in the context from `ptr` to a value; and addition (`Add lwfa1 lwfa2`, Lines 8–10) and multiplication (`Mul lwfa1 lwfa2`, Lines 11–13) are considered valid when both `lwfa1` and `lwfa2` are valid. Since ground values are natural numbers, subtraction (`Sub lwfa1 lwfa2`, Lines 14–17) has the additional constraint that the value represented by `lwfa2`, `v2`, must be less than that of `lwfa1`, `v1`; i.e. `v2` $\leq$ `v1`. An alternative approach might define `v1 - v2 = 0` when `v2` $>$ `v1`. A further alternative would be to represent ground values as integers, or generalise the ground values as an argument, possibly constrained by an interface.

Example 6.2. In the final paper, we will demonstrate both assertions that hold true and assertions that do not hold true using the Levenshtein Distance abstract interpretation from Example 6.1.

7 RELATED WORK

The calculation and bounding of resource usage is a topic of great interest in the programming language community, with approaches typically focussed on time, space, and type systems [9, 10, 15, 17, 18, 21, 33]. Other non-functional properties, such as information flow and leakage, have also been modelled using type systems [8, 34]. Energy consumption, which is of increasing interest to embedded systems programming [25], has also been represented as a function of program arguments [13, 23]. These approaches typically depend upon specific type systems or languages; accordingly, applying them to other languages can prove non-trivial [7]. Abstract interpretation offers an alternative approach to the verification and debugging of programs in languages that may not necessarily be best equipped for the desired techniques [24]. Examples include debugging of both imperative and logical programs [3, 5], and approaches to verification by Cousot [11]. More recently, The Ciao Preprocessor system (CiaoPP) [16, 24, 30] models Java [14] and XC [1] programs as sequences of Horn Clauses in order to debug and certify programs, using resource usage information that the system derives. A high-level comparison between CiaoPP and *Drive* is given by Brown *et al.* [7].

8 CONCLUSIONS AND FUTURE WORK

In the final paper, we will present an investigation of the soundness of the *Drive* framework for reasoning about non-functional properties of source code in C programs. We have defined a subset of C, $C_{\subset}$, around which we base our annotation definitions and novel abstract interpretation. We will extend these definitions in the final paper with covering functions and decision procedures, which will be used to determine, e.g., well-formedness of programs in $C_{\subset}$, and infer both abstract interpretations and contexts. Our definitions are, and will be, given as dependent type definitions in Idris, allowing soundness of covering functions and decision procedures to be ensured via type- and totality-checking. Our full implementation will be made available. We will define an operational semantics for $C_{\subset}$, and prove the soundness of our context inference with respect to these operational semantics. We will apply our Levenshtein Distance example at each stage in order to illustrate our approach on a concrete example. We intend to investigate how our novel abstract interpretation might be extended with rewriting rules that not only ensure and facilitate the automatic derivation of proofs of programmer-defined assertions, but can solve non-ground assertions in a type-safe manner.

REFERENCES

[1] 2011. XC Specification ver. 1.0 (X5965A). https://www.xmos.com/developer/xc-specification.

[2] Elvira Albert, Puri Arenas, Germán Puebla, and Manuel V. Hermenegildo. 2012. Certificate size reduction in abstraction-carrying code. *TPLP* 12, 3 (2012), 283–318.

[3] Giovanni Bacci and Marco Comini. 2010. Abstract Diagnosis of First Order Functional Logic Programs. In *LOPSTR (Lecture Notes in Computer Science)*, Vol. 6564. Springer, 215–233.

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *SEUS (Lecture Notes in Computer Science)*, Vol. 6399. Springer, 35–46.

[5] François Bourdoncle. 1993. Abstract Debugging of Higher-Order Imperative Languages. In *PLDI*. ACM, 46–55.

[6] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593.

[7] Christopher Brown, Adam D. Barwell, Yoann Marquer, Céline Minh, and Olivier Zendra. 2019. Type-Driven Verification of Non-functional Properties. In *PPDP*. Accepted for publication.

[8] Hongxu Chen, Alwen Tiu, Zhiwu Xu, and Yang Liu. 2018. A Permission-Dependent Type System for Secure Information Flow Analysis. In *CSF*. IEEE Computer Society, 218–232.

[9] Wei-Ngan Chin and Siau-Cheng Khoo. 2001. Calculating Sized Types. *Higher-Order and Symbolic Computation* 14, 2-3 (2001), 261–300.

[10] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9032. Springer, 406–431.

[11] Patrick Cousot. 2003. Automatic Verification by Abstract Interpretation. In *VMCAI (Lecture Notes in Computer Science)*, Vol. 2575. Springer, 20–24.

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[13] Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. 2017. Energy Transparency for Deeply Embedded Programs. *TACO* 14, 1 (2017), 8:1–8:26.

[14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[15] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*. ACM, 127–139.

[16] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.* 58, 1-2 (2005), 115–140.

[17] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62.

[18] John Hughes and Lars Pareto. 1999. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *ICFP*. ACM, 70–81.

[19] Graham Hutton. 2007. *Programming in Haskell.* Cambridge University Press, New York, NY, USA.

[20] Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. 2019. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In *WCET (OASICS)*, Vol. 72. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1:1–1:11.

[21] Ugo Dal Lago and Barbara Petit. 2013. The geometry of types. In *POPL*. ACM, 167–178.

[22] Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.* 69, 1-3 (2007), 56–67.

[23] Umer Liqat, Steve Kerrison, Alejandro Serrano, Kyriakos Georgiou, Pedro López-García, Neville Grech, Manuel V. Hermenegildo, and Kerstin Eder. 2013. Energy Consumption Analysis of Programs Based on XMOS ISA-Level Models. In *LOPSTR (Lecture Notes in Computer Science)*, Vol. 8901. Springer, 72–90.

[24] Pedro López-García, Luthfi Darmawan, Maximiliano Klemen, Umer Liqat, Francisco Bueno, and Manuel V. Hermenegildo. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *TPLP* 18, 2 (2018), 167–223.

[25] Jeremy Morse, Steven Kerrison, and Kerstin Eder. 2018. On the limitations of analysing worst-case dynamic energy of processing. *ACM Transactions on Embedded Computing Systems* 17, 3 (2 2018). https://doi.org/10.1145/3173042

[26] George C. Necula. 1997. Proof-Carrying Code. In *POPL*. ACM Press, 106–119.

[27] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer.* Springer.

[28] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2013. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *CoRR* abs/1308.5174 (2013). arXiv:1308.5174 http://arxiv.org/abs/1308.5174

[29] Christopher Schwaab, Ekaterina Komendantskaya, Alasdair Hill, Frantisek Farka, Ronald P. A. Petrick, Joe B. Wells, and Kevin Hammond. 2019. Proof-Carrying Plans. In *PADL (Lecture Notes in Computer Science)*, Vol. 11372. Springer, 204–220.

[30] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. 2014. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP* 14, 4-5 (2014), 739–754.

[31] Franck Slama and Edwin Brady. 2017. Automatically Proving Equivalence by Type-Safe Reflection. In *CICM (Lecture Notes in Computer Science)*, Vol. 10383. Springer, 40–55.

[32] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

[33] Pedro B. Vasconcelos and Kevin Hammond. 2003. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL (Lecture Notes in Computer Science)*, Vol. 3145. Springer, 86–101.

[34] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188.

Towards Time-, Energy- and Security-aware Functional Coordination

Julius Roeder
University of Amsterdam
Amsterdam, Netherlands
J.Roeder@uva.nl

Benjamin Rouxel
University of Amsterdam
Amsterdam, Netherlands
Benjamin.Rouxel@uva.nl

Clemens Grelck
University of Amsterdam
Amsterdam, Netherlands
C.Grelck@uva.nl

ABSTRACT

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches. An application organised according to the coordination paradigm consists of a collection of interacting components. Yet, we are neither aware of any adoption of the principle in the broader domain of low-powered safety-critical embedded systems, nor are we aware of time, energy and security aware approaches to coordination.

We aim at contributing to the community by bringing a coordination approach to real-time applications with the establishment of a Domain Specific Language (DSL). Our coordination workflow considers time and other non-functional properties, such as energy and security, as first-class citizens in application designs. We therefore aim at building a complete toolchain and workflow to compile a coordinate application to a final executable.

ACM Reference Format:
Julius Roeder, Benjamin Rouxel, and Clemens Grelck. 2020. Towards Time-, Energy- and Security-aware Functional Coordination. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Designing secure and safe Cyber Physical Systems (CPS) requires a tremendous amount of validations and certifications. To facilitate the whole process, a good design practice is to separate code source development from application structure design. This separation allows to start the validation at an early stage of the system design.

Coordination programming paradigm seems to us a promise solution as it allows to separate the high-level view of the structure from the low-level application units. An application organised according to the coordination paradigm consists of a collection of interacting components. Components, also known as actors or tasks, represent application features, sequential building blocks of application, implemented in a general-purpose programming language [11]. Given the focus of the safety-critical embedded systems domain, we exclusively work with the system-level programming language *C*. Hence, a component is technically a callable *C* function with certain restrictions on its functional behaviour, together with a set of non-functional properties, i.e. timing, energy and security.

The coordination language emphasizes communication, concurrency and synchronisation (referred to by the term *coordination*). It aims at describing component interactions in terms of precedences and data exchange. In contradiction with streaming languages (e.g. [23]) or data-flow languages (e.g. Lustre [1]), a coordination language is independent from the actual code, but it dictates, to the scheduler, how this code should be executed.

We aim at contributing to the community by bringing a coordination approach to real-time applications with the establishment of a Domain Specific Language (DSL). Our coordination workflow considers time and other non-functional properties, such as energy and security, as first-class citizens in application designs. We therefore aim at building a coordination language along with a complete toolchain to compile a coordinate application to a final executable.

Security with Cyber Physical System (CPS) relies on the implementation of proven protocols, encryption algorithms or scheduling obfuscation techniques. Such methods harden the system but also increases its requirement regarding computation and memory. Nevertheless some systems do not require to operate at a maximum security level all the time. For example, a military drone used for area recognition could adapt its security protocol in accordance with the mission state such as:

- low security level: taking off or landing from/to the base station,
- medium security level : navigating to/from the mission area,
- high security level : on site recognition.

When performing in low security level state, a drone can use a less resilient encryption system to send data to the base station, while it can use the more secure one when on mission site. This adaptation can result in less computation and therefore in power energy saving and longer effective mission time, while still guaranteeing the security and integrity of the data at all time.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Our solution is to embed different versions of the same task but with different security, time, and energy properties similarly to [19]. The scheduler will then be responsible to chose proper task versions according to current mission state.

We target CPS platforms executing on COTS multi-core heterogeneous processors where time, security and energy are safety and mission-critical. Our approach is not yet to improve timing analyses or, at first, scheduling policies themselves, but to make those secure and efficient by controlling the whole design path from specification to code generation. Hence, our first goal is to describe how to exploit our coordination language and how to integrate its semantic into state-of-the-art scheduling techniques, e.g. [18, 21]. And second, we will devise new scheduling policies that will execute smart scheduling decisions to achieve our goals regarding time, security and energy.

2 LITERATURE REVIEW

2.1 Coordination Language

Coordination is a well established computing paradigm with a plethora of languages, abstractions and approaches surveyed in [5]. Yet, we are neither aware of any adoption of the principle in the broader domain of critical embedded systems, nor are we aware of time&security-aware approaches to coordination. An example is the coordination language S-Net [11], from which we draw inspiration and experience for our Teamplay Coordination Language. However, like other coordination approaches S-Net merely addresses the functional aspects of coordination programming and has left out any non-functional requirements, not to mention time and security, in particular.

Notable exceptions in the otherwise pretty much uncharted territory of time&energy-aware coordination is Hume [13]. It was specifically designed with real-time systems in mind. Thus, guarantees on time (and space) consumption are key to Hume. However, the main motivation behind Hume was to explore how far high-level functional programming features, such as automatic memory management, higher-order functions, polymorphism, recursion, ... can be supported while still providing accurate real-time guarantees.

Another term referring to Coordination technology is "in-the-large" programming. It aims at describing a large view of an application, or its structure, while "in-the-low" programming considers only the implementation of each feature. Bondavalli et al. [4] presents a simple "in-the-large" programming language to describe the structure of a graph-based application. However they only model what we call component and simple edge, leaving time and security property aside. Their simple language also do not account for multi-version component nor complex edge structure as we do.

2.2 Data-flow programming language

Lustre [3, 12] was designed to program reactive system, such as automatic control and monitoring systems. Compared to general purpose programming language, it models the flow of data designates a low level programming language. The idea is to represent actions done on data at each time tick, link an electric circuit. The tick can be extended to represent periods and release times for tasks, but still an action is required to describe outputs for each tick (like reusing the last produced data). Compared to most dataflow programming language, Lustre is synchronous which seems necessary for time-sensitive applications. However, Lustre does not decorrelate the program source code from its structure as we do. The flow of data is extracted by the compiler through data dependencies of variables. We believe that separating the design of an application and its program source code is a good practice to facilitate analyses, certifications and increase reusability. We aim at expressing the flow of data with a much simpler and more explicit approach. We also act at a higher level by focusing on the interaction of components considered as black boxes. Hence, our approach is orthogonal to Lustre which can be used to program the inner side of each component.

In [1], Lustre is extended with meta-operators to integrate, as an intermediate representation (called Lustre++), a complete Model-Based Design tools from a high-level Simulink model to a low-level implementation. Still, this extension does not separate the design of the program structure and its actual code source. And Lustre++ remains at a too low level to only represent application structure as we intend to do.

In general, dataflow programming language include both application structure and code. Some are asynchronous [] and therefore unusable for time-sensitive applications, while synchronous one (Lustre, Esterel) [] aim at bringing timing constraint into programming paradigm.

2.3 Data-flow programming and graph description

On the other hand, we were also not able to find a graph description language from the broad world of dataflow paradigm, that allows us to fulfil our needs in term of time and security. A promising approach was the Dataflow Interchange Format (DIF) [14] which captures essential structure information while hiding implementation details of computation blocks. But timing and security are absent of its definition. StreamIT [23] language also describes graph-based streaming applications but it is restricted to fork-join graphs while we need to support arbitrary graphs with possible multiple sources/sinks.

From all existing coordination languages or dataflow description language, TCL gives the possibility :(1) to describe multiple versions of the same component to chose the right level of security according to the mission state, (2) to define stateful and stateless actors, such as in StreamIT [23], (3) to select the dataflow path depending on environmental or scheduling decisions.

2.4 Scheduling

2.4.1 Multi-version scheduling. Lastly, a multi-version scheduling technique is described in [17]. They employ multiple

version of tasks where each version different in term of energy usage. However they do not detail how to model such application. In addition, we target security problem, and we envision, in a near future, to join our techniques to provide full energy/time/security-aware scheduling policies.

2.4.2 Multi-mode scheduling.

2.5 Model-based paradigm

Model-Based Design (MBD) decorrelates source code programming and application structure with a high-level model design. One of the most known model-based programming language is the Unified Modelling Language (UML) [1] and a massive amount of UML profiles exists on embedded systems, with or without real-time capabilities.

SysML is one of the profile that target embedded systems, but it lakes from real-time capabilities.

Simulink does not have a denotational semantic, thus it is impossible to prove its faithfulness/correctness, (faithful between the generated code and the simulink model).

RW of modelling languages in section 2.3 of https://tel.archives-ouvertes.fr/tel-01773786/document (french thesis)

2.5.1 AUTOSAR. The objective of AUTomotive Open System ARchitecture (AUTOSAR) [10] is to establish an open industry standard for the automotive software architecture between suppliers and manufacturers. The standard comprises a set of specifications describing software architecture components and defining their interfaces. Software components are linked to a common interface that is recognized by an AUTOSAR runtime environment. AUTOSAR model is based on UML. The task model behind AUTOSAR includes periodic, sporadic, burst, concrete and arbitrary arrival patterns. With its event chains, AUTOSAR seems able to provide dependent tasks.

In [2], they studied the system model and what kind of information is available and so what kind of scheduling is possible with it. They identified the four groups of information required to perform a schedulability test. Then on a case study, they map what is provided by AUTOSAR to these groups. They finally apply the schedulability test for distributed systems with fixed priorities among dependent periodic tasks.

Even with its event chain, AUTOSAR fails to represent dataflow application.

2.5.2 AADL. Architecture Analysis & Design Language (AADL) [8] targets real-time system design. It provides formal modeling concepts for the description and analysis of application systems architecture in terms of distinct components and their interactions. It supports early predictions and analyses including performance, schedulability and reliability.

All common features to represent real-time systems task models are presents including (non-exhaustive): tasks, dependencies, period, deadline, etc .... But, there is not possibility to represent SDFs, CSDFs, or other multi-rates data transfers between components.

2.5.3 MoSaRT.

2.5.4 MARTE. MARTE [20] has become the standard for embedded real-time system engineering. It provides a common way to describe hardware and software aspects and how components interact with each other. However there is no mean to express different versions of a component and leave the scheduler decides which one to use.

2.5.5 Time4Sys. Time4Sys seems abandoned in 2018.

2.6 Unclassified yet

[6] aims at increasing parallelism while guaranteeing safety, their *HEPTAGON* language also includes source code of the application which require an additional step to get the binary. Their execution model separate communication from computation, they implement communication channels on a FPGA while computation occur on ARM cores. Mixed-criticality

3 COORDINATION WORKFLOW

An application organised according to the coordination paradigm consists of a collection of interacting components. Components are meaningful, sequential application building blocks, implemented in a general-purpose programming language [11]. Given the focus of critical embedded systems domain, we exclusively work with the system-level programming language *C*. Hence, a component is technically a callable *C* function with certain restrictions on its functional behaviour, together with a set of non-functional properties i.e. timing, energy and security.

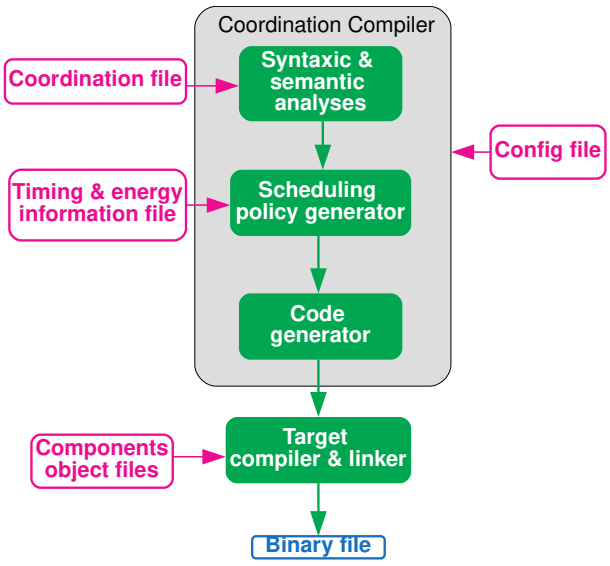


Figure 1: Coordination workflow

[1] https://www.uml.org

Figure 1 summarizes the workflow to build a coordinated application. The three main inputs of the workflow are:

(1) a *coordination file*: provided by the end-user to describe the application structure as well as non-functional properties, see details in Section 4,
(2) *timing and energy information*: provided by timing/energy harvesting tools such as AbsInt aiT [9],
(3) *object files*: provided by a C-compiler such as WCC [7], they contain compiled C-code for each component.

The fourth input file includes practical configurations i.e. target hardware description and security-level mission specifications, compiler passes to apply, ...

First the parser evaluates the *coordination file* and provides a graph representation of the application. This representation is then analysed and transformed according to the enabled compiler passes, e.g. deadlock-free check, security requirements. From the graph and timing & energy information, the schedule policy generator computes different schedules according to hardware specification and security levels configuration when an *off-line* schedule is asked, or performs schedulability analyses in case of *on-line* requirements. Generated schedules are passed to the code generation, e.g. dispatching schedule tables if off-line schedule applied. Finally, schedules and compiled object-files of the whole application are compiled and linked with the target toolchain to generate the final binary executable which can then be downloaded to the platform.

4 TEAMPLAY COORDINATION LANGUAGE

Our coordination language focuses on the design of arbitrary synchronous data-flow-oriented applications. It describes the graph structure by modelling the dependencies between components. Such applications are represented by a graph where vertices are component (a.k.a actors, tasks), while edges correspond to dependencies between components. A dependency defines a data exchange between a source and a sink through a FIFO channel, such data is often called token. A token can have different types according to the application, from primitive types to more evolve structures.

Similarly to periodic task models [16], a data-flow graph instance is called an iteration and a job is a task instance inside an iteration. Then, the DAG may iteratively executes until the end of time (or platform is shutdown). Hence, jobs execution order follows aforementioned constraint, job i finished before job i+1. But, the iteration j +1 can start before the completion of iteration j as long as jobs dependencies are satisfied. This allows to exploit job parallelism, i.e. pipelining [22].

Figure 2 presents the grammar written in pseudo-Xtext language which is given to ANTLR[2] in order to automatically create a lexer and a parser. Following is the description of each rule.

Rule *Application*, lines 1-7, describes the root element of our application. It is composed by an identifier, a deadline

```
1  Application: 'app' ID '{'
2          'deadline' TIME
3          'period' TIME
4          'datatypes' '{' (Datatype)* '}'
5          'components' '{' Component+ '}'
6          'edges' '{' Edge* '}'
7  '}';
8  Datatype: ID ':' STRING ':' INT ;
9  Component: ID '{'
10         ('inputs' ':' '[' (Connector)* ']')?
11         ('outputs' ':' '[' (Connector)* ']')?
12         Version*
13 '}';
14 Connector: '(' ID ',' INT ',' ID ')';
15 Version: 'version' ID '{'
16         ('deadline' INT )?
17         ('period' INT)?
18         ('targetArch' STRING)*
19         ('security' '[' INT ',' INT ']')?
20 '}';
21 Edge: (SingleEdge | DuplicateEdge | DataOrEdge |
       SchedOrEdge | EnvOrEdge);
22 SingleEdge:
23     (INT)? CompRef '->' CompRef;
24 DuplicateEdge:
25     (INT)? CompRef '->' CompRef  ( '&' CompRef )
           +;
26 DataOrEdge:
27     (INT)? 'DATA' Component ('/' Version)? ':' '
           (' '.' Connector '->'
28         ComponentRef ('|'  '.' Connector '->'
               ComponentRef)+ ')';
29 SchedOrEdge:
30     (INT)? 'SCHED' ComponentRef '->'
           ComponentRef ('|' ComponentRef)+;
31 CompRef: ID ('/' ID)? '.' ID ;
```

Figure 2: Pseudo-Xtext grammar for our coordination language

and a period. These timing information refer to an iteration of the graph.

Rules *Datatypes* (line 8), *Components* (lines 9-13) and *Edges* (lines 21-**??**) are children of an application. They respectively correspond to tasks and dependencies between tasks, more details are following in Sections 4.1 and 4.2 and contains the description constraints as stated in Section 4.

Rule *Datatype*, line 8, refers to the type of data exchanged by components. First is an identifier used to refer to this particular datatype inside de coordination file. Followed by a string that represents its implementation in user code, and a size in bits.

Rule *ComponentRef*, line 31, refers to the identifier of an already defined *Component*. A version defined for this component can be added, for fulfil some application constraint where a version of a component can be linked to only an other version of a component. The last *ID* of rule *ComponentRef* states the connector used to connect the component.

Rule *Condition*, line **??**, defines the selection type for selector edge. We identified two types of conditions:

[2] https://www.antlr.org

- environmental : depending on the battery status, the GPS location, or a user command reception the flow of data can vary ;
- scheduling : the scheduler decides which of the alternatives is the best according to some criteria, such as the security requirements.

Impacts on scheduler decisions depend on the scheduler nature. For off-line schedulers, if the condition is environmental, then both alternatives must be scheduled and the on-line dispatcher will block one. This drawback is a limitation of off-line schedulers, but not of on-line ones.

Terminals *ID*, *INT*, *STRING* and *TIME* are not described for space consideration. But they respectively catch an identifier (start with a letter followed with any alpha-numeric characters and some special characters, e.g. '_'), an integer, a string, a time duration in e.g. hours, milliseconds or cycles.

4.1 Component description

The TCL grammar, Figure 2 lines 9-13, defines a component (as known as task or actor) by two set of connectors, for respectively input and output edges, and a set of versions with a minimum of 1 version.

Connectors represent the interface of the components. What do then need to run with the input connectors, and will they produce with output connectors. Each connector includes a name, an amount of tokens, and a data type identifier. They are useful to perform a type checking analyses and guaranty that tokens produced along an edge are of the same type (or compatible one according to casting capabilities of the parser) as tokens consumed on that same edge.

If no input connectors (resp. output connectors) are given, then the component is a source (resp. sink) component.

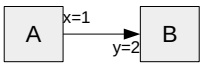
Each component is characterized by multiple versions in order to respond to different security requirements. To encompass the need for security of low powered real-time embedded system, we propose to have different component versions. This allows to switch from a low security level to a higher security level depending on current environment requirements. Hence, our coordination language offers to describe these versions. A component version, Figure 2 lines 15-20, is described with two security levels that defines the minimum and the maximum security level at which they are allowed to execute. The scheduler must therefore account for this information to properly choose the right version to activate.

It must be noted that when no version is given by the user, a default version is created by the compiler with widest range for the security value.

In addition to security levels, a component version can includes usual real-time properties (e.g. deadline, period) in order to enable periodical task model [16] representation with TCL. Our coordination language also allows mapping techniques targeting heterogeneous platform with a *targetArch* attribute placed on the component. It is worth noting that for homogeneous platforms, all component versions can have the same unique target architecture, or no *targetArch* attribute at all.

4.2 Dependencies representation

With dataflow-oriented application, tasks exchange data known as tokens. The token type depends on the application, e.g. a surveillance camera system would include tokens of type *image* while wireless devices would use integers. Dependencies therefore represent the flow of tokens in the graph. Following are the constructions we allow with our representation. Each construction are presented with both a graphical sketch, and a textual representation with respect to the grammar defined in Section **??**. For the graphical representation, components are grey boxes while dependencies are arrows.



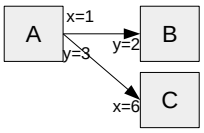
(a) Graphical sketch

```
A.x -> B.y
```

(b) Textual declaration

Figure 3: Simple edge

Figure 3 presents a simple edge between a source component A producing 1 token and a sink component B consuming 2 tokens, defined by Figure 2 line 23.



(a) Graphical sketch

```
A.x -> B.y
A.y -> C.x
```

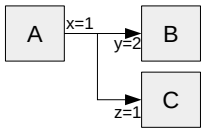
(b) Textual declaration

Figure 4: Multiple sinks edge

Figure 4 is an extrapolation of the previous dependency construction where source A produces 4 tokens and the first one goes to component B while the 3 others go to component C. Identically to [15] we impose the *top-down* ordering to determine the order of produced/consumed tokens. This allows to use scheduling policy from [18] where the order of produced/consumed tokens might have an impact on the scheduler decision. Our coordination language also allows to have more than one source for a component which is enabled by trivially deriving the two previous constructions.

Figure 5 presents a duplicating edge between a source component A producing 1 token and two sink components B and C consuming respectively 2 tokens and 1 token, as defined by Figure 2 line 25. The *duplication* dependency imposes to duplicate the token produced by component A on parts of the edge going to the two sinks. Hence, components B and C will work on the same copy of the same token.

Figure 6 allows conditional dependencies, as defined by Figure 2 line 28. In this case, component B and component C are dependent of component A but only one is allowed to actually execute depending to an environmental or scheduling



(a) Graphical sketch

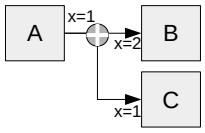
```
A.x -> B.y & C.z
```

(b) Textual declaration

```
A.x -> B.y
A.x -> C.z
```

(c) Other textual declaration

Figure 5: Tokens duplication edge



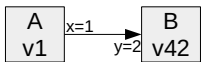
(a) Graphical sketch

```
A.x -> ENVIRONMENT
       ( B.x | C.x )
```

(b) Textual declaration

Figure 6: Path selection edge

decision. From Figure 2 line **??** is a dual rule that places a condition between multiple sources going to only one sink.



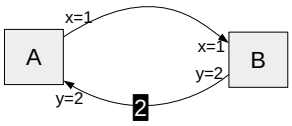
(a) Graphical sketch

```
A/v1.x -> B/v42.y
```

(b) Textual declaration

Figure 7: Simple edge with specific versions

Finally, our coordination language allows to constrain dependencies with component versions as in Figure 7 where a simple edge links the version x of component A and the version y of component B. This allows, for example, to guarantee that for component versions with overlapping security level, the two versions with the highest minimum security level are defined as dependent.



(a) Graphical sketch

```
A.x -> B.x
B.y -> A.y
```

(b) Textual declaration

Figure 8: Back edge

4.3 Towards scheduling

Prior to scheduling a graph unfolding is required to compute the repetition vector, i.e. number of times each actor needs to be fired, in order to consume all produced tokens as well as to produce as much as consumed tokens. A scheduler then generates a schedule of the graph which will be repeatedly executed on the board. Hence, the scheduler acts on one iteration of the graph [23].

5 CONCLUSION

In this paper we presented Teamplay Coordination Language, a simple DSL to coordinate a critical streaming applications while enforcing safety and security. We presented a grammar to describe the language and applied it to implement a Drone use-case. It is also worth mentioning that we implemented our grammar using Xtext-Eclipse[3] to provide a graphical editor. Hence, Xtext uses ANTLR inwards which allows us to generate a lexer/parser for any scheduler software a end-user might chose.

As future work, we intend to fill all boxes from the coordination workflow including an off-line and an on-line scheduler. Our future scheduling algorithms will guarantee an efficient, safe, secure and energy optimized application that we intend to effectively run on a drone after a code generation phase.

ACKNOWLEDGEMENT

This work is funded by the European Union Horizon2020 research and innovation programme under grant agreement No. 779882 (TeamPlay).

REFERENCES

[1] Mouaiad Alras, Paul Caspi, Alain Girault, and Pascal Raymond. 2009. Model-based design of embedded control systems by means of a synchronous intermediate model. In *2009 International Conference on Embedded Software and Systems*. IEEE, 3–10.

[2] Saoussen Anssi, Sara Tucci-Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. 2011. Enabling scheduling analysis for AUTOSAR systems. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 152–159.

[3] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[4] Andrea Bondavalli, Lorenzo Strigini, and Luca Simoncini. 1992. Dataflow-like languages for real-time systems: issues of computational models and notation. In *[1992] Proceedings 11th Symposium on Reliable Distributed Systems*. IEEE, 214–221.

[5] Giovanni Ciatto, Stefano Mariani, Maxime Louvel, Andrea Omicini, and Franco Zambonelli. 2018. Twenty years of coordination technologies: State-of-the-art and perspectives. In *International Conference on Coordination Languages and Models*. Springer, 51–80.

[6] Albert Cohen, Valentin Perrelle, Dumitru Potop-Butucaru, Marc Pouzet, Elie Soubiran, and Zhen Zhang. 2016. Hard Real Time and Mixed Time Criticality on Off-The-Shelf Embedded Multi-Cores. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

[7] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. 2006. Design of a wcet-aware c compiler. In *2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. IEEE, 121–126.

[3] https://www.eclipse.org/Xtext/

[8] Peter H Feiler, David P Gluch, and John J Hudak. 2006. *The architecture analysis & design language (AADL): An introduction*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

[9] Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer, 377–383.

[10] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. 2009. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Vol. 62. 5.

[11] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. 2010. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming* 38, 1 (2010), 38–67.

[12] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[13] Kevin Hammond and Greg Michaelson. 2003. Hume: a domain-specific language for real-time embedded systems. In *International Conference on Generative Programming and Component Engineering*. Springer, 37–56.

[14] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S Bhattacharyya. 2004. DIF: An interchange format for dataflow-based design tools. In *International Workshop on Embedded Computer Systems*. Springer, 423–432.

[15] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers* 100, 1 (1987), 24–35.

[16] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[17] Riad Nassiffe, Eduardo Camponogara, and George Lima. 2011. A Model for Reconfiguration of Multi-Modal Real-Time Systems under Energy Constraints. In *2011 Brazilian Symposium on Computing System Engineering*. IEEE, 127–132.

[18] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. 2019. Hiding communication delays in contention-free execution for SPM-based multi-core architectures. In *31th Euromicro Conference on Real-Time Systems (ECRTS19)*.

[19] Cosmin Rusu, Rami Melhem, and Daniel Moss'e. 2005. Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing* 1, 2 (2005), 271–283.

[20] Bran Selic and Sébastien Gérard. 2013. *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. Elsevier.

[21] Abhishek Singh, Pontus Ekberg, and Sanjoy Baruah. 2017. Applying real-time scheduling theory to the synchronous data flow model of computation. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[22] Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. 2014. Many-core scheduling of data parallel applications using SMT solvers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 615–622.

[23] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 179–196.

Language-Integrated Updatable Views

Rudi Horn
University of Edinburgh
United Kingdom
r.horn@ed.ac.uk

Simon Fowler
University of Edinburgh
United Kingdom
simon.fowler@ed.ac.uk

James Cheney
University of Edinburgh
United Kingdom
jcheney@inf.ed.ac.uk

ABSTRACT

Relational lenses are a modern approach to the *view update* problem in relational databases, inspired by research in *bidirectional transformations*. As introduced by Bohannon et al. [5], Relational Lenses allow the definition of updatable views by the composition of lenses performing individual transformations. Horn et al. [20] provided the first implementation of *incremental relational lenses*, which demonstrated that relational lenses can be implemented efficiently by propagating *changes* to the database rather than the replacing entire database state.

In this paper, we propose the first full integration of relational lenses in a functional programming language, by extending the Links web programming language. Our implementation supports fully dynamic predicates, generalising the previous work by Horn et al. [20]. We show how a subset of the Links intermediate representation can be compiled to a lens IR suitable for compilation to SQL, and provide the first implementation of typechecking of relational lenses, ruling out lenses which do not satisfy well-formedness conditions. We demonstrate the applicability of our approach through the case study of a curation interface for a scientific database application.

ACM Reference Format:
Rudi Horn, Simon Fowler, and James Cheney. 2019. Language-Integrated Updatable Views. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Relational databases are considered the *de facto* standard for storing data persistently. A typical use case where relational databases are especially prominent are web applications. Relational databases offer a ready to use method for storing and retrieving data efficiently for a broad range of contexts, without having to directly implement the intricate details required for high performance.

Running example: a music database. In the remainder of the paper, we consider an example music database, originally proposed by Bohannon et al. [5], shown in Figure 1. There are two tables: the `albums` table, which details the quantities of albums available, and the `tracks` table, which details the track name, date of release, rating, and the album on which the track is contained.

`albums`

album	quantity
Disintegration	6
Show	3
Galore	1
Paris	4
Wish	5

`tracks`

track	date	rating	album
Lullaby	1989	3	Galore
Lullaby	1989	3	Show
Lovesong	1989	5	Galore
Lovesong	1989	5	Paris
Trust	1992	4	Wish

Figure 1: Music Database

Language-Integrated Query. Programs interface with relational databases using the *Structured Query Language* (SQL). To query the database, the host application needs to take the user input (e.g., a track name), generate an SQL query, issue it to the database server, and then process the result in a way that aligns with the result of the query.

Our application could generate the query by using string concatenation and then assume the result will be in a known format containing records of track names of type **string** and years of type **int**. However, such an approach leaves many possible sources of error, most of which are related to a lack of cross-checking of the different stages of execution. The application could have bugs in query generation, which could result in incorrect queries or even security flaws. Furthermore, a generated query may not produce a result of the type that the application expects, resulting in a runtime error. The user experience of the programmer is also poor, as tooling is able to provide little help and the programmer must write code in two different languages, while being mindful not to introduce any bugs in the application. We refer to this as an *impedance mismatch* between the host programming language and SQL [11].

Existing work on *language integrated query* (LINQ) allows queries to be expressed in the host language [10, 33]. Rather than generating an SQL query using string manipulations, the query is written in the same syntax as the host programming language. The user need not worry about how the query is generated, and the code that

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

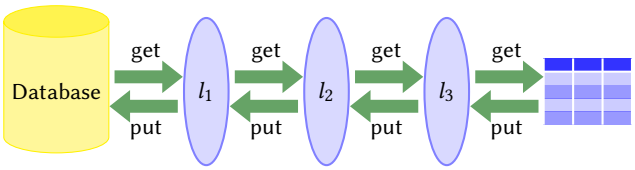


Figure 2: Relational Lenses

performs the database query is automatically type-checked at compile time. Consequently, LINQ provides a type-safe and easy-to-use method for expressive queries.

As an example, consider the following function, which queries the `albums` table and returns all albums with a given album name.

```
fun getAlbumsByName(albumName) {
  for (a <-- albums)
    where (a.album == albumName)
    [a]
}
```

The corresponding SQL for `getAlbumsByName("Galore")` would be:

```
SELECT * FROM albums AS a WHERE a.album == "Galore"
```

LINQ approaches are invaluable for querying databases, but still take a relatively fine-grained approach for data manipulation (updates). The programmer is required to explicitly determine which changes were made at the application level. All modifications made by the user then need to be translated into equivalent insertions, updates and deletions for each table. In contrast, a typical user workflow consists of fetching a subset of the database, called a *view*, making changes to this view, and then propagating the changes to the database. Defining views that can be updated directly is known as the *view-update problem*, a long-standing area of study in the field of databases [4].

Relational Lenses. A recent approach to the relational database view update problem is to define views using composable *relational lenses* [5]. Lenses are a form of *bidirectional transformation* [16]. With relational lenses, instead of defining the view using a general SQL query, the programmer defines the view using individual lenses, which are known to behave in a correct manner and can be composed to define a view. Bohannon et al. [5] define lenses for performing projections, selections and joins. Figure 2 shows the composable nature of relational lenses.

Relational lenses can be considered a form of *asymmetric* lenses, in which we have a forward (*get*) direction to fetch the data, and a reverse (*put*) direction to make updates [18]. A bidirectional transformation is defined as *well-behaved* if it satisfies round-tripping guarantees:

$$\textsc{GetPut} \quad \text{put } s \ (\text{get } s) = s$$
$$\textsc{PutGet} \quad \text{get } (\text{put } s \ v) = v$$

Relational lenses are equipped with typing rules defining requirements that ensure operations on lenses are well-behaved. The type system for relational lenses tracks the attribute types of the defined view as well as constraints, including *predicates* and *functional dependencies*, which are not easily expressible in a ML-like type system.

From theory to practice. The theory of relational lenses was developed over a decade ago by Bohannon et al. [5], but until recently there has been little work on implementing them. Horn et al. [20] recently presented the first practical implementation using an incremental semantics. However, that paper concentrates primarily on performance rather than language integration, leading to two primary drawbacks: the requirement that *selection predicates* must be known completely statically, and limited *static typechecking*, meaning that errors are detected mostly at runtime.

Predicates. Some of the relational lens constructors, such as the select lens, require user supplied functions for filtering rows. Such functions, called *predicates*, determine whether or not an individual record should be included. Predicates are a function of type $r \rightarrow$ **bool** where r is the type of the input record and a return value of **true** indicates a match. A *select* lens selects only rows which match the predicate. Links supports *row types* [29], which allow predicates to be defined over any record which contains a given set of fields.

Bohannon et al. [5] treat predicates as abstract (finite or infinite) sets, without giving a computational syntax: the predicate function for a set Q can be defined as $\lambda x.\ x \in Q$. Sets allow predicates to be defined in an abstract form while still being amenable to mathematical reasoning, but such an approach does not scale to a practical implementation. In practice the user should define a predicate as a function from a record (in this case consisting of `album` and `year` fields) to a boolean value:

```
fun(x) { x.album = "Galore" && x.year == 1989 }
```

Some of the lens typing rules require static checks on predicates. The above predicate contains only *static* information, and is thus a *closed function* which can be checked at compile-time. Alas, such checks become problematic when the programmer would like to define a function which depends on information only available at runtime, such as a parameter in a web request. For example, consider the following function which adapts the `getAlbumsByName` function to use relational lenses.

```
fun getAlbumsByNameL(albumName) {
  var albumLens = lens albums where album -> quantity;
  var selectLens = select from albumsLens where
    fun(a) { a.album == albumName };
  get selectLens;
}
```

The `getAlbumsByNameL` function begins by defining a lens, `albumsLens`, over the `albums` table. A *functional dependency* $\overrightarrow{X} \rightarrow \overrightarrow{Y}$ states that the columns in $\overrightarrow{Y}$ are *uniquely determined* by $\overrightarrow{X}$; in our case the `album -> quantity` clause in the `albumsLens` definition states that the `quantity` attribute is uniquely determined by the `album` attribute.

As `albumsName` is supplied as a parameter to the `getAlbumsByNameL` function, the anonymous predicate supplied to `lensselect` can only be completely known at runtime. A dynamic predicate is therefore *not* closed. In general, this may mean that variables in the closure of a dynamic predicate may not be available until runtime, and may themselves refer to functions. While it is possible to statically know the *type* of the function, and thus rule out a class of errors, relational lenses require finer-grained checks which require a more in-depth analysis of the predicate. As an example, a select lens is

only well-formed if the predicate does not rely on the output of a functional dependency.

If we required the function to be fully known at compile time, a programmer could not define predicates that depend on user input. Thus, there is a tradeoff between static correctness and programming flexibility. We provide an implementation which supports both defining static predicates, and also having lenses which are only checked at runtime.

The original work on relational lenses only considered predicates in an abstract sense. We are first to investigate the practicalities of including and checking relational lens predicates in a practical programming language.

Functional Dependencies. Another obstacle is the handling of functional dependencies, which are an important part of the type system for relational lenses. Functional dependencies are constraints that apply to the data, and specify which fields in a table uniquely determine other fields. An example of a functional dependency would be `album -> quantity`, where in a set of records, all records that have the same album name should have the same quantity. A large part of the complexity of working with functional dependencies is due to *Armstrong's axioms* [2], discussed further in Section 3.

The typing rules given by Bohannon et al. [5] are important for showing soundness of relational lenses: without ensuring all the requirements are met, it is not possible to ensure the lenses are well behaved. We take the existing work by Bohannon et al. [5] and concretise the design, showing the algorithms required to implement the rules in practice.

1.1 Contributions

The primary technical contribution of this paper is the first full design and implementation of relational lenses in a typed functional programming language. As outlined above, relational lenses involve a number of nonstandard features, with a type system that tracks information about attributes, predicates and functional dependencies. Embedding relational lenses in Links leads to a number of design choices, which we have currently resolved as follows:

- How to interface Links's existing support for records and row typing with the attribute types in relational lenses. This is fairly straightforward; we treat record types as attribute sets enriched with the field value types.
- How to construct the predicates used in relational lenses. We support two approaches: *static* predicates, which are essentially a concrete syntax for SQL-style predicates, and *dynamic* predicates, in which other Links features can be used to construct predicates, which will therefore not be fully known (or checkable) until run-time.
- How to manage and check side-conditions involving functional dependencies. Functional dependencies are currently checked as part of the type system, but they also need to be available at run time in order to check side-conditions that relate predicates and functional dependencies.

This paper makes three concrete contributions:

(1) An implementation of *dynamic predicates* on relational lenses. We define a *static predicate IR* (§2.1) consisting of record projection and operations which can be easily evaluated and compiled to SQL. We then define a functional *dynamic predicate IR* (§2.2), and show how it can be evaluated and compiled to the static predicate IR.
(2) An implementation of the typing rules for relational lenses, adapted to the setting of a functional programming language (§3). Static predicates can be fully checked at compile time, whereas the typing rules can be used to check certain aspects of dynamic predicates at runtime (§3.4).
(3) An example tierless web application of a curation interface for a real-world scientific database, tying together relational lenses with the model-view-update architecture for frontend web development (§4).

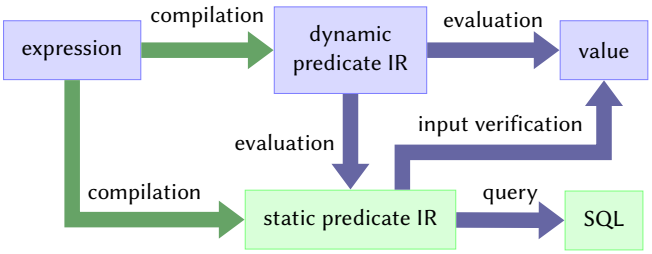


Figure 3: Links evaluation pipeline

The remainder of the paper proceeds as follows: §2 describes the implementation of static and dynamic predicates; §3 describes the implementation of static typechecking for relational lenses; §4 describes the case study; §5 describes related work; and §6 concludes.

2 PREDICATES

In this section, we describe the design and implementation of predicates for relational lenses.

Figure 3 shows how the predicate IR integrates with the rest of Links. Constructs in blue boxes (expressions, the Links internal representation (IR), and values) are existing Links constructs. Green boxes are newly added constructs by the lenses implementation. Green arrows indicate type checking / compilation stages, while blue arrows indicate actions performed during runtime / evaluation.

Notation. Labels are identifiers which refer to a the name of a column in a database. We write X, Y to denote a label. We refer to sets of labels as $\overrightarrow{X}$, $\overrightarrow{Y}$. We refer to sets of sets of labels as $\underline{X}$, $\underline{Y}$.

2.1 Static Predicate IR

Static predicates are closed functions which can be evaluated on a record to produce a value. Well-typed boolean predicates produce a boolean value and can either be evaluated directly or translated into a SQL expression.

Figure 4 shows the syntax, typing rules, and evaluation of the static predicate IR.

Syntax. Predicate terms P, Q consist of constants c (number, boolean or string values), labels X, boolean negation ($\neg P$) or binary operations ($P \odot Q$). The binary operator $\odot$ is defined as one of either a comparison operator $\odot_{cmp}$, an arithmetic operator $\odot_{arith}$ or a boolean operator $\odot_{bool}$.

Syntax

Base Types	D	::=	**bool** | **int** | **string**
Type environments	Φ	::=	$\cdot \mid \Phi, X : A$
Eval. environments	ρ	::=	$\cdot \mid \rho, X = c$
Terms	P, Q	::=	$c \mid X \mid P \odot Q \mid \neg P$
Operators	$\odot$	::=	$\odot_{arith} \mid \odot_{cmp} \mid \odot_{bool}$
	$\odot_{cmp}$	::=	$< \mid \leq \mid > \mid \geq \mid \neq \mid =$
	$\odot_{arith}$	::=	$+ \mid - \mid * \mid \div$
	$\odot_{bool}$	::=	$\wedge \mid \vee$

Typing rules $\boxed{\Phi \vdash P : A}$

$$\textsc{T-Constant}\ \frac{c \text{ of type } A}{\Phi \vdash c : A} \qquad \textsc{T-Field}\ \frac{X : A \in \Phi}{\Phi \vdash X : A} \qquad \textsc{T-Arith}\ \frac{\Phi \vdash P : \textbf{int} \qquad \Phi \vdash Q : \textbf{int}}{\Phi \vdash P \odot_{arith} Q : \textbf{int}}$$

$$\textsc{T-Bool}\ \frac{\Phi \vdash P : \textbf{bool} \qquad \Phi \vdash Q : \textbf{bool}}{\Phi \vdash P \odot_{bool} Q : \textbf{bool}} \qquad \textsc{T-Compare}\ \frac{\Phi \vdash P : A \qquad \Phi \vdash Q : A}{\Phi \vdash P \odot_{cmp} Q : \textbf{bool}}$$

$$\textsc{T-Neg}\ \frac{\Phi \vdash P : \textbf{bool}}{\Phi \vdash \neg P : \textbf{bool}}$$

Evaluation $\boxed{\mathcal{E}[\![P]\!]_\rho}$

$$\mathcal{E}[\![c]\!]_\rho = c \qquad \mathcal{E}[\![X]\!]_\rho = \rho(X) \qquad \mathcal{E}[\![P \odot Q]\!]_\rho = \mathcal{E}[\![P]\!]_\rho \,\hat{\odot}\, \mathcal{E}[\![Q]\!]_\rho$$

$$\mathcal{E}[\![\neg P]\!]_\rho = \hat{\neg}\mathcal{E}[\![P]\!]_\rho$$

Figure 4: Static predicate IR

Typing. We write D for base types, and a typing context Φ maps record labels X to types A. Evaluation environments ρ map record labels X to constants c.

The typing environment stays constant during type derivations, as none of the syntax rules include the option for defining further variables. The rule T-Field uses the context to look up the type of a variable in Φ. The function $\mathrm{dom}(\Phi)$ returns the domain of the typing environment (i.e., the labels of the record upon which the predicate operates). Any constant expressions assume the corresponding type D of the primitive constant. The typing rules of binary operations depend on the operator used. Binary boolean operations require the inputs to be boolean and produce a boolean, while arithmetic operators require integer inputs and produce an integer. Comparison operations take any input and produce a boolean. The unary negation operator negates a boolean predicate.

Evaluation. Evaluation is performed with respect to an evaluation context ρ, which maps identifiers to values. Constants evaluate to constants, and variables evaluate to the value for the specified identifier in the evaluation context. Binary operations first evaluate each of the arguments and then apply the specific interpretation of the operator, denoted as $\hat{\odot}$. Negation evaluates the argument and then performs a boolean negation on the result.

We say that an evaluation environment ρ is *compatible with a typing context* Φ, written $\Phi \vdash \rho$, if for all $x : A \in \Phi$ we have that $x = c \in \rho$, and $\Phi \vdash c : A$. It is straightforward to see that well-typed predicates evaluated under a compatible evaluation environment always reduce to a constant.

Proposition 2.1. *If $\Phi \vdash P : A$ and $\Phi \vdash \rho$, then $\mathcal{E}[\![E]\!]_P = c$ for some constant c such that $\Phi \vdash c : A$.*

Proof. By induction on the derivation of $\Phi \vdash P : A$. □

As a result, we know that a well-typed boolean predicate will always evaluate to a boolean value.

Compilation to SQL. As predicates are binary expressions which refer to column names, compilation from the predicate IR to SQL expressions is a mechanical translation into SQL syntax. As an example, the predicate

$$\texttt{rating} > 5 \wedge \texttt{album} = \text{“Galore”}$$

can be translated as the `WHERE` clause:

```
WHERE rating > 5 AND album = "Galore"
```

2.2 Dynamic Predicate IR

Relational lens predicates in the form presented in Figure 4 define a static predicate which can be straightforwardly converted into SQL. While it is possible for the programmer to define predicates in this form directly, static predicates do not allow a developer to dynamically generate predicates based on user or other input, and do not allow a developer to make use of other functions which are only known at runtime. To overcome this restriction, *dynamic* predicates allow predicates which can be passed to the relational lens constructor at runtime.

Syntax. Figure 5 shows the syntax and typing rules of the dynamic predicate IR, a subset of the Links intermediate representation which can be used to construct dynamic predicates. IR terms L, M, N include the usual variables x, constants c, lambda abstractions $\lambda x.\ M$, and application $L\ M$. Additionally, the IR includes the unit value (or empty record) $\langle\rangle$, record extension $\langle X = V; W\rangle$, record projection $M.X$, variant injection $(X\ M)^A$, case analysis **case** N $\{\overrightarrow{X = N}\}$, conditionals **if** L **then** M **else** N, and binary operations $M \odot N$. The typing rules are standard.

Evaluation. The relational lens constructor expects a predicate in the form $A_{\text{row}} \to \textbf{bool}$, where A_{row} is a record type of an entry in the lens view. Figure 6 shows a big-step evaluation relation on the dynamic predicate IR. Note that the big-step evaluation relation is partial: currently conditionals depending on a database value $\mathcal{R}.X$ are unsupported and will fail at runtime, but we plan to address this as immediate future work.

The *outer* translation $M \downarrow_\phi V$ evaluates a function taking a variable x of base record type A_{row}. It binds value x to a distinguished record value $\mathcal{R}$ in the environment and evaluates the function body using the *inner* evaluation relation $M \Downarrow_\phi V$, which states that term M evaluates to value V under evaluation environment ϕ.

Syntax

Types	A, B, C	::=	D
		|	$A \to B$
		|	$\langle \overrightarrow{X : A} \rangle \mid [\overrightarrow{X : A}]$
Base types	D	::=	**bool** | **int** | **string**
Base record types	A_{row}	::=	$\langle \overrightarrow{X : D} \rangle$
Labels	X		
Values	V, W	::=	$\lambda x.M \mid c$
		|	$\langle\rangle \mid \langle \overrightarrow{X = V} \rangle \mid \mathcal{R}.X$
		|	$(X\ V)^A \mid V \odot W$
Terms	L, M, N	::=	$x \mid c$
		|	$\lambda x.\ M \mid L\ M$
		|	$\langle\rangle \mid \langle X = M; N \rangle \mid M.X$
		|	$(X\ M)^A \mid$ **case** $M\ \{\overrightarrow{X = N}\}$
		|	**if** L **then** M **else** N
		|	$M \odot N$

Typing rules

$$\text{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{T-Const}\ \frac{c \text{ of type } A}{\Gamma \vdash c : A} \qquad \text{T-Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \text{T-Unit}\ \frac{}{\Gamma \vdash \langle\rangle : \langle\rangle}$$

$$\text{T-Extend}\ \frac{\Gamma \vdash M : B \qquad \Gamma \vdash N : \langle \overrightarrow{X : A} \rangle \qquad X \notin \overrightarrow{X}}{\Gamma \vdash \langle X' = M; N \rangle : \langle X' : B \cdot \overrightarrow{X : A} \rangle} \qquad \text{T-Project}\ \frac{\Gamma \vdash M : \langle X_i : A_i \rangle_{i \in I} \qquad j \in I}{\Gamma \vdash M.X_j : A_j}$$

$$\text{T-Variant}\ \frac{A = [(X_i : A_i)_{i \in I}] \qquad \Gamma \vdash M : A_j \qquad j \in I}{\Gamma \vdash (X_j\ M)^A : A} \qquad \text{T-Case}\ \frac{\Gamma \vdash M : [(X_i : A_i)] \qquad (\Gamma, x_i : A_i \vdash N_i : B)_i}{\Gamma \vdash \textbf{case}\ M\ \{(X_i(x_i) \mapsto N_i)_i\} : B}$$

$$\text{T-If}\ \frac{\Gamma \vdash L : \textbf{bool} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N : A}$$

$$\text{T-ArithOp}\ \frac{\Gamma \vdash M : \textbf{int} \qquad \Gamma \vdash N : \textbf{int}}{\Gamma \vdash M \odot_{arith} N : \textbf{int}} \qquad \text{T-CompareOp}\ \frac{\Gamma \vdash M : D \qquad \Gamma \vdash N : D}{\Gamma \vdash M \odot_{cmp} N : \textbf{bool}}$$

$$\text{T-BoolOp}\ \frac{\Gamma \vdash M : \textbf{bool} \qquad \Gamma \vdash N : \textbf{bool}}{\Gamma \vdash M \odot_{bool} N : \textbf{bool}}$$

Figure 5: Dynamic Predicate IR

Translation to Static Predicate IR.. Figure 7 defines the translation from IR values to the static predicate IR. The translation $\mathcal{V}[\![-]\!]$ is only defined on terms of type **bool**: constants translate to themselves; operations translate to operations in the predicate language; and projections on the special value $\mathcal{R}$ are translated into variable references. Base row types are translated into predicate typing environments through the use of the $\mathcal{R}[\![-]\!]$ translation.

Outer Predicate Evaluation $\boxed{V \downarrow_\phi W}$

$$\frac{M \Downarrow_{\phi, x = \mathcal{R}} V}{\lambda x.M \downarrow_\phi V}$$

Dynamic Predicate Evaluation $\boxed{M \Downarrow_\phi V}$

$$\frac{}{V \Downarrow_\phi V} \qquad \frac{\phi(x) = V}{x \Downarrow_\phi V} \qquad \frac{L \Downarrow_\phi \textbf{true} \qquad M \Downarrow_\phi V}{\textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N \Downarrow_\phi V}$$

$$\frac{L \Downarrow_\phi \textbf{false} \qquad M \Downarrow_\phi V}{\textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N \Downarrow_\phi V} \qquad \frac{M \Downarrow_\phi W \qquad N \Downarrow_\phi \langle \overrightarrow{X = V} \rangle}{\langle X' = M; N \rangle \Downarrow_\phi \langle X' = W \cdot \overrightarrow{X = V} \rangle}$$

$$\frac{M \Downarrow_\phi \langle (X_i = V_i)_{i \in I} \rangle \qquad j \in I}{M.X_j \Downarrow_\phi V_j} \qquad \frac{M \Downarrow_\phi \mathcal{R}}{M.X \Downarrow_\phi \mathcal{R}.X}$$

$$\frac{L \Downarrow_\phi \lambda x.N \qquad M \Downarrow_\phi V \qquad N \Downarrow_{\phi, x = V} W}{L\ M \Downarrow_\phi W}$$

$$\frac{M \Downarrow_\phi (X\ V)^A \qquad j \in I \qquad N_j \Downarrow_{\phi, x_j = V} W}{\textbf{case}\ M\ \{(X_i(x_i) \mapsto N_i)_{i \in I}\} \Downarrow_\phi W}$$

$$\frac{M \Downarrow_\phi V \qquad N \Downarrow_\phi W}{M \odot N \Downarrow_\phi V \odot W}$$

Figure 6: Dynamic Predicate Evaluation

Translation from values to predicates $\boxed{\mathcal{V}[\![V]\!] = P}$

$$\begin{array}{rcl} \mathcal{V}[\![c]\!] & = & c \\ \mathcal{V}[\![V \odot W]\!] & = & \mathcal{V}[\![V]\!] \odot \mathcal{V}[\![W]\!] \\ \mathcal{V}[\![\mathcal{R}.X]\!] & = & X \end{array}$$

Translation on base record types $\boxed{\mathcal{R}[\![\langle \overrightarrow{X : D} \rangle]\!] = \Phi}$

$$\mathcal{R}[\![\langle X_1 : D_1, \ldots, X_n : D_n \rangle]\!] \quad = \quad X_1 : D_1, \ldots, X_n : D_n$$

Figure 7: Translation from values to predicates

We begin by showing that evaluation preserves typing. We extend the notion of environment compatibility to term typing environments, writing $\Gamma \vdash \phi$ if for each $x : A \in \Gamma$ we have that $x = V \in \phi$, with $\Gamma \vdash V : A$.

Lemma 2.2. *Suppose* $\Gamma \vdash M : A$ *and* $\Gamma \vdash \phi$ *for some* ϕ. *If* $M \Downarrow_\phi V$, *then* $\Gamma \vdash V : A$.

Proof. By induction on the derivation of $\Gamma \vdash M : A$. □

Knowing that evaluation preserves typing, we can show that if the translation of an arbitrary predicate function is defined, then it will translate into a well-typed predicate in the static predicate IR.

Proposition 2.3 (Correctness of Dynamic Predicates). *Given* Γ, ϕ *such that* $\Gamma \vdash \phi$, *and a record type* $A_{row} = \langle \overrightarrow{X : D} \rangle$, *if* $\Gamma \vdash V :$

Functional Dependencies

$$F, G ::= \overrightarrow{X} \rightarrow \overrightarrow{Y}$$

Armstrong's Axioms $\boxed{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}}$

FD-ID
$$\frac{\overrightarrow{X} \rightarrow \overrightarrow{Y} \in \mathcal{F}}{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}}$$

FD-Transitive
$$\frac{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y} \qquad \mathcal{F} \vDash \overrightarrow{Y} \rightarrow \overrightarrow{Z}}{F \vDash \overrightarrow{X} \rightarrow \overrightarrow{Z}}$$

FD-Refl.
$$\frac{\overrightarrow{Y} \subseteq \overrightarrow{X}}{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}}$$

FD-Aug
$$\frac{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}}{\mathcal{F} \vDash \overrightarrow{X}\ \overrightarrow{Z} \rightarrow \overrightarrow{Y}\ \overrightarrow{Z}}$$

FD-Composition
$$\frac{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y} \qquad \mathcal{F} \vDash \overrightarrow{X'} \rightarrow \overrightarrow{Y'}}{\mathcal{F} \vDash \overrightarrow{X}\ \overrightarrow{X'} \rightarrow \overrightarrow{Y}\ \overrightarrow{Y'}}$$

FD-Decomposition
$$\frac{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}\ \overrightarrow{Z}}{\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}}$$

Figure 8: Armstrong's functional dependency axioms.

$A_{row} \rightarrow \textbf{\textit{bool}}$ *and* $\mathcal{V}[\![V \downarrow_\phi W]\!] = P$ *for some predicate* P*, then* $\mathcal{R}[\![A_{row}]\!] \vdash P : \textbf{\textit{bool}}$.

3 TYPECHECKING RELATIONAL LENSES

In this section, we show by example how naïve composition of lens combinators can give rise to ill-formed lenses, and show how such ill-formed lenses can be ruled out using static and dynamic checks. We take as our basis the rules proposed by Bohannon et al. [5], and show how they can be adapted to the setting of a functional programming language. We begin by showing how functional dependencies can be handled, and then look at each lens combinator in turn.

3.1 Functional Dependencies

Functional dependencies are constraints restricting combinations of records. A functional dependency $\overrightarrow{X} \rightarrow \overrightarrow{Y}$ requires that two records with the same values for $\overrightarrow{X}$ should have the same values for $\overrightarrow{Y}$. We use $\mathcal{F}$ and $\mathcal{G}$ to denote sets of functional dependencies. In the context of functional dependencies we use the notation $\overrightarrow{X}\ \overrightarrow{Y}$ to denote the union of the sets $\overrightarrow{X}$ and $\overrightarrow{Y}$. It is possible to derive functional dependencies from other functional dependencies. $\mathcal{F} \vDash \overrightarrow{X} \rightarrow \overrightarrow{Y}$ specifies that the functional dependency $\overrightarrow{X} \rightarrow \overrightarrow{Y}$ can be derived from the set of functional dependencies $\mathcal{F}$. The derivation rules for $\vDash$ are shown in Figure 8. Finally, we write $\mathcal{F} \vDash \mathcal{G}$ when every functional dependency in $\mathcal{G}$ can be derived from $\mathcal{F}$, and $\mathcal{F} \equiv \mathcal{G}$ for the symmetric closure of $\vDash$ on sets of functional dependencies.

Bohannon et al. [5] impose a special restriction on functional dependencies called *tree form*. Tree form requires that functional dependencies form a forest, meaning that column names can be partitioned into sets forming a directed acyclic graph with at most one incoming edge per node.

An implementation of the relational type checking rules requires verification that the functional dependencies of a lens are in tree form. We restrict the allowed functional dependencies accepted by our tree form checks to sets of functional dependencies that only require the FD-Decomposition rule to be applied, and reject any functional dependency inputs for which the tree form can only be derived by applying further rules. We argue the FD-Decomposition rule is useful, as it allows the definition of a table with a set of functional dependencies, e.g., $A \rightarrow BCD$ and allows this table to be joined to another table which only contains a join key that is a subset of this table, e.g., a table with the functional dependencies $CD \rightarrow E$. In this case the functional dependency of the left table is split into $A \rightarrow B$ and $A \rightarrow CD$. We define the functions $\mathsf{lefts}(\mathcal{F})$ as the functions which for each functional dependency $\overrightarrow{X} \rightarrow \overrightarrow{Y}$ returns the multiset of $\overrightarrow{X}$ values (written $\wr\overrightarrow{X}\int$), and $\mathsf{rights}(\mathcal{G})$ as the function which returns the multiset of $\overrightarrow{Y}$ values.

$$\begin{aligned} \mathsf{lefts}(\mathcal{F}) &\triangleq \wr X \mid X \rightarrow Y \in \mathcal{F}\int \\ \mathsf{rights}(\mathcal{F}) &\triangleq \wr Y \mid X \rightarrow Y \in \mathcal{F}\int \end{aligned}$$

We verify that functional dependencies are in tree form using the $\mathsf{inTreeForm}(\mathcal{F})$ function. The first restriction that tree form requires ($\mathsf{isDisjoint}(\mathsf{dom}\,(\mathsf{lefts}(\mathcal{F})))$) is that for all functional dependencies $\overrightarrow{X} \rightarrow \overrightarrow{Y}$, none of the left-hand sides may overlap. An example of an invalid combination is $A \rightarrow C; A\ B \rightarrow D$, resulting in two nodes in the graph, A and $A\ B$, which are not disjoint. We use set semantics because multiple functional dependencies with the same $\overrightarrow{X}$ are allowed, since they produce the same node. We then require that the multiset of $\overrightarrow{Y}$'s should be disjoint by enforcing $\mathsf{isDisjoint}(\mathsf{rights}(\mathcal{F}))$. This restriction prevents two functional dependencies from pointing to the same node, such as $A \rightarrow C; B \rightarrow C$ which would cause the node C to have two incoming edges. To detect duplicates we check disjointness in multiset semantics.

The $\mathsf{inTreeForm}(\mathcal{F})$ function applies the function $\mathsf{splitFDs}(\mathcal{F})$ to the functional dependencies, which uses the FD-Split rule to ensure that functional dependencies are in tree form and that functional dependencies have disjoint nodes. The function takes each functional dependency $\overrightarrow{X} \rightarrow \overrightarrow{Y}$, and then splits it into multiple dependencies $\overrightarrow{X} \rightarrow \overrightarrow{Z}$, where each $\overrightarrow{Z}$ is a subset of $\overrightarrow{Y}$ and appears in some other functional dependency $\overrightarrow{Z} \rightarrow \cdot$. To ensure that no part of the functional dependency is lost, all remaining columns $\overrightarrow{Z'}$ in $\overrightarrow{Y}$ not covered by any rules are summed together into a rule $\overrightarrow{X} \rightarrow \overrightarrow{Z'}$. The resulting set of functional dependencies is checked to ensure that all the nodes are disjoint ($\mathsf{isDisjoint}(\mathsf{nodes}(\mathsf{splitFDs}(\mathcal{F})))$) and there are no cycles $\mathsf{isAcyclic}(\mathcal{F})$. The function $\mathsf{isAcyclic}(\mathcal{F})$ is left out for brevity, as it performs standard cycle detection.

The support function $\mathsf{supp}\left(\overrightarrow{X}\right)$ returns the underlying set of a bag, removing all duplicates. We denote the number of occurrences of an element $\overrightarrow{X}$ in the set $\underline{X}$ as $m_{\underline{X}}(\overrightarrow{X})$.

$$
\begin{array}{llll}
\text{Types} & A, B & ::= & \cdots \mid \textbf{table of}\ \Phi \mid \textbf{lens of}\ (\Phi, P, \mathcal{F}) \\
 & & \mid & \textbf{record set of}\ \Phi \\
\text{Terms} & L, M, N & ::= & \cdots \mid \textbf{lens}\ M\ \textbf{with}\ \mathcal{F} \\
 & & \mid & \textbf{select}_Q\ \textbf{from}\ M \\
 & & \mid & \textbf{join}\ M\ \textbf{with}\ N\ \textbf{delete\_left} \\
 & & \mid & \textbf{drop}\ X'\ \textbf{determined by}\ (\overrightarrow{X}, V)\ \textbf{from}\ M \\
 & & \mid & \textbf{get}\ M \mid \textbf{put}\ M\ \textbf{with}\ N
\end{array}
$$

Figure 9: Syntax of lens types and terms

$$
\begin{aligned}
\mathsf{splitFD}(\overrightarrow{Z}, \overrightarrow{X} \to \overrightarrow{Y}) &\triangleq \left\{ \overrightarrow{X} \to \overrightarrow{Y}' \mid \overrightarrow{Y}' \in \overrightarrow{Z},\ \overrightarrow{Y}' \subseteq \overrightarrow{Z} \right\} \\
&\quad \cup \left\{ \overrightarrow{X} \to \overrightarrow{Y}' \mid \overrightarrow{Y}' = \overrightarrow{Y} - \bigcup \overrightarrow{Z},\ \overrightarrow{Y}' \neq \emptyset \right\} \\
\mathsf{splitFDs}(\mathcal{F}) &\triangleq \bigcup_{\overrightarrow{X} \to \overrightarrow{Y} \in \mathcal{F}} \mathsf{splitFD}(\mathsf{lefts}(\mathcal{F}), \overrightarrow{X} \to \overrightarrow{Y}) \\
\mathsf{nodes}(\mathcal{F}) &\triangleq \mathsf{supp}\,(\mathsf{lefts}(\mathcal{F})) \cup \mathsf{supp}\,(\mathsf{rights}(\mathcal{F})) \\
\mathsf{isDisjoint}(\underline{X}) &\triangleq \forall \overrightarrow{Y}, \overrightarrow{Z} \in \underline{X}.\ \overrightarrow{X} \cap \overrightarrow{Y} = \emptyset \vee \overrightarrow{X} = \overrightarrow{Y} \wedge m_{\underline{X}}(\overrightarrow{Y}) = 1 \\
\mathsf{inTreeForm}(\mathcal{F}) &\triangleq \mathsf{isDisjoint}(\mathsf{supp}\,(\mathsf{lefts}(\mathcal{F}))) \\
&\quad \wedge \mathsf{isDisjoint}(\mathsf{rights}(\mathcal{F})) \\
&\quad \wedge \mathsf{isDisjoint}(\mathsf{nodes}(\mathsf{splitFDs}(\mathcal{F}))) \\
&\quad \wedge \mathsf{isAcyclic}(\mathsf{splitFDs}(\mathcal{F}))
\end{aligned}
$$

3.2 Lens Types

Figure 9 shows the additional types and terms for the lens constructs. A **lens** type contains the set of columns defined as a static predicate IR typing context Φ, mapping record labels to column types. Each lens has a *restriction predicate* P which defines valid records which can be used as input during **put**. Similarly any set of records applied to the lens must satisfy the set of functional dependencies $\mathcal{F}$. A *table type* **table of** Φ is the type of handles to the database. The *record set type* **record set of** ϕ describes a set of records where the record type corresponds to the predicate typing environment Φ.

In the remainder of the section, we describe each lens combinator and its typing rule in turn.

3.3 Rules

We now introduce the rules we use to typecheck relational lenses. These are adapted from the rules as defined by Bohannon et al. [5] to make use of our concrete predicate syntax.

Lens Primitives. The rule T-Lens is used to lift Links tables into Relational Lenses. In doing so, the programmer can define arbitrary functional dependencies applying to the table, which may be more extensive than the key definitions provided by Links. A lens primitive is assigned the default predicate constraint **true**. All columns referred to by functional dependencies should be part of the table record type Φ.

$$
\textsc{T-Lens}\quad \frac{\Gamma \vdash M : \textbf{table of}\ \Phi \qquad \bigcup \mathsf{nodes}(\mathcal{F}) \subseteq \mathsf{dom}\,(\Phi)}{\Gamma \vdash \textbf{lens}\ M\ \textbf{with}\ F : \textbf{lens of}\ (\Phi, \textbf{true}, \mathcal{F})}
$$

Select Lens. Let us assume we have a lens l_1 which is the join of the two tables albums and tracks. We might first define a lens l_2 to find popular albums for which the stock is too low, by only returning the albums where `quantity < rating`.

track	date	rating	album	quantity
Lullaby	1989	3	Galore	1
Lovesong	1989	5	Galore	1
Lovesong	1989	5	Paris	4
Trust	1992	4	Wish	4

We might then decide to further limit this view by defining a lens l_3 which only shows the tables with the album Galore.

track	date	rating	album	quantity
Lullaby	1989	3	Galore	1
Lovesong	1989	~~5~~ 4	Galore	1

The user then notices that the rating for *Lovesong* is not correct, and changes it from 5 to 4. Calling **put** on l_3 would yield the updated view for l_2:

track	date	rating	album	quantity
Lullaby	1989	3	Galore	1
Lovesong	1989	~~5~~ 4	Galore	1
Lovesong	1989	~~5~~ 4	Paris	4
Trust	1992	4	Wish	4

Since the rating of the track Lovesong is 4 and not lower than the quantity of the album Paris, the updated view for l_2 violates the predicate requirement `quantity < rating`.

To prevent such an invalid combination of lenses, the select lens needs to ensure that the underlying lens has no predicate constraints on any fields which may be changed by functional dependencies. The set of fields which can be changed by functional dependencies $\mathcal{F}$ is outputs($\mathcal{F}$). A predicate P ignores the set $\overrightarrow{X}$ if the result of evaluating the predicate ρ is not affected by changing any fields in $\overrightarrow{X}$. Using our predicate semantics, we can traverse the syntax tree and determine if the predicate references any variables in $\overrightarrow{X}$. It is necessary for P to ignore outputs($\mathcal{F}$), which prevents l_3 from being well-typed.

The T-Select rule also needs to ensure that the resulting lens only accepts records that satisfy the given predicate Q as well as any existing constraints P that already apply to the underlying lens. The resulting lenses constraint predicate can thus be defined as $P \wedge Q$. The full select lens typing rule can be defined as:

$$
\textsc{T-Select}\quad \frac{\begin{array}{ll}\Gamma \vdash M : \textbf{lens of}\ (\Phi, P, \mathcal{F}) & \Phi \vdash Q : \textbf{bool} \\ \mathsf{inTreeForm}(\mathcal{F}) & Q\ \text{ignores outputs}(\mathcal{F})\end{array}}{\Gamma \vdash \textbf{select}_Q\ \textbf{from}\ M : \textbf{lens of}\ (\Phi, P \wedge Q, \mathcal{F})}
$$

Join Lens. Join lenses have limitations on what functional dependencies the underlying tables can have. Let us assume that there is another table `reviews` which contains album reviews by users. The table has the functional dependency `user album -> review`[1].

user	review	album
musicfan	4	Galore
90sclassics	5	Galore
thecure	5	Paris

The `reviews` table is joined with the `tracks` table to produce the lens l_1. Now let us assume the user tries to delete the first "90sclassics" record.

user	review	track	date	rating	album
musicfan	4	Lullaby	1989	3	Galore
musicfan	4	Lovesong	1989	5	Galore
~~90sclassics~~	~~5~~	~~Lullaby~~	~~1989~~	~~3~~	~~Galore~~
90sclassics	5	Lovesong	1989	5	Galore
thecure	5	Lovesong	1989	5	Paris

Then there is no way to define correct put behaviour. If the user's review is deleted then the other entry by the same user would also be removed from the joined table. If the track is deleted, then the entry from the other user for the same track would also be removed.

The issue is resolved by requiring that one of the tables is completely determined by the join key. The added functional dependency restriction ensures that each entry in the resulting view is associated with exactly one entry in the left table. In this case if the reviews table contained a single review per track, it would be possible to delete any individual record by only deleting the entry in the reviews table. In practice we need to show that we can derive the functional dependency $\overrightarrow{X} \cap \overrightarrow{Y} \rightarrow \overrightarrow{Y}$, where $\overrightarrow{X} \cap \overrightarrow{Y}$ are the join columns and $\overrightarrow{Y}$ is the set of columns of the right table. It is possible to check if this functional dependency can be derived by calculating the transitive closure of $\overrightarrow{X} \cap \overrightarrow{Y}$ and then checking if $\overrightarrow{Y}$ is a subset.

Join lenses come in different variants with varying deletion behaviours: A variant that always deletes the entry from the left table, a variant that tries to delete from the right table and otherwise deletes from the left table, and a variant that deletes the entries from both tables if possible. The type checking for these three variants is similar, so we only discuss the delete left lens. The rule T-Join-Left requires us to also show that P ignores outputs($\mathcal{F}$) and Q ignores outputs($\mathcal{G}$). The resulting lens should have the predicate $P \wedge Q$ since the record constraints of both input lenses apply to the output lens.

T-Join-Left

$$\frac{\begin{array}{c}\Gamma \vdash M : \textbf{lens of } (\Phi, P, \mathcal{F}) \qquad \Gamma \vdash N : \textbf{lens of } (\Phi', Q, \mathcal{G}) \\ \mathcal{G} \vDash \mathsf{dom}(\Phi) \cap \mathsf{dom}(\Phi') \rightarrow \mathsf{dom}(\Phi') \\ \mathsf{inTreeForm}(\mathcal{F}) \qquad \mathsf{inTreeForm}(\mathcal{G}) \\ P \text{ ignores outputs}(\mathcal{F}) \qquad Q \text{ ignores outputs}(\mathcal{G})\end{array}}{\Gamma \vdash \textbf{join } M \textbf{ with } N \textbf{ delete\_left} : \textbf{lens of } (\Phi \cup \Phi', P \wedge Q, \mathcal{F} \cup \mathcal{G})}$$

[1]This example does not satisfy functional dependency tree form. If it instead only had the functional dependencies `user -> review`, the same problem would occur.

Drop Lens. The *drop* lens allows a more fine-grained notion of relational projection, allowing us to remove a column from a view. Note that this is not to be confused with the SQL `DROP` statement, which deletes a record. Let us assume we define the lens l_1 as the select of `year > 1990` $\vee$ `rating > 4`.

track	date	rating	album
Lovesong	1989	5	Galore
Lovesong	1989	5	Paris
Trust	1992	4	Wish

We can then define the lens l_2 as l_1, but dropping column `date` determined by `track` to yield the table:

track	rating	album
Lovesong	~~5~~ 3	Galore
Lovesong	~~5~~ 3	Paris
Trust	4	Wish

What would the new predicate constraint be? It cannot reference the field `year`, since it does not exist anymore. If it is `rating > 4` then the last record violates it, hence the only valid predicate would be `true`. If we then change the `rating` from 5 to 3 for the track *Lovesong* however, it no longer satisfies the predicate since it is from year 1989 and the rating is only 3.

The underlying issue is the dependency between the dropped field `date` and the field `rating`. It is not possible to determine a predicate P which specifies if any `rating` value is valid independently of the drop column rate. Without being able to construct such a P, a lens cannot be well-typed.

We must show that an arbitrary predicate P can be deconstructed into the predicate $P_{X'} \wedge P_{\overrightarrow{X}}$ which should be equivalent to P, where $P_{X'}$ only depends on the field X' and $P_{\overrightarrow{X}}$ only depends on the fields $P_{\overrightarrow{X}}$, all fields except for X'. The predicate $P_{\overrightarrow{X}}$ becomes the record constraint applying to the lens l_2. Values for fields $\overrightarrow{X}$ are considered valid if they satisfy $P_{\overrightarrow{X}}$. Any record applied using **put** will be assigned a valid value for X' by the put lens. Since no changes to the fields $\overrightarrow{X}$ can affect whether a value for X' is considered valid, any values satisfying $P_{\overrightarrow{X}}$ will satisfy P when any value which satisfies $P_{X'}$ is chosen.

Partial evaluation. For the T-Drop rule we need to perform simplification of predicates using partial evaluation. By performing partial evaluation we mean the substitution of some variables that occur in a predicate without affecting any of the others. Simplification in this case means reducing the structure of a predicate as far as possible by substituting operators by their results if both of the operands are known, as well as performing *short-circuit evaluation* on boolean operands where one of the operands evaluates to **true** or **false** and outcomes can be eliminated.

Even in an impure, call-by-value language such as Links, this short-circuit simplification is behaviour-preserving since predicates must be pure. In the implementation, this is enforced by a type-and-effect system [9, 23].

We say that a typing context Φ is *weakly compatible with an execution environment* E, written $\Gamma \tilde{\vdash} E$, if for each $X = c \in E$ we have that either $X \notin \Gamma$ or $X : A \in \Gamma$ and $\Gamma \vdash c : A$.

With this, we can show that partial evaluation preserves typing.

$$\boxed{\mathcal{P}[\![P]\!]_E}$$

$$\mathcal{P}[\![X]\!]_E = \begin{cases} X & X \notin \mathcal{D}(E) \\ A & X : A \in E \end{cases}$$

$$\mathcal{P}[\![c]\!]_E = c$$

$$\mathcal{P}[\![P \odot Q]\!]_E = \begin{cases} \textbf{false} & \mathcal{P}[\![P]\!]_E = \textbf{false}, \odot \text{ is } \wedge \\ \textbf{false} & \mathcal{P}[\![Q]\!]_E = \textbf{false}, \odot \text{ is } \wedge \\ \textbf{true} & \mathcal{P}[\![P]\!]_E = \textbf{true}, \odot \text{ is } \vee \\ \textbf{true} & \mathcal{P}[\![Q]\!]_E = \textbf{true}, \odot \text{ is } \vee \\ c_1 \hat{\odot} c_2 & c_1 = \mathcal{P}[\![P]\!]_E, c_2 = \mathcal{P}[\![Q]\!]_E \\ \mathcal{P}[\![P]\!]_E \odot \mathcal{P}[\![Q]\!]_E & \text{otherwise} \end{cases}$$

$$\mathcal{P}[\![\neg P]\!]_E = \begin{cases} \hat{\neg} c & c = \mathcal{P}[\![P]\!]_E \\ \neg \mathcal{P}[\![P]\!]_E & \text{otherwise} \end{cases}$$

Figure 10: Partial evaluation of predicates

$$\begin{aligned} \mathrm{gv}(c) &= \{\{\}\} \\ \mathrm{gv}(X) &= \{\{X\}\} \\ \mathrm{gv}(P \odot Q) &= \begin{cases} \mathrm{gv}(P) \cup \mathrm{gv}(Q) & \odot \text{ is } \wedge \\ \mathrm{gv}(P) \times_\cup \mathrm{gv}(Q) & \text{otherwise} \end{cases} \\ \mathrm{gv}(\neg X) &= \{\{X\}\} \\ \mathrm{gv}(\neg c) &= \{\{\}\} \end{aligned}$$

Figure 11: The *grouped variables* function.

Proposition 3.1. *If* $\Phi \vdash P : A$ *and* $\Phi \vdash\!\!\!\sim E$, *then* $\Phi \vdash \mathcal{P}[\![P]\!]_E : A$.

Proof. By induction on the derivation of $\Phi \vdash P : A$. □

T-Drop

$$\frac{\mathcal{F} \equiv \mathcal{G} \cup \left\{\overrightarrow{X} \to X'\right\} \qquad \Gamma \vdash e_1 : \textbf{lens of } (\Phi \cup \left\{X' : \tau\right\}, P, \mathcal{F}) \qquad \overrightarrow{Y} = \mathrm{dom}(\Phi) \qquad \neg \mathrm{hasGrouped}\left(\mathrm{gv}(P), X'\right) \qquad \Gamma \vdash V : \tau \qquad P_{\overrightarrow{X}} = \mathcal{P}[\![P]\!]_{\{X'=V\}} \qquad (P = \textbf{false}) \vee (P_{\overrightarrow{X}} \neq \textbf{false})}{\Gamma \vdash \textbf{drop } X' \textbf{ determined by } (\overrightarrow{X}, V) \textbf{ from } M : \textbf{lens of } (\Phi, P_{\overrightarrow{X}}, \mathcal{G})}$$

To determine if the predicate P can be split into the two separate predicates, we use the *grouped variables* $\mathrm{gv}(P)$ helper function shown in Figure 11. The $\mathrm{gv}(P)$ function takes a predicate as an input, and determines a set of columns sets. Each column set shows a probable dependency between the variables in the set. A variable dependency indicates that changing a variable to value which satisfies a different input may still change the result of the predicate. Consider a predicate which contains $A > B$ as a term. The value of B influences whether a value of A is considered valid and vice versa, which introduces a dependency between the two variables. If such a dependency is found between the dropped column and any other column, then that predicate is rejected.

The $\mathrm{gv}(P)$ function requires the input predicate to be in negation normal form, meaning that all negations in a predicate are on

$$\begin{aligned} \mathrm{nnf}(c) &= c \\ \mathrm{nnf}(X) &= X \\ \mathrm{nnf}(P \odot Q) &= \mathrm{nnf}(P) \odot \mathrm{nnf}(Q) \\ \mathrm{nnf}(\neg c) &= \hat{\neg} c \\ \mathrm{nnf}(\neg X) &= \neg X \\ \mathrm{nnf}(\neg(P \odot Q)) &= \begin{cases} \mathrm{nnf}(\neg P) \vee \mathrm{nnf}(\neg Q) & \odot \text{ is } \wedge \\ \mathrm{nnf}(\neg P) \wedge \mathrm{nnf}(\neg Q) & \odot \text{ is } \vee \\ \mathrm{nnf}(P) \odot_\neg \mathrm{nnf}(Q) & \text{otherwise} \end{cases} \end{aligned}$$

Figure 12: Calculating negation normal form

variables. This can be calculated by using De Morgan's laws and by negating all binary operators. A function to perform this normalisation is shown in Figure 12. $\odot_\neg$ denotes the negated operator of $\odot$ (i.e. $>_\neg$ becomes $\geq$), and $\hat{\neg} c$ of the negated interpretation of the constant c.

The $\mathrm{gv}(P)$ function contains a simple implementation which overapproximates dependencies between variables. Constants contain no variable dependencies, and individual labels create a singleton group with that label. Binary operators generally result in a cross product union ($\times_\cup$) of the variable groups of the operands. We define the cross product union as the union of each pair in the cross product between the two sets, i.e.:

$$\underline{X} \times_\cup \underline{Y} := \left\{\overrightarrow{X} \cup \overrightarrow{Y} \mid \overrightarrow{X} \in \underline{X}, \overrightarrow{Y} \in \underline{Y}\right\}$$

One exception to this rule is the logical and operator $\wedge$, which returns the union of the two dependency sets. Negated variables and constants result in the same dependency set as the non negated version. Consider the predicate

$$(A = 1 \wedge B = 2) \vee (A = 1 \wedge B = 3),$$

which can be simplified to

$$A = 1 \wedge (B = 2 \vee B = 3).$$

The $\mathrm{gv}(P)$ function returns the groups $\{\{A; B\}\}$ for the first equation, and $\{\{A\}; \{B\}\}$ for the second equation. The $\{A; B\}$ group returned for the first variant of the predicate indicates a dependence between the variables A and B, while the grouped variables for the second equation does not. We accept this over approximation for the function $\mathrm{gv}(P)$, as false positives can usually be rewritten by the programmer.

If the column X' does not appear in a group with any of the columns in $\overrightarrow{X}$, then we know that it would be possible to deconstruct the predicate P into the two parts as required. To actually determine $P_{\overrightarrow{X}}$, we make use of the fact that the default value V should satisfy the predicate $P_{X'}$ (i.e. $\mathcal{E}[\![P_{X'}]\!]_{(X'=V)} = \textbf{true}$), and so partially evaluating P with $(X' = V)$ evaluates to $P_{\overrightarrow{X}} \wedge \mathcal{E}[\![P_{X'}]\!]_{X'=V} = P_{\overrightarrow{X}} \wedge \textbf{true} = P_{\overrightarrow{X}}$.

We can check if X' appears in any groups using the hasGrouped $(X, \underline{Y})$ function. The function determines if there is any group that contains

columns from $\underline{Y}$ in addition to the column X'. If hasGrouped $(X, \underline{Y})$ returns **false** then it is possible to split P into $P_{\overrightarrow{X}} \wedge P_{X'}$. The function is defined as:

$$\text{hasGrouped}\ (X, \underline{Y}) \triangleq \exists \overrightarrow{Y} \in \underline{Y}.\ \overrightarrow{Y} \cap \{X\} \neq \emptyset \wedge \overrightarrow{Y} \setminus \{X\} \neq \emptyset.$$

Lens Get. Finally we define typing rules for making use of relational lenses. Since Links is not dependently typed we drop the constraints which apply to the view and specify that calling **get** returns a set of records which all have the type Φ[2].

$$\text{T-Get}\ \frac{\Gamma \vdash e_1 : \textbf{lens of}\ (\Phi, P, \mathcal{F})}{\Gamma \vdash \textbf{get}\ e_1 : \textbf{record set of}\ \Phi}$$

Lens Put. Just as with T-Get, we have no way of statically ensuring that the input satisfies P and $\mathcal{F}$, so we only statically check that the updated view is a set of records matching type Φ. Instead we only ensure that the set of records satisfies $\mathcal{F}$ and P using runtime checks which can fail.

To ensure that the constraint P applies to each record ρ in a view, runtime checks ensure that $\mathcal{E}[\![P]\!]_\rho$ = **true**. Functional dependency constraints can be checked by projecting the set of records down to each functional dependency and determining if any two records violate a functional dependency.

$$\text{T-Put}\ \frac{\Gamma \vdash e_1 : \textbf{lens of}\ (\Phi, P, \mathcal{F}) \qquad \Gamma \vdash e_2 : \textbf{record set of}\ \Phi}{\Gamma \vdash \textbf{put}\ e_1\ \textbf{with}\ e_2 : \textbf{unit}}$$

3.4 Typechecking Dynamic Predicates

The typing rules defined in Section 3.3 assume static predicates, since they require knowledge of the restriction predicate in order to determine if the generated lens is well-behaved.

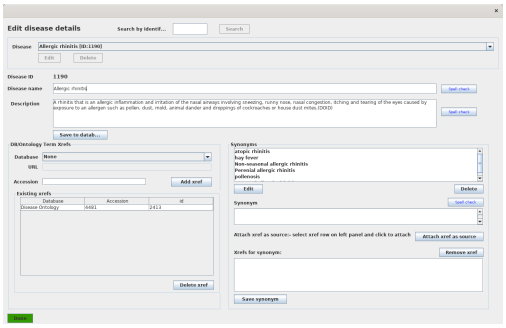
If a dynamic predicate is used in any lens combinator, the same checks are performed, but checking of predicates must be deferred to runtime. In this case, we require the programmer to acknowledge that the lens construction may fail at run-time. We introduce a special lens, the **check** lens, which the user must incorporate prior to using the lens in a **get** or **put** operation.

4 CASE STUDY: CURATED SCIENTIFIC DATABASES

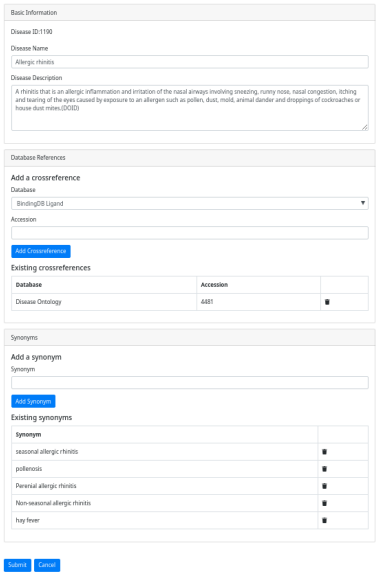
In this section, we illustrate the use of relational lenses in the setting of a larger Links application: part of the curation interface for a scientific database. Scientific databases collect information about a particular topic, and are *curated* by subject matter experts who manually enter and update entries.

The IUPHAR/BPS Guide to Pharmacology (GtoPdb) [24] is a curated scientific database which collects information on pharmacological targets, such as receptors and enzymes, and *ligands* such as pharmaceuticals which act upon targets. GtoPdb consists of a PostgreSQL database, a Java/JSP web application frontend to the database, and a Java GUI application used for data curation.

[2]Links has no native support for sets, so lists are used instead



(a) Java curation interface



(b) Links reimplementation

Figure 13: Curation interfaces for Diseases

In parallel work, we have implemented a workalike frontend application in Links, using the Links LINQ functionality. In this section, we demonstrate how we are beginning to use relational lenses for the curation interface, and show how relational lenses are useful in tandem with the Model-View-Update (MVU) paradigm pioneered by the Elm programming language [1].

4.1 Disease Curation Interface

One section of GtoPdb collects information on diseases, collecting information such as the drug name, description, crossreferences to other databases, and relevant drugs and targets. In this section, we describe a curation interface for diseases, where all interaction with the database occurs using relational lenses.

Figure 13a shows the official Java curation interface. The main data entries edited using the curation interface are the name and description of the disease; the crossreferences for the disease which refer to external databases; and the synonyms for a disease. As an example, a synonym for "allergic rhinitis" is "hayfever". Note that

the curation interface does not edit ligand or target information; curation of ligand-to-disease and target-to-disease links are handled by the ligand and target curation interfaces respectively.

Figure 13b shows the curation interface as a Links web application. The remainder of the section concentrates on the Links reimplementation.

4.2 Links Reimplementation

In the original implementation of Links [10], requests invoked Links as a CGI script. Modern Links web applications execute as follows:

(1) A Links application is executed, which registers URLs against page generation functions, and starts the webserver
(2) A request is made to a registered URL, and the server runs the corresponding page generation function
(3) The page generation function may spawn server processes, make database queries, and register processes to run on the client, before returning HTML to the client
(4) The client application spawns any client processes, and renders the HTML
(5) Client processes can communicate with server processes over a WebSocket connection.

4.2.1 Architecture. The disease curation interface consists of a persistent server process, and a client process which spawned by the Links MVU library.

Upon page creation, the application creates lenses to the underlying tables: the lenses retrieve data from and propagate changes to the database. Since lenses only exist on the server and cannot be serialised to the client, we spawn a process which awaits a message from the client with the updated data.

4.2.2 Tables and Lenses. We begin by defining the records we need, and handles to the underlying database and its tables.

First, we define a database handle, `db`, to the `gtopdb` database.

```
var db = database "gtopdb";
```

Next, we define type aliases for the types of records in each table. The disease curation interface uses tables describing four entity types: disease data (`DiseaseData`), metadata about external databases (`ExternalDatabase`), links from diseases to external databases (`DatabaseLink`), and disease synonyms (`Synonym`). (Note that `"prefix"` appears in quotes as `prefix` is a Links keyword).

```
typename DiseaseData =
  (disease_id: Int, name: String,
   description: String, type: String);

typename ExternalDatabase =
  (database_id: Int, name: String, url: String,
   specialist: Bool, "prefix": String);

typename DatabaseLink =
  (disease_id: Int, database_id: Int, placeholder: String);

typename Synonym =
  (disease_id: Int, synonym: String);
```

We will need to join the `ExternalDatabase` and `DatabaseLink` tables in order to render the database name of each external database link. It is therefore useful to define a type synonym for the record type resulting from the join:

```
typename JoinedDatabaseLink =
  (disease_id: Int, database_id: Int, placeholder: String,
   name: String, url: String,
   specialist: Bool, "prefix": String);
```

Next, we can define handles to each database table. The **`with`** clause specifies a record type denoting the column name and type of each attribute in the table, and the **`tablekeys`** clause specifies the primary keys (i.e., sets of attributes which uniquely identify a row in the database) for each table. We show only the definition of `diseaseTable`; the definitions for `databaseTable`, `dbLinkTable`, and `synonymTable` are similar.

```
var diseaseTable =
  table "disease" with DiseaseData
  tablekeys [["disease_id"]] from db;
```

The ID of the disease to edit (`diseaseID`) is provided as a `GET` parameter to the page, and thus we need a dynamic predicate as not all information is known statically. With the description of the entities and tables defined, we can describe the relational lenses over the tables. We work in a function scope where `diseaseID` has been extracted from the `GET` parameters.

```
fun diseaseFilter(x) { x.disease_id == diseaseID }

# Disease lenses
var diseasesLens = lens diseaseTable default;
var diseasesLens =
  check (select from diseasesLens by diseaseFilter);

# Database link lenses
var dbLens = lens databaseTable
  with { database_id -> name url specialist "prefix" };

var dbLinksLens = lens dbLinkTable default;
var dbLinksLens =
  check (select from dbLinksLens by diseaseFilter);
var dbLinksJoinLens = check (
  join dbLinksLens with dbLens
    on database_id delete_left);

# Synonym lenses
var synonymsLens = lens synonymTable default;
var synonymsLens =
  check (select from synonymsLens by diseaseFilter);
```

We create a lens over a table using the **`lens`** keyword, writing **`default`** when we do not need to specify functional dependencies. The `dbLens` lens specifies a functional dependency from `database_id` to each of the other columns, as knowledge of this dependency is required when constructing a join lens.

We need not filter the `databaseTable` table since we wish to display all external databases. The `diseaseLens`, `dbLinksLens`, and `synonymsLens` lenses make use of the **`select`** lens combinator, allowing us to consider only the records relevant to the given `diseaseID`. Note that each entity has a `disease_id` field: as a result, we can make use of Links' row typing system [23, 29] to define a *single* predicate, `diseaseFilter`, for each select lens.

The `dbLinksJoinLens` lens joins the external database links with the data about each external database by using the **`join`** lens combinator, stating that if a record is deleted from the view, then it should be deleted from the `dbLinkTable` rather than the `dbLens` table. Joining these two tables is only possible because `database_id` uniquely determines each column of the `databaseTable` table; as the lens uses a dynamic predicate, this property is checked at runtime.

4.2.3 Model. In implementing the case study, we make use of the *model-view-update* (MVU) paradigm, pioneered by the Elm programming language [1]. MVU is similar to the model-view-controller design pattern in that it splits the state of the system from the rendering logic. In contrast to MVC, MVU relies on explicit message passing to update the model. The key interplay between MVU and relational lenses is that MVU allows the model to be directly modified in memory, and relational lenses allow the changes in the model to be directly propagated to the database *without* writing any marshalling or query construction code.

Model.

```
typename DiseaseInfo =
  (diseaseData: DiseaseData,
   databases: [ExternalDatabase],
   dbLinks: [JoinedDatabaseLink],
   synonyms: [Synonym]);

typename Model =
  Maybe(
    (diseaseInfo: DiseaseInfo,
     selectedDatabaseID: Int,
     accessionID: String,
     newSynonym:String,
     submitDisease: (DiseaseInfo) {}~> ()));
```

The model (`Model`) contains all definitions retrieved from the database (`DiseaseInfo`), as well as the current value of the various form components for adding database links (`selectedDatabaseID` and `accessionID`) and synonyms (`newSynonym`). Finally, the model contains a function `submitDisease` which commits the information to the database. Note that the `{}~>` function arrow denotes a function which cannot be run on the database, and does not perform any effects. The `Model` type is wrapped in a `Maybe` constructor to handle the case where the application tries to curate a nonexistent disease.

Initial model. We now show how we construct the initial model. We begin by fetching the data from each lens, using the **`get`** primitive. We include type annotations for clarity, but they are not required.

```
var (diseases: [DiseaseData])       = get diseasesLens;
var (dbs: [ExternalDatabase])       = get dbLens;
var (dbLinks: [JoinedDatabaseLink]) = get dbLinksJoinLens;
var (synonyms: [Synonym])           = get synonymsLens;
```

Next, we spawn a server process which awaits the submission of an updated `DiseaseInfo` record. The `Submit` message contains the updated record along with a client process ID `notifyPid` which is notified when the query is complete.

The `submitDisease` function takes an updated `DiseaseInfo` process ID and sends a `Submit` message to the server. The **`spawnWait`** keyword spawns a process, waits for it to complete, and returns the retrieved value. In our case, we use **`spawnWait`** to only navigate away from the page once the query has completed.

```
var pid = spawn {
  receive {
    case Submit(diseaseInfo, notifyPid) ->
      put diseasesLens with [diseaseInfo.diseaseData];
      put dbLinksJoinLens with diseaseInfo.dbLinks;
      put synonymsLens with diseaseInfo.synonyms;
      notifyPid ! Done
  }
};

sig submitDisease : (DiseaseInfo) {}~> ()
fun submitDisease(diseaseInfo) {
  spawnWait {
    pid ! Submit(diseaseInfo, self());
    receive { case Done -> () }
  };
  redirect("/editDiseases")
}
```

Given the above, we can construct the initial model. Recall that the result of **`get`** `diseasesLens` is a *list* of `DiseaseInfo` records. As `disease_id` is the primary key for the `disease` table, we know that the result set must be either empty or a singleton list. Finally, we can initialise the model with the data retrieved from the database along with the `submitDisease` function and default values for the form elements.

```
var (initialModel: Model) = {
  switch(diseases) {
    case [] -> Nothing
    case d :: _ ->
      var diseaseInfo =
        (diseaseData = d,
         databases = dbs,
         dbLinks = dbLinks,
         synonyms = synonyms);

      Just((diseaseInfo = diseaseInfo,
            accessionID = "",
            newSynonym = "",
            selectedDatabaseID = hd(dbs).database_id,
            submitDisease = submitDisease))
  }
};
```

The model is rendered to the page using a `view` function which takes a model and produces some HTML to display. Interaction with the page produces *messages* which cause changes to the model. Finally, submission of the form causes the `submitDisease` function to be executed, which in turn sends a `Submit` message to the server to propagate the changes to the database using the lenses.

4.3 Discussion

In this section, we have described part of the curation interface for a real-world scientific database. Our application is a tierless web application with the client written using the Model-View-Update architecture.

Relational lenses allow seamless integration between all three layers of the application. Lenses with dynamic predicates allow us to retrieve the relevant data from the database; the data is used as part of a model which is changed directly as a result of interaction with the web page; and the updated data entries can be committed directly back into the database. At no point does a user need to write a query (or update): every interaction with the database uses only lens primitives.

The primary limitation of the implementation at present is that it does not currently support auto-incrementing primary keys, which are commonly used in relational databases. We plan to address this as immediate future work.

5 RELATED WORK

Edit Lenses. Edit lenses are a form of bidirectional transformation where, rather than translating directly between one data structure and another, the changes to a data structure are tracked and then translated into changes to the other data structure [19]. They can be particularly useful in the case of *symmetric lenses* in situations where neither of the data structures contain all of the data, and thus none of the sources can be considered the 'source' [18]. Changes could be described by *insert*, *update* and *delete* commands, and will usually result in similar insert, update or deletion commands for the other data structure.

Relational lenses as defined by Bohannon et al. are generally not considered edit lenses, as they directly translate the entire view to an updated source when performing get. *Incremental Relational Lenses* on the other hand take the updated view and compute a delta which is then translated into a delta to the source tables [20].

The language integration aspect of relational lenses is not dependent on the semantics used to perform relational updates. Instead it only relies on all of the relational lens typing rules in §3.3 to be satisfied. As long as this is the case, both the incremental and the non-incremental relational put semantics are ensured to be well-behaved.

Put-based Lenses. Bidirectional lenses are often defined in a form where we specify what we would like and then get the reverse direction for free. A common issue with this approach is that not every get function defines a unique well-behaved put function. As such, defining a bidirectional transformation by only specifying the forward direction is generally not sufficient. An alternative approach recently used is to rather have the programmer instead only specify the *put* semantics, which then uniquely define the *get* semantics [15, 21].

A *putback* approach to bidirectional transformations has been recently proposed by Asano et al. [3] for relational data as well. Asano et al. define a language which allows the specification of update queries, for which the forward query can automatically be derived. They support splitting views vertically for defining behaviour specific to columns and horizontally for behaviour specific to rows. For each of the different sections of the view they can then define the update behaviour, which can be simple checks or actual update semantics.

Tierless web programming. SMLServer [14] was among the first functional frameworks to allow interaction with a relational database. SMLServer acted as a plugin to a webserver such as AOLServer. The interaction model was through standard SQL queries, with results accessed using a `fold` function. Ur/Web [8] is a tierless web programming language. For database access, Ur/Web supports a statically-typed SQL DSL, along with atomic transactions and functional combinators on results. Neither framework supports language-integrated views.

Many other languages and frameworks support tierless web programming. Hop.js [31] builds on the Hop programming language [30] and allows tierless web programming in JavaScript. A particular strength is the use of *services* which allow distributed interaction patterns which go beyond the traditional client-server model, as well as providing rich integration with the existing JavaScript ecosystem. Eliom [27, 28] is a tierless functional programming framework building on top of the OCaml programming language. A particular distinguishing feature of Eliom is the ability to explicitly assign locations to functions and variables through the use of *section annotations*. The authors formalise the language as a core λ-calculus and describe compilation passes to client and server languages. ScalaLoci [32] is a Scala framework for tierless application programming. A key concept behind ScalaLoci is that data transfer between tiers uses the *reactive programming* paradigm. Haste.App [13] is a Haskell Embedded DSL allowing web applications to be written directly in Haskell. None of these languages have direct support for database integration, but since the languages are embedded DSLs or frameworks, it becomes possible to use the database functionality provided by other libraries. We are not aware of any work providing relational lenses as a library in any programming language.

Task-oriented programming (TOP) [26] is a high-level paradigm centred around the idea of a *task*, which can be thought of as a unit of work with an observable value. The TOP paradigm is implemented in the iTask system [25], a DSL written in the Clean programming language [6]. An *editor* is a task which interacts with a user. *Editlets* [12] are editors with customisable user interfaces, which can allow multiple users to interact with shared data sources. Much like incremental relational lenses [20], Editlets communicate *changes* in the data as opposed to the entire data source, however the user must specify this behaviour manually. Furthermore, shared data sources tend to correspond to mutable reference cells in ML, whereas relational lenses are more specialised to relational databases and include combinators specific to relational algebra.

Language-integrated Query. Language-integrated query is a rich area of research. Cooper [9] describes an effect type system and normalisation algorithm for Links language-integrated query, which defines a subset of Links code which can be compiled to efficient SQL with the knowledge that compilation will not fail at runtime. Giorgidze et al. [17] introduce *Database-Supported Haskell*, which allows Haskell query code to be run on the database directly rather than in memory.

Language-integrated query has been incorporated into the .NET framework, allowing its use in widely-used languages such as C# and F# [22]. Cheney et al. [7] describe a technique based on normalisation of quoted terms, showing that their technique can avoid

issues such as query avalanches, thus showing improvements relative to stock F# LINQ.

6 CONCLUSION

In this paper, we have presented the first full integration of relational lens combinators in a functional programming language, by extending the Links web programming language.

The *select* lens allows a user to work with a subset of the data in a particular table: a *predicate* allows a user to define the rows to consider. We have considered two forms of predicate: *static* predicates, which rely only on information known statically, and *dynamic* predicates, which can use information known only at runtime. We have shown typed intermediate representations for both static and dynamic predicates, and shown how we translate dynamic predicates into static predicates such that they can be translated into SQL.

In their original paper on relational lenses, Bohannon et al. [5] proposed a type system for relational lens combinators. We have provided the first implementation of their type system, fully statically checking lenses with static predicates, and providing lightweight dynamic checks for lenses with dynamic predicates.

As immediate future work, we plan to extend the predicate IRs to handle more complex functions, and remove the current partiality in the translation to the static predicate IR. We also plan to extend our implementation of relational lenses with the ability to handle auto-incrementing database fields.

REFERENCES

[1] 2019. Elm: A delightful language for reliable webapps. http://www.elm-lang.org. Accessed on 2019-07-04.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.

[3] Yasuhito Asano, Soichiro Hidaka, Zhenjiang Hu, Yasunori Ishihara, Hiroyuki Kato, Hsiang-Shang Ko, Keisuke Nakano, Makoto Onizuka, Yuya Sasaki, Toshiyuki Shimizu, et al. 2018. A View-based Programmable Architecture for Controlling and Integrating Decentralized Data. *arXiv preprint arXiv:1803.06674* (2018).

[4] François Bancilhon and Nicolas Spyratos. 1981. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 557–575.

[5] Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. 2006. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 338–347.

[6] T. H. Brus, Marko C. J. D. van Eekelen, M. O. van Leer, and Marinus J. Plasmeijer. 1987. CLEAN: A language for functional graph writing. In *FPCA (Lecture Notes in Computer Science)*, Vol. 274. Springer, 364–384.

[7] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ICFP*. ACM, 403–416.

[8] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *POPL*. ACM, 153–165.

[9] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *DBPL (Lecture Notes in Computer Science)*, Vol. 5708. Springer, 36–51.

[10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*. Springer, 266–296.

[11] George Copeland and David Maier. 1984. Making smalltalk a database system. In *ACM Sigmod Record*, Vol. 14. ACM, 316–325.

[12] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Editlets: type-based, client-side editors for iTasks. In *IFL*. ACM, 6:1–6:13.

[13] Anton Ekblad. 2016. High-performance client-side web applications through Haskell EDSLs. In *Haskell*. ACM, 62–73.

[14] Martin Elsman and Niels Hallenberg. 2003. Web Programming with SMLserver. In *PADL (Lecture Notes in Computer Science)*, Vol. 2562. Springer, 74–91.

[15] Sebastian Fischer, Zhengjiang Hu, and Hugo Pacheco. 2012. *Putback" is the essence of bidirectional programming.* Technical Report. Technical Report GRACE-TR 2012-08, GRACE Center, National Institute of ....

[16] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17.

[17] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. 2010. Haskell Boards the Ferry - Database-Supported Program Execution for Haskell. In *IFL (Lecture Notes in Computer Science)*, Vol. 6647. Springer, 1–18.

[18] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric lenses. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 371–384.

[19] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit lenses. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 495–508.

[20] Rudi Horn, Roly Perera, and James Cheney. 2018. Incremental relational lenses. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 74.

[21] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. 2014. Validity checking of putback transformations in bidirectional programming. In *International Symposium on Formal Methods*. Springer, 1–15.

[22] Dinesh Kulkarni, Luca Bolognese, Matt Warren, Anders Hejlsberg, and Kit George. 2007. LINQ to SQL: .NET Language-Integrated Query for Relational Data. *MSDN .NET Framework Developer Center (http://msdn. microsoft. com/enus/library/bb425822. aspx)* (2007).

[23] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102.

[24] Adam J. Pawson, Joanna L. Sharman, Helen E. Benson, Elena Faccenda, Stephen P.H. Alexander, O. Peter Buneman, Anthony P. Davenport, John C. McGrath, John A. Peters, Christopher Southan, Michael Spedding, Wenyuan Yu, Anthony J. Harmar, and NC-IUPHAR. 2013. The IUPHAR/BPS Guide to PHARMACOLOGY: an expert-driven knowledgebase of drug targets and their ligands. *Nucleic Acids Research* 42, D1 (11 2013), D1098–D1106. https://doi.org/10.1093/nar/gkt1143 arXiv:http://oup.prod.sis.lan/nar/article-pdf/42/D1/D1098/3598949/gkt1143.pdf

[25] Rinus Plasmeijer, Peter Achten, and Pieter W. M. Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. In *ICFP*. ACM, 141–152.

[26] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *PPDP*. ACM, 195–206.

[27] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: tierless Web programming from the ground up. In *IFL*. ACM, 8:1–8:12.

[28] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A Core ML Language for Tierless Web Programming. In *APLAS (Lecture Notes in Computer Science)*, Vol. 10017. 377–397.

[29] Didier Rémy. 1993. *Type inference for records in a natural extension of ML.* Theoretical Aspects Of Object-Oriented Programming.

[30] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a language for programming the web 2.0. In *OOPSLA Companion*. ACM, 975–985.

[31] Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs. In *ICFP*. ACM, 180–192.

[32] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *PACMPL* 2, OOPSLA (2018), 129:1–129:30.

[33] Limsoon Wong. 2000. Kleisli, a functional query system. *Journal of Functional Programming* 10, 1 (2000), 19–56.

Shapes and Flattening

IFL 2019 Draft Paper

John Reppy
Computer Science
University of Chicago
USA
jhr@cs.uchicago.edu

Joe Wingerter
Computer Science
University of Chicago
USA
wings@cs.uchicago.edu

ABSTRACT

NESL is a first-order functional language with an apply-to-each construct and other parallel primitives that enables the expression of irregular nested data-parallel (NDP) algorithms. To compile NESL, Blelloch and others developed a global flattening transformation that maps irregular NDP code into regular flat data parallel (FDP) code suitable for executing on SIMD or SIMT architectures, such as GPUs.

While flattening solves the problem of mapping irregular parallelism into a regular model, it requires significant additional optimizations to produce performant code. Nessie is a compiler for NESL that generates CUDA code for running on Nvidia GPUs. The Nessie compiler relies on a fairly complicated *shape analysis* that is performed on the FDP code produced by the flattening transformation. Shape analysis plays a key rôle in the compiler as it is the enabler of fusion optimizations, smart kernel scheduling, and other optimizations.

In this paper, we present a new approach to the shape analysis problem for NESL that is both simpler to implement and provides better quality shape information. The key idea is to analyze the NDP representation of the program and then preserve shape information through the flattening transformation.

1 INTRODUCTION

Nessie [17, 20] is a compiler for the nested-data-parallel (NDP) language NESL [4, 5] that targets GPUs. The NESL language was originally designed by Guy Blelloch as a way to program irregular parallel algorithms on the wide-vector and SIMD architectures of the early to mid 1990s. The enabling technology for NESL was a global flattening transformation developed by Blelloch and others that maps irregular NDP code into regular flat data parallel (FDP) code suitable for SIMD execution [7, 9, 12, 16].

The Nessie compiler follows the same approach for compiling NESL to GPUs. The front half of the Nessie compiler does the usual parsing and type checking, followed by monomorphization (NESL has parametric polymorphism). We then apply a modified form of Keller's flattening transformation [9] to produce a flat-data-parallel (FDP) representation of the program. In the flat representation, the only sequences are flat sequences of base types (`bool`, `float`, *etc.*). The original nested-sequence structure is captured in associated *segment descriptors*. The back-end of our compiler converts the FDP representation to CUDA code by first applying a flow-based shape analysis to determine the size and structure of sequences [20], then converting the FDP form to our λ_{cu} [18, 24] representation, optimizing the λ_{cu} representation, and finally generating code from it. The shape analysis is key to enabling optimizations: it identifies sequences of the same size, which enables the compiler to find more opportunities for fusion, memory reuse, and other optimizations.

Examining the results of shape analysis, however, one observes that the information recovered by the analysis is present in the original NDP representation and can be more easily detected before flattening. Furthermore, there is information that is trivially determined by the NDP representation that is more difficult to extract from the flattened program. For example, a sequence of pairs in the NDP representation will be transformed to a pair of sequences in the FDP representation;[1] in such a case, it may be difficult for a conservative analysis to recognize that the two sequences must have the same length.

This paper describes a new approach to mapping from a monomorphic NDP IR to a FDP IR annotated with shape information. Instead of applying shape analysis after flattening, we instead first analyze the NDP IR and record the shape information by annotating the types of the NDP program. We then apply a flattening transformation that transcribes the shape information from the NDP types into the type of the segment descriptors, such that if two segment descriptors have the same type, then they are the same.

The fundamental problem with shape analysis on the post-flattening FDP IR is that the flattening process disassociates related structures. The example of sequences of pairs has already been mentioned, but there are other examples, such as the fact that segment descriptors and the sequences that they describe are not directly connected. Thus the analysis has to work hard to identify which descriptors are associated with which data sequences. By performing the shape analysis prior to flattening, we do not have to rely on analysis to make those connections; instead, they are readily available in the structure of the NDP IR. The thesis of this work is that shape analysis on the NDP IR is inherently simpler and more precise than our previous flow-based analysis on the FDP IR. Furthermore, our new annotation language for shape information allows us to keep track of more precise shape information than before.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Flattening converts sequences of pairs to pairs of sequences (the so-called "AoS to SoA" transformation).

The rest of the paper is organized as follows. We first provide some background about the NESL language, the flattening transformation, and the need for shape information. In Section 3, we present a representation of types with shape annotations. We then describe how we analyze an NDP kernel language to produce a representation with shape information. In Section 5, we describe the flattening transformation that produces the FDP representation, where the segment descriptors are given types that reflect the shape of the sequences that they describe. We then describe the current status of our work and discuss future plans; finally we present concluding remarks in Section 8.

2 BACKGROUND

In this section, we describe the context in which we are operating, including a quick introduction to NESL, a brief description of the flattening transformation, and motivation for why shape information is important.

2.1 NESL

NESL is a first-order data-parallel functional language that supports data parallelism in two ways: through a parallel sequence comprehension (*apply-to-each*) and through a set of parallel primitive operators. Apply-to-each allows the programmer to map an arbitrary computation over a sequence. For example, the following function squares each element of the sequence `xs`:

```
function sqr (xs) = { x * x : x in xs };
```

It is also possible to map a computation over multiple sequences of the same length, as in this example that computes the element-wise product of `xs` and `ys`.

```
function prod (xs, ys) =
  { x * y : x in xs; y in ys };
```

Notice that NESL uses *zip* semantics for iteration over multiple sequences. To compute the outer product of these two sequences, we nest the iteration over `ys` inside the iteration over `xs`.

```
function outer (xs, ys) =
  { { x * y : y in ys } : x in xs };
```

An apply-to-each may include an optional predicate to specify which elements to apply the computation to; for example, we could compute the product of the positive elements of `xs` and `ys` as follows:

```
function prod_if_pos (xs, ys) =
  { x * y : x in xs; y in ys
  | x >= 0 and y >= 0 };
```

In this case, the size of the resulting sequence is unknown at compile time.

As seen in the `outer` example above, the computations that are mapped by an apply-to-each may themselves be parallel computations; thus the term *nested-data parallelism* is used for this programming pattern. The NDP model is well-suited to matrix operations; for example, we can use nested parallelism to multiply a matrix by a sequence:

```
function dotp (xs, ys) =
  sum({ x * y : x in xs; y in ys });

function mxv (m, v) =
  { dotp(row, v) : row in m };
```

p	::=	$fn\ p$	function definition
	|	f	
fn	::=	**fun** $f\,(x_1, \ldots, x_k) = e$	function
e	::=	v	value
	|	**if** v **then** e_1 **else** e_2	conditional
	|	$f\,(v_1, \ldots, v_k)$	function application
	|	**let** x **=** r **in** e	let binding
r	::=	e	right-hand-side expression
	|	$p\,(v_1, \ldots, v_k)$	primop application
	|	$\langle v_1, v_2 \rangle$	pair
	|	$\{\, e : b_1, \ldots, b_k \,\}$	parallel map
	|	$[\, v_1, \ldots, v_k \,]$	sequence construction
	|	$[\, \tau \,]$	empty sequence
v	::=	x	variable
	|	n	constant
b	::=	x **in** xs	binding

Figure 1: Syntax of NKL

The outer apply-to-each applies `dotp` in parallel to each row of the matrix. Within each call to `dotp`, the elements of the row are all multiplied in parallel, then added up with the parallel `sum` operation.

While the previous example is a computation over a rectangular matrix, nested parallelism does not have to be regular in NESL. For example, the computation of a sparse matrix (represented by rows of index-element pairs) times a dense sequence can be coded as

```
function sparse_mxv (sm, v) = {
    sum({ x * v[i] : (i, x) in sv }) : sv in sm
  }
```

This function has the same nested structure as before, but the amount of work per row varies. The fact that nested parallel computations can be arbitrary parallel computations, gives NESL a great deal of expressiveness that can be used to implement a wide variety of irregular parallel computations [21].

In addition to the parallel apply-to-each construct, NESL has a library of parallel sequence operations. These include reductions, such as the `sum` function above, prefix-scans, permutations, and various other operations on sequences [4].

2.2 NKL

We present our shape analysis and flattening techniques using a nested-data-parallel kernel language called NKL.[2] This language, whose abstract syntax is given in Figure 1, captures the important features of NESL in a more concise syntax. A program in NKL is a sequence of function definitions terminated by an exported main function. The expression language is normalized (*i.e.*, all arguments are either variables or constants) to simplify the shape analysis and flattening transformation. The last three right-hand-side forms support parallel sequences.

[2] This language is inspired by Keller's NKL [9].

Note that NKL's parallel map is more restrictive than NESL's general apply-to-each construct, since it does not include the optional predicate expression. We can represent the NESL expression

```
{ e : x1 in xs1, ..., xk in xsk | e' }
```

by the expression

```
let flgs = { e' : x1 in xs1, ..., xk in xsk };
    xs'1 = filter(xs1, flgs);
    ...
    xs'k = filter(xsk, flgs)
in
  { e : x1 in xs'1, ..., xk in xs'k }
```

where `filter` is a builtin function that takes two sequences of equal length and projects out the elements of the first that correspond to true entries in the second.

NKL types are defined by the following grammar:

τ	::=	$[\tau]$	sequence type
	|	$\tau_1 \times \tau_2$	pair type
	|	**int** | **bool** | **float** | $\cdots$	base types

and function types have the form

$$(\tau_1, \ldots, \tau_n) \to \tau$$

NKL programs are assumed to be explicitly monomorphically typed, but we omit type annotations to simplify the notation. Instead, we use "typeof(f)" to denote the type of functions and "typeof(x)" to denote the type of variables when necessary.

2.3 Flattening

Flattening is a global program transformation that converts nested data parallelism into flat data parallelism [6, 7, 9]. A key aspect of this transformation is that nested sequences are transformed into *segmented sequences*, which are flat sequences of atomic data values paired with segment descriptors that define the original sequences's nesting structure. The flattening transformation also coverts from "Array-of-Structs" (AoS) data layout to "Struct-of-Arrays" (SoA) layout. For example, the nested sequence:

```
[ [ (true, 1), (false, 2), (true, 3) ],
  [ ],
  [ (false, 4) ] ]
```

is represented by two data sequences

```
[ true, false, true, false ]
[ 1, 2, 3, 4 ]
```

and two segment descriptors that specify the lengths of the sequences at each level of nesting from inside out.

```
[ 3, 0, 1 ]
[ 3 ]
```

In practice, there are many different ways to represent segment descriptors that provide different benefits for different operations [1, 25]. For example, using an array of boolean segment-start flags allows segmented-scans to be defined using a flat scan operation [2].

In addition to affecting the data representation, the flattening transformation also affects the program structure. Code that is inside a parallel apply-to-each must be *lifted* to compute over sequences. For example, a function (or primitive operation) application

```
{ f(e1, e2) : x in xs }
```

is lifted to

```
if (empty xs)
  then []
  else let ys1 = { e1 : x in xs };
           ys2 = { e2 : x in xs }
       in
         f↑(ys1, ys2)
```

where $f^\uparrow$ is f lifted to work on sequences (*i.e.*, it is the parallel map of f over the zip of the argument sequences). The test for the empty input is the termination condition for recursion. To complete the above transformation, we recursively flatten the parallel maps of e_1 and e_2.

The flattening transformation uses bookkeeping operations to handle multiple levels of nested apply-to-each constructs (*i.e.*, we do not lift already lifted operations). Informally, these operations can be given the following types:

$\mathcal{S}$	:	$([[\tau]]) \to [\mathbf{int}]$	extract segment descriptor
$\mathcal{F}$	:	$([[\tau]]) \to [\tau]$	flatten one-level of nesting
$\mathcal{P}$	:	$([\tau], [\mathbf{int}]) \to [[\tau]]$	partition using descriptor

For example, the nested expression

```
{ { f(x, y) : x in xs, y in ys }
  : xs in xss, ys in yss
}
```

will be lifted to

```
let xs = F(xss);
    ys = F(yss);
    sd = S(xss)
in
  if (empty xs)
    then []
    else P (f↑ (xs ys), sd)
```

Because of the way that nested sequences are represented after flattening, these bookkeeping operations are constant-time.

Lastly, conditionals inside apply-to-each constructs are replaced with partitioning of the inputs. For example, we transform

```
{ if e then e1 else e2 : x in xs }
```

to

```
let flgs = { e1 : x in xs };
    (xs1, xs2) = SPLIT(xs, flgs);
    res1 = { e1 : x in xs1 };
    res2 = { e2 : x in xs2 }
in
  COMBINE(res1, res2, flgs)
```

Here the **SPLIT** and **COMBINE** operations are used to split and recombine sequences based on the sequence of booleans *flgs*.

2.4 The importance of shape information

Shape analysis is the process of statically identifying the nesting structure and sizes of sequences. Shape analysis plays a key rôle in the compiler as it is the enabler for a number of important optimizations, including

- Kernel fusion — Our compiler performs several kinds of fusion on parallel computations [17, 18, 24]. These include simple producer-consumer fusion; horizontal fusion of maps, scans, and reductions; and filter fusion [19]. The validity of these transformations requires that the compiler be able to reason about the sizes and structure of the argument sequences, which is information provided by the shape analysis.
- Memory reuse — reusing previously allocated memory for new results requires both lifetime analysis and information about the size of the memory object. The latter information is provided by shape analysis.
- Segment descriptor optimization — by attaching shape information to segment descriptors, we can identify when segment descriptors are redundant. We can also use this information to optimize their representation in some situations.
- Optimizing for rectangular structure — while, in general, nested sequences in NESL are irregular, there are examples of programs that impose a regular rectangular structure on arrays (*e.g.*, matrix multiplication). Segmented computations on such nested sequences can take advantage of the rectangular structure to gain efficiency. Shape analysis can discover when rectangular structure is implied by the computation.

Currently, we support the first two of these optimizations, with plans for adding the other two.

3 SHAPES

Previous work on type systems for sequence shapes has focused on rectangular shapes [10, 22, 23]. In this work, we are interested in tracking information about the *irregular* shapes that arise in NDP programs. For example, consider the following NESL program that adds two nested sequences:

```
function mm (xss:[[float]], yss:[[float]]) =
  { { x + y : x in xs; y in ys }
  : xs in xss, ys in yss };
```

While `xss` and `yss` are irregular, they must have the same shape. So at the outer level, we infer that `xss` and `yss` have the same number of rows, and at the inner level, we infer that each row of `xss` have the same number of elements as the corresponding row of `yss`.

Consider a slightly different example that adds the sequence `ys` to each row of the sequence of sequences `xss`:

```
function mv (xss: [[float]], ys: [float]) =
  { { x + y : x in xs; y in ys } : xs in xss };
```

In this case, we know that each row of `xss` must have the same length as `ys` and, thus we infer that `xss` must have rectangular shape.

The shape system that we describe below has been designed both to represent these kinds of structural properties and to provide a way to infer them via analysis.

3.1 Representing dimensions

We assume a countable set of dimension variables (α, β, *etc.*) and dimension-function variables (ϕ, ψ, *etc.*). The syntax of dimension expressions is given in Figure 2. In the syntax, we distinguish between *fixed dimensions* d, which specify a single (possibly unknown) size of a sequence, and *varying dimensions* v, which are used to specify the size of nested arrays that may be irregular.

$$
\begin{array}{rcll}
d & ::= & n & \text{known size} \\
 & | & \alpha & \text{dimension variable} \\
 & | & \phi(\alpha) & \text{applied dimension function} \\
 & | & d_1 + d_2 & \text{dimension addition} \\
 & | & \sum_{\alpha=1}^{d} \phi(\alpha) & \text{summation} \\
v & ::= & d & \text{fixed dimension} \\
 & | & \phi & \text{dimension function}
\end{array}
$$

Figure 2: Shapes — fixed and variable dimensions

$$
\begin{array}{rcll}
\widehat{\tau} & ::= & [\widehat{\tau} \,\#\, v] & \text{sequence type} \\
 & | & \widehat{\tau}_1 \times \widehat{\tau}_2 & \text{pair type} \\
 & | & \widehat{\iota} & \text{base types} \\
\widehat{\iota} & ::= & \mathbf{int}(v) & \text{integer type} \\
 & | & \mathbf{bool} \mid \mathbf{float} \mid \cdots & \text{other base types}
\end{array}
$$

Figure 3: Annotated types

We use the term *shape* to describe either a fixed or variable dimension, and *shape variable* to describe both dimension and dimension-function variables.

Shape expressions are organized into three levels based on the quality of information that they provide.

Varying dimensions:	ϕ
Fixed unknown dimensions:	α, $\phi(\alpha)$, $d_1 + d_2$, $\sum_{\alpha=1}^{d} \phi(\alpha)$
Fixed known dimensions:	n

Varying dimensions are the most general, then we have fixed dimensions of unknown size, and the most precise are known fixed dimensions.

3.2 Annotated types

We augment NKL the NKL types (τ) from Section 2.2 with shape information to form *annotated types* ($\widehat{\tau}$), the syntax of which is given in Figure 3. We use the dimensions to annotate types in two ways:

(1) The type $[\widehat{\tau} \,\#\, v]$ is a sequence type annotated with the shape v of the sequence. When the sequence is the outermost sequence in a type expression, v must be a fixed dimension d, but a nested sequences may have a varying dimension represented by a dimension function ϕ.
(2) The type $\mathbf{int}(v)$ represents integers, where v tracks the value of the integer. For sequences of integers, we use dimension functions to represent represent the fact that the value of the integer depends on which element it is.

Here are some examples of annotated types:

$[[\mathbf{float} \,\#\, 5] \,\#\, 5]$	a 5×5 matrix
$[[\mathbf{float} \,\#\, \phi] \,\#\, 5]$	a 5-element sequence of sequences
$\mathbf{int}(17)$	the integer 17
$[\mathbf{int}(\phi) \,\#\, \alpha]$	a sequence of integers

For an annotated type $\widehat{\tau}$, we follow the convention of writing τ for the type with its annotations erased.

When we analyze an iteration over a sequence, we need to ensure that the element type is valid (i.e., that the outermost dimension is not

varying). To help with this requirement, we define the *instantiation* of a dimension with index α, written $v@\alpha$, as follows:

$$\begin{aligned} d@\alpha &= d \\ \phi@\alpha &= \phi(\alpha) \end{aligned}$$

and then define the instantiation of an annotated type with index α as

$$\begin{aligned} [\widehat{\tau} \# v]@\alpha &= [\widehat{\tau} \# v@\alpha] \\ \widehat{\tau_1} \times \widehat{\tau_2}@\alpha &= \widehat{\tau_1}@\alpha \times \widehat{\tau_2}@\alpha \\ \mathbf{int}(v)@\alpha &= \mathbf{int}(v@\alpha) \\ \widehat{\tau}@\alpha &= \widehat{\tau} \quad \text{otherwise} \end{aligned}$$

This operation is needed when stripping off a sequence-type constructor to ensure that the resulting type is well formed.

3.3 Annotated function types

Most functions and operations in NESL will be shape-polymorphic. Furthermore, the result shape of a function may not be determined by the shapes of its arguments. Therefore, we use the following general pattern for specifying the annotated type of a function:

$$\forall \vec{\alpha}, \vec{\phi}.(\widehat{\tau_1}, \ldots, \widehat{\tau_k}) \to \exists \vec{\beta}, \vec{\psi}.\widehat{\tau}$$

Returning to the two example functions (`mm` and `mv`) from the beginning of this section, we give them the following annotated types:

`mm` : $\forall \alpha, \phi.([[\mathbf{float} \# \phi] \# \alpha], [[\mathbf{float} \# \phi] \# \alpha]) \to [[\mathbf{float} \# \phi] \# \alpha]$

`mv` : $\forall \alpha, \beta.([[\mathbf{float} \# \beta] \# \alpha], [\mathbf{float} \# \beta]) \to [[\mathbf{float} \# \beta] \# \alpha]$

Notice that the type of `mm` captures the fact that the arguments and results must all have the same shape, albeit an irregular one. On the other hand, the type of `mv` shows that the first argument must have rectangular shape with the row dimensions equal to the second argument's dimension.

There are two sources of uncertainty about result sizes: conditionals and builtin functions like `filter`. We discuss the types we assign to the builtin functions in Section 4.2.

4 SHAPE ANALYSIS

We present our shape analysis and lifting techniques using the NKL kernel language NKL given in Figure 1.

4.1 Pre-lifting

Before shape analysis, we perform a *pre-lifting* pass over the program. This pass first identifies functions that are invoked inside parallel filter or foreach constructs (or inside other lifted functions) and then adds lifted versions as necessary. For the function definition

$$\mathbf{fun}\ f\ (x_1, \ldots, x_k) = e$$

where f has been identified as being used in a parallel context, we add the lifted definition

$$\mathbf{fun}\ f^{\uparrow}\ (xs_1, \ldots, xs_k) = \{\, e : x_1\ \mathbf{in}\ xs_1, \ldots . x_k\ \mathbf{in}\ xs_k \,\}$$

to the program. The reason for applying the pre-lifting transformation prior to shape analysis is that lifting the annotated function types is fairly complicated, whereas doing shape analysis on a pre-lifted function is no different than analysis on any other function.

$$\begin{aligned}
+_{\mathbf{int}} &: \forall \alpha, \beta.(\mathbf{int}(\alpha), \mathbf{int}(\beta)) \to \mathbf{int}(\alpha + \beta) \\
*_{\mathbf{int}} &: \forall \alpha, \beta.(\mathbf{int}(\alpha), \mathbf{int}(\beta)) \to \exists \gamma.\mathbf{int}(\gamma) \\
\texttt{length} &: \forall \alpha.([\widehat{\tau} \# \alpha]) \to \mathbf{int}(\alpha) \\
\texttt{lengths} &: \forall \alpha, \phi.([[\widehat{\tau} \# \phi] \# \alpha]) \to [\mathbf{int}(\phi) \# \alpha] \\
\texttt{iota} &: \forall \alpha.(\mathbf{int}(\alpha)) \to \exists \phi.[\mathbf{int}(\phi) \# \alpha] \\
\texttt{!} &: \forall \alpha, \beta.([\widehat{\tau} \# \alpha], \mathbf{int}(\beta)) \to \widehat{\tau} \\
\texttt{++} &: \forall \alpha, \beta.([\widehat{\tau} \# \alpha], [\widehat{\tau} \# \beta]) \to [\widehat{\tau} \# \alpha + \beta] \\
\texttt{concat} &: \forall \alpha, \phi.([[\widehat{\tau} \# \phi] \# \alpha]) \to \exists \psi.[\widehat{\tau} \# \sum_{\beta=1}^{\alpha} \psi(\beta)] \\
\texttt{sum}_{\mathbf{float}} &: \forall \alpha.([\mathbf{float} \# \alpha]) \to \mathbf{float} \\
\texttt{filter} &: \forall \alpha.([\widehat{\tau} \# \alpha], [\mathbf{bool} \# \alpha]) \to \exists \beta.[\widehat{\tau} \# \beta]
\end{aligned}$$

Figure 4: Annotated types for builtins

Identifying which functions will require lifting is somewhat similar to the problem of identifying maximal sequential subexpressions in vectorization avoidance [11] and could be performed at the same time.

4.2 Shape types for library functions

A key source of shape information comes from the application of primitive operations and library functions. Figure 4 presents a sample of these. The integer addition operator ($+_{\mathbf{int}}$) has a special type that tracks the value; other integer operators, such as multiplication return a result with unknown value. The `length` function returns an integer that is equal to the dimension of the argument sequence; likewise, the `lengths` function tracks the original subsequence lengths. The type of the `iota` function captures the fact that the length of the result is determined by the value of the argument. For example, we can reason that `iota (length xs)` will have the same length as `xs`. The types of `++` and `concat` reflect the fact that the length of their results will be the sum of the lengths of their arguments.

4.3 Shape analysis

Shape analysis can be viewed as a type inference problem. In our implementation, we assign annotated types to variables and functions, and also generate a set C of equality constraints. We track three kinds of constraints: equality of dimensions ($d_1 = d_2$), equality of dimension functions ($\phi = \psi$), and equality between constant functions and dimensions ($\phi = d$).

Equality constraints are induced by setting two annotated types that have the same underlying structure equal.

$$\begin{aligned}
\mathbb{C}[\![\, [\widehat{\tau_1} \# v_1] = [\widehat{\tau_2} \# v_2] \,]\!] &= \mathbb{C}[\![\, \widehat{\tau_1} = \widehat{\tau_2} \,]\!] \cup \{v_1 = v_2\} \\
\mathbb{C}[\![\, \widehat{\tau_{11}} \times \widehat{\tau_{12}} = \widehat{\tau_{21}} \times \widehat{\tau_{22}} \,]\!] &= \mathbb{C}[\![\, \widehat{\tau_{11}} = \widehat{\tau_{21}} \,]\!] \cup \mathbb{C}[\![\, \widehat{\tau_{12}} = \widehat{\tau_{22}} \,]\!] \\
\mathbb{C}[\![\, \mathbf{int}(v_1) = \mathbf{int}(v_2) \,]\!] &= \{v_1 = v_2\} \\
\mathbb{C}[\![\, \iota = \iota \,]\!] &= \emptyset
\end{aligned}$$

In the case where we get a constraint of the form $\phi = d$ or $\phi(\alpha) = d$ (where d is not indexed by α), then we mark ϕ as being a *constant* function.

We currently do not do any arithmetic reasoning to determine if two dimensions are equal. Instead, we have found that it is sufficient to put the dimension expressions into canonical form and compare them for symbolic equality. We do check, however, for obvious inconsistencies when adding constraints to the set (*e.g.*, a dimension of 5 cannot be equal to a dimension of 3).

A key assumption that we make for the analysis is that the program is correct with respect to the dimensions of arrays (*i.e.*, it does not have out-of-bounds or unequal-length runtime errors). Because such errors will terminate the NESL program's execution (either with a runtime error or with a crash when using the "unsafe" compilation mode), being conservative about dimension equality constraints does not seem beneficial. As we discuss in Section 6, we might be able to use the shape analysis to identify where to place bounds checks.

The shape analysis proceeds by processing each function definition in turn. We initially annotate the types of the function by assigning fresh shape variables. Over the course of analyzing the function body, constraints will be placed on these variables. When analysis of the function is complete, we resolve the variables to their canonical representation and then close over the free shape variables to produce the function's annotated type.

In the remainder of this section, we describe how our shape analysis handles several of the NKL constructs.

4.3.1 Conditional. For conditionals, we compute an annotated type for each of the arms of the conditional. If $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are the annotated types of the conditional's arms, then we know that $\tau_1 = \tau_2$, since we start with a well-typed program. Thus, the annotated types $\widehat{\tau}_1$ and $\widehat{\tau}_2$ can only differ in their shapes. Given two shapes υ_1 and υ_2 that appear in corresponding positions of $\widehat{\tau}_1$ and $\widehat{\tau}_2$, we compute a new shape as follows:

- If $C \vdash \upsilon_1 = \upsilon_2$, where C is the constraint set, then we use υ_1.
- If $\upsilon_1 = \phi_1(\alpha)$ and $\upsilon_2 = \phi_2(\alpha)$, then we use $\psi(\alpha)$, where ψ is fresh. Note that since υ_1 and υ_2 share the same index (α), we preserve the iteration structure in the result.
- If $\upsilon_1 = \alpha_1, \upsilon_2 = \alpha_2$, and these are not nested inside a sequence, then we use a fresh dimension variable β as the shape.
- Otherwise, we use a fresh dimension-function variable ψ as the shape.

4.3.2 Function application. Consider the function application $f(v_1, \ldots, v_k)$, where f has the annotated type

$$\forall \vec{\alpha}, \vec{\phi}.(\widehat{\tau_1}, \ldots, \widehat{\tau_k}) \to \exists \vec{\beta}, \vec{\phi}.\widehat{\tau}$$

We instantiate the function's type by replacing the bound variables with fresh shape variables to get the type

$$(\widehat{\tau'_1}, \ldots, \widehat{\tau'_k}) \to \widehat{\tau'}$$

Then, for each argument v_i with type $\widehat{\tau''_i}$. we add the constraints induced by $\mathbb{C}[\![\, \widehat{\tau'_i} = \widehat{\tau''_i} \,]\!]$ to the set of constraints. We use a similar approach to handling primitive-operator applications.

4.3.3 Parallel map. A parallel map induces equality constraints on the outermost dimension of the argument arrays. For example, the outermost dimensions of `xs` and `ys` must be equal in the following expression:

```
{ x * y : x in xs; y in ys }
```

Furthermore, the outermost dimension of the result will also be equal to the other dimensions. It is important to note that in this situation we are only imposing equality constraints on the outermost dimension, since the shapes and types of the elements do not have to match.

The one subtlety with analyzing parallel maps is reconstructing nested structure without losing shape information. Consider the expression

```
{ { x * x : x in xs } : xs in xss }
```

where `xss` has the annotated type $[[\textbf{float} \# \phi] \# \alpha]$. When we analyze the outer parallel map, we need to assign a valid annotated type to `xs`. We use $[\textbf{float} \# \phi]@\beta = [\textbf{float} \# \phi(\beta)]$ as the type of `xs`, where β is a fresh variable that represents the iteration index of the map. In this example, the result type of the inner map will be $[\textbf{float} \# \phi(\beta)]$, we then generalize the shape $\phi(\beta)$ to ϕ and get the type $[[\textbf{float} \# \phi] \# \alpha]$ for the whole expression.

4.3.4 Detecting rectangular structure. We conclude the discussion of shape analysis by revisiting the `mv` example from Section 3.

```
function mv (xss: [[float]], ys: [float]) =
  { { x + y : x in xs; y in ys } : xs in xss };
```

Shape analysis will begin by assigning the types of the parameters:

$$\texttt{xss} : [[\textbf{float} \# \phi] \# \alpha]$$
$$\texttt{ys} : [\textbf{float} \# \beta]$$

When processing the outer parallel map, we give `xs` the type

$$\texttt{xs} : [\textbf{float} \# \phi(\gamma)]$$

Processing the inner map, requires setting the (outermost) dimensions of `xs` and `ys` to be equal — $\{\phi(\gamma) = \beta\}$ — which means that ϕ must be a constant function. We end up with a result type of $[\textbf{float} \# \beta]$ for the inner map and $[[\textbf{float} \# \phi(\gamma)] \# \alpha]$ for the outer map, and thus, `mv` has the type we predicted in Section 3.3.

$$\begin{array}{rcll}
\overline{\tau} & ::= & \texttt{[:}\iota\texttt{:]} & \text{base-sequence type} \\
& | & \overline{\tau_1} \times \overline{\tau_2} & \text{pair type} \\
& | & \iota & \text{base type} \\
& | & \sigma & \text{shape descriptor type} \\
\sigma & ::= & \textbf{sd}(d * \upsilon) & \text{segment descriptor} \\
& | & \textbf{sz}(d) & \text{base-sequence size}
\end{array}$$

Figure 5: Flattened types

5 FLATTENING WITH SHAPES

Once we have annotated the NKL program with shape information, the next step is to flatten the program into the FDP representation. The syntax of the FDP language, called FKL, is a subset of NKL, with the parallel map constructs removed. The types for FKL are different, however, and are shown in Figure 5. Notice that sequences in FKL may only contain base types and that we have types for segment descriptors and sequence sizes. The segment-descriptor

type $\mathbf{sd}(d * v)$ describes a segment descriptor with d segments, where v defines the size of the segments, while the base-sequence size type $\mathbf{sz}(d)$ specifies the size of an associated base sequence as being d. We do not transfer all of the shape information from the annotated types to the flattened types — specifically we no longer track integer values — but by including the shape information in the segment descriptors, we can determine when two segment descriptors are the same.

5.1 Translating types

The first part of the flattening transformation is the conversion from types annotated with shape information to FKL types. This translation is defined as follows:

$$\begin{aligned}
\mathbb{T}[\![\, [\iota \,\#\, v] \,]\!]\, d &= \mathbf{sz}(d) \times \texttt{[:}\mathbb{T}[\![\, \iota \,]\!]\, d \texttt{:]} \\
\mathbb{T}[\![\, [[\widehat{\tau} \,\#\, v] \,\#\, d'] \,]\!]\, d &= \mathbf{sd}(d * v) \times \mathbb{T}[\![\, [\widehat{\tau} \,\#\, \lfloor\, v \,\rfloor] \,]\!]\, (d' \circledast d) \\
\mathbb{T}[\![\, [\widehat{\tau_1} \times \widehat{\tau_2} \,\#\, d'] \,]\!]\, d &= \mathbb{T}[\![\, [\widehat{\tau_1} \,\#\, d'] \,]\!]\, d \times \mathbb{T}[\![\, [\widehat{\tau_2} \,\#\, d'] \,]\!]\, d \\
\mathbb{T}[\![\, \widehat{\tau_1} \times \widehat{\tau_2} \,]\!]\, d &= \mathbb{T}[\![\, \widehat{\tau_1} \,]\!]\, d \times \mathbb{T}[\![\, \widehat{\tau_2} \,]\!]\, d \\
\mathbb{T}[\![\, \mathbf{int}(v) \,]\!]\, d &= \mathbf{int} \\
\mathbb{T}[\![\, \iota \,]\!]\, d &= \iota \quad \text{otherwise}
\end{aligned}$$

where the second argument (d) to $\mathcal{T}$ is the cumulative size of the array. We use an initial value of 1 when translating an annotated type and we use the dimension multiplication operator "$\circledast$", which is defined as

$$\begin{aligned}
n \circledast m &= n * m \\
d \circledast d' &= \alpha \quad \text{where } \alpha \text{ is fresh}
\end{aligned}$$

This allows us to statically determine the actual size of the data sequence in the case where we have a rectangular array of known dimensions. We also use the notation $\lfloor\, d \,\rfloor$, which is defined as follows:

$$\begin{aligned}
\lfloor\, d \,\rfloor &= d \\
\lfloor\, \phi \,\rfloor &= \alpha \quad \text{where } \alpha \text{ is fresh}
\end{aligned}$$

This notation is used when stripping off a level of sequence so that the resulting type is well formed.

Here are some examples of the type translation:

$$\begin{aligned}
\mathbb{T}[\![\, [[\mathbf{float} \,\#\, 5] \,\#\, 5] \,]\!]\, 1 &= \mathbf{sd}(5 * 5) \times \mathbf{sz}(25) \times \texttt{[:}\mathbf{float}\texttt{:]} \\
\mathbb{T}[\![\, [[\mathbf{float} \,\#\, \phi] \,\#\, 5] \,]\!]\, 1 &= \mathbf{sd}(5 * \phi) \times \mathbf{sz}(\alpha) \times \texttt{[:}\mathbf{float}\texttt{:]} \\
\mathbb{T}[\![\, [\mathbf{int}(\phi) \,\#\, \alpha] \,]\!]\, 1 &= \mathbf{sz}(\alpha) \times \texttt{[:}\mathbf{int}\texttt{:]} \\
\mathbb{T}[\![\, [\mathbf{bool} \times [\mathbf{float} \,\#\, \phi] \,\#\, \alpha] \,]\!]\, 1 &= \\
&\quad (\mathbf{sz}(\alpha) \times \texttt{[:}\mathbf{bool}\texttt{:]}) \times (\mathbf{sd}(\alpha * \phi) \times \mathbf{sz}(\beta) \times \texttt{[:}\mathbf{float}\texttt{:]})
\end{aligned}$$

5.2 Target operations

In Section 2.3, we described how the $\mathcal{S}$, $\mathcal{F}$, and $\mathcal{P}$ operators are used to eliminate excess levels of lifting. These operations are implemented at the meta-level in our translation. In the case of the $\mathcal{P}$ operator, we use a type-indexed definition.[3]

$$\mathcal{P}_{\widehat{\tau}} : \forall \alpha, \phi.([\widehat{\tau} \,\#\, \alpha], [\mathbf{int}(\phi) \,\#\, \alpha]) \to [[\widehat{\tau} \,\#\, \phi] \,\#\, \alpha]$$

[3] It needs to be type indexed because of the way that we have chosen to represent sequences of pairs.

The definition of these meta operations is as follows:

$$\begin{aligned}
\mathcal{S} &= \lambda\langle sd,\ seq\rangle.sd \\
\mathcal{F} &= \lambda\langle sd,\ seq\rangle.seq \\
\mathcal{P}_{\widehat{\tau_1} \times \widehat{\tau_2}} &= \lambda(\langle a,\ b\rangle,\ sd).\langle \mathcal{P}_{\widehat{\tau_1}}(a, sd),\ \mathcal{P}_{\widehat{\tau_2}}(b, sd)\rangle \\
\mathcal{P}_{[\widehat{\tau} \# v]} &= \lambda(seq, sd).\langle sd,\ seq\rangle
\end{aligned}$$

We also need the primitive $\# : (\sigma) \to \mathbf{int}$ that returns the length of the sequence described by a shape descriptor.

We also introduce several FKL primitives for working with sequences, which are described in Figure 6. The **SPLIT** and **COMBINE** primitives are used to implement conditionals and the **FILTER** primitive is used to implement parallel filters. The **DIST**, **INDEX**, **SDIST**, and **TDIST** primitives are used to generate argument sequences. Lastly, the $\mathbf{SD}_{\sigma}$ primitive creates a segment descriptor of the specified type.

As will be shown below, the **DIST** primitive is used to replicate constants and variables to form sequences that can be used as arguments to lifted operations. In the case where the value being replicated is not a base value, the replication operation is more complicated, since the result must conform to the flattened-type representation described in the previous section. Therefore, we define a type-indexed function $\texttt{dist}_{\widehat{\tau}}$ that expands to the necessary code.

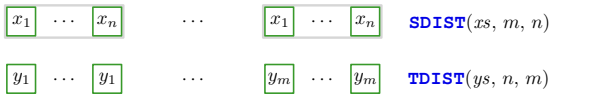
$$\begin{aligned}
\texttt{dist}_{\widehat{\tau_1} \times \widehat{\tau_2}} &= \lambda(\langle a,\ b\rangle, n).\langle \texttt{dist}_{\widehat{\tau_1}}(a, n),\ \texttt{dist}_{\widehat{\tau_2}}(b, n)\rangle \\
\texttt{dist}_{[\widehat{\iota} \# v]} &= \lambda(\langle sd,\ seq\rangle, n). \\
&\qquad \langle n,\ \langle \mathbf{SD}_{\mathbf{sd}(n*v)}(\mathbf{SDIST}(sd,\ n,\ ,\ \#(sd))), \\
&\qquad\quad \mathbf{SDIST}(seq,\ n,\ sd)\rangle\rangle \\
\texttt{dist}_{[\widehat{\tau} \# v]} &= \lambda(\langle sd,\ seq\rangle, n). \\
&\qquad \langle n,\ \langle \mathbf{SD}_{\mathbf{sd}(n*v)}(\mathbf{SDIST}(sd,\ n,\ ,\ \#(sd))), \\
&\qquad\quad \mathbf{SDIST}(\texttt{dist}_{\widehat{\tau}}(seq,\ n),\ n,\ sd)\rangle\rangle \\
\texttt{dist}_{\widehat{\iota}} &= \lambda(x, n).\langle n,\ \mathbf{DIST}(x,\ n)\rangle
\end{aligned}$$

Here the **SDIST** primitive is used to create multiple copies of a sequence and we also use it to build the segment descriptors for the nested sequences.

The **TDIST** primitive is required to support nesting of parallel maps. For example, consider the expression

```
{ { x + y : x in xs } : y in ys }
```

where $xs = [x_1, \ldots, x_n]$ and $ys = [y_1, \ldots, y_m]$. This expression is flattened to a lifted addition ($+^{\uparrow}$) applied to two sequences, where the first argument is the xs sequence replicated m times and the second sequence is the concatenation of n copies of y_1, followed by n copies of y_2, up to m copies of y_m. The **TDIST** primitive is used to create this second sequence, as illustrated in the following diagram:



Note that the **TDIST** primitive could be defined in terms of **DIST** as follows:

$$\mathbf{TDIST}(ys,\ n,\ m) = \mathbf{DIST}^{\uparrow}(ys,\ \mathbf{DIST}(n,\ m))$$

COMBINE	:	$(\text{sz}(d_1) \times \texttt{[:}\iota\texttt{:]}, \text{sz}(d_2) \times \texttt{[:}\iota\texttt{:]}, \text{sz}(d) \times \texttt{[:bool:]}) \rightarrow \texttt{[:}\iota\texttt{:]}$	*merge two sequences according to the boolean flags*
DIST	:	$(\iota, \text{int}) \rightarrow \texttt{[:}\iota\texttt{:]}$	*replicate the first argument by given length*
FILTER	:	$(\texttt{[:}\iota\texttt{:]}, \texttt{[:bool:]}, \text{sz}(d)) \rightarrow \text{sz}(d_1) \times \texttt{[:}\iota\texttt{:]}$	*filter out elements of the first argument where the corresponding element in the second argument is true*
INDEX	:	$(\texttt{[:int:]}, \texttt{[:int:]}, \texttt{[:int:]}, \text{sz}(d)) \rightarrow \text{sz}(d') \times \texttt{[:int:]}$	*given a sequence of start values, strides, and lengths, build a flattened sequence index sequences*
SPLIT	:	$(\texttt{[:}\iota\texttt{:]}, \texttt{[:bool:]}, \text{sz}(d)) \rightarrow (\text{sz}(d_1) \times \texttt{[:}\iota\texttt{:]}) \times (\text{sz}(d_2) \times \texttt{[:}\iota\texttt{:]})$	*split the first argument according to the values in the second argument*
SDIST	:	$(\texttt{[:}\iota\texttt{:]}, \text{int}, \text{int}) \rightarrow \texttt{[:}\iota\texttt{:]}$	*replication of the first argument; the second argument is the number of copies and the third argument is the length of the first argument*
$\textbf{SD}_\sigma$	:	$(\text{int}, \texttt{[:int:]}) \rightarrow \sigma$	*build a segment descriptor from a length and sequence of segment lengths*
TDIST	:	$(\texttt{[:}\iota\texttt{:]}, \text{int}, \text{int}) \rightarrow \texttt{[:}\iota\texttt{:]}$	*transposed replication of the first argument; the second argument is the number of copies and the third argument is the length of the first argument*

Figure 6: Primitive sequence operations for FKL

but it is useful to have it as a primitive. As with `DIST`, we define a type-indexed function $\texttt{tdist}_{\overline{\tau}}$ to handle the cases where the first argument is not a sequence of base values. We also define $\texttt{combine}_{\widehat{\tau}}$ as a type-indexed version of the **COMBINE** primitive.

5.3 Flattening

Our presentation of the flattening transformation roughly follows that of Keller [9], except that we also track shape information. We organize the translation by the syntactic forms found in Figure 1 and define three corresponding flatting translations for terms inside a parallel map.

$\mathbb{E}[\![e : \vec{b}]\!]_{sd}$	flatten expression e
$\mathbb{R}[\![r : \vec{b}]\!]_{sd}$	flatten right-hand-side r
$\mathbb{V}[\![v : \vec{b}]\!]_{sd}$	flatten value v

For each of these translations, $\vec{b}$ is the sequence of bindings for the map and sd is the size/segment descriptor of the result.

Although FKL is a normalized IR, we take the liberty of using direct-style notation for the result of the translation to simplify its specification. In the actual implementation, the result is normalized.

5.3.1 Flattening expressions. For expressions that are inside a parallel map context, we use the translation $\mathbb{E}[\![e : \vec{b}]\!]_{sd}$, where e is the expression being flattened, $\vec{b}$ is the sequence of bindings for the map, and sd is the size/segment descriptor of the result. Note that for the transformation of the conditional, we show the case where there is a single binding "x **in** xs," but this generalized to multiple bindings by simply adding a **SPLIT**/**COMBINE** pair for each additional binding.

$$\mathbb{E}[\![v : \vec{b}]\!]_{sd} = \mathbb{V}[\![v : \vec{b}]\!]_{sd}$$

$$\begin{aligned}
&\mathbb{E}[\![\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2 : \overrightarrow{x_i \textbf{ in } xs_i}]\!]_{sd} = \\
&\quad \textbf{let } flgs = \mathbb{E}[\![p : \overrightarrow{x_i \textbf{ in } xs_i}]\!]_{sd} \\
&\quad \overrightarrow{\textbf{let } \langle ys_i, zs_i \rangle = \texttt{split}_{\text{typeof}(x_i)}(xs_i, flgs, sd)} \\
&\quad \textbf{let } res_1 = \mathbb{E}[\![e_1 : \overrightarrow{x_i \textbf{ in } ys_i}]\!]_{\mathcal{S}(ys_1)} \\
&\quad \textbf{let } res_2 = \mathbb{E}[\![e_2 : \overrightarrow{x_i \textbf{ in } zs_i}]\!]_{\mathcal{S}(zs_2)} \\
&\quad \textbf{in} \\
&\qquad \texttt{combine}_{\text{typeof}(e_1)}(res_1, res_2, flgs)
\end{aligned}$$

$$\begin{aligned}
&\mathbb{E}[\![f\,(v_1, \ldots, v_k) : \vec{b}]\!]_{sd} = \\
&\quad \textbf{if } \#(sd) = 0 \\
&\qquad \textbf{then } \langle sd, [\,] \rangle \\
&\qquad \textbf{else } f^{\uparrow}(\mathbb{V}[\![v_1 : \vec{b}]\!]_{sd}, \ldots, \mathbb{V}[\![v_k : b]\!]_{sd}, sd)
\end{aligned}$$

$$\begin{aligned}
&\mathbb{E}[\![\textbf{let } y = r \textbf{ in } e : \vec{b}]\!]_{sd} = \\
&\quad \textbf{let } ys = \mathbb{R}[\![r : \vec{b}]\!]_{sd} \\
&\quad \textbf{in} \\
&\qquad \mathbb{E}[\![e : y \textbf{ in } ys, \vec{b}]\!]_{sd}
\end{aligned}$$

5.3.2 Flattening right-hand sides. For right-hand sides of let bindings, we use the translation $\mathbb{R}[\![r : \vec{b}]\!]_{sd}$, where r is the right-hand side being flattened, $\vec{b}$ is the sequence of bindings for the iteration, sd is the size/segment descriptor of the result. The flattening

translation of most right-hand-side forms is straightforward:

$$\mathbb{R}[\![\ e\ :\ \vec{b}\ ]\!]_{sd} = \mathbb{E}[\![\ e\ :\ \vec{b}\ ]\!]_{sd}$$

$$\mathbb{R}[\![\ p\,(v_1, \ldots, v_k)\ :\ \vec{b}\ ]\!]_{sd} = p^{\uparrow}\,(sd, \mathbb{V}[\![\ v_1\ :\ \vec{b}\ ]\!]_{sd}, \ldots, \mathbb{V}[\![\ v_k\ :\ \vec{b}\ ]\!]_{sd})$$

$$\mathbb{R}[\![\ \langle v_1, v_2\rangle\ :\ \vec{b}\ ]\!]_{sd} = \langle \mathbb{V}[\![\ v_1\ :\ \vec{b}\ ]\!]_{sd}, \mathbb{V}[\![\ v_2\ :\ \vec{b}\ ]\!]_{sd}\rangle$$

$$\mathbb{R}[\![\ [\,v_1, \ldots, v_k\,]\ :\ \vec{b}\ ]\!]_{sd} = \mathbb{V}[\![\ v_1\ :\ \vec{b}\ ]\!]_{sd}\ {+\!+_{\widehat{\tau}}}\ \cdots\ {+\!+_{\widehat{\tau}}}\ \mathbb{V}[\![\ v_k\ :\ \vec{b}\ ]\!]_{sd}$$
where $\text{typeof}(v_1) = \widehat{\tau}$

The case for flattening nested parallel maps is more complicated. Keller's translation relies on multiple translation passes to handle nested parallel maps, which results in multiple levels of lifting. Fortunately, the following identity

$$f^{\uparrow\uparrow}(\overrightarrow{xss_i}) = \mathcal{P}(f^{\uparrow}(\overrightarrow{\mathcal{F}(xss_i)}), \#(xss_1))$$

allows multiple levels of lifting to be reduced to a single level. In our approach, we lift the outer bindings in anticipation that they might be referred to inside the nested parallel map. Dead-code elimination will remove any unused liftings.

$$\begin{aligned}
&\mathbb{R}[\![\ \{\, e : \overrightarrow{x_i\ \texttt{in}\ xs_i}\,\}\ :\ \overrightarrow{y_j\ \texttt{in}\ ys_j}\ ]\!]_{sd} = \\
&\quad \texttt{let}\ sd' = \mathcal{S}_{\text{typeof}(xs_1)}(xs_1) \\
&\quad \texttt{let}\ \overrightarrow{xs'_i = \mathcal{F}(\mathbb{V}[\![\ xs_i\ :\ \overrightarrow{y_j\ \texttt{in}\ ys_j}\ ]\!]_{sd})} \\
&\quad \texttt{let}\ \overrightarrow{ys'_j = \texttt{tdist}_{\text{typeof}(y_j)}(ys_j, \#(sd'), \#(sd))} \\
&\quad \texttt{let}\ res = \mathbb{E}[\![\ e\ :\ \overrightarrow{x_i\ \texttt{in}\ xs'_i},\ \overrightarrow{y_j\ \texttt{in}\ ys'_j}\ ]\!]_{sd'} \\
&\quad \texttt{in} \\
&\qquad \mathcal{P}_{\text{typeof}(e)}(res, sd)
\end{aligned}$$

5.3.3 Flattening values. For values, we use the translation $\mathbb{V}[\![\ v\ :\ \vec{b}\ ]\!]_{sd}$, where v is the value being flattened, $\vec{b}$ is the sequence of bindings for the iteration, sd is the size/segment descriptor of the result.

$$\mathbb{V}[\![\ n\ :\ \vec{b}\ ]\!]_{sd} = \texttt{DIST}(n, sd)$$

$$\mathbb{V}[\![\ x\ :\ x\ \texttt{in}\ xs,\ \vec{b}\ ]\!]_{sd} = xs$$

$$\mathbb{V}[\![\ x\ :\ y\ \texttt{in}\ ys,\ \vec{b}\ ]\!]_{sd} = \mathbb{V}[\![\ x\ :\ \vec{b}\ ]\!]_{sd}$$

$$\mathbb{V}[\![\ x\ :\ \cdot\ ]\!]_{sd} = \texttt{dist}_{\text{typeof}(x)}(x, sd)$$

For base constants, we can use the **DIST** primitive to lift a single value to a sequence of values. If a variable x is in the binding list, then we can replace it with the corresponding sequence variable; otherwise, we use the type-indexed $\texttt{dist}_{\widehat{\tau}}$ operation to generate the necessary FKL code.

6 STATUS AND FUTURE WORK

We have implemented a prototype of the analysis and shape-preserving flattening transformations for NKL. The next step is to incorporate these phases into the Nessie compiler to produce λ_{cu} code. The main challenge is supporting the large library of builtin functions, many of which will have to be type indexed.

We have not addressed the *replication problem* that some NESL programs suffer from, but we believe that the techniques developed by other researchers to solve this problem are compatible with our approach [14, 15].

Vectorization avoidance is a technique to increase the granularity of primitive operations prior to flattening [11]. This technique identifies maximal subexpressions that are sequential and packages them up as indivisible operations that are not subject to decomposition by the flattening transformation. We have previously implemented this technique in the Nessie compiler and believe that we can combine the pre-lifting analysis from Section 4.1 with the analysis that identifies sequential subexpressions.

One of the major improvements of our new shape analysis is that it can identify regular rectangular arrays. There are some obvious opportunities for exploiting this information, such as using more compact representations of segment descriptors and more efficient segmented scan and reduction operations, but we have not worked out the details of these optimizations yet.

As we mentioned in Section 4.3, we assume that programs do not have out-of-bounds or unequal-length runtime errors. Consider the following contrived NESL code:

```
let xs = { x : x in xs | x < 0 };
    ys = { y : y in ys | y < 0 };
in
  { x + y : x in xs, y in ys }
```

For this program to run correctly, `xs` and `ys` must have equal length, so our analysis will make that assumption, but there is no evidence that they have the same length. In general, we believe that when equality constraints are introduced that constrain existentially quantified variables, then we are adding a constraint that potentially could be violated at runtime (but the program would fail). We can perhaps use an approach of tracking existential variables and then introducing dimension checks where they must be equal to some other dimension. We may be able to use some of the ideas from Futhark for this purpose [8].

Once we have sorted out the issue of bounds checking, it should be possible to prove the correctness of the shape analysis, but this is left for future work.

7 RELATED WORK

The flattening transformation dates back to Blelloch's Ph.D. research [3, 7], but Blelloch only described the transformation using informal English-language prose. Keller developed a more rigorous description [9, 12] in her dissertation, which we loosely follow here. Our approach differs from Keller in that we handle nested parallel maps directly, whereas Keller transforms the inner maps first, which results in multiple passes. Keller's transformation also extends the NESL model with support for inductive types, which we have not done. Leshchinskiy developed a formal tranformation for Data-Parallel Haskell (DPH) [13]. DPH not only supports inductive types, but also supports higher-order functions. He uses an approach of representing functions as pairs of closures; one for the original function and one for the lifted version of the function. None of these flattening transformations, however, tracks the shapes of irregular nested computations.

Analyzing the shapes of arrays has been an important research problem in the area of array-language compilers, but all of the work that we are aware of has focused on rectangular arrays. Scholz has

developed a type system for shapes in the functional array language Single-Assignment C (SAC) [22]. His system supports a limited form of dimension polymorphism, which is not part of our system, but it is limited to rectangular array shapes. A related idea, also for SAC, allows users to define constraints on the shapes of arrays that have polymorphic structure as a way to provide more information about the structure of shape-polymorphic code [23]. Their constraints are similar to the constraints that we induce during shape analysis. The Repa library for GHC uses Haskell's powerful type system to define shape polymorphic code [10]. The Futhark language uses a hybrid approach that combines static analysis of array shapes with a dynamic fallback mechanism that can check situations that were not possible to resolve statically [8]. They also use a novel program slicing method to extract code for computing array shapes at runtime. Like the other works listed, Futhark does not support irregular nested arrays.

8 CONCLUSION

We have described a new approach to the problem of determining shape information for the FDP code produced by the flattening transformation. We have prototyped this approach for a kernel language NKL and we are preparing to integrate it into Nessie, our NESL to CUDA compiler.

ACKNOWLEDGMENTS

Portions of this research were supported by the National Science Foundation under award CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

REFERENCES

[1] Lars Bergstrom and John Reppy. 2012. Nested Data-Parallelism on the GPU. In *ICFP '12*. ACM, New York, NY, 247–258.
[2] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *IEEE Computer* 38, 11 (Nov. 1989), 1526–1538.
[3] Guy E. Blelloch. 1990. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA.
[4] Guy E. Blelloch. 1995. *NESL: A nested data-parallel language (version 3.1)*. Technical Report CMU-CS-95-170. School of C.S., CMU, Pittsburgh, PA.
[5] Guy E. Blelloch. 1996. Programming parallel algorithms. *CACM* 39, 3 (March 1996), 85–97.
[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. 1994. Implementation of a portable nested data-parallel language. *JPDC* 21, 1 (1994), 4–14.
[7] Guy E. Blelloch and Gary W. Sabot. 1990. Compiling collection-oriented languages onto massively parallel computers. *JPDC* 8, 2 (1990), 119–134.
[8] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2014. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In *FHPC '14*. ACM, New York, NY, 31–42.
[9] Gabriele Keller. 1999. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. Ph.D. Dissertation. Technische Universität Berlin, Berlin, Germany.
[10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP '10*. ACM, New York, NY, 261–272. https://doi.org/10.1145/1863543.1863582
[11] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. 2012. Vectorisation Avoidance. In *HASKELL '12*. ACM, New York, NY, 37–48.
[12] Gabriele Keller and Martin Simons. 1996. A Calculational Approach to Flattening Nested Data Parallelism in Functional Languages. In *Concurrency and Parallelism, Programming, Networking, and Security (LNCS)*, Joxan Jaffar and Roland H. C. Yap (Eds.), Vol. 1179. Springer-Verlag, New York, NY, 234–243.
[13] Roman Leshchinskiy. 2005. *Higher-Order Nested Data Parallelism: Semantics and Implementation*. Ph.D. Dissertation. Technische Universität Berlin, Berlin, Germany.
[14] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. 2012. Work Efficient Higher-order Vectorisation. In *ICFP '12*. ACM, New York, NY, 259–270.
[15] Frederik M. Madsen. 2012. Flattening Nested Data Parallelism. Master's Project, DIKU. Available from http://hiperfit.dk/publications.
[16] Jan F. Prins and Daniel W. Palmer. 1993. Transforming High-Level Data-Parallel Programs into Vector Operations. In *PPoPP '93*. ACM, New York, NY, 119–128.
[17] John Reppy and Nora Sandler. 2015. Nessie: A NESL to CUDA Compiler. Presented at CPC 2015; London, UK, 13 pages. Available from https://nessie.cs.uchicago.edu.
[18] John Reppy and Joe Wingerter. 2016. λ_{cu} — An Intermediate Representation for Compiling Nested Data Parallelism. Presented at CPC 2016; Valladolid, Spain, 13 pages. Available from https://cpc2016.infor.uva.es.
[19] Amos Robinson, Ben Lippmeier, and Gabriele Keller. 2014. Fusing Filters with Integer Linear Programming. In *FHPC '14*. ACM, New York, NY, 53–62.
[20] Nora Sandler. 2014. Nessie: A New NESL Compiler. (June 2014). BA Honors Thesis, Department of Computer Science, University of Chicago.
[21] Scandal Project. [n. d.]. A library of parallel algorithms written in NESL. Available from http://www.cs.cmu.edu/~scandal/nesl/algorithms.html.
[22] Sven-Bodo Scholz. 2001. A Type System for Inferring Array Shapes. In *IFL '01 (LNCS)*, Thomas Arts and Markus Mohnen (Eds.). Springer-Verlag, New York, NY, 65–82.
[23] Fangyong Tang and Clemens Grelck. 2013. User-Defined Shape Constraints in SAC. Presented at IFL 2012; Oxford U.K., 19 pages. Available from www.sac-home.org.
[24] Joe Wingerter. 2017. λ_{cu} *— An Intermediate Representation for Compiling Nested Data Parallelism*. Master's thesis. University of Chicago.
[25] Yongpeng Zhang and Frank Mueller. 2012. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *ICPP '12*. IEEE Computer Society Press, Los Alamitos, CA, 340–349.

FatFast: traversing Fat Branches Fast in Haskell

Juan Carlos Sáenz-Carrasco
University of Sheffield, UK
jcsaenzcarrasco1@sheffield.ac.uk

Mike Stannett
University of Sheffield, UK
m.stannett@sheffield.ac.uk

ABSTRACT

Over a decade ago, the *finger tree* devised by Hinze and Paterson has been considered the default Haskell *sequence* data structure for its great performance ($O(\log n)$ amortised) in both searching and updating the elements for such as container. However, in the lack of an index for random access, one needs to look for an additional structure which provides such as reference. We present FatFast, a structure that offers the best of two worlds, a binary search trees *BST* for fast lookups embedded within an specialised version of that of *finger tree*s for the sequence management. This work is an extension of the FunSeqSet structure, a work submitted to WFLP2019.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Software and its engineering → Functional languages**.

KEYWORDS

data structures, functional programming, Haskell

ACM Reference Format:
Juan Carlos Sáenz-Carrasco and Mike Stannett. 2020. FatFast: traversing Fat Branches Fast in Haskell. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Amongst the purely data structures we can classify the way the data is stored in two main designs: *dictionaries* or *sets* and *sequences*. For the former, a relationship is needed in order to keep data *sorted*. This is useful when speed in searching is key. On the other hand, the latter structures take care for data being *ordered* , that is, the line up of the elements matters. Okasaki in [6] nicely describes a large compendium of the aforementioned structures specifically for the purely functional programming language Haskell. In this paper we consider the *default* structures in the online package repository Hackage: `Data.Set` and `Data.Sequence` which are available at http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Set.html and http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Sequence.html respectively. We need to clarify, however, that `Data.Sequence` is actually a specialisation of that in `Data.FingerTree` which corresponds to the original paper published by Hinze and Paterson in [1] and available at http://hackage.haskell.org/package/fingertree-0.1.4.2/docs/Data-FingerTree.html. In the remaining of the document we refer to the former as `Set`, `Seq` for the second, and to the latter simply as `FT`.

Before describing `FT` in detail, we must note that the notation used in this paper differs slightly from the one used by Hinze and Paterson, as some of the assumptions we make over such structure. The essential ideas, however, are the same. Discussion on the differences is presented alongside the corresponding sections. A `Set` is either a `Tip` (empty-valued) or a node (e.g. `Bin`) alogside its size (e.g. `Size`), its value of any type (e.g. `a`) and two subtrees (recursive definition of `Set`).

```
data Set a = Bin Size a (Set a) (Set a)
           | Tip
```

For the FT data type we start defining the inner elements and assign visual representations to facilitate further explanations in following sections. There are two type parameters that define a FT, the one dedicated to *measure* the tree and the one defining the type of the sequence it holds on the leaves. The former, lets call it **v**, is constrained to be an instance of **Monoid**, that is, a type able to perform a binary operation with a special value acting as the identity for such operation. The latter, we call it `a`, is polymorphic. We commence defining a `Node`.

```
data Node v a = Node2  v  a  a
              | Node3  v  a  a
```

We represent `Node` with a blue circle with a "N2" or a "N3" for the cases of `Node2` and `Node3` respectively. The following auxiliary type for a FT is `Digit` which represent the prefix and suffix of a FT.

```
data Digit a = One    a
             | Two    a  a
             | Three  a  a  a
             | Four   a  a  a  a
```

We represent the `Digit` type with a white dialogue box and a Roman numeral for each data constructor. Finally, the data type for a finger tree FT, represented pictorially with a crossed diamond when FT is empty and filled in blue otherwise.

```
data FingerTree v a
  = Empty
  | Single  a
  | Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)
```

The height of a FT is increased `Node`-wise through the recursive call. Notice that also the affixes are increased in this way.

Example

The following example in Figure 1 denotes the sequence of pairs of integers which its monoidal annotation or **v** type is the monoid $(\mathbb{N}, +, 0)$, in other words, a sequence of type `Seq (Int,Int)` from

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

`Data.Sequence` library. The sequence is `[(7,7),(7,9),(9,9),(9,5),(5,5),(5,9),(9,2),(2,2),(2,9),(9,4),(4,4),(4,9),(9,3),(3,3),(3,9),(9,7)]`. Note that **v** type is denoted with a yellow rounded square next to `Node` and `FingerTree` types.

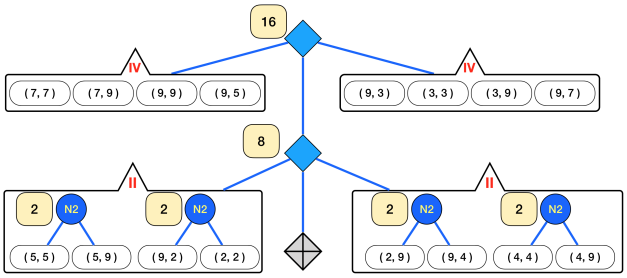


Figure 1: a FT with 16 elements

The following tables show the operators we shall use in the remainder of the paper for both structures `Set` and `FT`

Table 1: Performance of basic FT operators

Function	Description	Complexity
`viewl`	view the first element	$O(1)$
◁	inserting from the left	$O(1)$
▷	inserting from the right	$O(1)$
⋈	appending two trees	$O(\log(min(n_1, n_2)))$
`search`	looking for an element	$O(\log n)$

Table 2: Performance of basic SET operators

Function	Description	Complexity
`member`	testing membership	$O(\log n)$
`insert`	addition of new element	$O(\log n)$
`union`	disjoint set union	$O(m \times (\log((n/m) + 1)))$

When looking for an element, say `(4,4)` in the FT in Figure 1 we face two needs. The first need is discharged when we are provided its index, `10` (based 0), for which we simply use the *size* (performed by the monoidal annotation) of FT as part of the predicate when applying `search` function (i.e. last row in Table 1). The latter need arises when such index is *not* provided. Well, `Data.Seq` offers at least two functions to look for an element rather than the index, `elemIndex` (actually `elemIndexL` and `elemIndexR`), but this function takes linear time since it traverses the entire structure. Can we do it better ? We attempt to answer this question in the following sections.

1.1 Contributions

The main contributions of this paper are:

(1) adjusting the current `FT` datatype to reduce *fat*, that is, intentionally not to perform the monoidal operation where values may be repeated
(2) enhancing the current methods for inserting, appending and searching within the `FT` structure
(3) a comparative evaluation of the general case and our proposal, reducing fat by fasting.

1.2 Structure of this paper

This paper is organised as follows. Section 2 describes the type constructors where reductions of accumulation for the sets takes place. By avoiding monoidal accumulation in central annotations we redefine the `FT` as `FatFast` in Section 3. Section 4 redefines the current functions in `FT` in order to allow a query to perform efficiently without the accumulation factor. We provide a comparison between a the current `FT` and the non accumulative `FT` (i.e. reduced fat) for the same sequence in Section 5. Finally, we conclude with further work and remarks in Section 6.

2 ACCUMULATORS AND SEMI-ACCUMULATORS

Recalling the datatype from `FT`, we distinguish two places where the monoidal annotation is defined.

```
data FingerTree v a
...
  | Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)
```

Let us call the first and second v's the *fat*-factor or *global* accumulator since it is stored in the tree-spine. We call the third `v` the *local* accumulator which is stored in the (left and right) subtrees. The red arrows in Figure 2 point to global accumulators while the green arrows point to the *semi*-accumulators, next to the `N2` (i.e., `Node2`) and `N3` (i.e., `Node3`) data constructors.

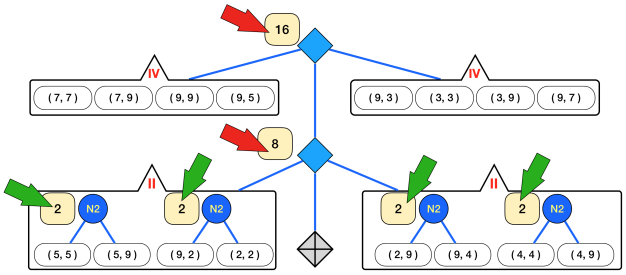


Figure 2: a FT with 16 elements, and arrows pointing to the accumulators (fat and semi)

In order to show the discrepancies of performance from the original paper by Hinze and Paterson and the `FatFast`, we firstly bring up the main operators altogether their bounds. In each case, n gives the number of vertices in the first (or only) tree operated upon; for those functions taking two trees as input, m is the number of vertices in the second tree. The result for ⋈ assumes that $m \leq n$ (if not, we can swap the order of the arguments before applying ⋈). This is shown in Table 1.

Take for instance the operator ◁, which takes an element `e` and a tree `t` and returns a new tree `t'`. This operation assumes that the monoidal operation is always performed in $O(1)$. This is the case when the applications provided in [1] meet the requirement, such

as random access with the arithmetic addition operator (+) or the priority queues with the strict functions `max` and `min`. When we introduce the `union` operator to deal with sets working as searching engines, we need to take into consideration the bounds in Table 2. Now, performing $\triangleleft$ takes $O(\log n)$ rather than constant and $\bowtie$ operator takes $O(m \times \log n)$ instead of logarithmic, but what is the worst case ?

Let us now show the same example from Figure 1 but this time our monoidal annotation is $(Set(\mathbb{N}, \mathbb{N}), \cup, \emptyset)$,

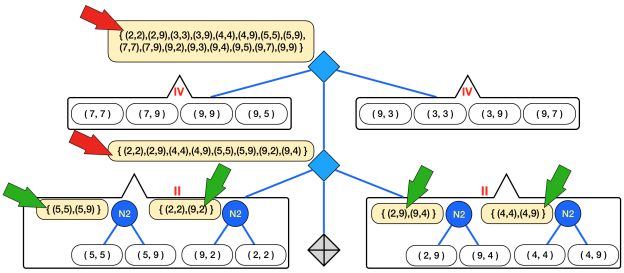


Figure 3: a FT with 16 elements, where its monoidal annotation is a *Set*

Following this approach, our runtime bound is tight to that of the $\cup$ (set-union) operation since it is performed any time an annotation is involved. The following shows a sample of the number of $\cup$-operations when looking just the half of a tree.

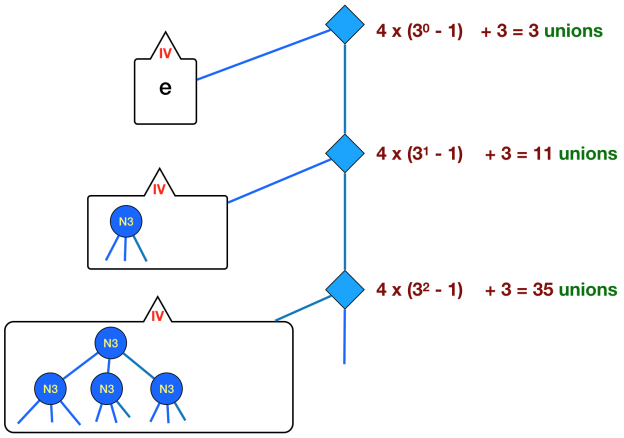


Figure 4: number of $\cup$-operations in half of a FT

That is, the number of $\cup$-operations per affix is around $(4 \times (3^h - 1) + 3$. To the double of the above we also need to sum up the spine union, i.e. the fat accumulator.

3 FASTING AT SPINAL MONOIDAL ANNOTATION

Clearly, the larger sets are located at the FT spine since they incorporate, i.e. join, the sets coming up from the prefix, suffix and middle parts. This is the piece we shall intentionally omit. In order to do so, we have at least two options, removing such type constructor from the FT datatype definition or simply avoid the computation for *measuring* such as annotation. The term *measuring* refers either the type class `Measured` or its function `measure`, which according to Hinze and Paterson is the mechanism to perform efficient operations (i.e. internal nodes) over a sequence (i.e. the leaf nodes).

The monoidal annotations grow for every insertion derived from either $\triangleleft$, $\triangleright$ or $\bowtie$. We shall show the changes in $\triangleleft$ function definition in this paper, but the remaining operations follow easily the same idea. From the original definition

```
(<|) :: (Measured v a) => a -> FingerTree v a
     -> FingerTree v a
a <| Empty          =  Single a
a <| Single b       =  deep (One a) Empty (One b)
a <| Deep v (Four b c d e) m sf
   = m `seq` Deep (measure a `mappend` v) (Two a b)
              (node3 c d e <| m) sf
a <| Deep v pr m sf
   = Deep (measure a `mappend` v) (consDigit a pr) m sf
```

We substitute `measure` next to `Deep` in both cases with the empty set, which is the identity element in the monoid and defined in Haskell with `mempty` function. The substitutions look like

```
Deep mempty (Two a b) (node3 c d e <| m) sf
```

and

```
Deep mempty (consDigit a pr) m sf
```

We call to this 'new' `FT` as `FatFast`, which is illustrated in Figure 5

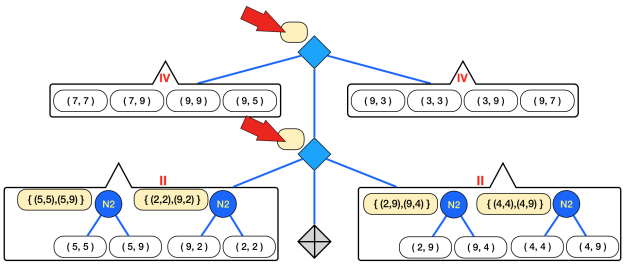


Figure 5: a 'thinner' FT with 16 elements, showing empty sets (red arrows) respect to those in Figure 3

4 A NEW SEARCH FOR SEMI-ACCUMULATOR TREES

The essence of the update operations redefined in previous Section remains the same as the original functions in `FT`. What is affected at the end is the query operation `search` (which includes a splitting) since the arguments on which this function feeds on are now reduced. Performing *the look up* at this stage may lead to inconsistent results.

To manage a query in `FatFast` we shall follow the top-down approach as the original. Unlike the current `search`, we evaluate both extremes of the tree and then we go down the spine through the middle component. The predicate analysis is the same as before, that is, the condition for an accurate search needs the predicate to

be monotonic. The central function from FT is searchTree which carries on the search assuming the element to look for *is in* the structure. We lost the chance for such as assumption since the set on the top of FatFast is now empty. Therefore, we need to look for the failure and success for the element in focus. Where the Split datatype is defined in the original search, we extend such a definition to include the failure cases, that is, from

```
data Split t a = Split t a t
```

into

```
data Built t a = NoBuilt | Built t a t
```

Our search within trees is defined as

```
searchT :: (Measured v a) => (v -> v -> Bool)
         -> FingerTree v a -> Built (FingerTree v a) a
searchT _ Empty      = NoBuilt
searchT p (Single x)
   | p (measure x) mempty = Built Empty x Empty
   | otherwise            = NoBuilt
searchT p (Deep _ pr m sf)
  | p vl mempty
    =  let  Split l x r     =  searchD p pr
       in  Built (maybe Empty digitToTree l) x (deepL r m sf)
  | p vr mempty
    =  let  Split l x r     =  searchD p sf
       in  Built (deepR pr  m  l) x (maybe Empty digitToTree r)
  | otherwise
    =  case (searchT p m) of
         NoBuilt        -> NoBuilt
         Built ml xs mr ->
            let  Split l x r     =  searchN p xs
            in   Built (deepR pr ml l) x (deepL r mr sf)
  where
    vl  = measure pr
    vr  = measure sf
```

5 EMPIRICAL RESULTS

This section presents experiments to evaluate how much running time costs in terms of performance. The experiments will show that, in practice, FatFast is faster as expected in the theoretical analysis (Section 2). We plot four implementations, all defining tuple of integers at the leaves and just one implementation with no sets as monoidal annotations, called Seq similar to that in Data.Sequence. The Full implementation is for the case when a FT contains sets in ever monoidal annotation. We call Semi to the FT containing sets only in the affixes (as in Figure5) and Top to the FT containing sets in the affixes and in the top of the FT.

5.1 Experimental Setup

Functions ◁ and ⋈ were tested by the authors in Haskell and compiled with ghc version 8.0.1 with optimisation -O2. The experiments were performed on a 2.2 GHz Intel Core i7 MacBook Pro with 16 GB 1600 MHz DDR3 running macOS High Sierra version 10.13.1 (17B1003). We imported the following libraries into our code from the online package repository Hackage: [2] code for finger trees, [4] for conventional sets .

The running time of a given computation was determined by the mean of three executions. In Figure 6 we insert trees into a forest of the same type of FT.

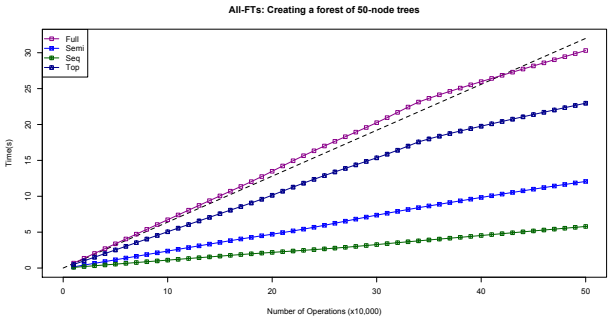


Figure 6: Generation of trees within a forest for all implementations

We plot queries for the k^{th} element when it is at position $n/2$, $n\times(3/4)$, $n+1$ (i.e. not in the forest) and $n/2$ in Figure 7 and Figure 8

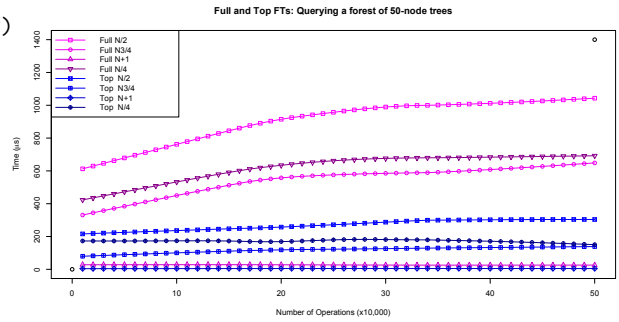


Figure 7: Looking for different tuples in Full and Top implementations

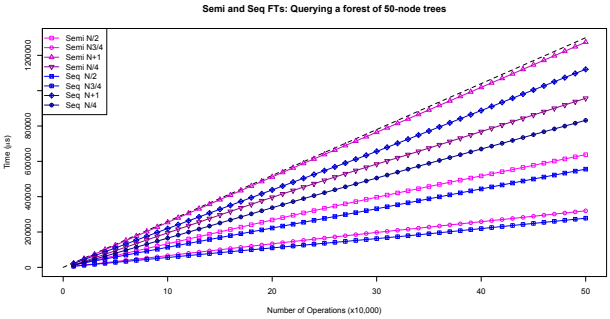


Figure 8: Looking for different tuples in Semi and Seq implementations

Finally, we plot the case when an update is performed. In this case we append two trees within the forest when the indices are *not* provided for the corresponding tuples in Figure **??**.

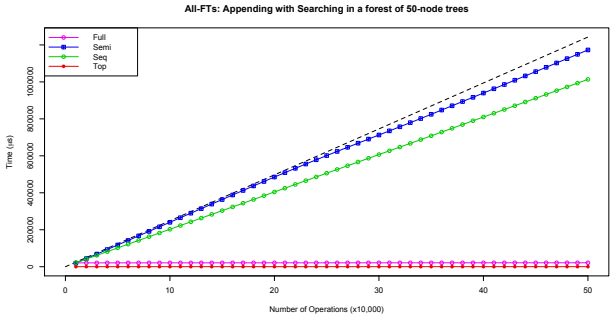


Figure 9: Appending two trees where the indices for the tuples involved are not provided

6 FURTHER WORK AND REMARKS

Finally, we state some conclusions and avenues for future work.

6.1 Conclusions

We have seen that monoidal annotations are a key factor in the performance of a finger tree, showing that original performance varies depending on such annotations, as shown in Section 2

We present `FatFast`, an specialised version on top of finger trees by Hinze and Paterson that enhances performance by a logarithmic factor for the abovementioned cases, in particular the `Set` as monoid.

6.2 Future Work

Since the internal dictionary-shape structure is the cause of the lack of performance we refer to potential cases which look promising but have not been implemented nor analysed from the functional approach. The first case is the work done by Iacono and Özkan in [3] where a skip lists are merged, i.e. disjoint set, in logarithmic time. Assuming that a purely functional implementation for skip lists is available, the performance for $\bowtie$ could be improved by a linear factor.

A second case is the parallelisation of the pure functions `measure` and the pattern matching at `searchT`. Some analyses on the strategies or the `Eval` and `Par` monads following Marlow in [5] should worth to try specially on large trees.

Finally, the current paper took into consideration the semi-strict data structure of `FT`. A combination of lazy structure for `FatFast` and other set-alike structure such `LazyPairingHeap` could provide different results at least in practice since not all the set-union are preformed straight away.

6.3 Source code

`FatFast` is freely available from: https://github.com/jcsaenzcarrasco/FatFast. This paper describes and refers the first version.

7 ACKNOWLEDGMENTS

The first author would like to thank the Consejo Nacional de Ciencia y Tecnología - CONACYT (the Mexican National Council for Science and Technology) for the financial support under the grant no. 580617/411550 and registration no. 214885

REFERENCES

[1] Ralf Hinze and Ross Paterson. 2006. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming* 16, 02 (2006), 197–217.
[2] Ralf Hinze and Ross Paterson. 2018. `Data.FingerTree`. http://hackage.haskell.org/package/fingertree. [Online; accessed 20-June-2019; version 0.1.4.1].
[3] John Iacono and Özgür Özkan. 2010. Mergeable Dictionaries. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I*. 164–175. https://doi.org/10.1007/978-3-642-14165-2_15
[4] Daan Leijen. 2017. `Data.Set`. https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Set.html. [Online; accessed 20-June-2019; version 0.5.10.2].
[5] Simon Marlow. 2011. Parallel and Concurrent Programming in Haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. 339–401. https://doi.org/10.1007/978-3-642-32096-5_7
[6] Chris Okasaki. 2018. `Data.Edison.Coll`. https://hackage.haskell.org/package/EdisonCore. [Online; accessed 20-June-2019; version 1.3.2.1].

Tensor Comprehensions in SaC — Extended Abstract —

Artjoms Šinkarovs
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Sven-Bodo Scholz
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
s.scholz@hw.ac.uk

ABSTRACT

We propose a new notation for data parallel operators on multi-dimensional arrays named *tensor comprehensions*. This notation combines the basic principle of array-comprehensions with syntactical shortcuts very close to those found in the so-called Tensor Notations used in Physics and Mathematics. As a result, complex operators with rich semantics can be defined concisely. The key to this conciseness lies in the ability to define shape-polymorphic operations combined with the ability to infer array shapes from the immediate context. The paper provides a definition of the proposed notation, a formal shape inference process, as well as a set of re-write rules that translates tensor comprehensions as a zero-cost syntactic sugar into standard SaC expressions. The practical application of this system is demonstrated at the example of a neural network, making it possible to save about 30% of code, maintaining the original performance but improving redability.

ACM Reference Format:
Artjoms Šinkarovs and Sven-Bodo Scholz. 2020. Tensor Comprehensions in SaC — Extended Abstract —. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Most array languages offer a variety of built-in operators that are commonly used in practice, *e.g.* linear algebra, tensor operations, *etc.* Typically, these operators are pure functions that can be easily composed, opening a great potential for program analysis and optimisations.

While these operations are very powerful when expressing homogeneous computations on entire arrays, they lead to rather cumbersome specifications when operating on subsets of the elements only; arrays have to be disassembled into sub-arrays before applying the desired functionality and re-assembled thereafter. As a simple example, consider incrementing all inner elements of a vector v[1]:

```
   take([1 ], v)
++ drop([1 ], drop([-1], v)) + 1
++ take([-1], v)
```

Note here, that negative first arguments to the functions `take` and `drop` relate to elements from the "right", *i.e.* starting from the highest index.

If we apply this approach to a matrix we need to create 5 sub-arrays, only one of which actually is being incremented. Looking at the generic case of an array of rank n, we need to involve $2^n + 1$ sub-arrays. Such inconvenient specifications can be avoided through the introduction of array comprehensions. Array comprehensions introduce explicit references to indices, enabling the identification of the relevant elements through index relations. Using the array comprehensions of SaC, named *with-loop*s, the above example can now be specified as

```
with {
  ([1] <= [i] < shape(v)-1) : v[i] + 1;
} : modarray( v);
```

A further benefit of such comprehensions is the straight-forward extensibility to higher ranks. For matrices `m`, we can write

```
with {
  ([1,1] <= [i,j] < shape(m)-1) : m[i,j] + 1;
} : modarray( m);
```

and in shape-polymorphic languages such as SaC, we can define inner increments for arrays `a` of arbitrary rank in the same fashion:

```
with {
  ( 0*shape(a)+1 <= iv < shape(a)-1) : a[iv] + 1;
} : modarray( a);
```

The expressive power of with-loops allows them to be used as vehicle to implement all build-in array operations in SaC, including `take`, `drop`, and `++` from above. While this is very convenient from the perspective of language design and implementation, it comes at a price: even trivial operations such as element-wise arithmetics require explicit index-range and shape specifications. Often, this does not matter, as shape-polymorphism enables the abstraction of generic operators such as an element-wise addition for arrays of arbitrary rank. However, in cases where many new operators are needed, the necessity to express shape ranges and ranks adds a lot of verbosity.

Looking at textbooks from Mathematics and Physics, we can see that these have the very same issues when describing tensor calculi. In typical tensor notations, it is customary to omit index ranges whenever they are "obvious" from the context. Likewise, index-ranges of summations and even the summation symbols themselves are omitted based on the occurrence of indices on both sides of equations.

[1]Here, we use the array operators as they are provided in the SaC standard library; similar operators can be found in all array languages.

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper, we describe a shortcut notation for with-loops and array comprehensions in general which imitates the tensor notations while, at the same time, guaranteeing a non-ambiguous semantics.

The contributions of this paper are threefold:

- new notation for array comprehensions;
- formal specification of its desugaring; and
- application of the proposed notation at the example of implementation of a neural network.

2 WITH-LOOPS

In this section we quickly recap the notion of the with-loop construct. The construct can be understood from the following example:

```
with {
   ([3] <= iv < [5]) : iv[0] + 1;  // generator part
} : genarray                       // operator
            ([8],                  // shape of the result
             42)                   // default element
```

which evaluates [42, 42, 42, 4, 5, 42, 42, 42] — a 1-dimensional array of 8 elements. Each with-loop consists of one or many generator expressions and an operator. Each generator expression defines a hyperrectangular set of indices and the expression that is evaluated for every index. In the above example, the index set is {[3], [4]} and the expression is `iv[0] + 1`, where `iv` binds to each index from the given index set. Finaly, the `genarray` operator constructs the new array of shape provided by the first argument and by the default element provided by the second argument. In case index sets intersect, the last definition is taken.

Finally, the with-loop provides the `fold` operator that can be understood from the following example:

```
with {
   ([0] <= iv < [10]) : iv[0]      // generator part
} : fold                           // operator
        (+,                        // binary function
         0)                        // default element
```

computes the sum of first 9 natural numbers.

As a more practical example, consider matrix multiplication of square matrices A and B of the same shape, written using *with-loop* notation:

```
with {
  ([0,0] <= iv < shape (A))
    : with {
       ([0] <= jv < shape (A)[0])
         : A[[iv[0], jv[0]]] * B[[jv[0], iv[1]]];
      } : fold (+, 0);
} : genarray (shape (A), 0)
```

The outer operates on the $n \times n$ index space, where $[n, n]$ is the shape of A and B. The inner with-loop performs a reduction of $A_{ik}B_{kj}$. As iv is a 2-element vector and jv is a 1-element vector, reshuffling of the indices involves selections into iv and jv.

Even though one can be convinced that the above code actually implements matrix multiplication, it is clear that not only this notation is difficult to read, but also it contains excessive information. In the outer with-loop, the generator spans across the entire index space of the array that we generate. Therefore, it would be convenient to specify the shape of the array once and introduce the notion of the generator that spans across the entire array. In this case the default element of the outer `genarray` is not needed.

Generally, one might ask whether default element is ever needed, as one might require the generators to partition the index set of the resulting array and never provide a default element. Even though this is possible in some cases, the case when with-loops evaluate an empty array becomes ambiguous. Consider the following expression:

```
with {
  ([0,0] <= iv < [0,0]) : e        // empty generator
} : genarray ([3,3], /* ? */)
```

The problem here is that when the generator is empty, the shape of the default element determines the shape of the resulting array. For example, if the default element would be 42, the resulting shape would be [3, 3]. In case the default element would be [42, 42, 42], the sahpe of the resulting array would be [3, 3, 3]. One could say that the shape of the expression in the generator can be used instead of the default shape. The are two problems with that. First, e potentially refers to the bound variable, therefore we cannot evaluate the shape of e unless we bind iv to some value. Unfortunately, there are no valid values for iv, as the generator is empty. Secondly, the generator expression can lead to non-termination or an error, *e.g.* `iv[0]/0`, therefore we also cannot evaluate it.

3 NEW NOTATION

Our new notation is built on the idea of a set comprehensions, but instead of generating a set, we generate a multi-dimensional array:

```
// Variable    Expression     Generator
{  iv       -> e           | g          ;
   ...                                   }
```

Array comprehensions contain one or more partitions, each of which is very simliar to the genarray with-loop. The iv is a bound variable, e is an expression as in the with-loop, and g defines index positions at which e will be evaulated. The final result is obtained by unifying all the partitions of the array comprehension.

The following features of array comprehensions are different from the with-loop construct.

Multiple generators. Array comprehensions contain a list of variable-expression-genertor triplets. Each triplet represents a subpartition of the index-space that is being computed. For example:

```
{ iv -> 1 | [0] <= iv < [5]  ;
  iv -> 2 | [5] <= iv < [10] }
```

evaluates a vector of 10 elements, where first five of them are ones and the rest are twos.

Simplified generators. Array comprehensions make it possible to omit specification of the lower bound, upper bound or the entire generator. For example, the following forms of expressions are supported:

```
{ iv -> e | iv < ub  }      // upper bound only
{ iv -> e | lb <= iv }      // lower bound only
{ iv -> e }                 // no generator
```

where e is the body of the array comprehension and lb and ub are expressions for the lower and upper bounds. In all three cases, we attempt to infer the full generator. In the first case this is straightforward: the lower bound is simply `ub*0`. Resolving the second case

depends on some external knowledge about the shape of the array we are evaluating. The third case requires analysis of e so that we could infer the range of the expression from the access patterns within e. For example,

```
{ iv -> a[iv] }
```

is a legal expression, with the index range spanning from `0*shape(a)` to `shape(a)`. As array selections may contain arbitrarily complex expressions, it is not always possible to compute the index bound of e. In these cases the inference algorithm produces an error and asks the user to provide additional shape information for e.

Index Pattern Matching. Index components can be bound to variables:

```
{ [i,j] -> i+j | [i,j] < [3,3] }
```

Additionally, we support two patterns: a single dot and three dots. A dot makes it possible to skip the component of an index vector without binding it to a variable. For example `[i,.,j]` defines a tree-dimensional index space, where the frist and the second component of the index vector are bound to i and j correspondingly. The tripple dot can be used once in a pattern, and it skips all the index components except those that are specified by the pattern on the left and on the right. For example: `[i,...]` defines an index space of any rank where the first index component is bound to i. The `[i,...,j]` expression also defines an index space of any rank, and it binds the first and the last index components to i and j. Both dot patterns can be used simultaneously: `[.,i,...,j,.]`.

The last partition. The new notation does not require specification of neither the default element nor its shape. Instead the shape of the default element is infered. However, as mentioned before, this cannot be done in all cases. Our convention that the last partition of the array comprehension determines the shape of the entire result helps to overcome this problem. We achieve two goals simultaneously: we provide the missing shape information for the entire expression; and we provide the default element in case the inference failed. For example:

```
{ [i,j] -> i+j | [3,3] <= [i,j] ;
  iv    -> 0   | iv < [10,10]   }
```

If there were no second partition, the index space of that expression would have been underspecified.

With the new notation, matrix multiplication example reduces to:

```
{ [i,j] -> sum({ [k] -> A[i,k]*B[k,j] }) }
```

where `sum` is library function that can be implemented using a fold with-loop.

4 DESUGARING

In the full paper we provide a formal specification of the inference algorithm, and the translation scheme from the new notation to SaC code.

TBD.

5 EXAMPLE

We used the new notation to encode a convolutional neural network in SaC. The entire code can be found at https://github.com/SacBase/CNN. We made sure that the new syntax is correctly parsed and interpreted by the compiler and that desugaring does not introduce any performance overheads.

As a quick example consider a shape-polymorphic formulation of the convolution computed on the array `in` with weights `w`.

```
float[*] conv(float[*] in, float[*] w)
{
  s = shape(in) - shape(w) + 1;
  return {iv -> sum({ov -> in[iv+ov] * w[ov]}) | iv < s};
}
```

After computing the resulting shape s, we use a doubly-nested array comprehension. The outer one defines an index-space bound by s where every element is a sum of component-wise multiplication of w and a subarray of s that starts at offset iv. The `sum` is a library function that can be defined by means of a fold with-loop. Note that we did not specify the default element and shape in the inner comprehension and we only had to specify the upper bound in the outer one.

6 RELATED WORK

TBD.

7 CONCLUSIONS & FUTURE WORK

TBD.