Draft Proceedings of the 29th Symposium on Implementation and Application of Functional Languages (IFL 2017)

Nicolas Wu (Editor)
University of Bristol

August 29, 2017

Foreword

I am delighted to welcome you to IFL 2017, the 29th Symposium on Implementation and Application of Functional Languages, held in Bristol, UK, from 30 August until 1 September, 2017. The goal of the IFL symposia is to bring together researchers actively engaged in the implementation and application of functional and function-based programming languages. IFL is a venue for researchers to present and discuss new ideas and concepts, work in progress, and publication-ripe results related to the implementation and application of functional languages and function-based programming.

The call for papers generated 24 submissions, all of which were accepted for presentation and are contained in these draft proceedings. The submissions were screened by the programme committee chair to make sure they are within the scope of IFL. It should be stressed, however, that these contributions are not peer-reviewed publications. Following the IFL tradition, IFL 2017 will use a post-symposium review process to produce formal proceedings, which will be published. After the symposium, authors will be given the opportunity to incorporate the feedback from discussions at the symposium and will be invited to submit a revised paper for the formal review process. From the revised submissions, the programme committee will select papers for the formal proceedings considering their correctness, novelty, originality, relevance, significance, and also clarity.

The programme consists of 24 presentations, two tutorial tracks, and one invited keynote tutorial track. The tutorial tracks are presented by Oleg Kiselyov on "Systematic Generation of Optimal Code", and by Melinda Tóth, István Bozó, and Tamás Kozsik on "Pattern Candidate Discovery and Parallelization Techniques". Edwin Brady, from the School of Computer Science at the University of St Andrews, is the invited speaker of IFL 2017. His tutorials are entitled "Towards an Implementation of Idris in Idris".

Putting together IFL 2017 was truly a team effort. I am grateful to the Department of Computer Science, Helen Cooke and Lindsay Coleman in particular, for administrative support. I would like to thank the members of the programme committee for accepting my invitation and for their work in putting together the programme. I am also very grateful to Erlang Solutions, S N Systems, Credit Call, and the University of Bristol Computer Science Society for sponsoring undergraduate attendees to the symposium. Last but not least I would like to thank Matthew Pickering for putting together the website and helping with the organisation, and also to Adam Beddoe, Samantha Frohlich, Charlie Harding, Jamie Luxford, Ibrahim Qasim, and Alessio Zakaria for their help in running the symposium.

Nicolas Wu
Chair of IFL 2017
University of Bristol
Bristol, UK, August 2017

Contents

Organisation

Chair

Nicolas Wu, University of Bristol, UK

Programme Committee

Kenichi Asai, Ochanomizu University, Japan

Sandrine Blazy, University of Rennes 1, France

Carlos Camarao, Universidade Federal de Minas Gerais, Brazil

Stephen Dolan, University of Cambridge, UK

Jurriaan Hage, Utrecht University, Netherlands

Yukiyoshi Kameyama, University of Tsukuba, Japan

Benjamin Lerner, Brown University, USA

Bas Lijnse, Radboud University, Netherlands

Garrett Morris, University of Kansas, USA

Miguel Pagano, Universidad Nacional de Córdoba, Argentina

Tomas Petricek, Alan Turing Institute, UK

Maciej Piróg, University of Wrocław, Poland

Exequiel Rivas, Universidad Nacional de Rosario, Argentina

Neil Sculthorpe, Nottingham Trent University, UK

Melinda Tóth, Eötvös Loránd University, Hungary

Phil Trinder, Glasgow University, UK

Kanae Tsushima, National Institute of Informatics, Japan

Marcos Viera, Universidad de la Republica, Uruguay

Meng Wang, University of Kent, UK

Organising Committee

Matthew Pickering, University of Bristol, UK

Converting Haskell to Coq (Extended Abstract)

Antal Spector-Zabusky
University of Pennsylvania
Philadelphia, PA, USA
antals@seas.upenn.edu

Abstract

We are building a tool, `hs-to-coq`, which can automatically translate a significant portion of Haskell into Coq. This tool will be used, as part of the DeepSpec project, to verify real-world Haskell programs. In particular, we plan to use `hs-to-coq` to verify components of GHC, starting with (1) the "Call Arity" analysis pass that is used to optimize function calls; and (2) the "Core Lint" pass that typechecks GHC's internal representation.

Keywords Haskell, Coq, Program Translation, Program Verification

ACM Reference Format:
Antal Spector-Zabusky. 2017. Converting Haskell to Coq (Extended Abstract). In *Proceedings of 29th Symposium on Implementation and Application of Functional Languages, Bristol, UK, August 30–September 1, 2017 (IFL'17),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

One of our strongest weapons in the fight against incorrect software is program verification. One of the only ways to be sure that software does what it's supposed to do is to verify it against a specification. And when verifying real-world software, it is not sufficient to write these proofs about models: bugs often lie in the *implementations* of algorithms, not simply in their design. We must instead verify the actual programs themselves – we must provide *live specifications* – which demands computer-assisted (mechanical) verification.

However, the state of the art in mechanical verification has not yet progressed to be able to verify real-world programs; for example, to produce a verified C compiler, Leroy wrote CompCert [5] instead of verifying an existing C compiler such as GCC [6]. One obstacle to verifying existing programs is that they are rarely written in languages conducive to verification: few to no widely-used languages have a formal semantics (a "core calculus" is insufficient to capture all the code that is actually written). One of the few ways to ensure that programs are verifiable is to write them inside a proof assistant. For example, GCC, which is unverified, is written in C and C++, while CompCert, which is mechanically verified, is written in the Coq proof assistant [4].

Many languages other than Coq make claims towards being able to produce correct software. Different languages use different techniques; Haskell programmers, in particular, claim that Haskell makes it easier to write correct software due to its

1. purity;
2. functional paradigm; and
3. powerful type system.

There are other languages which are purely functional with powerful type systems, including Coq. If all our Haskell programs were written in Coq, then we could verify them with Coq's powerful logic. Alas, the two languages are not the same. But perhaps we can bridge that gap.

As part of the DeepSpec project [1], we are developing a tool, descriptively named `hs-to-coq`, for translating Haskell source code to Coq source code. It is a Haskell program that uses GHC [7] (a Haskell compiler written in Haskell) to parse Haskell source files, then analyzes the resulting Haskell AST and outputs a Coq source file. This tool is a work in progress, but can already translate a significant portion of Haskell, including

- inductive data types, including parametric and mutually recursive data types;
- simple structurally-recursive functions;
- record selectors and record-based pattern-matching;
- pattern-matching with simple guards; and
- type signatures.

Our eventual goal with this tool is to translate pieces of GHC itself into Coq and verify them. GHC's internals operate mostly on an internal language called *Core*, which is intended to implement System FC [8], a variant of System F. Our translation efforts are focused on translating the definition of Core and on verifying individual Core-to-Core passes. In particular, we are exploring verifying the "Call Arity" analysis pass [3], which determines how many arguments are passed to functions when they're called; this is used to optimize away the overhead of having every function be fully curried. A model of this analysis has already been verified in Isabelle [2], but the actual implementation in GHC has not. We also hope to extend our translation efforts to "Core Lint", an optional sanity check that typechecks Core programs.

What's the logic behind doing this verification by translating Haskell to Coq? The two languages seem fairly compatible: both are purely functional languages with (co)inductively-defined data types; and Coq's type system is stronger (it even includes type classes!), so we should never have a well-typed Haskell term that corresponds to an ill-typed Coq term.

IFL'17, August 30–September 1, 2017, Bristol, UK
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, there are some very serious differences between Haskell and Coq as well. On the Coq side, most of the fundamental differences stem from the fact that Coq is *total* (specifically, *weakly normalizing*); termination is ensured in several different ways. Haskell, on the other hand, is Turing-complete and makes no normalization guarantees. In particular:

- In Coq, recursive functions must be structurally recursive, and corecursive functions must be guarded; in Haskell, there is unrestricted recursion (and thus corecursion).
- In Coq, all pattern matches must handle every case; in Haskell, pattern matches may be incomplete.
- In Coq, all data types must be strictly positive; in Haskell, there is no such restriction.
- In Coq, inductive and coinductive data types are distinguished, and thus so are recursion and corecursion; in Haskell, these are not distinguished, and many types (e.g., lists) are used both recursively (e.g., `length`) and corecursively (e.g., `cycle`).

Our current translation scheme requires the user to address these differences directly, such as by omitting and reimplementing the offending Haskell types and functions. We also insert axioms on the Coq side for representing the missing cases of partial functions, as well as functions like Haskell's `error`; to fully complete the translation and verification, we need to ensure that the Haskell code never fails, and so those axioms will eventually need to be addressed and removed.

On the Haskell side, the fundamental difference mostly stems from the fact that Haskell is a fully-fledged programming language that can interact with the real world, whereas Coq is a theorem prover and is self-contained. In particular, in Haskell, the `IO` type constructor wraps (purely!) input and output, mutable state, and all sorts of other machine-level functionality; Coq has no notion of these sorts of features at all. Our current translation does not address `IO` at all; we are focused on the functional core of Haskell. Using this core as much as possible is generally considered good style, and it is used to write most of the parts of GHC that we are concerned with verifying.

During the course of developing this tool, we have also run into other, less-fundamental, difficulties, including the following:

- Haskell is a very syntactically rich language; Coq is relatively minimal. Some features, such as the fall-through semantics of guards, were and are difficult to translate.
- Some features of Haskell that have direct analogues in Coq have different semantics. For example, type class method definition and module imports and exports exist in both languages, but do not work the same way.

The `hs-to-coq` tool is already capable of translating most of the data types that define the structure of Core; however, the metadata contained in variable names is cyclic and contains non-strictly-positive data types, so we do not yet have the ability to translate those data types. We are currently attempting to finalize the translation of Haskell terms in a modular fashion, so that we can begin to verify the definition of the Call Arity analysis itself.

Acknowledgments

The verification of GHC is an enormous task that is just beginning, and I am deeply indebted to my collaborators: Joachim Breitner, John Wiegley, Christine Rizkallah, Yao Li, and my advisor Stephanie Weirich. This work was supported by NSF award #1521523, *Expeditions in Computing: The Science of Deep Specification.*

References

[1] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. *DeepSpec: The Science of Deep Specification.* https://deepspec.org/

[2] Joachim Breitner. 2015. Formally Proving a Compiler Transformation Safe. *SIGPLAN Notices* 50, 12 (Aug. 2015), 35–46. https://doi.org/10.1145/2887747.2804312

[3] Joachim Breitner. 2017. Call Arity. *Computer Languages, Systems & Structures* (2017). https://doi.org/10.1016/j.cl.2017.03.001

[4] The Coq development team. 1989–2017. *The Coq Proof Assistant.* INRIA. http://coq.inria.fr

[5] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[6] The GNU Project. 1987–2017. *GCC, the GNU Compiler Collection.* https://gcc.gnu.org/

[7] The GHC Team. 1992–2017. *The Glasgow Haskell Compiler.* https://www.haskell.org/ghc/

[8] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. *SIGPLAN Notices* 48, 9 (Sept. 2013), 275–286. https://doi.org/10.1145/2544174.2500599

The sufficiently smart compiler is a theorem prover

Extended abstract

Joachim Breitner
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104, USA
joachim@cis.upenn.edu

ABSTRACT

That the Haskell Compiler GHC is capable of proving non-trivial equalities between Haskell code, by virtue of its aggressive optimizer, in particular the term rewriting engine in the *simplifier*. We demonstrate this with a surprising little code in a GHC plugin, explains the knobs we had to turn, discuss the limits of the approach and related applications of the same idea, namely testing that promises from Haskell libraries with domain-specific optimizations hold.

ACM Reference format:
Joachim Breitner. 2017. The sufficiently smart compiler is a theorem prover. In *Proceedings of Implementation and Application of Functional Languages, Bristol, UK, September 2017 (IFL'17),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Functional programming languages are advertised as well suited to formal reasoning. Nevertheless, a programmer who actually tries to rigorously reason that one function definition in, say, Haskell is equivalent to another receives no support his development tools and has to resort to manual, handwritten proofs or external tools.

This is unfortunate, as the one tool that every programmer uses, and which really knows what a program means, can actually do some of these proofs: The compiler!

If we assume the compiler to be sufficiently smart, then it will optimize any piece of source code to the "best" output code possible. It follows immediately that two semantically equivalent expressions will be compiled to the same output. So to prove that two expressions are equivalent, we just have to ask the compiler to compile both, compare the output, and if they are the same we know that the input expressions are the same. Done.

If this paragraph made you raise your eyebrows, you can lower them again: We are well aware that this bold approach poses a few questions:

(1) How do we "ask the compiler" in the first place? Can we include this in the user's program workflow?
(2) How do we compare the output?
(3) The sufficiently smart compiler does not exist. Will the actually existing compiler be able to prove any interesting equivalences?
(4) If the compiler says that it cannot prove two expressions to be equal, what, if anything, does that mean?
(5) If the compiler says that it can prove the two expressions to be equal, what does *that* mean, given that compilers have bugs too?
(6) Besides proofs of program equivalence, is there anything else we can do with this idea?

Our contributions are raising these questions, and giving preliminary answers, which we summarize here:

(1) Custom syntax, embedded in the actual code, would be most convenient, but even without extending the language (and hence the whole compiler pipeline, starting with the parser) it is possible to include the proof obligations in the Haskell code, to be picked later in the pipeline by a compiler plugin. This way, no new or additional tools have to be employed.
(2) There is no need to compare machine code: Almost all interesting optimizations are applied by GHC's simplifier, which implements a rewrite engine. It suffices to compare simplified terms in GHC's intermediate language Core.
(3) Yes. We proved the Functor, Applicative and Monad laws of a simple, but already non-trivial data structure, as well as a number of rewrite rules from `hlint`'s database.
(4) It certainly does not mean that the expressions are not equivalent. But inspecting the differences in the simplified expressions can still provide useful insights, such as differences in strictness.
(5) It does mean that the expressions are equivalent. In a not completely tongue-in-cheek sense the only semantics for Haskell that matters in practice is the one defined by how the compiler compiles a program. Its proofs are therefore vacuously correct.
(6) The author of a Haskell library who carefully crafts it so that certain code (e.g. composition of list-processing functions) is optimized to more efficient, less pleasant code (e.g. explicit recursion) can use the same technique in his normal library test suite, ensuring that his promises continue to hold, even as the compiler is upgraded to newer versions.

1.1 Pedestrian Proofs

Consider a Haskell library author who wants to provide instances for the Functor, Applicative and Monad type classes. He would not

IFL'17, Bristol, UK

DOI: 10.1145/nnnnnnn.nnnnnnn

```
data Succs a = Succs a [a]

getCurrent :: Succs t → t
getCurrent (Succs x _) = x

getSuccs :: Succs t → [t]
getSuccs (Succs _ xs) = xs

instance Functor Succs where
  fmap f (Succs o s) = Succs (f o) (map f s)

instance Applicative Succs where
  pure x = Succs x [ ]
  Succs f fs ⟨∗⟩ Succs x xs
    = Succs (f x) (map ($x) fs ++ map f xs)

instance Monad Succs where
  Succs x xs ≫= f = Succs y (map (getCurrent ∘ f) xs ++ ys)
    where Succs y ys = f x
```

Figure 1: A non-trivial functor with type class instances

```
  pure (∘) ⟨∗⟩ Succs u us ⟨∗⟩ Succs v vs ⟨∗⟩ Succs w ws
= Succs (∘) [ ] ⟨∗⟩ Succs u us ⟨∗⟩ Succs v vs ⟨∗⟩ Succs w ws
= Succs (u∘) (map (∘) us) ⟨∗⟩ Succs v vs ⟨∗⟩ Succs w ws
= Succs (u ∘ v) (map ($v) (map (∘) us) ++ map (u∘) vs)
      ⟨∗⟩ Succs w ws
= Succs (u ∘ v) (map (($v) ∘ (∘)) us ++ map (u∘) vs)
      ⟨∗⟩ Succs w ws
= Succs ((u ∘ v) w) (map ($w) (map (($v) ∘ (∘)) us
      ++ map (u∘) vs) ++ map (u ∘ v) ws)
= Succs ((u ∘ v) w) (map (($w) ∘ ($v) ∘ (∘)) us
      ++ map (($w) ∘ (u∘)) vs ++ map (u ∘ v) ws)
= Succs (u (v w)) (map (λu. u (v w)) us
      ++ map (λv. u (v w)) vs ++ map (λw. u (v w)) ws)
= Succs (u (v w)) (map ($(v w)) us
      ++ map u (map ($w) vs ++ map v ws))
= Succs u us ⟨∗⟩ Succs (v w) (map ($w) vs ++ map v ws)
= Succs u us ⟨∗⟩ (Succs v vs ⟨∗⟩ Succs w ws)
```

Figure 2: A manual proof of the second Applicative law

only be expected to implement these methods, as shown in Figure 1, but also ensure that these instances fulfill the laws of the type class. He can do so by testing (and indeed, property-based testing [2] would nicely work here) or proving the desired properties; in this paper we focus on the latter.

If the user had written this code in the language of a theorem prover such as Gallina, then he could simply prove if in that system. Alas, Haskell is not a theorem prover, so what are the options? Certainly he could re-implement the code in a different system and perform the proofs there (and maybe the translation can be automated to some extent [6, 7]), but that requires knowing the other system first, and involves quite some extra work. So especially with smaller proof tasks such as this one, the author is likely to do the proofs "by hand", maybe in a comment of the file, and write long chains of equations as in Figure 2.

1.2 The Compiler is in the Know

Assume the programmer had added functions to the module that represent the left-hand-side and the right-hand-side of the one of the proof obligations, as follows:

```
functor_law1_rhs :: Succs a → Succs a
functor_law1_rhs x = x
functor_law1_lhs :: Succs a → Succs a
functor_law1_lhs x = fmap id x
```

What will the compiler make of that? We can view the intermediate code using `-ddump-simpl` and list the relevant lines:

```
$ ghc -O -ddump-simpl Succs.hs|grep functor_law1_lhs
functor_law1_lhs :: forall a. Succs a -> Succs a
functor_law1_lhs = functor_law1_rhs
```

This means that the compiler already knows that the left-hand side and the right-hand side of this equations are the same! How is it that it knows that, and why does this equation appear in the code?

We invoked GHC with the `-O` flag, telling it to optimize the code. Three optimizations play a role in this example:

- *Inlining* [4]: The call to `fmap` is replaced with the definition in Figure 1, and we obtain
  `functor_law1_lhs =`
  `  λx. case x of Succs o s → Succs (id o) (map id s).`
  Further inlining replaces `id o` with `o`.
- *Rewrite rules* [5]: The base library code contains a number of rewrite rules which allow the compiler to know the identity `map id xs = xs`. Therefore, we obtain
  `functor_law1_lhs =`
  `  λx. case x of Succs o s → Succs o s.`
- *Common subexpression elimination*: Given this code, the compiler knows that the case analysis is redundant, and simplifies this function definition further to
  `functor_law1_lhs = λx. x.`
  But this is precisely the definition of `functor_law1_rhs`! So, again by common subexpression elimination, the compiler lets both names refer to the same function:
  `functor_law1_lhs = functor_law1_rhs`

At this point, the programmer feels silly. Why did he spend the effort of manually proving these equations when they are already known by the tool that he uses already, namely the compiler?

2 PROOF BY COMPILATION

To get rid of this silliness we need to provide a way for the user to declare equations that he wants the compiler to check, and then actually check them.

A slightly simplified view of the compiler pipeline is shown in the top line of Figure 3; the most relevant component for us is the simplifier in the middle: It receives the program in GHC's intermediate language Core to simplify, optimize and rewrite it. Its output (dumped via `-ddump-simpl`) is where we discovered the statement of the desired equality.

So one approach would be to create a dedicated tool based on the compiler code (or ghc-the-library) runs this pipeline only until the simplifier and then checks if the desired equations hold. But one goal here is to *not* require the programmer to reach out to a new tool,

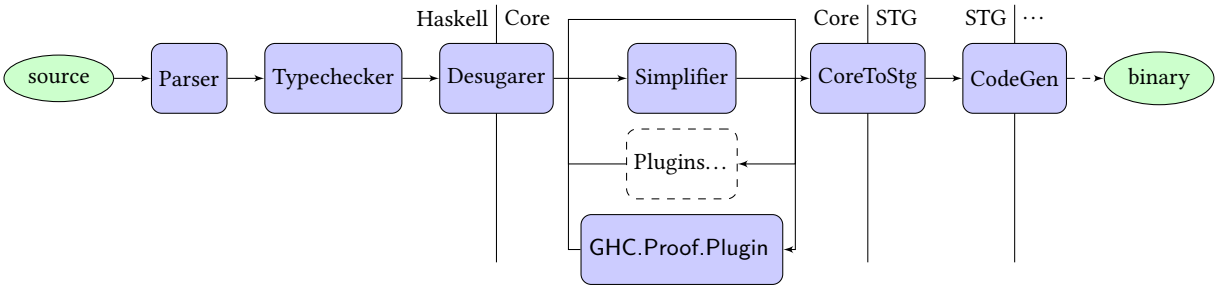


Figure 3: The GHC pipeline and the plugin hook

```
module GHC.Proof where

data Proof = Proof

(===) :: a → a → Proof
_ === _ = Proof
infix 0 ===
 {-# INLINE [0] (===) #-}
```

Figure 4: The proof obligation combinator

and we have to use the existing hooks provided by the compiler. One such hook is provided in the form of *GHC plugins*, which are dynamically loaded additional core-to-core passes functions performed by GHC.

2.1 Syntax for proof obligations

Unfortunately, the plugin interface is intended to introduce additional optimization passes, not new language features, and does not allow use to define new syntax for the proof obligations. But we can work around that limitation: The module GHC.Proof shown in Figure 4 exports the operator (===) whose type allows it to relate any two expressions of the same type.

A proof obligation is now simply an occurrence of (===) bound to a top-level name. The return type of the operator allows us to give descriptive type signatures to these names:

```
 {-# OPTIONS_GHC -O -fplugin GHC.Proof.Plugin #-}
import GHC.Proof
...
functor_law1 :: Succs a → Proof
functor_law1 = fmap id x === x
```

The actual implementation is irrelevant and is chosen so that at the end of the compilation pipeline (===) has been inlined and its arguments have been removed as dead code, so that embedded proof obligations do not increase the size of the compiled program.

2.2 The compiler plugin

The GHC flag `-fplugin GHC.Proof.Plugin`, which we can embed in the file as shown above, instructs the compiler to load our plugin, which inserts itself early in the optimization pipeline, directly after the desugarer. It

(1) scans the code for occurrences of lhs === rhs,
(2) passes lhs to the simplifier to obtain its optimized counterpart lhs', and likewise for rhs,
(3) compares lhs' and rhs' for alpha-equality.
(4) If all such pairs are alpha-equivalent, the plugin prints an encouraging message and compilation continues:

```
$ ghc -O Succs.hs
[1 of 1] Compiling Succs ( Succs.hs, Succs.o )
GHC.Proof: Proving functor_law1 ...
GHC.Proof proved 1 equalities
```

(5) But if they are not alpha-equivalent, e.g. trying to prove x <$ y === y, the plugin prints the differing left- and right-hand sides of the equation and aborts compilation:

```
$ ghc -O Succs.hs
[1 of 1] Compiling Succs ( Succs.hs, Succs.o )
GHC.Proof: Proving functor_law1 ...
GHC.Proof: Proving wrong_law ...
Proof failed
  Simplified LHS:
    case x of { Succs o s ->
    Succs.Succs @ a y
       (map @ a @ a ( _ [Occ=Dead] -> y) s)
    }
  Simplified RHS: x

Succs.hs: error: GHC.Proof could not
  prove all equalities
```

Since the plugin works at the stage of Core, these expressions are necessarily Core expressions and include, for example, explicit type applications (`Succs @ a`).

All heavy lifting is done by existing parts of the compiler (simplification and checking for alpha-equality), so the plugin itself simple and short (< 100 lines of code).

2.3 Gory implementation details

Nevertheless, there are a few technicalities that the plugin has to get right for this to work:

2.3.1 Pipeline location. A GHC compiler plugin inserts itself into the long list of Core-to-Core transformations performed by GHC. Ideally, we would like to install our plugin first, so that we can get our hands on the proof in their most pristine form and have full control over the optimizations. Unfortunately, the output of the desugarer (cf. Figure 3) is still incomplete. For example, important information about a function defined in the current module is not

yet available at the use-site of that function, including information required to inline that function. But without inlining, many proofs do not go through.

The first Core-to-Core pass is a "gentle" run of the simplifier, which takes care of this. Therefore, we insert the plugin as a the second pass, right after that.

2.3.2 Optimizations to run. The default Core-to-Core optimization pipeline contains, besides multiple invocations of the general simplifier, a few more specific passes: Common Subexpression Elimination (CSE), Demand analysis, Call Arity, Float-out and Float in, Worker-Wrapper transformation. Which of these do we want to run?

Most of them, such as the worker-wrapper transformation, make the code more efficient, but also more complex, and we do not expect them to be useful in these kind of proofs. Therefore, the small pipeline set up by the simplifier consists of multiple runs of the simplifier, interspersed with CSE.

One run of the simplifier can be configured by its *phase*, a small number that is counted down towards zero. Phases are used by complex arrangements of rewrite rules such as list fusion ([5]) to order rules. We run the simplifier in phases 2 and 1. We do not run a phase 0 because in this phase, recursive functions like `map` get inlined, which does not enable any further interesting transformations, but makes the output of a failed proof harder to understand.

The main purpose of the CSE run is to simplify useless case analyses expressions like **case** e **of** Succs x xs → Succs x xs to e.

2.3.3 Aggressive inlining. Inlining, i.e. replacing a function call with the body of the called function, is crucial for the effectiveness of this approach. GHC is generally very good at inlining, supporting, for example, cross-module inlining. But it refrains from inlining of larger functions when it does not see an obvious performance benefit, to avoid the program code size to explode.

In our case, however, there are more reasons for inlining than just performance, and code size is much less an issue. Therefore the plugin instructs the simplifier to be very aggressive when deciding whether to inline a function, and do not heed code size any more. Other ways to control inlining, such as the function arity, `NOINLINE` pragmas and phase control, are still adhered to.

3 WHAT CAN WE PROVE?

So far so simple. How far does this approach get us?

3.1 Type class instances

The type class instances in Figure 1 come with 9 laws. Of these, 7 laws go through immediately.

One failing law is

```
monad_law1 :: a → (a → Succs b) → Proof
monad_law1 a k = (return a >>= k) === k a
```

which fails with

```
GHC.Proof: Proving monad_law1 ...
Proof failed
    Simplified LHS:
        Succs.Succs
          @ b
          (case k a of { Succs y ys -> y })
          (case k a of { Succs y ys -> ys })
    Simplified RHS: k a
```

Upon closer inspection we can see that the two sides differ only in strictness and evaluation order: While the LHS is always a value and calls `k a` only if the fields of Succs are evaluated, the RHS immediately calls `k a`.

This can mean one of two things to the user: Either he learns that he indeed did not write a law-abiding monad instance, and may have to revise the definition. Or he decides that fast-and-loose reasoning [3] is fine and he is not worried about differences in the strictness. He can even encode this decision in the proof obligation and change it to

```
monad_law1 :: a → (a → Succs b) → Proof
monad_law1 a k = (k a `seq` return a >>= k) === k a
```

where the invocation of `seq` forces the left hand side to be strict in `k a`. The modified equality is proven by our plugin.

A similar problem arises with the other failing proof obligation, namely the third monad law:

```
monad_law3 :: Succs a → (a → Succs b)
                      → (b → Succs c) → Proof
monad_law3 m k h = m >>= (λx. k x >>= h) ===
                   (m >>= k) >>= h
```

This law only holds if `h` is strict, so in order for this to pass, the user would instead write

```
monad_law3 m k h = m >>= (λx. k x >>= (h $!)) ===
                   (m >>= k) >>= (h $!)
```

to force `h` to be strict in its argument.

3.2 Custom data type

The code in Figure 1 in the previous section makes heavy use of the built-in data type for lists. Were the proof in the previous section maybe only successful because the compiler handles handle list functions specially?

To test that, changed replaced the use of lists in Figure 1 with *bags*, which ordered collections represented by trees.[1] The Succs

```
module Bag where

data Bag a = Empty
           | Singleton a
           | Append (Bag a) (Bag a)

mapBag :: (a → b) → Bag a → Bag b
mapBag f Empty          = Empty
mapBag f (Singleton x)  = Singleton (f x)
mapBag f (Append b1 b2) = Append (mapBag f b1)
                                 (mapBag f b2)
append Empty b = b
append b Empty = b
append b1 b2 = Append b1 b2
```

Figure 5: The Bags data structure

[1] In fact, due to the $O(1)$ concatenation, bags are more a suitable data structure here anyways.

```
{-# NOINLINE [0] mapBag # -}
{-# RULES "map/id" mapBag (λx. x) = id # -}
{-# RULES "map/." forall f g b. mapBag f (mapBag g b) = mapBag (f ∘ g) b # -}
{-# RULES "map/Empty" forall f. mapBag f Empty = Empty # -}
{-# RULES "map/append" forall f b1 b2. mapBag f (b1 `append` b2) = mapBag f b1 `append` mapBag f b2 # -}

{-# NOINLINE [0] append # -}
{-# RULES "append/Empty1" forall b. Empty `append` b = b # -}
{-# RULES "append/Empty2" forall b. b `append` Empty = b # -}
{-# RULES "append/assoc" forall b1 b2 b3. (b1 `append` b2) `append` b3 = b1 `append` (b2 `append` b3) # -}
```

Figure 6: Domain-specific rewrite rules

code needs to construct empty bags, map functions over bags and append bags; these operations are implemented as Figure 5.

After switching to this data type, all proof obligations fail. So clearly, the compiler knows something about list operations that he does not know here. But that is not a problem: We can simply tell the compiler what he needs to know, in the form of rewrite rules. These rules, listed in Figure 6, can be shipped with the Bag code and will be applied whenever the compiler sees code that matches the left-hand side of such a rule.

The `NOINLINE` pragmas are required to prevent the compiler from inlining these functions before the rules had a change to be applied.

Rule "*append/assoc*" is especially interesting: The append operator is not really associative, as it is easy to find bags b1, b2 and b3 so that the expression on the left-hand side of the rule can be distinguished from the right-hand side of the rule. But both sides represent the same collection, and all functions over bags are expected to respect this. In this sense, the equation holds *morally*, and we, as the library designer, may want to allow the compiler to perform this transformation. Since this is domain-specific knowledge, we cannot expect even the smartest compiler to do this on its own.

With these rules in place all 9 type class laws are proven again.

3.3 HLint rules

The HLint tool analyses Haskell code and suggests improvements to it. It has a database of patterns and replacement, which has been the target of formal verification before [1]. At the time of writing, we looked at 92 proof obligations, of which GHC is able to prove 34.

It is notable that even expressions involving IO are within reach. The plugin can prove, for example, the following identity:

```
proof1 x = putStrLn (show x) === print x
```

4 DISCUSSION

Let us return to the questions from the introduction.

(1) We have seen that it is possible to specify these proof obligations without extending the surface syntax of Haskell. Nevertheless, it might be desirable to do so in the long run, as the current approach has a few shortcomings. If the plugin does not "find" the (===), a proof obligation might simply be unnoticed. This might happen if the user abstracts over this operator.

(2) We check the output at the Core level, where all interesting transformations have been applied, but the expressions still resemble the user's code. Currently we compare the expressions for alpha-equality, but this can be too strict. One would reasonably expect the two expressions

$$\lambda x\ y.\ \textbf{case}\ x\ \textbf{of}\ (x2, \_) \to \textbf{case}\ y\ \textbf{of}\ (y2, \_) \to (x2, y2)$$

and

$$\lambda x\ y.\ \textbf{case}\ y\ \textbf{of}\ (y2, \_) \to \textbf{case}\ x\ \textbf{of}\ (x2, \_) \to (x2, y2)$$

to be considered equal. A more liberal equivalence relation on Core terms could allow that – and would be useful in other places in the compiler, e.g. in CSE.

(3) The previous section shows that indeed non-trivial equalities are proven by this approach. If the compiler is not sufficiently smart yet, the programmer is able to educate it about further equalities.

GHC's rewrite rule system is already quite useful, but still rather simple compared to the systems in theorem provers, e.g. Isabelle's `simp` method. If more sophisticated term rewriting features are added to GHC to meet a demand created by this application, then Haskell library authors would also benefit from the additional power.

(4) Of course, this approach will never disprove an equality. If the compiler tells the user that it cannot prove an equality, it very likely means that it is simply out-of-reach for this approach. The user may investigate the given terms and determine if they ought to be equal and can add some some strictness annotations or additional rewrite rules.

(5) The trustworthiness of these "proofs" raise some interesting questions. On the one hand we can see the compiler as a huge pile of unverified and bug-infested code, and we have no formal guarantees that the compiler respects some implicit or explicit semantics of the code. So the compiler saying that an equality holds does not mean anything.

On the other hand, we can see the compiler as the practically relevant way of giving semantics to our code in the first place. And with regard to this implementation-defined semantics, the compiler is correct by definition!

(6) Asking the compiler questions about how he compiles certain expressions has useful applications beyond proving theorems to be equal.

When developing a well-performing Haskell libraries, e.g. the list functions in the base library or the `vector` library, the library authors pay close attention to how the

compiler compiles certain combinations of the library's functions. This is especially true when the Libra ships rewrite rules, as reading the Core output is the only way to make sure that the rules have the desired effect.

But when a new version of the compiler is released, it is not unlikely that it compiles the code in question slightly differently, and suddenly the optimizations do not happen in the intended way any more. But unless the library author manually checks the output, or has performance regression tests sensitive enough to notice the difference, this might go unnoticed.

But using the methods described here, the library author can explicitly list both the high-level code that it wants its user to be able to write and the low-level code that this should be compiled to (or rather, the Haskell source equivalent to the expected Core code), and ensure that this happens as part of the library's regular test suite.

In the context of test suites there is another interesting application of this idea. Imagine that the author of the Succs library would have used QuickCheck to test the type class laws instead. Whenever the the test suite is run the test suite would generate a hundred random inputs, pass them to the properties and compare the outcome left and right of the === operator. But, as we have seen now, after compilation, the exact same code is on both sides of the comparison!

In that case it would speed up the test suite to skip generating arguments for the property in the first place. Moreover, the test suite could list the test as "statically known to succeed", which may convey useful information to the programmer – maybe he expected to test a tricky property and a vacuously satisfied property indicates a bug in the test suite?

5 CONCLUSION

We find that GHC's is able to prove non-trivial program equalities just by virtue of its optimizer, and with – some convincing – is able to allow users to make use of this ability. We do not expect this method to succeed only for a limited class of equations, but it is unclear yet just how small or large this class is. Increasing the reach of the method may require more configurability (i.e. selectively disable or enable rules). Furthermore, every failed proof may point to a way in which the compiler's optimizer itself can be extended, which may in turn improve the compiler itself. Finally, even if the proving aspect of this work remains a cute hack, the outlined applications in testing may have actual impact.

REFERENCES

[1] Joachim Breitner, Brian Huffman, Neil Mitchell, and Christian Sternagel. 2017. Isabelle/HOLCF-Prelude. *Archive of Formal Proofs* (2017). https://www.isa-afp.org/entries/HOLCF-Prelude.shtml

[2] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*. ACM, 268–279.

[3] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. *SIGPLAN Not.* 41, 1 (2006). DOI: http://dx.doi.org/10.1145/1111320.1111056

[4] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming* 12, 5 (2002). DOI:http://dx.doi.org/10.1017/S0956796802004331

[5] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*.

[6] Tobias Rittweiler and Florian Haftmann. 2015. Haskabelle – converting Haskell source files to Isabelle/HOL theories. (2015). http://isabelle.in.tum.de/haskabelle.html

[7] Antal Spector-Zabusky. 2017. Converting Haskell to Coq. In *IFL*.

Extrapolate: generalizing counter-examples of functional test properties

Rudy Braquehais
University of York, UK
rmb532@york.ac.uk

Colin Runciman
University of York, UK
colin.runciman@york.ac.uk

Abstract

This paper presents a new tool called Extrapolate that automatically generalizes counter-examples found by property-based testing in Haskell. Example applications show that generalized counter-examples can make the task of finding faults easier and less time consuming. Extrapolate is able to produce more general results than similar tools. Although it is intrinsically unsound, as reported generalizations are based on testing, it works well in practice.

CCS Concepts • **Software and its engineering → Software testing and debugging**;

Keywords enumerative property-based testing, systematic testing, debug assistance, functional programming, Haskell.

ACM Reference Format:
Rudy Braquehais and Colin Runciman. 2017. Extrapolate: generalizing counter-examples of functional test properties. In *Proceedings of 29th symposium on Implementation and Application of Functional Languages, Bristol, UK, August 30 - September 1, 2017 (Pre-symposium proceedings (Accepted for Presentation) IFL 2017)*, 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Arguably, most programmers are familiar with the following situation: a failing test case has been reported by an automated testing tool; but, it is not immediately apparent what more general class of tests would trigger the same failure. The programmer may resort to painstaking step-by-step reevaluation of the reported failure in the hope of realizing where a fault lies. In this paper, we examine a less explored approach: the generalization of failing cases informs the programmer more fully and more immediately what characterises such failures. This information helps the programmer to locate more confidently and more rapidly the causes of failure in their program. We present Extrapolate, a tool to generalize counter-examples of test properties in Haskell. Several example applications demonstrate the effectiveness of Extrapolate.

Pre-symposium proceedings (Accepted for Presentation)
IFL 2017, August 30 - September 1, 2017, Bristol, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$00.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Example 1.1. Consider the following faulty `sort` function:

```
sort :: Ord a => [a] -> [a]
sort []     =  []
sort (x:xs) =  sort (filter (< x) xs)
            ++ [x]
            ++ sort (filter (> x) xs)
```

The function `sort` should have the following properties:

```
prop_sortOrdered :: Ord a => [a] -> Bool
prop_sortOrdered xs  =  ordered (sort xs)

prop_sortCount :: Ord a => a -> [a] -> Bool
prop_sortCount x xs  =  count x (sort xs)
                     == count x xs
```

Together these properties completely specify `sort`.

If we pass both properties to a property-testing library, such as QuickCheck or SmallCheck, we get something like:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK, passed 360 tests.

> check (prop_sortCount :: Int -> [Int] -> Bool)
*** Failed! Falsifiable (after 4 tests):
0 [0,0]
```

But, for the failing property, Extrapolate goes on to print:

```
Generalization:
x (x:x:_)
```

Some values have been generalized: the specific value `0` does not matter, `prop_sortCount x (x:x:_)` fails for any integer `x`; the tail value `_` does not affect the result.

And, if we include `count` as a *background function* (§3.3):

```
> let chk = check `withBackground`
>         [ constant "count" count ]
> chk (prop_sortCount :: Int -> [Int] -> Bool)
```

Extrapolate also prints:

```
Conditional Generalization:
x xs  when  count x xs > 1
```

Our faulty `sort` function fails exactly for lists that have repeated elements. □

1.1 Contributions

The main contributions of this paper are:

1. methods using automated black-box testing to generalize counter-examples of functional test properties by replacing constructors with variables. The described methods are able to produce more general results than previous similar techniques by allowing repeated variables and variables subject to side-conditions to appear in generalized counter-examples.
2. the design of the Extrapolate library, which implements these methods in Haskell and for Haskell test properties;
3. a selection of small case-studies, investigating the effectiveness of Extrapolate;
4. a comparative evaluation of generalizations performed by Extrapolate and similar tools for Haskell.

Despite the Haskell setting of the implementation and experiments, we expect similar techniques to be readily applicable in other functional programming languages.

1.2 Road-map

This paper is organized as follows:

- §2 describes how to use Extrapolate;
- §3 describes how Extrapolate works internally;
- §4 presents example applications and results;
- §5 discusses related work;
- §6 evaluates Extrapolate in comparison with similar tools for Haskell.
- §7 draws conclusions and suggests future work.

2 How Extrapolate is Used

Extrapolate is a library: modules using it should include "`import Test.Extrapolate`". Unless they already exist, instances of the `Listable` (§3.1) and `Generalizable` (§3.2) typeclasses are declared for needed user-defined datatypes (step 1). The `check` function should be applied to each test property (step 2).

Step 1. Provide class instances for used-defined types Extrapolate needs to know how to generate test values of properties — this capability is provided by instances of the `Listable` typeclass (§3.1). Extrapolate also needs to extract the structure of test values so that it can perform its generalization procedure — this capability is provided by instances of the `Generalizable` typeclass (§3.2). Extrapolate provides instances for most standard Haskell types and a facility to derive instances for user-defined data types using Template Haskell [36]. Writing

```
deriveListable ''<Type>
deriveGeneralizable ''<Type>
```

```
import Test.Extrapolate

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = sort (filter (< x) xs)
           ++ [x]
           ++ sort (filter (> x) xs)

prop_sortOrdered :: [Int] -> Bool
prop_sortOrdered xs = ordered (sort xs)
  where
  ordered (x:y:xs) = x <= y && ordered (y:xs)
  ordered _ = True

prop_sortCount :: Int -> [Int] -> Bool
prop_sortCount x xs =
  count x (sort xs) == count x xs

count :: Int -> [Int] -> Int
count x = length . filter (== x)

main :: IO ()
main = do
  check prop_sortOrdered
  chk   prop_sortCount
  where
  chk = check `withBackground`
        [constant "count" count]
```

Figure 1. Full program used to obtain the results in §1.

is enough to create the necessary instances. See §3.2 for how to define instances manually, and why that is desirable in some cases.

Step 2. Call the* `check` *function The function `check` tests properties, and, if counter-examples are found, it generalizes and reports those counter-examples.

Example 1.1 (revisited). Figure 1 shows the program used to obtain the results in §1. □

By default, Extrapolate:

- tests properties for up to 360 value assignments;
- considers side conditions up to size 5.

Extrapolate allows variations of these default settings.

3 How Extrapolate Works

Extrapolate works in three steps:

1. it tests properties searching for counter-examples, if any is found, steps 2 and 3 are performed; (§3.1);
2. it tries to generalize counter-examples by substituting variables for constants (§3.2);
3. it tires to generalize counter-examples by introducing variables subject to side conditions. (§3.3);

Example 3.1. In Haskell, the `Data.List` module declares the function nub, that removes duplicate elements from a list keeping only their first occurrences. Consider the following incorrect property about nub:

```
prop_nubid :: [Int] -> Bool
prop_nubid xs  =  nub xs == xs
```

When Extrapolate's *check* function is applied to the above property, it produces the following output:

```
> check prop_nubid
*** Failed!  Falsifiable (after 3 tests):
[0,0]

Generalization:
x:x:_

Conditional Generalization:
x:xs  when  elem x xs
```

Throughout this section we revisit this example five times to illustrate how each step of Extrapolate works. □

3.1 Searching for counter-examples

To test properties searching for counter-examples, Extrapolate uses LeanCheck [2] which defines the `Listable` typeclass used to generate test values *enumeratively*.

We have used LeanCheck in two other tools reported elsewhere: FitSpec [4] and Speculate [5]. More information on LeanCheck and the `Listable` typeclass can be found in its Haddock documentation [3].

Example 3.1 (1st revisit). When Extrapolate's *check* function is applied to `prop_nubid`, it is tested for each of the list arguments: `[]`, `[0]`, `[0,0]`. The property fails for the third list `[0,0]` and Extrapolate produces its first two lines of output:

```
*** Failed!  Falsifiable (after 3 tests):
[0,0]
```
□

3.2 Unconditional generalization

Listing generalizations After finding a counter-example, Extrapolate lists all of its possible generalizations, from most general to least general, formed by replacing one or more subexpressions with variables. Generality order is not total as some generalizations are incomparable. So we arbitrarily list variables to the left first.

Example 3.1 (2nd revisit). Recall the counter-example to `prop_nubid`: `[0,0]`, or, more verbosely, `0:0:[]`. Its generalizations from most general to least general are: `xs`, `x:xs`, `x:y:xs`, `x:x:xs`, `x:y:[]`, `x:x:[]`, `x:0:xs`, `x:0:[]`, `0:xs`, `0:x:xs`, `0:x:[]` and `0:0:xs`. □

Testing generalizations For each of these generalizations, Extrapolate tests whether the property *always* fails for the configured number of value assignments, reporting the first that does so. Although this process is unsound, as based on testing, it works well in practice (§4). Any variables that appear only once in generalized counter-examples are reported as "_".

Example 3.1 (3rd revisit). The first three generalizations are not counter-examples as there are possible assignments of values such that the property returns `True`:

- `xs` — `prop_nubid []` `= True`
- `x:xs` — `prop_nubid (0:[])` `= True`
- `x:y:xs` — `prop_nubid (0:1:[])` `= True`

The fourth generalization `x:x:xs` is tested and found to be apparently correct: the property fails for all tested assignments of values to the variables `x` and `xs`. Extrapolate produces its third and fourth lines of output:

```
Generalization:
x:x:_
```

Since the variable `xs` appears only once in the generalized counter-example, it is reported as "_". □

3.3 Conditional generalization

Background functions Before trying to discover conditional generalizations we must first decide which functions are allowed to appear in conditions.

The larger the number of functions allowed to appear in the background, the slower Extrapolate will take to produce a conditional generalization. So, we refrain from including a large set such as the entire Haskell Prelude in the background. If the algorithm used is ever refined to something faster we may be able to include a larger set by default (§7).

Each type has a default list of functions to be considered, declared as part of their `Generalizable` typeclass instance:

- for `Bools`, the function `not`;
- for lists, the functions `length` and `elem`.

When using `deriveGeneralizable`, by default, `Eq` instances have `==` and `/=` in the background and `Ord` instances have `<` and `<=` in the background. Extra background functions can be provided using the `` `withBackground` `` combinator (Figure 1). In our experiments (§4), we found it useful to include functions appearing in the properties.

Whenever a property is tested, the aforementioned functions are gathered for all types and sub-types of arguments of the property being tested.

Background constants Constants of the types being tested, obtained from their `Listable` instances, are also included in the background.

Example 3.1 (4th revisit). The background used when testing `prop_nubid :: [Int] -> Bool` is:

- `length :: [Int] -> Int`;

- `elem :: Int -> [Int] -> Bool`;
- `(==), (/=), (<), (<=) :: Int -> Int -> Bool`.

Along with enumerated constants of `Int` and `[Int]` type. Those are computed from the type of the property argument `[Int]`, having `Int` as its only sub-type. □

Side-conditions Extrapolate recursively enumerates expressions formed by type-correct applications of background functions to background constants, variables occurring in the counter-example and expressions. They are bound by the configured maximum size, in number of symbols.

Expressions which have a boolean type are used as candidate side-conditions.

Discarding neutral side-conditions Neutral conditions are those that yield generalizations that are equivalent to a simpler counter-example. For example:

```
[x,x]  when  x == 0
```

is equivalent to simply

```
[0,0]
```

So, any conditions of the form `<var> == <value>` or that are tested and found to be true for only one value for a variable are discarded.

Listing conditional generalizations For each generalization, Extrapolate lists all boolean conditions involving its variables and functions from the *background* up to the configured maximum size.

Testing conditional generalizations The first found to be true in more than 10% variable assignments is reported as a generalization.

This is similar the approach we adopted for Speculate [5]: there we find side-conditions to properties, whereas here we apply side-conditions to generalized counter-examples.

Example 3.1 (5th revisit). For simplicity, let's suppose Extrapolate has been configured to explore conditions of up to size 3.

For the first candidate generalization `xs`, only one candidate condition is listed:

1. `elem 0 xs`

Testing shows that the candidate generalization does *not* always falsify the property under this condition.

For the second candidate generalization `x:xs`, Extrapolate lists 12 candidate conditions:

1. `elem 0 xs`
2. `elem x []`
3. `elem x xs`
4. `0 /= x`
5. `x /= 0`
6. `x /= x`
7. `0 < x`
8. `x < 0`
9. `x < x`
10. `0 <= x`
11. `x <= 0`
12. `x <= x`

The counter-example falsifies the property for all tests under the third condition, `elem x xs`. Extrapolate reports its fifth and sixth lines of output:

```
Conditional Generalization:
x:xs  when  elem x xs
```
□

4 Example Applications and Results

In this section, we use Extrapolate to generalize counter-examples of properties about:

- a calculator library (§4.1);
- integer overflows (§4.2);
- a serializer and parser (§4.3);
- the XMonad window manager (§4.4).

The example applications from §4.1–§4.4 are adapted from Pike [32]. The example applications in §4.1, §4.3 and §4.4 are respectively of relative small, medium and large scale. The example application in §4.2 was originally crafted with the intention to to make the use of enumerative testing libraries, like Extrapolate, infeasible — we shall challenge that intention. Then, in §4.5 we give a summary of performance results for all these applications.

In this section, we evaluate Extrapolate on its own. Comparison with related work can be found in §6.

4.1 A calculator language

In this section, we use Extrapolate to find and generalize counter-examples to a property about the simple calculator language described by Pike [32].

Expressions to be calculated are represented by the Haskell datatype `Exp` and may contain integer constants, addition and division.

```
data Exp = C Int
         | Add Exp Exp
         | Div Exp Exp
```

The function `eval` evaluates `Exp`s and returns a `Maybe` value:

- `Nothing` when the calculation involves has a division by 0;
- `Just` an integer otherwise.

```
eval :: Exp -> Maybe Int
eval (C i)       = Just i
eval (Add e0 e1) = liftM2 (+) (eval e0) (eval e1)
eval (Div e0 e1) =
  let e = eval e1
  in if e == Just 0
       then Nothing
       else liftM2 div (eval e0) e
```

The following function noDiv0[1], returns True when no division by the literal 0 occurs in an expression.

```
noDiv0 :: Exp -> Bool
noDiv0 (C _)          =  True
noDiv0 (Div _ (C 0))  =  False
noDiv0 (Add e0 e1)    =  noDiv0 e0 && noDiv0 e1
noDiv0 (Div e0 e1)    =  noDiv0 e0 && noDiv0 e1
```

Using noDiv0, we define the following test property:

```
\e -> noDiv0 e ==> eval e /= Nothing
```

That is, if an expression contains no literal division by 0, evaluating it returns a Just value.

Using Extrapolate, we find a counter-example and two generalizations:

```
> check $ \e -> noDiv0 e ==> eval e /= Nothing
*** Failed! Falsifiable (after 20 tests):
Div (C 0) (Add (C 0) (C 0))

Generalization:
Div (C _) (Add (C 0) (C 0))

Conditional Generalization:
Div e1 (Add (C 0) (C 0))  when  noDiv0 e1
```

The property fails because it is not enough to test whenever any denominator is a literal zero constant, we should test that any *evaluates* to zero. The generalized counter-examples provide improved information for the programmer. Specifically, constructors that are unrelated to the fault are generalized to variables.

The generate the above conditional generalization we manually included noDiv0 in the list of *background* functions.

The following maximal generalizations (§6) are out-of-reach for the current implementation of Extrapolate:

```
Div e1 e2  when noDiv0 (Div e1 e2)
             && eval e2 == Just 0

Div e1 e2  when noDiv0 e1
             && noDiv0 e2
             && e2 /= (C 0)
             && eval e2 == Just 0
```

Their conditions have 9 and 16 symbols respectively. The search space of conditions of those sizes is to big for Extrapolate.

4.2 Stress test: integer overflows

As an initial motivating example, Pike [32] provides the following program and test property:

[1]Pike [32] originally called this function divSubTerms, we found it clearer to call it noDiv0.

```
type I  =  [Int16]
data T  =  T I I I I I  deriving  Show

toList :: T -> [[Int16]]
toList (T i j k l m) = [i,j,k,l,m]

pre :: T -> Bool
pre t  =  all ((< 256) . sum) (toList t)

post :: T -> Bool
post t  =  (sum . concat) (toList t) < 5 * 256

prop :: T -> Bool
prop t  =  pre t ==> post t
```

The property is incorrect because it fails to account for overflows caused by lists containing very large negatives. Pike [32] describes this as an example where enumerative property-based testing tools are infeasible due to the size of input space.

Infeasible by default And indeed, using the standard way to enumerate integers, Extrapolate, an enumerative property-based testing tool, is not able to cope with finding a counter-example to prop.

By default Extrapolate enumerates integers by alternating between positives and negatives of increasing magnitude:

```
list :: [Int16] = [0, 1, -1, 2, -2, 3, -3, ...]
```

With this standard definition, Extrapolate does not find a counter-example, even when testing a large number of test values — say a million:

```
> check `for` 1000000 $ prop
+++ OK, passed 1000000 tests.
```

Counter-examples will appear only much much later in the enumeration.

Feasible by tweaking enumeration Consider the following alternative definition:

```
list :: [Int16] =
  [ 0, 1, -1, maxBound,   minBound
     , 2, -2, maxBound-1, minBound-1
     , 3, -3, maxBound-2, minBound-2
     , ... ]
```

It alternates not only between positives and negatives but also between extremely small and extremely large values. The intuition is that, maxBound (32767) and minBound (-32768) are extreme values that are *likely* to break the property.

With this simple change, Extrapolate does find a counter-example in less than a second:

```
> check `for` 10000 $ prop
*** Failed! Falsifiable (after 8792 tests):
T [] [] [] [-1] [-32768]
```

Extrapolate does not find a generalization.

Despite using enumerative testing, Extrapolate is able to find faults caused by overflows by tweaking the integer enumeration. This technique can be extended to other enumerative property-based testing libraries, like SmallCheck [35], Lazy SmallCheck [34], Feat [15] or Neat [16].

Conditional generalization The function sum occurs twice in the property. If we include sum as a background function, Extrapolate is able to find the following conditional generalization:

```
> check  `for` 10000  `withConditionSize` 4
>   `withBackground`
>   [constant "sum" (sum :: [Int16] -> Int16)]
>   $ prop
Conditional Generalization:
T xs xs xs ((-1):xs) ((-32768):xs)
  when  0 == sum xs
```

Unfortunately, Extrapolate takes half an hour to find this conditional generalization. We don't expect users of Extrapolate to wait for that long for a conditional generalization – this specific example is a stress test. In §7 we list some avenues of future work to potentially reduce this runtime.

4.3 A serializer and parser

In this section, we apply Extrapolate to the parser and pretty printer for a toy language described by Pike [32]. For brevity, we omit details of the implementation here. It has two main functions:

```
show' :: Prog -> String
read' :: String -> Prog
```

The serializer code (show') has approximately 100 lines of code. The parser code (read') has approximately 200 lines of code. The parser includes a bug that switches the arguments of conjunction expressions.

When we test the property that serializing followed by parsing is an identity, Extrapolate reports a counter-example along with generalizations:

```
> check $ \e -> read' (show' e) == e
*** Failed! Falsifiable (after 96 tests):
Lang [] [Func (Var "a")
              [And (Int 0) (Bool False)] []]

Generalization:
Lang _ (Func _ (And (Int _) (Bool _):_) _:_)

Conditional Generalization:
Lang _ (Func _ (And e f:_) _:_)  when  e /= f
```

The reported conditional generalization clearly characterizes a set of failing cases: the property fails whenever there is an And expression with different operands. This characterization strongly hints at a programming error failing to distinguish operands correctly.

4.4 XMonad

XMonad [39] is a tiling window manager written in roughly 1700 lines of Haskell code. The XMonad developers defined over a hundred test properties.

In this section, we use Extrapolate to find an artificial bug introduced by Pike [32] in XMonad.

The function XMonad has a function

```
removeFromWorkspace ws =
  ws { stack = stack ws >>= filter (/=w) }
```

which removes the current item (or window) from the workspace.

The bug As described by Pike [32], we introduce an artificial bug, replacing /= by == simulating a typo:

```
removeFromWorkspace ws =
  ws { stack = stack ws >>= filter (==w) }
```

The property The following property prop_delete is one of XMonad's original test properties.

```
prop_delete x =
  case peek x of
    Nothing -> True
    Just i  -> not (member i (delete i x))
  where _ = x :: T
```

A counter-example In a regular enumerative property-based testing tool, we would get the following minimal counter-example for prop_delete:

```
> check prop_delete
Failed! Falsifiable (after 15 tests):
StackSet
  { current = Screen
      { workspace = Workspace
          { tag = 0
          , layout = 0
          , stack = Just ( Stack { focus = 'a'
                                 , up = ""
                                 , down = ""
                                 } )
          }
      , screen = 0
      , screenDetail = 0
      }
  , visible = []
  , hidden = []
  , floating = fromList []
  }
```

A generalization When we pass `prop_delete` to Extrapolate, we instead get the following output (again indented to improve readability):

```
> check prop_delete
*** Failed! Falsifiable (after 15 tests):
StackSet
  (Screen
    (Workspace 0 0 (Just (Stack 'a' "" "")))
    0 0)
  [] [] (fromList [])

Generalization:
StackSet
  (Screen (Workspace _ _ (Just _)) _ _) _ _ _
```

Compare the non-generalized counter-example above with its generalization. We can see quite clearly which parts of are actually related to the fault: what characterizes the failing cases is that an optional third argument (of type `Maybe Stack`) is present in the argument workspace. Or; as Pike [32] explained "it turns out what matters is having a `Just` value, which is the stack field that deletion works on!"

4.5 Performance

Our tool and examples were compiled using `ghc -O2` (Glasgow Haskell Compiler [40] version 8.0.1) under Linux. The platform was a PC with a 2.2Ghz 4-core processor and 8GB of RAM.

Performance results are summarized in Table 1. For all example applications, Extrapolate takes up to a second to produce unconditional generalizations. For the sort, calculator and parser examples, Extrapolate also takes up to a second to produce conditional generalizations. For the XMonad and overflow applications, it takes 20 seconds and 24 minutes to when considering conditional generalizations. The Table also includes results for other tools, to be discussed in §6.

5 Related Work

Since the introduction of QuickCheck [10–12], several other property-based testing libraries and techniques have been developed. For Haskell, we can cite SmallCheck, Lazy SmallCheck [34, 35] and Feat [15]. For Curry, there's EasyCheck [9]. For Erlang, there's PropEr [31] and QuviQ QuickCheck [1]. For CLEAN, there's GAST [26]. For Standard ML, there's QCheck [27]. For Ocaml, there's a port of QuickCheck [38].

Tracing and step-by-step evaluation A lot of research has been done on tracing and step-by-step evaluation. In the realm of Haskell, we can note tools such as Freja [29, 30], Hat [7, 8, 41] and Hood [17]. These tools facilitate the process of locating faults in programs. Extrapolate on other hand does not directly improve this process, but rather gives the programmer improved results to inform it. Except when a generalized counter-example makes it very obvious where the fault is, Extrapolate does not replace tools like Freja, Hat and Hood: it is more of a complement. Claessen et al. [13] elaborate on the relation between property-based testing and tracing.

Property discovery QuickSpec [14, 37] and Speculate [5] are tools capable of automatically conjecturing properties when given a collection of Haskell functions. Like Extrapolate, these tools rely mainly on testing to achieve their results. Extrapolate does not conjecture properties like these tools, but its generalized counter-examples can be *seen* as properties about faults. In Example 1.1, the counter-example `x (x:x:_)` can be seen as the following property:

```
\x xs -> not $ prop_sortCount x (x:x:xs)
```

Within this view, Extrapolate's generalization aims to find largest test space in which the negation of the property under test succeeds. Extrapolate actually uses some of the internal machinery of Speculate.

Program Synthesis Magic Haskeller [20–23] and IGOR II [18, 19, 24] are systems for program synthesis using inductive functional programming techniques [25]. They are able to produce programs based on a limited list of input-output bindings and a set of background functions. Similarly, there are PROGOL [28], FOIL [33] and GOLEM [6] for logic programs. There is a potential for the application of these systems and their techniques to generalize counter-examples Based on which test inputs pass or fail, generate a program to describe a set of counter-examples.

Lazy SmallCheck Lazy SmallCheck [34, 35] is a property-based testing tool that uses laziness to prune the search space. Before testing fully-defined test values, it tries partially defined test values: if a property fails or passes for a partially defined value, more defined variations of that value need not be tested. As a side-effect of this test strategy, Lazy SmallCheck is able to return partially defined values as counter-examples (see Table 3). These can be read as generalized counter-examples. In §6 we compare results of Lazy SmallCheck to those of Extrapolate.

SmartCheck Because QuickCheck tests values randomly, it does not always return small counter-examples, but relies on shrinking [10] to derive smaller counter-examples from larger ones. SmartCheck [32] is an extension to QuickCheck that provides two improvements: a better algorithm for shrinking and generalization of counter-examples. SmartCheck's generalization algorithm performs both universal and existential quantification but does not allow repeated variables. SmartCheck is perhaps the most closely related tool to Extrapolate, and Lee's paper [32] inspired the work reported here. In §6 we compare results of SmartCheck to those of Extrapolate.

Table 1. Summary of results for five different applications, testing properties with Extrapolate, SmartCheck and Lazy SmallCheck; #-symbols = #-constants + #-variables; #-constants = number of constants in the reported counter-examples; #-variables = number of variables in the reported counter-examples; generality = how general is the counter-example (○ = least general; ● = most general; × = degenerate); runtime = rounded elapsed time; space = peak memory residency.

Example & Property	Tool	#-symbols	#-constants	#-variables	generality	Runtime	Memory
Faulty `sort` (§1)	Ungeneralized	6	6	0	○	< 1s	20MB
`\x xs -> count x (sort xs)`	Lazy SmallCheck	6	6	0	○	< 1s	33MB
`== count x xs`	SmartCheck (min)	6	5	1	◔	< 1s	22MB
	SmartCheck (median)	12	11	1	◔	< 1s	22MB
	Extrapolate	6	2	4	◕	< 1s	22MB
	Extrapolate (side)	8	4	4	●	< 1s	22MB
Faulty `noDiv0` (§4.1)	Ungeneralized	8	8	0	○	< 1s	20MB
`\e -> noDiv0 e`	Lazy SmallCheck	8	7	1	◔	< 1s	33MB
`==> eval e /= Nothing`	SmartCheck (min/median)	7	6	1	×	< 1s	22MB
	Extrapolate	8	7	1	◔	< 1s	22MB
	Extrapolate (side)	10	8	2	◕	< 1s	22MB
Integer Overflow (§4.2)	Ungeneralized	10	10	0	○	< 1s	30MB
`\t -> pre t -> post t`	Lazy SmallCheck	– c.e. not found –				60m 00s	36MB
	SmartCheck (min)	10	5	5	×	< 1s	22MB
	SmartCheck (median)	16	11	5	×	< 1s	22MB
	Extrapolate	10	10	0	○	< 1s	30MB
	Extrapolate (side)	15	9	6	◔	24m 23s	50MB
Faulty parser (§4.3)	Ungeneralized	17	17	0	○	< 1s	22MB
`\e -> read' (show' e) == e`	Lazy SmallCheck	17	17	0	○	12s	36MB
	SmartCheck (min)	17	17	0	○	< 1s	23MB
	SmartCheck (median)	27	27	0	○	< 1s	23MB
	Extrapolate	14	7	7	◔	1s	26MB
	Extrapolate (side)	16	7	9	◕	1s	26MB
Faulty XMonad (§4.4)	Ungeneralized	16	16	0	○	< 1s	5MB
`prop_delete`	SmartCheck	13	9	4	◔	–	–
	Extrapolate	12	4	8	◕	< 1s	29MB
	Extrapolate (side)	– gen. not found –				20s	35MB

6 Comparative Evaluation

In this section, we revisit examples from §4 comparing results of Extrapolate with results of SmartCheck and Lazy SmallCheck: the two other tools for Haskell able to generalize counter-examples.

Criteria The following paragraphs define some of the criteria used in evaluating results of different tools: *generality* and *degenerate counter-examples*.

Generality The more general the counter-example, the better it is. We consider a generalized test-case description more general than another if:

- it strictly subsumes another;
- it includes a larger set of failing test cases.

We say that a generalization is maximal when no more general description exists.

Degenerate counter-examples When a reported generalized counter-example is *too general* and includes inputs that are *not* counter-examples, we say it is *degenerate*. Later in this section, we shall see examples of this.

Representatives and Median values SmartCheck randomly tests values and does not usually report the same counter-examples. In Tables 2–4 counter-examples for SmartCheck

Table 2. Counter-examples for the `count` property of sort (Example 1.1 from §1) reported by Extrapolate, SmartCheck and LazySmallCheck.

Tool	Counter-example
Ungeneralized	`0 [0,0]`
Lazy SmallCheck	`0 [0,0]`
SmartCheck (min)	`4 (4:4:x0)`
SmartCheck	`9 (8:17:9:9:x0)`
Extrapolate	`x (x:x:_)`
Extrapolate (side)	`x xs when count x xs > 1`

Table 3. Counter-examples for the property involving `noDiv0` (§4.1) reported by Extrapolate, SmartCheck and Lazy SmallCheck. Most generalized counter-examples reported by SmartCheck are degenerate.

Tool	Counter-example
Ungeneralized	`Div (C 0) (Add (C 0) (C 0))`
Lazy SmallCheck	`Div (C _) (Add (C 0) (C 0))`
SmartCheck	`Div x0 (Add (C (-5)) (C 5))`
Extrapolate	`Div (C _) (Add (C 0) (C 0))`
Extrapolate (side)	`Div e1 (Add (C 0) (C 0))` `when noDiv0 e1`

are median representatives only. In Table 1, the values for numbers of symbols, constructors and variables are the median of 1000 sample runs.

Summary Table 1 summarizes all results. For *most* examples, Extrapolate gives *more general* results than either Lazy SmallCheck or SmartCheck. For *all* examples, Extrapolate gives results *at least as general* as Lazy SmallCheck and SmartCheck.

All tools usually report their results within a second, a reasonable time when testing a property. Extrapolate is slower to produce conditional generalizations on the XMonad and overflow examples, taking respectively 20s and 24m. There are no significant differences in memory use.

Faulty sort (§1) See Table 2. The counter-example reported by Extrapolate has the same number of symbols as the one reported by Lazy SmallCheck. Lazy SmallCheck is not able to report a generalization as the property being tested is not lazy. Extrapolate is able to generalize the tail and the initial elements of the list whereas SmartCheck[2] is only able to generalize the tail. With the function `count` in the background, Extrapolate reports a *maximal* generalization – there is no more general description of the failing cases.

[2]Since SmartCheck only tries to generalize the first argument of properties (a design choice), the property had to be uncurried for it to report a generalization.

Table 4. Counter-examples for the property about integer overflows (§4.2) reported by Extrapolate, SmartCheck and Lazy SmallCheck. The generalized counter-example reported by SmartCheck is degenerate.

Tool	Counter-example
Ungeneralized	`T [] [] [] [-1] [-32768]`
Lazy SmallCheck	— timeout: c.e. not found —
SmartCheck	`T x3 (-21874:x0) x2 (-12585:[]) x1`
Extrapolate	`T [] [] [] [-1] [-32768]`
Extrapolate (side)	`T xs xs xs ((-1):xs) ((-32768):xs)` `when 0 == sum xs`

Calculator and faulty `noDiv0` (§4.1) See Table 3. Extrapolate and Lazy SmallCheck report the same generalization. SmartCheck uses randomization, so its results may vary between runs. The generalizations it reports in 96% of runs are *too* general – as they include values that are *not* counter-examples if we read `==>` as a logical implication. Indeed they are not failing test cases. Concerning the proposed counter-example reported in Table 3:

```
> let prop e = noDiv0 e ==> eval e /= Nothing
> let x0 = (Div (C 0) (C 0))
> prop (Div x0 (Div (C (-5)) (C 5)))
True
```

With the precondition falsified, the property holds.

Integer overflow (§4.2) Lazy SmallCheck is not able to find a counter-example even after running for an hour, replicating the result reported by Pike [32]. We did not use a customized integer enumeration (§4.2). Although Extrapolate fails to report an unconditional generalization, it reports a conditional generalization. SmartCheck reports a degenerate generalization that includes inputs that are not counter-examples. See:

```
> let x0 = [21874]
> let x1 = []
> let x2 = []
> let x3 = []
> prop (T x3 (-21874:x0) x2 (-12585:[]) x1)
True
```

Faulty parser (§4.3) See Table 5. Neither Lazy SmallCheck nor SmartCheck is able to report a generalization. Extrapolate is able to report both an unconditional generalization and to improve it in a further conditional generalization. Extrapolate reports counter-examples that are smaller than counter-examples reported by other tools.

Table 5. Counter-examples for the parser property (§4.3) reported by Extrapolate, SmartCheck and Lazy SmallCheck.

Tool	Counter-example
Ungeneralized (full record syntax)	`Lang {modules = [], funcs = [Func {fnName = Var "a", args = [And (Int 0) (Bool False)], stmts = []}]}`
Ungeneralized (simplified)	`Lang [] [Func (Var "a") [And (Int 0) (Bool False)] []]`
Lazy SmallCheck	`Lang [] [Func (Var "a") [And (Int 0) (Bool False)] []]`
SmartCheck	`Lang [] [Func (Var "U") [] [Return (And (Bool False) (Int 0))]]`
Extrapolate	`Lang _ (Func _ (And (Int _) (Bool _):_) _:_)`
Extrapolate (with side condition)	`Lang _ (Func _ (And e f:_) _:_)  when  e /= f`

Table 6. Counter-examples for `prop_delete` from XMonad (§4.4) reported by Extrapolate and SmartCheck. The SmartCheck counter-example is the one reported by Pike [32] in the original SmartCheck paper.

Tool	Counter-example
Ungeneralized	`StackSet (Screen (Workspace 0 0 (Just (Stack 'a' "" ""))) 0 0) [] [] (fromList [])`
SmartCheck	`StackSet (Screen (Workspace x0 (-1) (Just x1)) 1 1) x2 x3 (fromList [])`
Extrapolate	`StackSet (Screen (Workspace _ _ (Just _)) _ _) _ _ _`

Faulty XMonad (§4.4) See Table 6. SmartCheck[3] and Extrapolate give counter-examples of almost the same number of symbols, but the Extrapolate counter-example has fewer constant constructors and more variables. SmartCheck always assigns variable names whereas Extrapolate uses "_" for unrepeated variables, making it more immediately apparent where component values do not matter.

Versions used We have used the latest version possible for each tool:

- Lazy SmallCheck: version of 2014-07-07 — compiled with GHC 7.8.4;
- SmartCheck: version 0.2.4 — compiled with GHC 8.0.2;
- Extrapolate: version 0.2.3 — compiled with GHC 8.0.2.

7 Conclusions and Future Work

In summary, we have presented a tool that is able to generalize counter-examples of functional test properties. As set out in §2 and §3, Extrapolate enumerates and tests generalizations reporting the one that fails all tests. We have demonstrated in §4 Extrapolate's applicability to a range of examples. And after reviewing some related work §5, we have compared in §6 Extrapolate results with those reported by other tools.

Conclusions

Value of reported laws The conjectured generalizations reported by Extrapolate are surprisingly accurate in practice, despite their inherent uncertainty in principle. They can provide helpful insights into the source of faults. Allowing repeated variables and side-conditions makes possible the discovery of generalizations previously unreachable by other tools that provide similar functionality such as Lazy SmallCheck [34] and SmartCheck [32].

Ease of use Extrapolate requires no more programming effort than a regular property-based testing tool such as QuickCheck [11] or SmallCheck [35]. If only standard Haskell datatypes are involved, no extra `Listable` or `Generalizable` instances are needed. If user-defined data types can be freely enumerated without a constraining data invariant, instances can be automatically derived.

Future Work We note a number of avenues for further investigation that could lead to improved versions of Extrapolate or similar tools.

Type-by-type generalization Although sufficiently fast for the purposes of testing, the current algorithm to find counter-examples is very naïve: it generalizes sub-expressions to variables of several types at once causing an unnecessary combinatorial explosion. We believe runtime could be reduced by generalizing type-by-type.

First generalizing with one variable per type An interesting observation used in our previous work [5] and the work of Smallbone et al. [37] could be used to speed-up Extrapolate. For a property with several variables per type to be true, its one-variable-per-type instance should be true as well, for example:

$$\forall x\ y\ z.(x+y)+z = x+(y+z) \Rightarrow \forall x.(x+x)+x = x+(x+x)$$

[3] Due to time constraints we have not yet tested Lazy SmallCheck or SmartCheck on this example. The SmartCheck counter-example is as reported by Pike [32] in the original paper about SmartCheck.

The same is true for generalized counter-examples: if lists of `x:y:xs` always falsify a property, then lists of the form `x:x:xs` should as well. Based on this observation, we can test one-variable-per-type generalizations first, potentially reducing the generalization search space.

Generalizing from several counter-examples The current version of Extrapolate bases its generalizations on a single counter-example. As mentioned in §5, it may be possible to use techniques from inductive functional programming [25] base its generalizations on several counter-examples. This could potentially reduce the time needed to produce generalizations.

Parallelism As a way to improve performance, particularly when dealing with costly test functions such as in the XMonad (§4.4) or overflow (§4.2) examples, we could parallelize the testing of different enumerated generalizations among multiple processors.

Multiple generalizations Reported generalizations are derived from initial counter-examples. After finding a generalization, Extrapolate could search for other counter-examples and report additional generalizations. These could hopefully be of additional help in finding the source of faults.

Automatically include functions occurring in properties in the background In several examples (§4), we provided functions present in the property as background for side conditions improving generalized counter-examples. This could be done automatically to improve out-of-the-box results.

Availability

Extrapolate is freely available with a BSD3-style license from:

- https://hackage.haskell.org/package/extrapolate
- https://github.com/rudymatela/extrapolate

This paper describes Extrapolate as of version 0.2.3.

Acknowledgements

We thank Tim Atkinson and Lee Pike for their comments on earlier drafts.

Rudy Braquehais is supported by CAPES, Ministry of Education of Brazil (Grant BEX 9980-13-0).

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with QuviQ QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2–10.

[2] Rudy Braquehais. 2016–2017. LeanCheck. https://github.com/rudymatela/leancheck. (2016–2017).

[3] Rudy Braquehais. 2016–2017. LeanCheck's Documentation. https://hackage.haskell.org/package/leancheck/docs/Test-LeanCheck.html. (2016–2017).

[4] Rudy Braquehais and Colin Runciman. 2016. FitSpec: refining property sets for functional testing. In *Haskell'16*. ACM, 1–12.

[5] Rudy Braquehais and Colin Runciman. 2017. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Haskell'17 (to appear)*. ACM.

[6] R. Mike Cameron-Jones and J. Ross Quinlan. 1994. Efficient Top-down Induction of Logic Programs. *SIGART Bulletin* 5, 1 (Jan 1994), 33–42.

[7] Olaf Chitil, Maarten Faddegon, and Colin Runciman. 2016. A Lightweight Hat: Simple Type-Preserving Instrumentation for Self-Tracing Lazy Functional Programs. In *IFL 2016*. ACM, 1–14.

[8] Olaf Chitil, Colin Runciman, and Malcolm Wallace. 2001. Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *IFL 2000*. Springer, 176–193.

[9] Jan Christiansen and Sebastian Fischer. 2008. EasyCheck – Test Data for Free. In *Functional and Logic Programming*. Springer, 322–336.

[10] Koen Claessen. 2012. Shrinking and Showing Functions. In *Haskell'12*. ACM, 73–80.

[11] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00*. ACM, 268–279.

[12] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. In *Haskell '02*. ACM, 65–77.

[13] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. 2003. Testing and Tracing Lazy Functional Programs Using QuickCheck and Hat. In *AFP'03*. Springer, 59–99.

[14] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010*. Springer, 6–21.

[15] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Haskell'12*. ACM, 61–72.

[16] Jonas Duregård. 2016. *Automating Black-Box Property Based Testing*. Ph.D. Dissertation. Chalmers University of Technology.

[17] Andy Gill. 2001. Debugging Haskell by Observing Intermediate Data Structures. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 1. Also in Haskell'00.

[18] Martin Hofmann and Emanuel Kitzelmann. 2010. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *PEPM'10*. ACM, 93–100.

[19] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. 2010. Porting Igor II from Maude to Haskell. In *Approaches and Applications of Inductive Programming (AAIP'09). Revised Papers*. Springer, 140–158.

[20] Susumu Katayama. 2004. Power of Brute-Force Search in Strongly-Typed Inductive Functional Programming Automation. In *PRICAI 2004: Trends in Artificial Intelligence (LNAI 3157)*. Springer, 75–84.

[21] Susumu Katayama. 2005. Systematic search for lambda expressions. In *TFP 2005*.

[22] Susumu Katayama. 2010. Recent Improvements of MagicHaskeller. In *AAIP 2009. Revised Papers (LNCS 5812)*. Springer, 174–193.

[23] Susumu Katayama. 2012. An Analytical Inductive Functional Programming System That Avoids Unintended Programs. In *PEPM'12*. ACM, 43–52.

[24] Emanuel Kitzelmann. 2007. Data-driven induction of recursive functions from input/output-examples. In *Approaches and Applications of Inductive Programming (AAIP'07)*. 15–26.

[25] Emanuel Kitzelmann. 2010. Inductive Programming: A Survey of Program Synthesis Techniques. In *AAIP'09*. Springer, 50–73.

[26] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. 2003. GAST: Generic Automated Software Testing. In *Implementation of Functional Languages*. Lecture Notes in Computer Science, Vol. 2670. Springer, 84–100.

[27] Christopher League. 2007–2011. QCheck. https://github.com/standardml/qcheck. (2007–2011).

[28] Stephen Muggleton. 1995. Inverse entailment and progol. *New Generation Computing* 13, 3 (Dec 1995), 245–286.

[29] Henrik Nilsson. 1998. *Declarative Debugging for Lazy Functional Languages*. Ph.D. Dissertation.

[30] Henrik Nilsson and Jan Sparud. 1997. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering* 4, 2 (Apr 1997), 121–150.
[31] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*. ACM, 39–50.
[32] Lee Pike. 2014. SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization. In *Haskell'14*. ACM, 59–70.
[33] J. R. Quinlan and R. M. Cameron-Jones. 1993. FOIL: A midterm report. In *ECML-93: European Conference on Machine Learning*. Springer, 1–20.
[34] Jason S. Reich, Matthew Naylor, and Colin Runciman. 2013. Advances in Lazy SmallCheck. In *IFL'13*. Springer, 53–70.
[35] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell'08*. ACM, 37–48.
[36] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Haskell'02*. ACM, 1–16.
[37] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017).
[38] Roman Sokolov. 2008–2015. Ocaml-QuickCheck. https://github.com/camlunity/ocaml-quickcheck. (2008–2015).
[39] Don Stewart and Spencer Sjanssen. 2007. XMonad. In *Haskell '07*. ACM, 119–119.
[40] The GHC Team. 1992–2017. The Glasgow Haskell Compiler. https://www.haskell.org/ghc/. (1992–2017).
[41] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. 2001. Multiple-View Tracing for Haskell: a New Hat. In *Haskell'01*. ACM, 182–196.

Biorthogonality for a Lazy language

Daniel Fridlender
Alejandro Gadea
fridlend@famaf.unc.edu.ar
gadea@famaf.unc.edu.ar
FaMAF
Univ. Nacional de Córdoba
Córdoba, Argentina

Miguel Pagano
Leonardo Rodríguez
pagano@famaf.unc.edu.ar
lrodrig2@famaf.unc.edu.ar
FaMAF
Univ. Nacional de Córdoba
Córdoba, Argentina

Abstract

In this paper we extend the technique of biorthogonality for proving the correctness of a compiler for a lazy language, which, as far as we know, has only been preliminary explored by Rodríguez [17]. One of the technical difficulties arising in this context is the sharing of the heap between the realizer and the test. We considerably extend the language considered by Rodríguez by adding natural numbers and products, which may be used as a test case for tackling other data-types.

CCS Concepts • **Theory of computation** → *Logic and verification*; *Denotational semantics*;

Keywords lazy language, correct compiler, biorthogonality

ACM Reference format:
Daniel Fridlender, Alejandro Gadea, Miguel Pagano, and Leonardo Rodríguez. 2017. Biorthogonality for a Lazy language. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Acknowledgments

This material is based upon work supported by the Agencia Nacional de Promoción Científica y Tecnológica under Grant No. 2014-3508. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Agencia Nacional de Promoción Científica y Tecnológica.

1 Introduction

In this article we extend the use of biorthogonality to prove the correctness of a compiler for a core lazy language targeting the Sestoft abstract machine [19].

Biorthogonality [13, 14] is a modular technique useful for transferring compositional principles (as those involved in denotational models) to operational settings, which often lack them. In the case of the correctness of a compiler with respect to a denotational semantics, one seeks to prove that the low-level code (generated by the compiler) "implements" the mathematical entity denoted by the high-level expression being compiled. Following the novel reading of realizability by Krivine [9], Benton and Hur [1] observed that one can use biorthogonality to lift a relation between realizers and values of basic types (the realizers are code fragments which obviously implement denotations at the basic type) to realizer for higher-order types. This logical relation is parameterized on some notion of observation on configurations of the runtime environment. In the case of call-by-value languages (considered in [1, 7, 8]) the configurations can be split into the realizer and the tests; the same situation of a clear cut between realizers and tests arises in the case of call-by-name [16].

Our contribution consists in extending the technique of biorthogonality for proving the correctness of a compiler for a lazy language, which, as far as we know, has only been preliminary explored by Rodríguez [17]. One of the technical difficulties arising in this context is the sharing of the heap between the realizer and the test.

Related Work The first proof of correctness given by McCarthy and Painter [12] is celebrating its fiftieth anniversary [6]. More recently, fuelled by the advance of proof-assistants, the trade shifted to the mechanization of correctness proofs. One major milestone is the certified compiler for the C language developed by the CompCert project [11]. There have been several proposal for proving the correctness of functional languages: Chlipala [4] presents a certified compiler for the simply typed lambda calculus targeting assembly code; Benton and Hur [1] proves the correctness of the SECD machine as an execution environment for a call-by-value language. In a recent development, Tan et al. [20] documented a realistic compiler for CakeML targeting several architectures. As far as we know, correctness of compilers for lazy languages is a less explored area; the closest related works we can mention are the proof of adequacy between the denotational and operational semantics given by Launchbury [10] and its recent mechanization, and slightly correction, carried on by Breitner [3] in Isabelle.

In previous works, we have used the technique of biorthogonality to prove the correctness of a compiler for a core functional language using normal order evaluation [16] and also

with paper note.
Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

for a call-by-value one [5]. Rodríguez [17] has explored the use of biorthogonality for a lazy language with only one basic type. In this paper we considerably extends the language by adding naturals numbers and products.

2 Realizability for Lazy Evaluation

Let us briefly explain the use of realizability and biorthogonal operators for proving the correctness of a compiler for a call-by-name language; then we analyse the differences as we switch to a call-by-need strategy with sharing.

The basic intuition on using realizability for compiling correctness is that a piece of low-level code is a *realizer* for a high-level type if running the code finishes in a configuration with a value of the high-level type. For example, a code computing a numeral can be thought as a realizer of the type **int**. By using refinement types [7], one gets more informative realizers and can decide if a low-level code computes some particular numeral. In Krivine realizability one also takes into account the environment on which the low-level code is executed (i.e., not only the code segment but also the stack and the heap); the environment is thought as a *test* and a code has to satisfy every test to be counted as a realizer. In our case, a realizer satisfies a test if the whole configuration obtained by combining them computes the value of a high-level expression, but several results about realizers and tests can be developed without choosing a particular notion of satisfaction. These results provide means to compose low-level codes in a principled way.

Instead of starting with tests, Benton and Hur [1] identified what they called primitive realizers and defined tests and realizers by means of the orthogonal operators associated with any binary relation [2, p. 122]. Moreover, their primitive realizers are not realizers for some type, but are realizers for some denotational value of the given type. In the case of proving correctness of a compiler targeting the KAM, the test is the stack of the machine and the realizer is a closure. The stack contains the arguments to be consumed by the current closure; but if an argument is used more than once, then it will be computed as many times as it is needed. For example, the compilation of $(\lambda x\,.\,x + x)\ t$ will be executed by pushing the code corresponding of t to the stack and that code will be executed twice.

In a machine with lazy evaluation, where arguments are also evaluated only on demand and at most once, the stack contain pointers and the heap map pointers to closures which will be updated when the value of an argument has been found. Since realizers also have pointers, we need a more contrived definition of the satisfiability relation to account for the situation when they share common pointers.

3 High-Level Language

Our source language is the lambda calculus extended with constansts, binary operations, pairs, and conditional (in the form of testing for zero). It is mainly the same language of Launchbury [10] and Sestoft [19], although we use de Bruijn indices for variables and our non recursive let-expressions bind only one variable. We use $\bar{n}$ for the variable n, the constant m is denoted with $\lfloor m \rfloor$. Notice that the operand of an application is always a variable.

Definition 3.1 (Terms)**.**

$$\begin{aligned} t, t' \in \mathit{Term} \ ::= \ & \bar{n} \mid \lambda\, t \mid t\ \bar{n} \mid \diamond \mid \mathit{let}\ t\ \mathit{in}\ t' \\ & \mid (t, t') \mid \mathit{fst}\ \bar{n} \mid \mathit{snd}\ \bar{n} \\ & \mid \lfloor n \rfloor \mid t \odot t' \mid \mathit{ifz}\ t\ \mathit{then}\ t'\ \mathit{else}\ t'' \end{aligned}$$

In contrast with Launchbury and Sestoft who worked with an untyped language, we restrict our work to well-typed expressions over the following types.

Definition 3.2 (Types)**.**

$$\theta, \theta' \in \mathit{Type} \ ::= \ \mathbf{unit} \mid \mathbf{int} \mid \theta \to \theta' \mid \theta \times \theta'$$

Contexts assign types to variables; since we use de Bruijn indices, contexts, we use π as a meta-variable for them, are lists of types. The notation $\_ :: \_$ corresponds to append: our contexts grow to the left, thus keeping a close connection between the index and the position of its type in the context. In the spirit of turning a failure in a list of successes, we assume that the lookup function, denoted by $\_ \cdot \_$, returns the empty set if the index is not defined and a singleton when it is defined. A type judgment $\pi \vdash t : \theta$ says that t has type θ under the context π. A judgment is valid if one can build a derivation using the following rules.

Definition 3.3 (Typing Rules)**.**

$$\textsc{Abs}\ \frac{\theta :: \pi \vdash t : \theta'}{\pi \vdash \lambda\, t : \theta \to \theta'} \qquad \textsc{App}\ \frac{\pi \vdash t : \theta \to \theta'}{\pi \vdash t\ \bar{n} : \theta'}\ \pi \cdot n = \{\theta\}$$

$$\textsc{Var}\ \frac{}{\pi \vdash \bar{n} : \theta}\ \pi \cdot n = \{\theta\} \qquad \textsc{Unit}\ \frac{}{\pi \vdash \diamond : \mathbf{unit}}$$

$$\textsc{Let}\ \frac{\pi \vdash t : \theta \qquad \theta :: \pi \vdash t' : \theta'}{\pi \vdash \mathit{let}\ t\ \mathit{in}\ t' : \theta'}$$

$$\textsc{Pair}\ \frac{\pi \vdash t : \theta \qquad \pi \vdash t' : \theta'}{\pi \vdash (t, t') : \theta \times \theta'}$$

$$\textsc{Fst}\ \frac{\pi \vdash \bar{n} : \theta \times \theta'}{\pi \vdash \mathit{fst}\ \bar{n} : \theta} \qquad \textsc{Snd}\ \frac{\pi \vdash \bar{n} : \theta \times \theta'}{\pi \vdash \mathit{snd}\ \bar{n} : \theta'}$$

$$\textsc{Int}\ \frac{}{\pi \vdash \lfloor m \rfloor : \mathbf{int}} \qquad \textsc{BinOp}\ \frac{\pi \vdash t : \mathbf{int}\ \pi \vdash t' : \mathbf{int}}{\pi \vdash t \odot t' : \mathbf{int}}$$

$$\textsc{Ifz}\ \frac{\pi \vdash t : \mathbf{int} \qquad \pi \vdash t' : \theta \qquad \pi \vdash t'' : \theta}{\pi \vdash \mathit{ifz}\ t\ \mathit{then}\ t'\ \mathit{else}\ t'' : \theta}$$

Denotational Semantics

A set-theoretic semantics is enough for our simply-typed language, because it lacks a recursion operator. The denotation of each type is a set. The semantics is the *intended* one:

integers expressions are interpreted as integers; as usual, unit, products, and functional types get their semantics from the cartesian structure of the category.

Definition 3.4 (Semantic of types)**.**

$$[\![\,\mathbf{unit}\,]\!] = \{\diamond\} \quad [\![\,\theta \to \theta'\,]\!] = [\![\,\theta\,]\!] \to [\![\,\theta'\,]\!]$$
$$[\![\,\mathbf{int}\,]\!] = \mathbb{Z} \qquad [\![\,\theta \times \theta'\,]\!] = [\![\,\theta\,]\!] \times [\![\,\theta'\,]\!]$$

The semantics of contexts are also given as products; we say that a tuple $\rho \in [\![\,\pi\,]\!]$ is an *environment*.

Definition 3.5 (Semantic of contexts)**.**

$$[\![\,[]\,]\!] = \{\diamond\} \qquad [\![\,\theta :: \pi\,]\!] = [\![\,\theta\,]\!] \times [\![\,\pi\,]\!]$$

We give meaning only to well-typed terms –what Reynolds [15] called *intrinsic semantics*: by recursion on derivations, we assign to each typing derivation with conclusion $\pi \vdash t : \theta$ a total function from $[\![\,\pi\,]\!]$ to $[\![\,\theta\,]\!]$. The denotational semantics is given in Fig. 1. For conciseness, we elide the derivation and write the context and the type as sub-indices. We extend tuples to the left (assuming (d, ρ) is the same as its flattening) and write $\rho \swarrow n$ the n-th projection from the tuple ρ, when using it we assume $n < |\rho|$.

4 The Abstract Machine

Launchbury proposed a natural semantics for lazy evaluation in order to get a more pleasant setting to study operational properties. Soon after, Sestoft [19] took Launchbury's semantics as the basis for deriving an abstract machine. Sestoft's machine is a variant of the Krivine Abstract Machine, with an additional component, the *heap*, wich is used to share evaluations. A configuration is a quadruple (Γ, i, η, s) consisting of the *heap* Γ, the *instruction* i, the *environment* η, and the *stack* s.

While in KAM both the environment and the stack are lists of *closures* (pairs of an instruction and environments), in Sestoft's machine they contain *pointers*; the heap is a map from those pointers to closures. Besides pointers, the stack might also contain *markers* which are used to update the heap once a value is found, to compute a binary operation, and for projecting from pairs.

Definition 4.1 (Components)**.**

$$\begin{array}{lll}
iv \in \mathit{Val} & ::= & \texttt{Grab} \triangleright i \mid \texttt{Unit} \mid \texttt{Pair}\,(i, i') \mid \underline{n} \\
i, i' \in \mathit{Code} & ::= & iv \mid \texttt{Access}\ n \mid \texttt{BOp}\ i\ i' \\
 & \mid & \texttt{Push}\ n \triangleright i \mid \texttt{Let}\ i \triangleright i' \\
 & \mid & \texttt{Snd}\ n \mid \texttt{Fst}\ n \mid \texttt{IfZ}\ i\ i'\ i'' \\
p, q \in \mathit{Pointer} & & \\
m \in \mathit{Marker} & ::= & \sharp p \mid \sharp_1 p \mid \sharp_2 p \mid ?p \mid \odot p \\
k \in \mathit{Cont} & ::= & \texttt{IfZ}\,\square\,(i, i') \mid \texttt{BOp}\,\square\,i \mid \texttt{BOp}\,\underline{n}\,\square \\
\alpha \in \mathit{MClos} & ::= & (i, \eta) \mid (k, \eta) \\
\eta \in \mathit{MEnv} & ::= & [] \mid p :: \eta \\
s \in \mathit{Stack} & ::= & [] \mid p :: s \mid m :: s \mid \pi_i :: s \\
\Gamma, \Delta \in \mathit{Heap} & ::= & [] \mid (p, \alpha) :: \Gamma \\
w \in \mathit{Conf} & ::= & (\Gamma, i, \eta, s)
\end{array}$$

We will use without notice the isomorphism between functions with finite domains and finite maps (conceived as lists of tuples). A pointer p is in an environment η if there is a position n such that $\eta \cdot n = p$; for stacks we count pointers regardless if they are marked or not. The pointers of a closure are those of its environment; and those of a heap are given by its domain and those in the closures of its image.

Definition 4.2 (Pointers)**.**

$$ptr(\eta \in \mathit{MEnv}) = \bigcup_{n \in \mathbb{N}} \eta \cdot n$$
$$ptr(s \in \mathit{Stack}) = \bigcup_{n \in \mathbb{N}} s \cdot n$$
$$ptr((i, \eta) \in \mathit{MClos}) = ptr(\eta)$$
$$ptr(\Gamma \in \mathit{Heap}) = dom(\Gamma) \cup \bigcup_{p \in dom(\Gamma)} ptr(\Gamma\, p)$$

We write $\Gamma[\,p \mapsto \alpha\,]$ to denote the update of the heap at the point(er) p:

$$\Gamma[\,p \mapsto \alpha\,]\,q = \begin{cases} \alpha & \text{if } p = q \\ \Gamma\, q & \text{otherwise} \end{cases}$$

The transition semantics for the machine is given in Fig. 2. The first rule corresponds to updating the heap because a value has been reached and the marker in the top of the stack indicates the pointer to be updated. The second and third rules update a component of a pair. The following two rules correspond to the evaluation of a strict binary operation; notice that, in the second rule, we update the corresponding pointer in the heap after computing the final value. The marker $?p$ is used to point to a closure with the instructions for continuing after the evaluation of the guard.

The next four rules are taken directly from Sestoft. An instruction $\texttt{Grab} \triangleright i$ is executed by moving the pointer from the stack to the environment and continuing the execution with i. The stack accumulates (pointers to) the code of the arguments and moving a pointer to the environment roughly means that the body of an abstraction can access to its argument. The instruction $\texttt{Access}\ n$ will dereference the pointer $\eta \cdot n$ and start the execution of its code; moreover this transition add a marker in the stack to force an update of the heap if we reach a value. The instruction $\texttt{Push}\ n \triangleright i$ inserts the n-th pointer from the environment into the stack. That pointer was created and added to the environment by an instruction $\texttt{Let}\ i \triangleright i'$, which increments the heap by inserting a new pointer p associated with the closure (i, η), η being the current environment. The freshness of p is guaranteed by the condition $p \notin ptr(\Gamma) \cup ptr(\eta) \cup ptr(s)$. One can avoid the no-determinism in the election of the new pointer by choosing a concrete representation for pointers and giving a deterministic algorithm for the generation of new names.

Two other instructions also create pointers, but in these cases they are associated with the other kind of closures.

$$[\![\lambda\, t]\!]_{\pi,\,\theta\to\theta'}\,\rho = d \mapsto [\![t]\!]_{\theta::\pi,\,\theta'}\,(d,\,\rho) \qquad [\![\bar{n}]\!]_{\pi,\,\theta}\,\rho = \rho \swarrow n$$

$$[\![t\ \bar{n}]\!]_{\pi,\,\theta'}\,\rho = [\![t]\!]_{\pi,\,\theta\to\theta'}\,\rho\,(\rho \swarrow n) \qquad [\![\diamond]\!]_{\pi,\,\mathbf{unit}}\,\rho = \diamond$$

$$[\![\, let\ t\ in\ t']\!]_{\pi,\,\theta'}\,\rho = [\![t']\!]_{\theta::\pi,\,\theta'}\,([\![t]\!]_{\pi,\,\theta}\,\rho,\,\rho) \qquad [\![(t,\,t')]\!]_{\pi,\,\theta\times\theta'}\,\rho = ([\![t]\!]_{\pi,\,\theta}\,\rho,\,[\![t']\!]_{\pi,\,\theta'}\,\rho)$$

$$[\![\lfloor m\rfloor]\!]_{\pi,\,\mathbf{int}}\,\rho = m \qquad [\![\, fst\ n]\!]_{\pi,\,\theta}\,\rho = \pi_1([\![\bar{n}]\!]_{\pi,\,\theta\times\theta'}\,\rho)$$

$$[\![t\odot t']\!]_{\pi,\,\mathbf{int}}\,\rho = ([\![t]\!]_{\pi,\,\mathbf{int}}\,\rho)\odot([\![t']\!]_{\pi,\,\mathbf{int}}\,\rho) \qquad [\![\, snd\ n]\!]_{\pi,\,\theta'}\,\rho = \pi_2([\![\bar{n}]\!]_{\pi,\,\theta\times\theta'}\,\rho)$$

$$[\![\, ifz\ t\ then\ t'\ else\ t'']\!]_{\pi,\,\theta}\,\rho = \begin{cases} [\![t']\!]_{\pi,\,\theta}\,\rho & \text{if } [\![t]\!]_{\pi,\,\mathbf{int}}\,\rho = 0 \\ [\![t'']\!]_{\pi,\,\theta}\,\rho & \text{if } [\![t]\!]_{\pi,\,\mathbf{int}}\,\rho = n+1 \end{cases}$$

Figure 1. Intrinsic denotational semantics

$$\begin{array}{lcll}
(\Gamma,\, iv,\, \eta,\, \sharp p :: s) & \longmapsto & (\Gamma[\,p \mapsto (iv,\,\eta)\,],\, iv,\, \eta,\, s) & \\
(\Gamma,\, iv,\, \eta,\, \sharp_1 p :: s) & \longmapsto & (\Gamma[\,p \mapsto (\mathtt{Pair}\ (iv, i'),\, \eta')\,],\, iv,\, \eta,\, s) & \text{with } (\mathtt{Pair}\ (i, i'),\, \eta') = \Gamma\, p \\
(\Gamma,\, iv,\, \eta,\, \sharp_2 p :: s) & \longmapsto & (\Gamma[\,p \mapsto (\mathtt{Pair}\ (i, iv),\, \eta')\,],\, iv,\, \eta,\, s) & \text{with } (\mathtt{Pair}\ (i, i'),\, \eta') = \Gamma\, p \\
(\Gamma,\, \underline{n},\, \eta',\, \odot p :: s) & \longmapsto & (\Gamma[\,p \mapsto (\mathtt{BOp}\ \underline{n}\ \square,\, \eta)\,],\, i',\, \eta,\, \odot p :: s) & \text{with } \Gamma p = (\mathtt{BOp}\ \square\ i',\, \eta) \\
(\Gamma,\, \underline{m},\, \eta',\, \odot p :: s) & \longmapsto & (\Gamma[\,p \mapsto \underline{n \odot m}\,],\, \underline{n \odot m},\, \eta,\, s) & \text{with } \Gamma p = (\mathtt{BOp}\ \underline{n}\ \square,\, \eta) \\
(\Gamma,\, \underline{0},\, \eta,\, ?p :: s) & \longmapsto & (\Gamma,\, i',\, \eta',\, s) & \text{with } \Gamma p = (\mathtt{IfZ}\ \square\ (i', i''),\, \eta') \\
(\Gamma,\, \underline{n+1},\, \eta,\, ?p :: s) & \longmapsto & (\Gamma,\, i'',\, \eta',\, s) & \text{with } \Gamma p = (\mathtt{IfZ}\ \square\ (i', i''),\, \eta') \\
(\Gamma,\, \mathtt{Grab}\ \triangleright\ i,\, \eta,\, p :: s) & \longmapsto & (\Gamma,\, i,\, p :: \eta,\, s) & \\
(\Gamma,\, \mathtt{Access}\ n,\, \eta,\, s) & \longmapsto & (\Gamma,\, i',\, \eta',\, \sharp p :: s) & \text{with } n < |\,\eta\,|, p = \eta \cdot n, (i',\, \eta') = \Gamma\, p \\
(\Gamma,\, \mathtt{Push}\ n\ \triangleright\ i,\, \eta,\, s) & \longmapsto & (\Gamma,\, i,\, \eta,\, p :: s) & \text{with } p = \eta \cdot n \\
(\Gamma,\, \mathtt{Let}\ i\ \triangleright\ i',\, \eta,\, s) & \longmapsto & (\Gamma[\,p \mapsto (i,\,\eta)\,],\, i',\, p :: \eta,\, s) & \text{with } p \notin ptr(\Gamma) \cup ptr(\eta) \cup ptr(s) \\
(\Gamma,\, \mathtt{BOp}\ i\ i',\, \eta,\, s) & \longmapsto & (\Gamma[\,p \mapsto (\mathtt{BOp}\ \square\ i',\, \eta)\,],\, i,\, \eta,\, \odot p :: s) & \text{with } p \notin ptr(\Gamma) \cup ptr(\eta) \cup ptr(s) \\
(\Gamma,\, \mathtt{IfZ}\ i\ i'\ i'',\, \eta,\, s) & \longmapsto & (\Gamma[\,p \mapsto (\mathtt{IfZ}\ \square\ (i', i''),\, \eta)\,],\, i,\, \eta,\, ?p :: s) & \text{with } p \notin ptr(\Gamma) \cup ptr(\eta) \cup ptr(s) \\
(\Gamma,\, \mathtt{Fst}\ n,\, \eta,\, s) & \longmapsto & (\Gamma,\, i,\, \eta',\, \sharp p :: \pi_1 :: \sharp_1 p :: s) & \text{with } p = \eta \cdot n, \Gamma\, p = (i, \eta') \\
(\Gamma,\, \mathtt{Snd}\ n,\, \eta,\, s) & \longmapsto & (\Gamma,\, i,\, \eta',\, \sharp p :: \pi_2 :: \sharp_2 p :: s) & \text{with } p = \eta \cdot n, \Gamma\, p = (i, \eta') \\
(\Gamma,\, \mathtt{Pair}\ (i, i'),\, \eta,\, \pi_1 :: s) & \longmapsto & (\Gamma,\, i,\, \eta,\, s) & \\
(\Gamma,\, \mathtt{Pair}\ (i, i'),\, \eta,\, \pi_2 :: s) & \longmapsto & (\Gamma,\, i',\, \eta,\, s) &
\end{array}$$

Figure 2. Transitions of the machine

The computation of a binary operator creates a frame [18] that keeps track of which operand is being computed. The execution of the conditional branching associates the new pointer with the pair of instructions to be followed depending on the evaluation of the guard which is put in the control position.

The last four rules deal with the execution of projections: to project a component, say the first with `Fst` n, we must evaluate the code associated with the pointer given by $\eta \cdot n$; we also push into the stack markers for updating that pointer, projecting the first component –in the case that the code computes a pair–, and finally to update the first component. The last two transitions are the actual projection of the component.

5 Compiler and its Correctness

In this section we introduce the compiler and prove its correctness with respect to the denotational semantics. When the language permits recursive definition or has some fixed-point operator, correctness involves two aspects. First, the compilation of a diverging term must produce a low-level code which makes the machine run forever; secondly if the meaning of a term is a constant, its compilation should be a code such that the machines stops after some finite steps and reaches a configuration with that constants.

In contrast with the denotational semantics, we do not use any typing information in the compilation, thus we can compile any term. Not withstanding, we prove correctness only for well-typed terms and the proof relies on the typing derivations.

Definition 5.1 (Compiler)**.**

$$\begin{array}{llll}
(\!|\,\lambda\, t\,|\!) & = \mathtt{Grab}\ \triangleright\ (\!|\, t\,|\!) & (\!|\,\bar{n}\,|\!) & = \mathtt{Access}\ n \\
(\!|\, t\ \bar{n}\,|\!) & = \mathtt{Push}\ n\ \triangleright\ (\!|\, t\,|\!) & (\!|\,\diamond\,|\!) & = \mathtt{Unit} \\
(\!|\, let\ t\ in\ t'\,|\!) & = \mathtt{Let}\ (\!|\, t\,|\!)\ \triangleright\ (\!|\, t'\,|\!) & (\!|\,\lfloor m\rfloor\,|\!) & = \underline{m} \\
(\!|\, (t,\, t')\,|\!) & = \mathtt{Pair}\ ((\!|\, t\,|\!), (\!|\, t'\,|\!)) & (\!|\, fst\ \bar{n}\,|\!) & = \mathtt{Fst}\ n \\
(\!|\, t\odot t'\,|\!) & = \mathtt{BOp}\ (\!|\, t\,|\!)\ (\!|\, t'\,|\!) & (\!|\, snd\ \bar{n}\,|\!) & = \mathtt{Snd}\ n \\
\multicolumn{4}{l}{(\!|\, ifz\ t\ then\ t'\ else\ t''\,|\!) = \mathtt{IfZ}\ (\!|\, t\,|\!)\ (\!|\, t'\,|\!)\ (\!|\, t''\,|\!)}
\end{array}$$

Observations

In order to apply biorthogonality as explained in Sec. 2 we need to introduce some notions useful for defining the *satisfiability* relation which will induce the orthogonal operators.

We say that a heap is *well-formed* when all its pointers are included in its domain. For example, one clearly checks that the heap $\Gamma = \{ p \mapsto (\texttt{Unit}, [\, q \,]) \}$ is not well-formed because $q \notin dom(\Gamma)$. The predicate $wf(\Gamma)$ establishes that Γ is well-formed, and is extended to realizers $(\Gamma,\ \alpha)$ and tests $(\Gamma,\ s)$.

Definition 5.2 (Well-formed heap)**.**

$$\begin{aligned} wf(\Gamma) &\text{ if and only if } ptr(\Gamma) = dom(\Gamma) \\ wf(\Gamma,\ \alpha) &\text{ if and only if } wf(\Gamma) \text{ and } ptr(\alpha) \subseteq dom(\Gamma)\,, \\ wf(\Gamma,\ s) &\text{ if and only if } wf(\Gamma) \text{ and } ptr(s) \subseteq dom(\Gamma)\ . \end{aligned}$$

Let us state some useful and self-evident lemmas about well-formedness.

Lemma 5.3. *If $wf(\Gamma)$, then for any $p \in dom(\Gamma)$, $wf(\Gamma,\ \Gamma\, p)$. If $wf(\Gamma)$ and $ptr(\alpha) \subseteq dom(\Gamma)$, then $wf(\Gamma[\, p \mapsto \alpha \,])$.*

We say that the heaps Γ and Δ are *compatible* if they coincide in their common pointers.

Definition 5.4. The heaps Γ and Δ are compatible, written $\Gamma \bowtie \Delta$, if $\Gamma\, p = \Delta\, p$, for all $p \in dom(\Gamma) \cap dom(\Delta)$.

The union of heaps is well-defined when they are compatible.

Definition 5.5. If $\Gamma \bowtie \Delta$, their *union* $\Gamma \cup \Delta$ with domain $dom(\Gamma) \cup dom(\Delta)$ is defined by

$$(\Gamma \cup \Delta)\, p = \begin{cases} \Gamma\, p & \text{if } p \in dom(\Gamma) \\ \Delta\, p & \text{if } p \in dom(\Delta)\ . \end{cases}$$

Let us state some basic facts about the relation $\bowtie$ and the union of compatible heaps.

Lemma 5.6. *The relation $\bowtie$ is reflexive and symmetric. Moreover, assuming $\Delta \bowtie \Delta'$, then $\Gamma \bowtie \Delta$ and $\Gamma \bowtie \Delta'$ if and only if $\Gamma \bowtie (\Delta \cup \Delta')$.*

The union of compatible heaps is both associative and commutative; well-formedness is preserved by the union.

Lemma 5.7. *If $wf(\Gamma)$, $wf(\Delta)$, and $\Gamma \bowtie \Delta$, then $wf(\Gamma \cup \Delta)$.*

In Krivine realizability an *observation* $\perp\!\!\!\perp$ is a set of configurations closed by anti-execution (i.e., if $w' \in \perp\!\!\!\perp$ and $w \longmapsto w'$, then $w \in \perp\!\!\!\perp$). We impose some additional restrictions on observations.

Definition 5.8. A set $\perp\!\!\!\perp \subseteq Conf$ is an *observation* if it is closed by anti-execution and satisfies the following inductive clauses:

$$O_1\ \frac{(\Gamma,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}{(\Gamma \cup \Delta,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}\ wf(\Gamma,\ s), wf(\Gamma,\ (i,\ \eta)),\ \Gamma \bowtie \Delta$$

$$O_2\ \frac{(\Gamma,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}{(\Gamma,\ i,\ \eta,\ \sharp p :: s) \in \perp\!\!\!\perp}\ \Gamma\, p = (i,\ \eta)$$

$$OP_1\ \frac{(\Gamma,\ i,\ \eta,\ s) \in \perp\!\!\!\perp}{(\Gamma,\ i,\ \eta,\ \sharp_1 p :: s) \in \perp\!\!\!\perp}\ \Gamma\, p = (\texttt{Pair}\ (i, i'),\ \eta)$$

$$OP_2\ \frac{(\Gamma,\ i',\ \eta,\ s) \in \perp\!\!\!\perp}{(\Gamma,\ i',\ \eta,\ \sharp_2 p :: s) \in \perp\!\!\!\perp}\ \Gamma\, p = (\texttt{Pair}\ (i, i'),\ \eta)$$

The first rule O_1 asserts that the heap can be extended with pointers which are irrelevant for the execution. Their irrelevance is evident from the conditions, since $ptr(\eta) \cup ptr(s) \subseteq dom(\Gamma)$ and $wf(\Gamma)$, every pointer in the configuration $(\Gamma,\ i,\ \eta,\ s)$ is in the domain of Γ, therefore $dom(\Delta) - dom(\Gamma)$ is a set of pointers which will not be dereferenced in the execution, in spite of being in the heap $\Gamma \cup \Delta$.

The other three rules allow to add markers to update some component of the heap by adding new markers to the stack. Intuitively these rules imply that the mechanism for avoiding the repeated evaluation of the same argument is an optimization and should not affect the observation.

One valid observation is given by the set of terminating configurations (regardless of the particular values of the components). Clearly adding irrelevant pointers does not affect termination; it can, however, alter the particular (name of) pointers chosen on the execution. Pushing an update marker into the stack does not either cause a terminating configuration to loop, although it may enable some further steps. Notice that the restriction $wf(\Gamma)$ in O_1 is essential for avoiding the following situation. Let $\perp\!\!\!\perp$ be the set of terminating configurations and let $\Gamma = \{ p \mapsto (\texttt{Access}\ 0, [\, q \,]) \}$ be a not well-formed heap (since $q \notin dom(\Gamma)$) and $\Delta = \{ q \mapsto (\texttt{Access}\ 0, [\, q \,]) \}$. Clearly $(\Gamma, \texttt{Access}\ 0, [\, p \,], s) \in \perp\!\!\!\perp$, at some point the machine gets stuck because q is undefined. As soon as we combine the heaps, q is defined and a loop is produced, thus $(\Gamma \cup \Delta, \texttt{Access}\ 0, [\, p \,], s) \notin \perp\!\!\!\perp$.

In contrast with the KAM, where a closure is a realizer by itself, in Sestoft machine the heap also takes part since it gives meaning to the pointers occurring in the environment of the closure. Similarly, pointers in the stack also reference to the heap, thus tests should be pairs of a stack and a heap. In order to combine a test and a realizer we shall enforce that their heaps are compatible and well-formed.

Definition 5.9 (Satisfiability)**.** Let $\perp\!\!\!\perp \subseteq Conf$ be an observation, then $\models \subseteq (Heap \times MClos) \times (Heap \times Stack)$ is the smallest relation satisfying

$$\frac{wf(\Gamma,\ \alpha),\ wf(\Delta,\ s),\ \Gamma \bowtie \Delta \text{ imply } (\Gamma \cup \Delta,\ \alpha,\ s) \in \perp\!\!\!\perp}{(\Gamma,\ \alpha)\ \models\ (\Delta,\ s)}$$

Let us explicit the orthogonal operators associated with this relation. As is well-known, they form an antitone Galois connection, thus their compositon yields a closure operator.

Definition 5.10. Let $\perp\!\!\!\perp \subseteq \mathit{Conf}$ be an observation, which determines the relation $\_ \models \_$.

$$X^{\perp} = \{\, (\Delta,\, s) \mid \text{for all } (\Gamma,\, \alpha) \in X,\, (\Gamma,\, \alpha) \models (\Delta,\, s) \,\}$$

$$Y^{\top} = \{\, (\Gamma,\, \alpha) \mid \text{for all } (\Delta,\, s) \in Y,\, (\Gamma,\, \alpha) \models (\Delta,\, s) \,\} \quad .$$

Realizers, tests, and logical relations

A possible reading for realizers is that they approximate denotational elements (of course, this terminology is more sensible when the language has some form of recursion). Under this analysis, primitive realizers are direct approximations; for example a closure $\lfloor m \rfloor, \eta$ paired with any heap constitutes a primitive realizer of the denotational value $m \in \mathbb{Z}$. Looking not so closely at primitive realizers for arrow types, one unveils a well-known situation: we are introducing a family of logical relations; in doing so, we also furnish the low-level setting with the required structure to be model of the high-level language. Non-primitive realizers are, in turn, defined taking the biorthogonal of the primitive realizers.

Definition 5.11 (Realizers)**.** Let $\perp\!\!\!\perp \subseteq \mathit{Conf}$ be an observation.

$$\frac{\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma)}{(\Gamma,\, \mathtt{Unit},\, \eta) \in \mathcal{R}_P^{\mathbf{unit}}(\diamond)} \qquad \frac{\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma)}{(\Gamma,\, \underline{m},\, \eta) \in \mathcal{R}_P^{\mathbf{int}}(m)}$$

$$\frac{\begin{array}{c}\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma) \quad \text{for all } d, \Gamma', \alpha \quad \mathit{wf}(\Gamma',\, \alpha)\ \Gamma \bowtie \Gamma' \\ (\Gamma',\, \alpha) \in \mathcal{R}^{\theta}(d)(\Gamma \cup \Gamma') \bowtie \{\, p \mapsto \alpha \,\} \\ (\Gamma \cup \Gamma' \cup \{\, p \mapsto \alpha \,\},\, i,\, p :: \eta) \in \mathcal{R}^{\theta'}(f\ d)\end{array}}{(\Gamma,\, \mathtt{Grab} \triangleright i,\, \eta) \in \mathcal{R}_P^{\theta \to \theta'}(f)}$$

$$\frac{\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma) \quad (\Gamma,\, i,\, \eta) \in \mathcal{R}^{\theta}(d_0) \quad (\Gamma,\, i',\, \eta) \in \mathcal{R}^{\theta'}(d_1)}{(\Gamma,\, \mathtt{Pair}\ (i, i'),\, \eta) \in \mathcal{R}_P^{\theta \times \theta'}((d_0, d_1))}$$

$$\mathcal{R}^{\theta}(d) = \mathcal{R}_P^{\theta}(d)^{\perp\top} \quad .$$

The set of tests for a denotational value $d \in [\![\, \theta \,]\!]$ is given by the orthogonal of its realizers, $\mathcal{T}^{\theta}(d) = \mathcal{R}^{\theta}(d)^{\perp}$. From the adjuction underlying the Galois connection we know $\mathcal{R}^{\theta}(d)^{\perp} = \mathcal{R}_P^{\theta}(d)^{\perp}$. The approximation relation is given in terms of realizers.

Definition 5.12 (Approximation)**.** Let $\perp\!\!\!\perp \subseteq \mathit{Conf}$.

$$(\Gamma,\, \alpha)\ \blacksquare^{\theta}\ d \text{ if and only if } (\Gamma,\, \alpha) \in \mathcal{R}^{\theta}(d) \quad .$$

We know $(\Gamma,\, \mathtt{Unit},\, \eta)\ \blacksquare^{\mathbf{unit}}\ \diamond$ whenever $\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma)$, because the biorthogonal operator is inflationary. Intuitively, to probe $(\Gamma,\, \mathtt{Grab} \triangleright i,\, \eta)\ \blacksquare^{\theta \to \theta'}\ f$, we have to show that extending the environment with an approximation for $d \in [\![\, \theta \,]\!]$ we get an approximation for $f\ d \in [\![\, \theta' \,]\!]$. Clearly such an extension of the environment is through a pointer; so if we assume $(\Gamma',\, \alpha)\ \blacksquare^{\theta}\ d$, we take $p \in \mathit{Pointer}$ and extend the heap with the closure α. That is possible only if $(\Gamma \cup \Gamma') \bowtie \{\, p \mapsto \alpha \,\}$, which admits both the situation where p is already defined in the heap and also a fresh pointer $p \notin \mathit{dom}(\Gamma \cup \Gamma')$. Under all these assumptions, we must show $(\Gamma \cup \Gamma' \cup \{\, p \mapsto \alpha \,\},\, i,\, p :: \eta)\ \blacksquare^{\theta'}\ f\ d$.

One strategy to conclude $(\Gamma,\, \alpha) \in \mathcal{R}^{\theta}(d) = \mathcal{T}^{\theta}(d)^{\top}$ is proving that $(\Gamma,\, \alpha) \models (\Delta,\, s)$ holds for any $(\Delta,\, s) \in \mathcal{T}^{\theta}(d)$. After expanding the definition of the satisfiability relation, the strategy reduces to deduce $(\Gamma \cup \Delta,\, \alpha,\, s) \in \perp\!\!\!\perp$ with the hypotheses $\mathit{wf}(\Gamma,\, \alpha)$, $\mathit{wf}(\Delta,\, s)$ and $\Gamma \bowtie \Delta$.

The rule O_1 allows us to extend the heap of a realizer by taking its union with other heap.

Lemma 5.13. *If* $\mathit{wf}(\Gamma,\, \alpha)$, $\Gamma \bowtie \Delta$ *and* $(\Gamma,\, \alpha)\ \blacksquare^{\theta}\ d$, *then* $(\Gamma \cup \Delta,\, \alpha)\ \blacksquare^{\theta}\ d$.

Proof. We assume $\mathit{wf}(\Gamma \cup \Delta,\, \alpha)$; let us take $(\Delta',\, s) \in \mathcal{T}^{\theta}(d)$ such that $\mathit{wf}(\Delta',\, s)$ and $(\Gamma \cup \Delta) \bowtie \Delta'$ for proving $(\Gamma \cup \Delta \cup \Delta',\, \alpha,\, s) \in \perp\!\!\!\perp$.

Since $(\Gamma \cup \Delta) \bowtie \Delta'$, we get $\Gamma \bowtie \Delta'$ and $\Delta \bowtie \Delta'$. Since $(\Gamma,\, \alpha)\ \blacksquare^{\theta}\ d$, we can conclude $(\Gamma \cup \Delta',\, \alpha,\, s) \in \perp\!\!\!\perp$. By Lemma 5.7 we know that the union is well-formed, $\mathit{wf}(\Gamma \cup \Delta')$. Moreover we can easily see $(\Gamma \cup \Delta') \bowtie \Delta$, thanks to $\Gamma \bowtie \Delta$ and $\Delta \bowtie \Delta'$. We may apply the rule O_1 to conclude $(\Gamma \cup \Delta' \cup \Delta,\, \alpha,\, s) \in \perp\!\!\!\perp$. □

We extend the approximation relation from types to typing contexts.

Definition 5.14 (Approximation for environments)**.**

$$\frac{}{(\Gamma,\, [])\ \blacksquare^{[]}\ \diamond} \qquad \frac{(\Gamma,\, \eta)\ \blacksquare^{\pi}\ \rho \quad p \in \mathit{dom}(\Gamma) \quad (\Gamma,\, \Gamma\, p)\ \blacksquare^{\theta}\ d}{(\Gamma,\, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d,\, \rho)}$$

The following properties follow immediately from the definition of approximations for contexts.

Lemma 5.15. *Let* $(\Gamma,\, \eta)\ \blacksquare^{\pi}\ \rho$.

1. $|\, \eta \,| = |\, \pi \,|$ *and* $\mathit{ptr}(\eta) \subseteq \mathit{dom}(\Gamma)$.
2. *For any* $n < |\, \eta \,|$, $(\Gamma,\, \Gamma\, (\eta \cdot n))\ \blacksquare^{\pi \cdot n}\ \rho \swarrow n$.

As a direct consequence of Lemma 5.13, the extension of the heap is also valid for realizers of environments.

Lemma 5.16. *If* $\mathit{wf}(\Gamma)$, $\Gamma \bowtie \Delta$ *and* $(\Gamma,\, \eta)\ \blacksquare^{\pi}\ \rho$, *then* $(\Gamma \cup \Delta,\, \eta)\ \blacksquare^{\pi}\ \rho$.

Proof. We proceed by induction on π. The base case $\pi = []$ is trivial since $(\Gamma \cup \Delta,\, [])\ \blacksquare^{[]}\ \diamond$. Let us consider the inductive case with $\theta :: \pi$; we assume $(\Gamma,\, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d,\, \rho)$ to prove $(\Gamma \cup \Delta,\, p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d,\, \rho)$. By inversion on the hypothesis, we have $(\Gamma,\, \eta)\ \blacksquare^{\pi}\ \rho$ and $(\Gamma,\, \Gamma\, p)\ \blacksquare^{\theta}\ d$. The inductive hypothesis gives us $(\Gamma \cup \Delta,\, \eta)\ \blacksquare^{\pi}\ \rho$. On the other hand, since $\mathit{wf}(\Gamma)$, by Lemma 5.3 we get $\mathit{wf}(\Gamma,\, \Gamma\, p)$, therefore we can apply Lemma 5.13 to deduce $(\Gamma \cup \Delta,\, \Gamma\, p)\ \blacksquare^{\theta}\ d$. □

The following results will show that the approximation relations are modular, meaning that one can combine tests and realizers to form realizers of more complex types. For

example, an approximation for an environment $(d, \rho) \in [\![\theta :: \pi]\!]$ is built from approximations for $d \in [\![\theta]\!]$ and $\rho \in [\![\pi]\!]$.

Lemma 5.17. *Let* $(\Gamma, \eta) \blacksquare^{\pi} \rho$ *with* $wf(\Gamma)$, *and* $(\Gamma', \alpha) \blacksquare^{\theta} d$ *with* $wf(\Gamma', \alpha)$, *and* $\Gamma \bowtie \Gamma'$. *If* $(\Gamma \cup \Gamma') \bowtie \{p \mapsto \alpha\}$, *then* $(\Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}, p :: \eta) \blacksquare^{\theta :: \pi} (d, \rho)$.

Proof. Let $\Gamma'' = \Gamma \cup \Gamma' \cup \{p \mapsto \alpha\}$. We must show $(\Gamma'', \eta) \blacksquare^{\pi} \rho$ and $(\Gamma'', \Gamma'' p) \blacksquare^{\theta} d$. The first item easily follows from $(\Gamma, \eta) \blacksquare^{\pi} \rho$ and $wf(\Gamma)$, by Lemma 5.16. Since $\Gamma'' p = \alpha$, for the second part we have to prove $(\Gamma'', \alpha) \blacksquare^{\theta} d$. From $\Gamma' \bowtie \Gamma$ and $\Gamma' \bowtie \{p \mapsto \alpha\}$, we get $\Gamma' \bowtie (\Gamma \cup \{p \mapsto \alpha\})$. Then we apply Lemma 5.13 on $(\Gamma', \alpha) \blacksquare^{\theta} d$ and $wf(\Gamma', \alpha)$, to conclude $(\Gamma' \cup \Gamma \cup \{p \mapsto \alpha\}, \alpha) = (\Gamma'', \alpha) \blacksquare^{\theta} d$. □

We may also construct a test for a function $f \in [\![\theta \to \theta']\!]$ by combining a realizer of some possible argument $d \in [\![\theta]\!]$ and a test for the result $f\ d \in [\![\theta']\!]$.

Lemma 5.18. *Let us assume* $\Gamma \bowtie \Delta$, $(\Gamma, \alpha) \in \mathcal{R}^{\theta}(d)$ *such that* $wf(\Gamma, \alpha)$, *and* $(\Delta, s) \in \mathcal{T}^{\theta'}(f\ d)$ *with* $wf(\Delta, s)$. *For any* p *such that* $(\Gamma \cup \Delta) \bowtie \{p \mapsto \alpha\}$, $(\Gamma \cup \Delta \cup \{p \mapsto \alpha\}, p :: s) \in \mathcal{T}^{\theta \to \theta'}(f)$.

Proof. We introduce $\Gamma' = \Gamma \cup \Delta \cup \{p \mapsto \alpha\}$; let us show $(\Gamma', p :: s) \in \mathcal{R}_P^{\theta \to \theta'}(f)^{\perp}$. So we take a primitive realizer for the function $(\Gamma'', \hat{\alpha}) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ such that $wf(\Gamma'', \hat{\alpha})$ and $\Gamma' \bowtie \Gamma''$, to prove $(\Gamma' \cup \Gamma'', \hat{\alpha}, p :: s) \in \perp\!\!\!\perp$.

By definition of primitive realizer for arrow types $\mathcal{R}_P^{\theta \to \theta'}(f)$ we have $\hat{\alpha} = (\texttt{Grab} \triangleright i, \eta)$ for some $i \in Code$, $\eta \in MEnv$. Moreover, since $\Gamma'' \bowtie \Gamma$ and $\Gamma'' \cup \Gamma \bowtie \{p \mapsto \alpha\}$, we know, again from $(\Gamma'', \hat{\alpha})$ being a primitive realizer for f, $(\Gamma'' \cup \Gamma \cup \{p \mapsto \alpha\}, i, p :: \eta) \in \mathcal{R}^{\theta'}(f\ d)$. Notice that $(\Gamma' \cup \Gamma'', \hat{\alpha}, p :: s) \longmapsto (\Gamma' \cup \Gamma'', i, p :: \eta, s)$; so we might use anti-execution after proving this last configuration is in the pole $\perp\!\!\!\perp$. Let $\Delta' = \Gamma'' \cup \Gamma \cup \{p \mapsto \alpha\}$, it clearly holds $\Delta \bowtie \Delta'$ and $wf(\Delta')$, so we can conclude $(\Delta' \cup \Delta, i, p :: \eta, s) \in \perp\!\!\!\perp$, by combining the primitive realizer and the test of $f\ d$. □

We may combine approximations of a function with approximations of arguments to get an approximation for the result.

Lemma 5.19. *Let* $(\Gamma, \eta) \blacksquare^{\pi} \rho$ *such that* $wf(\Gamma)$, $n < |\eta|$, *and* $\theta = \pi \cdot n$. *If* $(\Gamma, i, \eta) \blacksquare^{\theta \to \theta'} f$, *then* $(\Gamma, \texttt{Push}\ n \triangleright i, \eta) \blacksquare^{\theta'} f\ (\rho \swarrow n)$.

Proof. Let $d = \rho \swarrow n$, $p = \eta \cdot n$. We take a test for the function $(\Delta, s) \in \mathcal{T}^{\theta'}(f\ d)$ assuming that $wf(\Delta, s)$ and $\Gamma \bowtie \Delta$ in order to prove $(\Gamma \cup \Delta, \texttt{Push}\ n \triangleright i, \eta, s) \in \perp\!\!\!\perp$.

Since $(\Gamma, \eta) \blacksquare^{\pi} \rho$ and $n < |\eta|$, we get by Lemma 5.15 $(\Gamma, \Gamma p) \in \mathcal{R}^{\theta}(d)$. We may use Lemma 5.18, because $(\Gamma \cup \Delta) \bowtie \{p \mapsto \Gamma p\}$, to know that $(\Gamma \cup \Delta, p :: s) \in \mathcal{T}^{\theta \to \theta'}(f)$. Thus we know $(\Gamma \cup \Delta, i, \eta, p :: s) \in \perp\!\!\!\perp$. This finish the proof as $\perp\!\!\!\perp$ is closed by anti-execution. □

Since environments give denotation to free variables, which are compiled with an `Access` instruction, it is intuitive to think that an approximation for an environment can be used with `Access` to get an approximation for each variable.

Lemma 5.20. *Let* $(\Gamma, \eta) \blacksquare^{\pi} \rho$ *and* $wf(\Gamma)$. *For any index* $n < |\eta|$, $(\Gamma, \texttt{Access}\ n, \eta) \blacksquare^{\pi \cdot n} \rho \swarrow n$.

Proof. Let $\theta = \pi \cdot n$, $d = \rho \swarrow n$, and $p = \eta \cdot n$. We take a test for d, $(\Delta, s) \in \mathcal{T}^{\theta}(d)$ such that $wf(\Delta, s)$ and $\Gamma \bowtie \Delta$, in order to prove $(\Gamma \cup \Delta, \texttt{Access}\ n, \eta, s) \in \perp\!\!\!\perp$. By Lemma 5.15, we know $(\Gamma, \Gamma p) \blacksquare^{\theta} d$; therefore $(\Gamma \cup \Delta, \Gamma p, s) \in \perp\!\!\!\perp$. Consequently by rule O_2 we get $(\Gamma \cup \Delta, \Gamma p, \sharp p :: s) \in \perp\!\!\!\perp$. Since we have $(\Gamma \cup \Delta, \texttt{Access}\ n, \eta, s) \longmapsto (\Gamma \cup \Delta, \Gamma p, \sharp p :: s)$ and $\perp\!\!\!\perp$ is closed by anti-execution, we have finished. □

The next lemma asserts that a test for the first component of a tuple can be turned into a test for the tuple by pushing into the stack the markers for projecting and updating the component. Of course, there is a symmetric lemma for the second projection.

Lemma 5.21. *If* $(\Delta, s) \in \mathcal{T}^{\theta_1}(d_1)$ *such that* $wf(\Delta, s)$, $wf(\Gamma)$ *and* $p \in ptr(\Gamma)$ *with* $(\Gamma, \Gamma p) \in \mathcal{R}^{\theta_1 \times \theta_2}((d_1, d_2))$, *and* $\Gamma \bowtie \Delta$, *then* $(\Gamma \cup \Delta, \pi_1 :: \sharp_1 p :: s) \in \mathcal{T}^{\theta_1 \times \theta_2}((d_1, d_2))$

Proof. We take a primitive realizer $(\Gamma', \alpha') \in \mathcal{R}_P^{\theta_1 \times \theta_2}((d_1, d_2))$ such that $wf(\Gamma', \alpha')$, $wf(\Gamma \cup \Delta, \pi_1 :: \sharp_1 p :: s)$ and $\Gamma' \bowtie \Gamma \cup \Delta$ to prove $(\Gamma' \cup \Gamma \cup \Delta, \alpha, \pi_1 :: \sharp_1 p :: s) \in \perp\!\!\!\perp$. As before we will use anti-execution to deduce that. Since (Γ', α) is a primitive realizer, we know $\alpha = (\texttt{Pair}\ (i, i'), \eta)$; since $(\Gamma' \cup \Gamma \cup \Delta, \alpha, \pi_1 :: \sharp_1 p :: s) \longmapsto (\Gamma' \cup \Gamma \cup \Delta, i, \eta, \sharp_1 p :: s)$, we prove that this last configuration is in the observation. At this point, the rule OP_1 allows us to change the goal to $(\Gamma' \cup \Gamma \cup \Delta, i, \eta, s) \in \perp\!\!\!\perp$.

We reached a configuration where all the pointers in $dom(\Gamma)$ are irrelevant, thus we may appeal to O_1 and prove $(\Gamma' \cup \Delta, i, \eta, s) \in \perp\!\!\!\perp$. It is easy to check that the side conditions are satisfied: we deduce $wf(\Gamma' \cup \Delta, s)$ and $wf(\Gamma' \cup \Delta, (i, \eta))$ from $wf(\Delta, s)$ and $wf(\Gamma', \alpha')$; then $\Gamma' \cup \Delta \bowtie \Gamma$ follows almost directly from $\Gamma' \bowtie \Gamma \cup \Delta$ (and thus $\Gamma' \bowtie \Gamma$), we also know $\Gamma \bowtie \Delta$, therefore we conclude by Lemma 5.6.

In order to finish the proof we note that $(\Gamma', i, \eta) \in \mathcal{R}^{\theta_1}(d_1)$, by inversion on the primitive realizer of the pair, and $(\Delta, s) \in \mathcal{T}^{\theta_1}(d_1)$ is one of the tests it satisfies. Since the three preconditions ($wf(\Gamma', (i, \eta))$, $wf(\Delta, s)$ and $\Gamma' \bowtie \Delta$) of the satisfiability relation are met, we know $(\Gamma' \cup \Delta, i, \eta, s) \in \perp\!\!\!\perp$. □

The fundamental lemma of logical relations states, in this setting, that the compilation of any well-typed term approximates its denotation.

Theorem 5.22. *If* $\pi \vdash t : \theta$, $wf(\Gamma)$, *and* $(\Gamma, \eta) \blacksquare^{\pi} \rho$, *then* $(\Gamma, (\!| t |\!), \eta) \blacksquare^{\theta} [\![t]\!]_{\pi, \theta}\ \rho$.

Proof. We proceed by induction on the derivation $\pi \vdash t : \theta$. When we face an introduction rule, we prove that the compilation is a primitive realizer; in these cases, the premise $ptr(\eta) \subseteq dom(\Gamma)$ is fulfiled by Lemma 5.15.

(**Abs**) Let $f = [\![\lambda\, t]\!]_{\pi,\, \theta \to \theta'}\, \rho$. We have to prove $(\Gamma,\ \texttt{Grab} \triangleright (\!| t |\!),\ \eta) \in \mathcal{R}_P^{\theta \to \theta'}(f)$. We take $d \in [\![\theta]\!]$, $p \in \mathit{Pointer}$, and $(\Gamma',\ \alpha) \in \mathcal{R}^{\theta}(d)$. We assume furthermore $wf(\Gamma',\ \alpha)$, $\Gamma \bowtie \Gamma'$, and $(\Gamma \cup \Gamma') \bowtie \{ p \mapsto \alpha \}$.
Let $\Gamma'' = \Gamma \cup \Gamma' \cup \{ p \mapsto \alpha \}$; applying Lemma 5.17 we get $(\Gamma'',\ p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d,\ \rho)$. Since $wf(\Gamma'')$, we may use the inductive hypothesis to get

$$(\Gamma'',\ (\!| t |\!),\ p :: \eta)\ \blacksquare^{\theta'}\ [\![t]\!]_{\theta :: \pi,\, \theta'}\, (d,\ \rho) = f\ d\ .$$

Which proves $(\Gamma,\ \texttt{Grab} \triangleright (\!| t |\!),\ \eta) \in \mathcal{R}_P^{\theta \to \theta'}(f)$ as we wanted.

(**Var**) Our hypotheses are $(\Gamma,\ \eta)\ \blacksquare^{\pi}\ \rho$ and $n < |\, \pi \,| = |\, \eta \,|$, so we get by Lemma 5.20 exactly what is needed:

$$(\Gamma,\ \texttt{Access}\ n,\ \eta)\ \blacksquare^{\theta}\ \rho\ \swarrow\ n = [\![\bar{n}]\!]_{\pi,\, \theta}\, \rho\ .$$

(**App**) Let $\theta = \pi \cdot n$. We have to prove $(\Gamma,\ (\!| t\ \bar{n} |\!),\ \eta)\ \blacksquare^{\theta'}\ [\![t\ \bar{n}]\!]_{\pi,\, \theta'}\, \rho$. By the inductive hypothesis on the first premise, we have $(\Gamma,\ (\!| t |\!),\ \eta)\ \blacksquare^{\theta \to \theta'}\ [\![t]\!]_{\pi,\, \theta \to \theta'}\, \rho$, Then, by Lemma 5.19 we get

$$(\Gamma,\ \texttt{Push}\ n \triangleright (\!| t |\!),\ \eta)\ \blacksquare^{\theta'}\ [\![t]\!]_{\pi,\, \theta \to \theta'}\, \rho\, (\rho\ \swarrow\ n)\ .$$

(**Unit**) It is trivial, since $(\Gamma,\ \texttt{Unit},\ \eta) \in \mathcal{R}^{\mathbf{unit}}(\diamond)$.

(**Let**) To show $(\Gamma,\ (\!|\ \mathit{let}\ t\ \mathit{in}\ t' |\!),\ \eta)\ \blacksquare^{\theta'}\ [\![\ \mathit{let}\ t\ \mathit{in}\ t']\!]_{\pi,\, \theta'}\, \rho$, we let $d = [\![t]\!]_{\pi,\, \theta}\, \rho$ and $d' = [\![t']\!]_{\theta :: \pi,\, \theta'}\, (d,\ \rho)$; then $[\![\ \mathit{let}\ t\ \mathit{in}\ t']\!]_{\pi,\, \theta'}\, \rho = d'$. Let us take $(\Delta,\ s) \in \mathcal{T}^{\theta'}(d')$ such that $wf(\Delta,\ s)$ y $\Gamma \bowtie \Delta$, to prove

$$(\Gamma \cup \Delta,\ \texttt{Let}\ (\!| t |\!) \triangleright (\!| t' |\!),\ \eta,\ s) \in \perp\!\!\!\perp\ .$$

Let $\alpha = ((\!| t |\!),\ \eta)$, then by anti-execution it is enough to prove $((\Gamma \cup \Delta)[p \mapsto \alpha],\ (\!| t' |\!),\ \eta,\ s) \in \perp\!\!\!\perp$, for a fresh pointer $p \notin \mathit{ptr}(\Gamma \cup \Delta) \cup \mathit{ptr}(\eta) \cup \mathit{ptr}(s)$.
From the first inductive hypothesis we get $(\Gamma,\ \alpha)\ \blacksquare^{\theta}\ d$. Since $\Gamma \bowtie \Gamma$ and $\Gamma \bowtie \{ p \mapsto \alpha \}$, we deduce, by Lemma 5.17, $(\Gamma \cup \{ p \mapsto \alpha \},\ p :: \eta)\ \blacksquare^{\theta :: \pi}\ (d,\ \rho)$. Then we can apply the other inductive hypothesis and obtain $(\Gamma \cup \{ p \mapsto \alpha \},\ (\!| t' |\!),\ p :: \eta)\ \blacksquare^{\theta :: \pi}\ d'$.
Since $\Gamma \cup \{ p \mapsto \alpha \} \bowtie \Delta$ and also the equivalence $(\Gamma \cup \Delta)[p \mapsto \alpha] = \Gamma \cup \{ p \mapsto \alpha \} \cup \Delta$ of heaps holds, we conclude $((\Gamma \cup \Delta)[p \mapsto \alpha],\ (\!| t' |\!),\ \eta,\ s) \in \perp\!\!\!\perp$.

(**Pair**) Let $d_1 = [\![t_1]\!]_{\pi,\, \theta_1}\, \rho$ and $d_2 = [\![t_2]\!]_{\pi,\, \theta_2}\, \rho$. Notice that this is a introductory rule, so we prove that $(\Gamma,\ \texttt{Pair}\ ((\!| t_1 |\!), (\!| t_2 |\!)),\ \eta)$ is a primitive realizer. The inductive hypotheses tell us $(\Gamma,\ (\!| t_1 |\!),\ \eta)\ \blacksquare^{\theta_1}\ d_1$ and $(\Gamma,\ (\!| t_2 |\!),\ \eta)\ \blacksquare^{\theta_2}\ d_2$; so we conclude

$$(\Gamma,\ \texttt{Pair}\ ((\!| t_1 |\!), (\!| t_2 |\!)),\ \eta)\ \blacksquare^{\theta_1 \times \theta_2}\ (d_1,\ d_2)\ .$$

(**Fst**) Let $(d_1,\ d_2) = \rho\ \swarrow\ n$; using Lemma 5.15 we know $\eta \cdot n = p\ (\Gamma,\ \Gamma\, p) \in \mathcal{R}^{\theta_1 \times \theta_2}((d_1,\ d_2))$.
We must prove $(\Gamma,\ \texttt{Fst}\ n,\ \eta) \in \mathcal{R}^{\theta_1}(d_1)$; which is the same as proving that for any test $(\Delta, s) \in \mathcal{T}^{\theta_1}(d_1)$, such that $wf(\Gamma,\ (\texttt{Fst}\ n, \eta))$, $wf(\Delta,\ s)$, and $\Gamma \bowtie \Delta$, the configuration is in the observation: $(\Gamma \cup \Delta, \texttt{Fst}\ n, \eta, s) \in \perp\!\!\!\perp$. By anti-execution we might prove $(\Gamma \cup \Delta, \Gamma\, p, \sharp p :: \pi_1 :: \sharp_1 p :: s) \in \perp\!\!\!\perp$. Using the rule O_2 we can remove the update marker on the top of the stack and prove $(\Gamma \cup \Delta, \Gamma\, p,\ \pi_1 :: \sharp_1 p :: s) \in \perp\!\!\!\perp$.
The Lemma 5.21 comes handy to finish the proof because we have a realizer for the product type, thus we can combine it with the test $(\Gamma \cup \Delta,\ \pi_1 :: \sharp_1 p :: s)$; the hypotheses of the lemma are obviously true. In order to conclude $(\Gamma \cup \Delta, \Gamma\, p, \pi_1 :: \sharp_1 p :: s) \in \perp\!\!\!\perp$ we check that the preconditions of the satisfiability relation also hold.

(**Snd**) Analogous to the previous case. □

We notice that the previous theorem is agnostic of the *observation* $\perp\!\!\!\perp$. By choosing a particular observation we can obtain a proof that the compiler is correct with respect to closed terms of a basic type.

Theorem 5.23. *Assume* $wf(\Gamma)$ *and* $[] \vdash t : \mathbf{unit}$*, then* $(\Gamma,\ (\!| t |\!),\ [],\ []) \longmapsto^{*} (\Delta,\ \texttt{Unit},\ \eta,\ [])$ *for some heap* $\Delta \in \mathit{Heap}$ *and machine environment* $\eta \in \mathit{MEnv}$.

Proof. The intended observation is the set of configurations reaching a blocking state with `Unit` as the code and the empty stack: $\perp\!\!\!\perp = \{ w \in \mathit{Conf} \mid w \longmapsto^{*} (\Delta,\ \texttt{Unit},\ \eta,\ []) \}$.

Let us omit the proof that $\perp\!\!\!\perp$ fulfil the four conditions stated in Definition 5.8. Since $wf(\Gamma)$ and $(\Gamma,\ [])\ \blacksquare^{[]}\ \diamond$, by Theorem 5.22 we have $(\Gamma,\ (\!| t |\!),\ [])\ \blacksquare^{\mathbf{unit}}\ [\![t]\!]_{[],\, \mathbf{unit}}\, \diamond = \diamond$.

We may conclude if we prove that $(\Gamma,\ []) \in \mathcal{T}^{\mathbf{unit}}(\diamond)$, which is obvious because the only primitive realizer satisfies it; i.e., $(\Gamma,\ \texttt{Unit},\ \eta,\ []) \in \perp\!\!\!\perp$. □

One may wonder if we have put aside the most difficult part of the proof of correctness, namely proving a similar theorem for higher-order types. Let us remark, that correctness for higher-order types is aptly captured by the notion of approximation: one can deduce that a function is correctly implemented by a low-level code if it computes what it should when provided with enough arguments.

6 Conclusion

We have developed the machinery of realizability combined with biorthogonality for proving the correctness of a compiler targeting Sestoft machine. As far as we know, this is the first work using such techniques for lazy evaluation. We are working on addressing recursive definitions, a most important feature to consider a more realistic language. Adding recursion will introduce diverging computations and one needs to refine the notion of correctness: the compiler must preserve the semantics of terminating programs but also must produce low-level code that hangs the machine when compiling programs whose denotation is the least element of the corresponding domain. We expect to deal with recursive let-expressions in a new paper.

Another feature we would like to add is algebraic data-types; we expect that coproducts can be handled in a similar way to products. Moreover we plan to streamline the extension of the machine with algebraic data-types by following

Sestoft ideas closer, instead of having specific markers in the stack for products and coproducts. In further works we consider the inclusion of recursive data-types. In a different direction, we plan to complete the mechanization in Coq of the whole technical development.

References

[1] Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.* 44, 9 (Aug. 2009), 97–108.

[2] Garret Birkhoff. 1940. *Lattice Theory.* Vol. 25.

[3] Joachim Breitner. 2016. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation.* Ph.D. Dissertation. Karlsruher Institut für Technologie.

[4] Adam Chlipala. 2007. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.* 42, 6 (June 2007), 54–65.

[5] Alejandro Gadea, Emmanuel Gunther, and Miguel Pagano. 2017. The importance of being Extrinsic: coherence and adequacy for a call-by-value language. In *Programming Languages - 21th Brazilian Symposium SBLP 2017.*

[6] Graham Hutton and Patrick Bahr. 2017. Compiling a Fifty Year Journey. (2017). In preparation.

[7] Guilhem Jaber and Nicolas Tabareau. 2010. Krivine realizability for compiler correctness. In *Workshop LOLA 2010, Syntax and Semantics of Low Level Languages.* Edinburgh, United Kingdom.

[8] Guilhem Jaber and Nicolas Tabareau. 2011. The Journey of Biorthogonal Logical Relations to the Realm of Assembly Code. In *Workshop LOLA 2011, Syntax and Semantics of Low Level Languages.* Toronto, Canada.

[9] Jean-Louis Krivine. 1994. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic* 68, 1 (1994), 53 – 78.

[10] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 144–154. https://doi.org/10.1145/158511.158618

[11] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[12] John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science 1*, Vol. 19. American Mathematical Society, 33–41.

[13] Andrew M. Pitts. 2000. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* 10, 3 (2000), 321–359. http://journals.cambridge.org/action/displayAbstract?aid=44651

[14] A. M. Pitts and I. D. B. Stark. 1998. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics*, A. D. Gordon and A. M. Pitts (Eds.). Cambridge University Press, 227–273.

[15] John C. Reynolds. 2003. What Do Types Mean?: From Intrinsic to Extrinsic Semantics. In *Programming Methodology*, Annabelle McIver and Carroll Morgan (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 309–327.

[16] Leonardo Rodríguez, Miguel Pagano, and Daniel Fridlender. 2016. Proving Correctness of a Compiler Using Step-indexed Logical Relations. *Electr. Notes Theor. Comput. Sci.* 323 (2016), 197–214. https://doi.org/10.1016/j.entcs.2016.06.013

[17] Leonardo Rodríguez. 2017. *Compilación Certificada sobre Máquinas Abstractas de Evaluación Normal.* Ph.D. Dissertation. FaMAF, Universidad Nacional de Córdoba.

[18] Peter Selinger. 2003. From Continuation Passing Style to Krivine's Abstract Machine. Manuscript. (2003). Available in Peter Selinger's web site.

[19] Peter Sestoft. 1997. Deriving a lazy abstract machine. *Journal of Functional Programming* 7, 3 (1997), 231–264.

[20] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In ***ICFP** '16: Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming.* ACM Press, 60–73. https://doi.org/10.1145/2951913.2951924

Wolfram for data processing and visualization

Stijn Schildermans
ES&S-DTAI
KU Leuven
Diepenbeek, Belgium
stijn.schildermans@kuleuven.be

Kris Aerts
ES&S-DTAI
KU Leuven
Diepenbeek, Belgium
kris.aerts@kuleuven.be

Abstract

Wolfram is a modern multi-paradigm, mainly functional programming language with a strong focus on user-friendliness and intuitive use. Therefore it is highly declarative in nature and focuses on a transparent and uniform syntax. Furthermore, Wolfram possesses several unique features that greatly add to the scope of the language and simplify tasks such as software deployment.

This paper focuses on the use of Wolfram as a functional programming language for data processing and visualization. The main goal is to provide insight in the benefits and weaknesses of this language compared to other (functional) technologies for these specific tasks. This is done via a case study in which Wolfram code for a specific task is compared to analogue solutions in different languages.

During the case study it quickly became apparent that the Wolfram syntax is unique and intuitive to use. Because of its very declarative nature complex tasks often take only a few lines of code, and are easy to perform. On the other hand, issues with interpretation of code started to appear when Wolfram was used for more complex tasks. The performance of Wolfram also appeared to be sub-par compared to other popular programming languages. Therefore, this case study determined that Wolfram is a very suitable language for quickly creating small applications for data processing and visualization, but for complex and computationally expensive applications other languages might be more suitable.

Keywords Wolfram, Case study, Performance, Data processing, Data visualization

ACM Reference format:
Stijn Schildermans and Kris Aerts. 2017. Wolfram for data processing and visualization. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, Bristol, UK, August 30–September 1, 2017 (IFL'17)*, 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Wolfram [6] is a modern programming language that aims to combine features from several programming paradigms to create an intuitive programming language that can be used for quickly and easily performing a multitude of tasks ranging from data processing to creating interactive web forms. Wolfram also possesses many interesting features that can not be found in any other programming language.

One application domain in which an intuitive and functional programming language might prosper is that of data processing and visualization. This is the case because the functional programming mindset -viewing the program as a transformation from a certain input to a certain output through the application of first-order functions- converges perfectly with the needs of this application domain. Therefore this paper aims to explore the specific benefits and weaknesses of using Wolfram for this task.

As an extension to this core objective this paper also explores some of the features of Wolfram that go beyond those of most other programming languages and are useful for the studied application domain. After all, Wolfram is mostly known from the search engine Wolfram|Alpha, which is directly integrated in the language. Wolfram also provides an integrated cloud platform, which promises to simplify the deployment process greatly. It is evident that these features should be taken into account when forming an opinion about the suitability of Wolfram for the task of data processing and visualization.

Thus, the core goal of this paper is to form a judgment on the value of Wolfram as a platform for performing tasks such as data processing and visualization, considering the syntax and properties of the language itself, as well as the additional features the language provides that can not be found in alternative programming languages.

This paper is based on the main author's Master's thesis [3]. The case study discussed in this paper is described in more detail in said thesis.

2 Background: the Wolfram language

The Wolfram programming language originates from Mathematica [8], which is a commonly used platform for scientific computation and modelling. Although the language is syntactically radically different from the commonly used programming languages today, it is easy to use and intuitive. This section gives a brief overview of the Wolfram syntax and design principles, as well as the most important unique features it possesses that form an added value for the task of data processing and visualization.

IFL'17, August 30–September 1, 2017, Bristol, UK
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The official Wolfram language reference documentation [7] is very complete and concise. This documentation forms the main source for this section. Interested readers are encouraged to explore the language further via this medium.

2.1 Syntax

In Wolfram, everything from a mathematical operation to a complex geometrical figure is displayed as a uniform symbolic expression. These expressions are formed by a head which serves as a name, and a set of arguments. Expressions are used for declaring constants as well as variables and functions.

Wolfram supports two kinds of assignments: immediate assignment and delayed assignment. Immediate assignments are calculated immediately and denoted using '=' as in 'a = 2', while delayed assignments are recalculated every time the assigned expression is evaluated. Delayed assignments are denoted with ':=' as in 'b := 5'.

Functions in Wolfram are just expressions that were assigned using the delayed assignment method and take an argument list. There is no further distinction between functions and other data in Wolfram. This is a good example of how Wolfram considers functions as first class citizens. The fact that constants and functions can have exactly the same notation is mathematically pure and a strong functional feature of the language. The code sample below illustrates the principles described above.

```
a = 2;
b := 5
c[x__] := 5
d[x__] := Max[x]
```

The expressions a and b are resp. an immediate and delayed assigned constant. The expressions c and d illustrate that constants and functions can have exactly the same notation, which is mathematically pure and is a strong functional feature of the language.

Wolfram is partly because of its functional nature a highly declarative language. Code is interpreted flexibly by the run time environment which partially compensates for mistakes by the programmer. Furthermore, the language is dynamically typed to further simplify the programming process.

Wolfram also incorporates many well known features from the functional programming paradigm. One example is the possibility to map functions on lists. Wolfram provides several ways of doing this, and even gives the programmer the possibility to use very compact syntax for this common task. The code sample below illustrates this.

```
Map[f, {a, b, c}]
f /@ {a, b, c}
```

The lines of code illustrated above have the same effect. Both map the function f on the list {a,b,c}. Wolfram has many features like this for applying functions directly to complex data structures. Interested readers are referred to [4].

2.2 Features

Besides the elegant and compact syntax the Wolfram language has several unique features that make tasks like data processing and visualization among others a lot easier than in other languages. This section summarizes the most important ones for the studied application domain.

2.2.1 Wolfram|Alpha

The Wolfram language contains several built-in functions that allow the developer to directly interact with the Wolfram|Alpha search engine programmatically. This is a very powerful tool that can be used to collect relevant side information about all kinds of data and real-world entities. This feature can be very useful for data processing. The code below illustrates this.

```
Bristol["Population"]
```

Wolfram|Alpha is used to interpret 'Bristol' in the code sample above. This results in a real-world entity of the type 'city'. This can be interpreted as an object with a set of properties that can be queried. In this example, the property 'Population' is extracted from the real-world entity. The result of this code is '60 147 people'.

Note that anything that has to be interpreted by Wolfram|Alpha has to be entered into a special search field that can be invoked anywhere in the code by pressing 'ctrl + enter'. This also implies that Wolfram code is best always written using a dedicated IDE, i.e. a version of Mathematica[1] or the online Wolfram Development Platform[2].

2.2.2 Data visualization functions

Another interesting feature of the Wolfram language is that it contains several built-in functions that make generating complex graphics from a data set a matter of a few lines of code. Functions like *ListPlot* generate graphs from a list of input data very easily. The code below generates a simple list plot from some arbitrary input data.

[1] https://www.wolfram.com/mathematica/
[2] https://www.wolfram.com/development-platform/

```
data = {{1,1},{2,5},{3,1}};

ListLinePlot[data]
```

Figure 1 shows the result of this code.

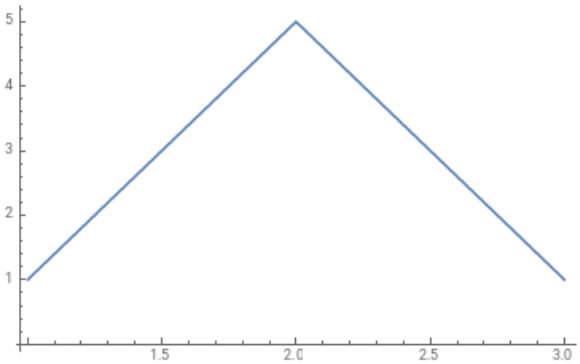


Figure 1. Simple list plot in Wolfram.

The result of this minimal code is a simple graph that already provides all essential features to interpret the data. Through the use of so called 'options' these graphics can be customized further in many ways. It is possible to add plot labels, a legend, a custom color scheme, etc. The customization options are numerous and thorough. This again shows the highly declarative nature of Wolfram. Thanks to this approach the code is kept as compact as possible while maintaining possibilities for customization.

2.2.3 Wolfram Cloud

A final feature of the Wolfram Language that is interesting for any type of application, is the possibility it provides to instantly deploy any Wolfram code to the integrated Wolfram Cloud through the use of the built-in function *CloudDeploy*. Wolfram also has dedicated functions for creating RESTful web APIs, and even embedding cloud-deployed Wolfram code directly in other web pages. For example, the code below generates a simple *Listlineplot* like in section 2.2.2, but this time this plot is deployed to the cloud in the form of an API that takes a sequene of integers as input, and returns a graph corresponding to the input as a png image.

```
CloudDeploy[APIFunction[
"d"->DelimitedSequence["Integer"],
ExportForm[ListLinePlot[#],"PNG"]&]]
```

The code above shows just how compact and declarative the Wolfram language is for quickly creating APIs for data visualization. The returned image can be directly embedded in any other web page or program. Wolfram even provides functions that generate the code required for this.

Note that the input to the *ListLinePlot* function is a list of numbers in this case, rather than a nested list of x,y-coordinates like in section 2.2.2. Wolfram automatically interprets a plain list in this case as a list of x-y coordinates, with the x-coordinates being an ascending sequence of natural numbers starting at 1. Thus, the resulting list plot will in this case be identical to the results from section 2.2.2, provided the parameter *d* gets the value '1,5,1'. This is a great example of the dynamic way in which Wolfram interprets code, which often makes the task of quickly creating simple services for data processing and visualization very easy compared to other programming languages.

3 Method

As stated above the main goal of this paper is to determine whether Wolfram is a suitable language for data processing and visualization, and why Wolfram should or should not be chosen for this task above other languages. The method used to determine this is a case study. Since this paper is based on the Master's thesis of the main author, said case study directly originates from [3].

In [3] several programming languages are compared by writing a recursive algorithm that generates Pascal's triangle. This is a mathematical structure in which numbers are arranged in a triangular shape, and each element is the sum of the two elements directly above it, with all the 'border elements' having the value '1' [1]. This is demonstrated in figure 2.

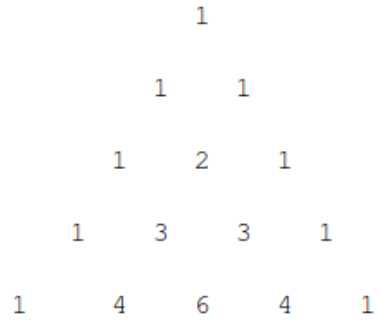


Figure 2. Pascal's triangle [1].

The studied languages are Java, C#, F# and Wolfram. Since this paper only covers Wolfram, only the results for Wolfram will be discussed in detail, and the other languages will be used as a reference. All implementations are then compared based on several predetermined parameters, being:

- Readability,
- Code length,
- Ease of use,
- Performance.

The performance of the studied languages in [3] was compared from within Wolfram using the J/Link [9] and .NET/Link [5] libraries. These libraries start an independent Java resp. .NET run time environment on the local machine and import the results in Wolfram for further processing. In this case this allowed for a uniform timing of the performance of the various tested languages. From within Wolfram the Pascal's triangle algorithm is called for different sizes of the triangle, and the time needed to compute the entire triangle is measured programmatically for each implementation. This is repeated for 10 iterations. The average time needed for generating the triangles is calculated and visualized using Wolfram.

Both the code for generating Pascal's triangle -requiring a lot of computation- and for processing and comparing the timings, are use cases for determining the suitability of Wolfram for data processing and visualization tasks. Appendix A provides the full Wolfram script that processes the data from the performance tests. The function takes a list of language names as input and custom decoder functions were written to link these names to the actual data to be generated and the function to be used for curve fitting in the visualization step. The result of this is a plot such as figure 3.

4 Results

This section discusses the results of the comparative case study described above. The results are divided into several major sections that are discussed in detail. Finally, a general overview is given and a direct comparison is made with the languages Java, C# and F#.

4.1 Implementation

This section discusses implementation-oriented criteria concerning the suitability of Wolfram for the tasks of data processing and visualization. Several advantages of Wolfram are illustrated, as well as some drawbacks.

4.1.1 Intuitive and concise code

As stated in section 3, this case study is based upon the aforementioned recursive Pascal's triangle algorithm. This can be seen as an algorithm that generates/processes data and can be identically implemented in many languages, thus making a transparent comparison possible. The said algorithm is very easy to implement in Wolfram. The code below shows the implementation used for this research.

```
ptr[s_Integer] /; s == 1 := {{1}}
ptr[s_Integer] :=
  Table[ptrv[i, j], {i, 1, s},
    {j, 1, i}]
ptrv[r_Integer, c_Integer]
  /; c == 1 || c == r := 1
ptrv[r_Integer, c_Integer] :=
  ptrv[r - 1, c - 1] + ptrv[r - 1, c]
```

The code above is a good illustration of the compactness of the Wolfram language. Like many functional languages it uses pattern matching for determining control flow as a compact and elegant alternative for if-statements from the imperative programming world. This results in very readable and concise code. Furthermore the declarative and flexible nature of the language is clear from the use of the Table function for generating a nested list in which the length of the inner lists is dependent on the index of the outer list; a task that is relatively complicated in imperative languages and usually requires two nested for loops. The code below shows a reference implementation in Java with the same functionality.

```
public static int[][]
pascalTriangle(int size) {
    int[][] points = new int[size][];
    for (int i = 0; i < size; i++) {
        int[] row = new int[i + 1];
        for (int j = 0; j <= i; j++)
            row[j] =
            getValueAtPoint(i, j);
        points[i] = row;
    }
    return points;
}
public static int
getValueAtPoint(int row, int col) {
    if (col == 0 || col == row)
        return 1;
    else return
    getValueAtPoint(row - 1, col - 1)
    + getValueAtPoint(row - 1, col);
}
```

4

It is clear that the Java version is much longer, and contains much more clutter and syntactical elements. Also the Java version is much less readable, and the many variables that have to be explicitly assigned and read make it more challenging to get the algorithm error-free.

The compact and intuitive syntax might set Wolfram apart from imperative languages, but many other functional languages like Haskell and F# share these properties. However, generally during this research Wolfram appeared to possess the most compact and intuitive syntax of all tested languages, and this without sacrificing readability. Below is the F#-equivalent of the recursive Pascal's triangle algorithm for comparison.

```
let rec pascalTriangleValue row col =
match col with
    | 1 -> 1
    | col when col = row -> 1
    | _ -> pascalTriangleValue
           (row-1) (col-1)
           + pascalTriangleValue
           (row-1) col
let pascalTriangle size = [1..size]
    |> List.map (fun r->[1..r]
    |> List.map
    (fun c->pascalTriangleValue r c))
```

The F# code is a lot more compact and readable than the Java version, but is still longer than the Wolfram code. Also, it is clear that this code adheres much more to classic functional principles, while Wolfram takes its own, unique approach. In conclusion to this section, it can be said that Wolfram has a unique, very compact and very declarative syntax that is greatly beneficial for tasks like data processing.

4.1.2 Unexpected behavior

The many design choices of the language that are oriented to ease of use have an effect on the overall clarity of what the code is exactly doing, which can sometimes lead to unexpected behavior. A good example is the dynamic typing system of Wolfram. In combination with the flexible interpretation of code this means that Wolfram code almost never crashes, but this also means that functions even work on data that is not suitable for that function, and just return an unexpected result. This can be easily demonstrated by a simple code example:

```
AllTrue[{{1}}, ListQ]
AllTrue[{1}, ListQ]
AllTrue[1, ListQ]
```

The function AllTrue expects a list and a function that returns a boolean as input, and returns a boolean depicting whether or not all the elements in the list comply with the boolean test function. The function ListQ assesses whether or not its input is a list.

In the first expression in the code sample above, the input is a list with one element, being itself a list with one element. The expression thus evaluates to 'True' as expected.

In the second expression however, the list of input arguments contains only the number one, which is not a list, so the expression evaluates to 'False'.

In the third expression, the first input element is not even a list, so most other programming languages would throw an exception at this point. Wolfram does not however, and reinterprets the input '1' so that the expression does evaluate successfully. Strangely though, this expression evaluates to 'True', while intuitively 'False' should be the correct answer. This demonstrates that the great flexibility of Wolfram might indeed lead to unexpected results, and this can happen quite easily. For complex programs this might be a serious concern, as behavior like this might induce errors that are difficult to find. Luckily, Wolfram contains several functions that can give the programmer stricter control over the evaluation of code[3].

4.2 Powerful features

What really sets Wolfram apart from the other functional languages for the task of data processing and visualization are the built-in features described previously in this paper. The instant access to the wealth of information that Wolfram|Alpha provides and the flexibility of the Wolfram language to interpret this information can be of great value when processing certain kinds of data. One example that proved particularly useful during this research is the flexible way Wolfram can interpret units of time. Dates and times can be directly subtracted from each other and instantly used in graphs and other applications. All formatting is abstracted from the programmer, and Wolfram|Alpha is used for many conversions behind the scenes. The code below illustrates this feature.

```
t = Now - 1 wk - 5 min
```

[3] http://reference.wolfram.com/language/guide/EvaluationControl.html

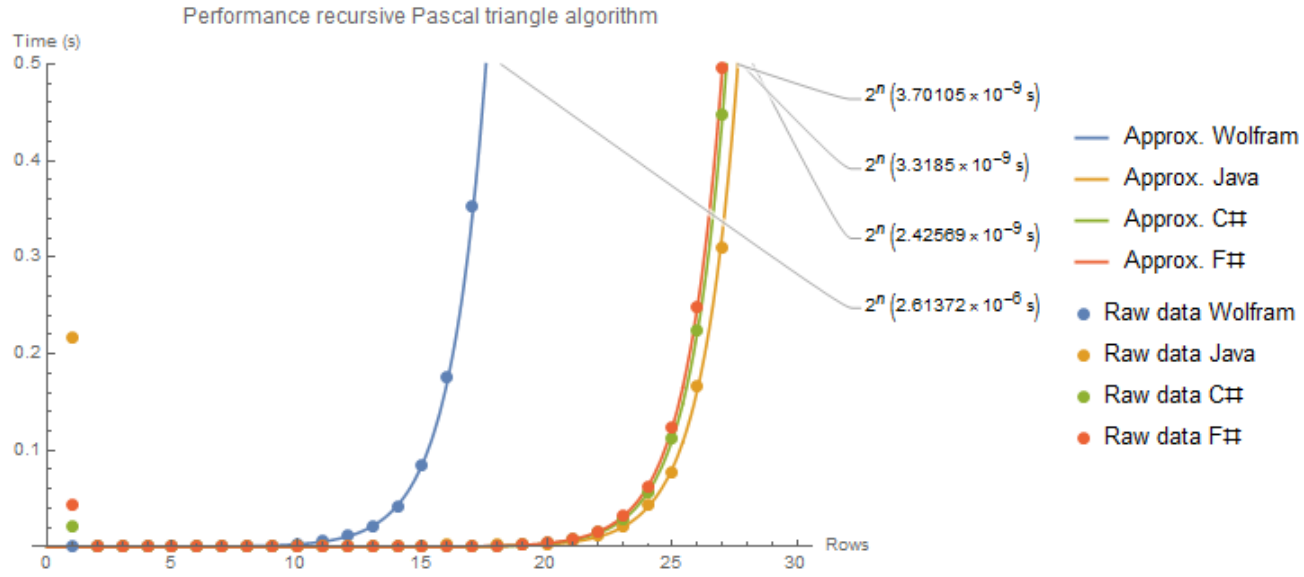


Figure 3. Complex plot in Wolfram.

The result of this code is a DateTime expression that represents the point in time 1 week and 5 minutes before the variable t was assigned. This DateTime expression can be used seamlessly in other code. Wolfram|Alpha interprets the terms 'wk' and 'min' and converts them to the appropriate form. This makes working with dates and times much more intuitive in Wolfram than in other languages.

The features described in section 2.2 have also proven to be very useful for this research. The intuitive data visualization functions made visualizing test results fast and easy to implement. Deploying Wolfram code was in this case study also remarkably easier than deploying programs written in any other programming language, thanks to the integrated Wolfram Cloud. This is a major advantage for creating quick services for processing and visualizing data, that can be easily integrated into other applications as demonstrated in section 2.2.3.

4.3 Elegant and flexible data visualization

This section discusses the experiences during this case study for visualizing the results of the performance tests discussed in section 3. The code used to visualize the results of the performance tests in this case study is shown in appendix A. In this example input data is first processed, and then displayed in a plot, together with an approximation of the data generated by curve-fitting. Labels, a legend, etc. were all added by means of options, as discussed in section 2.2.2. The result of this code is shown in figure 3.

As can be seen in figure 3, these options are a powerful tool with many possibilities. Each data visualization function has many options that allow the programmer to do virtually anything he would reasonably want to do in a very compact and intuitive way.

4.4 Performance

The performance of Wolfram was tested and compared with that of Java, C# and F# using the method described in section 3. Wolfram turned out to be far slower than any of its competitors. Table 1 shows the relative performance of Wolfram compared to the other languages tested in [3]. The resulting numbers are a factor that indicates the time needed by Wolfram to perform this task compared to the time needed by the other languages.

Table 1. Relative performance of Wolfram compared to the other languages tested in [3].

Language	Wolfram
Java	1070
C#	800
F#	700

These results indicate that Wolfram is indeed much slower than any of the other tested languages in [3]. This has serious repercussions for the application domain of data processing and to some extent also visualization. Wolfram turns out to be 1000 times slower than Java, which is of course unacceptable for any application where hardware resources are limited, or where performance is of any kind of importance.

4.5 General

Table 2 shows a comparison between Wolfram and all the other tested programming languages in [3] based on several parameters discussed in this case study. The table depicts a qualitative judgment, relative to the other languages.

Table 2. Comparison between Wolfram and several other programming languages [3].

Language	Readability	Length	Ease of use	Performance
Java	OK	20	Good	Excellent
C#	OK	25	Good	Good
F#	Good	10	Bad	Good
Wolfram	Good	5	Excellent	Poor

Compared to the other tested languages Wolfram scores very well in the implementation-oriented categories. The only major weakness of Wolfram is performance. The very high-level approach of Wolfram and the great ease of use that is induced by it clearly have a cost when it comes to performance. Also the fact that Wolfram is an interpreted rather than a compiled language can contribute significantly to its poor performance [2]. This might be a serious concern for many data processing tasks, although the many advantages of the language might compensate for this fundamental problem in many cases.

It is also important to note that the compact, typeless syntax in combination with the flexible interpretation of Wolfram code might lead to unexpected results as described in section 4.1.2. This is especially a drawback for large and/or complicated programs and can lead to bugs that are difficult to detect.

5 Conclusion

Wolfram is a modern programming language that takes a unique approach to software development through its very compact and uniform syntax, which encompasses many functional features as well as aspects from other programming paradigms. Thanks to the declarative and functional nature and several unique features such as very easy cloud deployment and integration with Wolfram|Alpha, Wolfram is an excellent language for applications such as data processing and visualization from an implementation standpoint, although for large projects the dynamic typing system and flexible interpretation of code might be counter productive at times.

The largest drawback of Wolfram for the studied application domain however is its very poor performance. This case study showed relative performance numbers that are not acceptable for computationally expensive tasks compared to other popular programming languages; both functional and object-oriented.

Wolfram is thus a powerful tool for quickly creating and deploying applications or services for data processing and visualization, as long as the task at hand is not too large in scope or too computationally expensive. However, when performance is an issue, other programming languages might be more suitable.

Acknowledgments

Special thanks Tom Schrijvers who was closely involved in the constitution of this paper and provided comprehensive and vital feedback.

References

[1] 2017. Pascal's Triangle. (2017). http://mathforum.org/workshops/usi/pascal/pascal_intro.html

[2] John S. Riley. [n. d.]. Interpreted vs. Compiled Languages. ([n. d.]). http://dsbscience.com/freepubs/start_programming/node6.html

[3] Stijn Schildermans. 2017. *Exploratie van functionele en declaratieve ontwikkelmethodes voor cloud programming.* Master's thesis. UHasselt.

[4] Wolfram. [n. d.]. Applying Functions. ([n. d.]). https://www.wolfram.com/language/fast-introduction-for-programmers/en/applying-functions/

[5] Wolfram. 2017. .NET/Link User Guide. (2017). http://reference.wolfram.com/language/NETLink/tutorial/Overview.html

[6] Wolfram. 2017. Wolfram: Computation Meets Knowledge. (2017). http://www.wolfram.com/

[7] Wolfram. 2017. Wolfram Language and System Documentation Center. (2017). http://reference.wolfram.com/language/

[8] Wolfram. 2017. Wolfram Mathematica. (2017). https://www.wolfram.com/mathematica/

[9] Wolfram. 2017. Writing Java Programs That Use the Wolfram Language. (2017). http://reference.wolfram.com/language/JLink/tutorial/WritingJavaProgramsThatUseTheWolframLanguage.html#15141

A Wolfram code for processing results of performance tests

```
testPerformance
[langs_List, iterations_Integer] :=
testPerformance[langs,iterations,0.5]
testPerformance[langs_List,
iterations_Integer, yRange_]:=Catch[
testData = Table[ReleaseHold[#],
{i, iterations}] &
/@ ((data[#]) &) /@ langs;
avg = ({First[First[#]],
Mean[[#[[2]] &) /@ #]} &)
/@ Map[Flatten,
Table[(Cases[#, {i, _}] &)
/@ #, {i, Length[#[[1]]]}], {2}]&
/@ testData;
listPlot = ListPlot[avg,
PlotStyle -> PointSize[Large],
PlotRange -> {0, yRange},
PlotLegends -> ("Raw data " <> # &)
/@ langs,
TargetUnits -> "Seconds"];
tupels = Table[{avg[[i]],
fittingFunction[langs[[i]]]},
{i, Length[langs]}];
fit = Fit[(#[[1]])[[2;;]],
#[[2]], n ] & /@ tupels;
plot = Plot[Evaluate[fit],
{n, 0, Max[Length[#[[1]]] &
/@ testData]},
PlotLabel ->
"Performance Recursive Pascal Triangle",
PlotRangeClipping -> False,
PlotRange -> {0, yRange},
AxesLabel -> {"Rows", "Time (s)"},
PlotLegends -> ("Approx. " <> # &)
/@ langs,
PlotLabels -> fit];
Show[plot, listPlot,
ImageSize -> Large],
InvalidInputException];
```

Type Error Customization in GHC (draft)

Controlling expression-level type errors by type-level programming

Alejandro Serrano
A.SerranoMena@uu.nl
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

Jurriaan Hage
J.Hage@uu.nl
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

Abstract

Embedded domain specific languages (DSLs) are a common pattern in the functional programming world, providing very high-level abstractions to programmer. Unfortunately, this abstraction is broken when type errors occur, leaking details of the DSL implementation. In this paper we present a set of techniques for customizing type error diagnosis in order to avoid this leaking. These techniques have been implemented in the GHC Haskell compiler.

Our customizations are declared in the type signatures of functions provided by the DSL, leading to type error message that are context-dependent: the same kind of error can be reported in a different way depending on the particular expression in which it occurs. We make use of the ability to manipulate constraints using type-level programming which is already present in GHC, and which enables reuse and abstraction of common type error patterns.

ACM Reference format:
Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC (draft). In *Proceedings of 29th Symposium on Implementation and Application of Functional Languages, Bristol, United Kingdom, 30 August – 1 September 2017 (IFL 2017),* 15 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Domain Specific Languages (or DSLs for short) are a widely used technique. Their main advantage is the effectiveness with which domain experts can describe the solution to a problem, even with little prior programming experience. Examples abound, ranging from SQL for database processing to describing drawings (Yorgey 2012) and even music harmony analysis (Koops et al. 2013). There are two main approaches to developing DSLs: in the *external* approach a new compiler toolchain is built, including a parser, a code generator and associated tooling. The drawback of such a standalone DSL is the amount of work required; for that reason frameworks have been designed to help developers in this task (Voelter 2013).

The second approach is that of *internal* or *embedded* DSLs (Hudak 1996). Embedded DSLs are developed as libraries in a host language, in order to reuse all the machinery which is already implemented for that host. Another advantage of embedded DSLs is the ease with which DSLs can be combines in a single application. The embedded approach is common in the functional programming world and is the focus of this paper. From now on, when we talk of DSLs we mean embedded DSLs.

DSLs provide a very high level of abstraction to the programmer, who communicates with the compiler using domain terms. Alas, this abstraction is broken when a type error occurs. From the point of view of the compiler, a DSL is merely a library, and by itself cannot phrase error messages in domain terms. The result is that errors leak details of the implementation of the DSL to the user, a serious disadvantage (Hage 2014). Some authors of DSLs include a special section in the documentation devoted to explain the most common type errors, like in `diagrams` (Yorgey 2016, section 5.5), whereas others like `persistent` (Snoyman 2012) directly document the library internals. Both solutions are clearly suboptimal.

This problem has been acknowledged by several authors, who have proposed various mechanisms for DSL authors to customize the errors generated by the compiler (Christiansen 2014; Heeren et al. 2003a; Plociniczak et al. 2014; Serrano and Hage 2016b; Wazny 2006), so that type error messsages can be phrased in terms of the domain instead of the underlying encoding of that domain in the host language. This paper introduces type error customization in GHC, the main implementation of the Haskell language (Marlow et al. 2010), showing that powerful customized type error diagnosis is available for a language as large and complex as Haskell.

1.1 Type Errors

Before we describe our solution, we need to understand the different sources of errors during type checking. We stress that our focus is on type errors, so we assume that code has already been successfully parsed and names have already been resolved; errors may only arise from problems with typings.

Earlier in this paper, we mentioned `diagrams`, a DSL for representing drawings and animations. The fundamental type in this library is *QDiagram b v n m*, where the type parameters represent, in order, the drawing back-end – such as SVG or bitmap–, the vector space – `diagrams` deals with 2-D and 3-D images in a uniform way –, the type of numeric coordinates, and the type used for annotating the diagrams. In most cases, we can only combine diagrams when all of these type parameters coincide. One such example is the *atop* combinator used to arrange diagrams vertically:

$$\begin{aligned} atop \;::\; & (OrderedField\ n, Metric\ v, Semigroup\ m) \\ & \Rightarrow QDiagram\ b\ v\ n\ m \to QDiagram\ b\ v\ n\ m \\ & \to QDiagram\ b\ v\ n\ m \end{aligned}$$

The first group of type errors consists of those arising from *inconsistencies*. In this case, an expression in the program is expected to have two different incompatible types. A simple example is when we pass to *atop* the arguments *True* and `'c'`:

```
Couldn't match type 'QDiagram b v n m' with type 'Bool'
```

This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213).

IFL 2017, Bristol, United Kingdom

DOI: 10.1145/nnnnnnn.nnnnnnn

Inconsistencies might also arise at a deeper level. For example, the arguments to *atop* might be rightful diagrams, but living in different vector spaces. In that case, the default error message:

```
Couldn't match type 'V2' with type 'V3'
```

already assumes some knowledge of the internals of the library, namely that *V2* and *V3* are the types used to represent the two- and three-dimensional vector spaces, and that this information is encoded in the diagram type. A better diagnosis uses domain terms:

```
Vector spaces do not coincide: V2 vs. V3
```

Since version 8.0, users of GHC have the ability to produce custom inconsistency errors during type resolution by means of a special construct *TypeError* (Diatchki 2015). The archetypal example is providing a better user experience when using the *show* function which converts a value into its string representation. In the general case, we do not want to allow functions to appear as arguments to *show*. The solution is to write an instance of the *Show* type class which has one *TypeError* constraint as context:

```
instance TypeError (Text "Functions cannot be shown")
      ⇒ Show (a → b)
```

Now, if the user tries to compile *show* ($\lambda x \to x$), he or she receives that custom message instead of the more cryptic[1]

```
No instance for (Show (t0 -> t0))
arising from a use of 'show'
```

Haskell, among other languages, supports ad-hoc overloading via type classes. Functions can require an instance of a type class for the specific type the polymorphic function is applied to. The *atop* function introduced previously uses such a mechanism: it requires the vector space v to have a *Metric* instance, the number field n must be ordered and the annotation type m must form a semigroup. Operationally, each type class requirement is transformed into an extra argument which is filled in automatically by the compiler as part of the typing process.

This mechanism gives rise to a second source of errors: an instance of a given type class is not available for a specific type. This is the case in the previous example where a function cannot be shown. Given the operational view we have just explained, it makes sense for the compiler to signal an error, since without an instance there is no way to fill in the extra argument coming from the type class requirement. We note that programs written in languages featuring implicit parameter resolution – such as Scala, Agda, Idris, and Coq – can also exhibit this kind of error.

If you think of type checking as a process where constraints between types are generated and then solved – an intuition we shall formalize in just a moment –, these errors come from constraints which are *left undischarged*.

As with inconsistency errors, left-undischarged errors are also amenable to customization. One good example is `persistent`, a type-safe database library developed for Haskell. When using this library, each data type which can be persisted needs to be annotated in a certain way (Snoyman 2012), resulting in an instance of the *PersistEntity* type class. This requirement is visible in the types of the query and update operations:

```
insertUnique
  :: (MonadIO m, PersistUniqueWrite backend, PersistEntity record)
  ⇒ record → ReaderT backend m (Maybe (Key record))
```

If we find out that an instance of *PersistEntity* is missing, we can do better than plainly reporting this fact: we can point the user to the documentation discussing how to properly annotate the data type. Even more, the user need not be aware of the fact that type classes are involved in the implementation, we can just output:

```
Data type 'Person' is not declared as an entity.
Hint: entity definition can be automatically derived.
Check the documentation at http://www.yesodweb.com/...
```

We note that the implicit parameter system in Scala has some support for customization in the form of the *@implicitNotFound* annotation (Scala Team Scala Team). Whenever an implicit value cannot be inferred for a given trait, the declared message is shown instead of the compiler's default one.

```
@implicitNotFound (msg = "Data type is not an entity")
trait PersistEntity [A] {...}
```

A third kind of error which may arise during type checking is *ambiguity* errors. Ambiguous types are reported when for a given type variable the compiler is unable to find a ground type to unify with, nor is it able to quantify over the type variable. In our own experience, these problems do not often call for a domain-specific type error diagnosis solution, but rather for more generic techniques such as *defaulting* (Heeren and Hage 2005; Marlow et al. 2010). For that reason, we consider in this paper only inconsistencies and left-undischarged errors.

1.2 Context-Dependent Errors

We have discussed how several languages provide support for customizing left-undischarged errors and generate inconsistency errors with a custom message. Is this enough to provide good error messages for embedded DSLs? Not at all! When using a DSL, the context in which an expression appears gives meaningful information which should be considered when building an error message.

As an example, let us consider a version of *atop* in which each single constraint over the types appears as a single constraint in the source code. In particular, instead of a single equality constraint between the arguments,

$$QDiagram\ b_1\ v_1\ n_1\ m_1 \sim QDiagram\ b_2\ v_2\ n_2\ m_2$$

we indicate the equalities between the sub-components. Note that we follow GHC's convention of denoting the equality between types t_1 and t_2 as $t_1 \sim t_2$.

```
atop :: (d1 ~ QDiagram b1 v1 n1 m1,
         d2 ~ QDiagram b2 v2 n2 m2,
         b1 ~ b2, v1 ~ v2, n1 ~ n2, m1 ~ m2,
         OrderedField n1, Metric v1, Semigroup m1)
      ⇒ d1 → d2 → d1   -- Argument types replaced by variables
```

From the compiler's point of view, the equality $b_1 \sim b_2$ is no different from $v_1 \sim v_2$. But from the DSL point of view, these are quite different: the former checks an agreement between back-ends, whereas the latter speaks about vector spaces. If an inconsistency is found between values of v_1 and v_2 we want to report:

```
Vector spaces do not coincide: V2 vs. V3
```

[1] In newer versions of GHC, that error message is extended with a hint (`maybe you haven't applied a function to enough arguments?`). However, this special reporting is baked into the compiler, DSL authors are not able to provide such hints.

It is not only the position of a variable in a longer type, what custom message to report also depends on the function being used. Consider the following combinator:

$$\begin{aligned} &\mathit{strokePath} \;::\; (\mathit{TypeableFloat}\ n, \mathit{Renderable}\ (\mathit{Path}\ \mathit{V2}\ n)\ b) \\ &\qquad \Rightarrow \mathit{Path}\ \mathit{V2}\ n \to \mathit{QDiagram}\ b\ \mathit{V2}\ n\ \mathit{Any} \end{aligned}$$

In this case we demand that the argument lives in a two-dimensional space by setting the corresponding type parameters to *V2*. Whereas in the *atop* case there are two vector spaces with equal status, in the case of *strokePath* the two-dimensional requirement is key, and thus we prefer a message along the lines of:

```
'strokePath' requires a V2 path, but was passed a V3
```

The dependence of the wording of the type error message on the context is something we cannot achieve with any of the previous solutions. *TypeError*s can generate new errors or replace left-undischarged ones with inconsistencies, but only reflecting on information at the type level. In a similar fashion, *@implicitNotFound* belongs to a particular trait; the same custom message is reported regardless of where the trait is required.

1.3 Contributions

In this paper we present two mechanisms for context-dependent type error customization. The first technique (§ 3) is very simple to implement, but has some limitations with respect to ordering the checks done inside the compiler. In § 4 we present another technique which lifts these limitation.

Both techniques have several advantages compared to previous proposals on type error customization:

- They do not require changes to libraries: a wrapper providing better error messages can be developed independently. This also opens the door to distributing several versions of the same API whose error messages target specific users.
- Error customizations blend with other type-level constructs. That means that we can use the type-level facilities in GHC, such as type families, to provide facilities for abstraction and reuse when implementing our customizations.
- Custom messages become part of the exported type interface, not part of a document which users might not be aware of or may become outdated.

We have implemented both techniques inside of GHC, and developed a library of useful combinators targeting common type error patterns (§ 5). Using this library, we have wrapped some popular libraries like the aforementioned `diagrams` and `persistent`. An example of the result is given in Figure 1 (:∼: represents inequality). Note that we can piggy-back on the *Diagrams.Combinators* version in the implementation. In Appendix B we fully describe the annotations needed for the `path` library.

2 Constraint-based Type Checking

Type checking in GHC, starting with version 7, is based on the OutsideIn(X) system (Vytiniotis et al. 2011). OutsideIn(X) belongs to the family of type systems whose operation is based on constraints; this also includes other functional languages such as ML (Pottier and Rémy 2005) and Swift (Swift Team 2016). OutsideIn(X) is itself generic, and is instantiated for a constraint system X. In the specific case of GHC, X includes type classes and type families.

Constraint-based type checkers perform their work in two phases. The communication between phases is mediated via *constraints*, which are specific to each type system, although almost all of them include some form of type equality. In the case of GHC, constraints conform to the following syntax:

$$\begin{array}{lcll} \text{Primitive constraints} & P & ::= & \tau_1 \sim \tau_2 \mid \mathsf{C}\ \tau_1\ \ldots\ \tau_n \\ \text{Simple constraints} & Q & ::= & \varepsilon \mid Q_1, Q_2 \mid P \\ \text{(Extended) constraints} & C & ::= & Q \mid C_1, C_2 \mid \exists \overline{\alpha}.(Q \supset C) \end{array}$$

Primitive constraints include type class instance constraints, such as $\mathit{Show}\ [\mathit{Int}]$, in addition to equalities. Implication constraints are needed to model pattern matching over generalized algebraic data types (GADTs), where different branches refine the information we have about types. These are called extended constraints in OutsideIn(X); for conciseness we shall drop the adjective. At this point, the addition of the level of simple constraints, including only conjunction, in between primitive and normal ones is not standard; the reason for introducing it shall become apparent later.

The first phase in the type checking process, usually called *generation* or *gathering*, traverses the Abstract Syntax Tree of the program and produces a set of constraints which the types of the expressions need to satisfy. In some cases, this traversal also involves the propagation of some information, like pushing type signatures towards the leaves of the tree. In the case of OutsideIn(X), generation is represented by a judgment $\Gamma \vdash e : \tau \rightsquigarrow C$ which states that under environment Γ the expression e is assigned a type τ given that the constraint set C is satisfied. We refrain from describing the entire judgment, the fragment given in Figure 2 is enough for our purposes. The syntax $[\overline{a \mapsto \alpha}]$ represents the substitution of the rigid variables in the type signature of x by fresh type variables.

After constraints have been gathered we need to *solve* them. From a high-level point of view solving is described by a judgment $\mathcal{Q} \vdash C_w \rightsquigarrow Q_r; \theta$ stating that under a set of axioms $\mathcal{Q}$ the wanted constraints C_w can be simplified to a set of simple constraints Q_r and a type assignment θ. In this case, axioms refer to global information like the declared instances for every type class. For example, if we take $\mathcal{Q}$ to contain the following declarations:

$$\textbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a \qquad \textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]$$

Then we have that $\mathcal{Q} \vdash \mathit{Ord}\ a, \mathit{Eq}\ [a] \rightsquigarrow \mathit{Ord}\ a$, since $\mathit{Eq}\ [a]$ is simplified to $\mathit{Eq}\ a$, and we know that this second constraint holds because we have $\mathit{Ord}\ a$.

Under the hood, the solver consists of a driver which takes care of implications and a rewriting procedure working over simple constraints. In a simplified form, rewriting is expressed as a judgment $\mathcal{Q} \vdash \langle \varphi; Q_a; Q_w \rangle \hookrightarrow \langle \varphi'; Q'_a; Q'_w \rangle$. The set of axioms does not change throughout the process, but the ongoing substitution φ, the assumed constraints Q_a – coming from the implications gathered from pattern matching over a GADT – and the wanted constraints Q_w do. Each rewrite step applies a single simplification to the constraint sets, like splitting a type equality headed by a constructor into type equalities on its arguments:

$$\begin{array}{lcl} \mathcal{Q} & \vdash & \langle \varphi\ ;\ (\mathsf{T}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{T}\ \rho_1\ \ldots\ \rho_n), Q_a\ ;\ Q_w \rangle \\ & \hookrightarrow & \langle \varphi\ ;\ \tau_1 \sim \rho_1, \ldots, \tau_n \sim \rho_n, Q_a\ ;\ Q_w \rangle \\ \mathcal{Q} & \vdash & \langle \varphi\ ;\ Q_a\ ;\ (\mathsf{T}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{T}\ \rho_1\ \ldots\ \rho_n), Q_w \rangle \\ & \hookrightarrow & \langle \varphi\ ;\ Q_a\ ;\ \tau_1 \sim \rho_1, \ldots, \tau_n \sim \rho_n, Q_w \rangle \end{array}$$

At each of these steps the solver might detect an inconsistency, such as $\mathit{Int} \sim \mathit{Bool}$. In that case, a special constraint $\bot$ is generated.

This description of solving allows us to formally define what inconsistency and left-undischarged errors are:

```
atop :: CustomErrors [
    [d1 :∞: QDiagram b1 v1 n1 m1 :⇛: Text "Arg. #1 to 'atop' must be a diagram",
     d2 :∞: QDiagram b2 v2 n2 m2 :⇛: Text "Arg. #2 to 'atop' must be a diagram"],
    [DoNotCoincide "Back-ends" b1 b2, DoNotCoincide "Vector spaces" v1 v2,
     DoNotCoincide "Numerical fields" n1 n2, DoNotCoincide "Query annotations" m1 m2],
    [Check (OrderedField n1), Check (Metric v1), Check (Semigroup m1)]
  ] ⇒ d1 → d2 → d1
atop = Diagrams.Combinators.atop
```

Figure 1. Type signature for *atop* with custom error diagnosis

$$\frac{\overline{\alpha}\ \text{fresh} \qquad (x : \forall \overline{a}.Q \Rightarrow \tau) \in \Gamma}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow [\overline{a \mapsto \alpha}]Q}$$

$$\frac{\alpha\ \text{fresh} \qquad \Gamma, (x : \alpha) \vdash e : \tau \rightsquigarrow C}{\Gamma \vdash (\lambda x \to e) : \alpha \to \tau \rightsquigarrow C}$$

$$\frac{\Gamma \vdash f : \tau_1 \rightsquigarrow C_1 \qquad \Gamma \vdash e : \tau_2 \rightsquigarrow C_2 \qquad \alpha\ \text{fresh}}{\Gamma \vdash f\ e : \alpha \rightsquigarrow C_1, C_2, \tau_1 \sim (\tau_2 \to \alpha)}$$

Figure 2. Constraint generation for a subset of Haskell

- Inconsistencies occur when a constraint has been rewritten to $\bot$. Thus, what is considered an inconsistency depends on the type system at hand. In a real implementation, each of these $\bot$ constraints keeps track of the constraints it came from, in order to report the corresponding error message.
- Left-undischarged errors are reported whenever the set of residual constraints was expected to be empty – like when checking a function definition against its type signature – but this was not the case.

In most cases, in the presence of inconsistencies left-undischarged constraints are not reported.

2.1 The *Constraint* kind

Starting with version 7.4, GHC considers constraints as types of the special kind *Constraint* (Bolingbroke 2011). This implies that all the constructs available for type-level programming can be used also to manipulate constraints. The simplest example is giving an alias to a set of constraints. If we have the following declaration:

```
type JSONSerializable a = (FromJSON a, ToJSON a)
```

we can write *JSONSerializable Person* instead of both *FromJSON Person* and *ToJSON Person*. The kind of *JSONSerializable* is inferred by the compiler to be $* \to Constraint$, where $*$ is the kind of ground types.

More interesting is the combination of the *Constraint* kind and type families, a form of type-level functions in GHC. The following example applies a type class to all the elements of a type-level list:

```
type family All (c :: k → Constraint) (xs :: [k]) where
  All c []       = ()
  All c (x : xs) = (c x, All c xs)
```

This way, *All Show* [*Int*, *Bool*] is equivalent to (*Show Int*, *Show Bool*). For the previously introduced *TypeError*, we can use type families to abstract common patterns in those cases. For example, it is very common that a type class is not implementable for functions. We can write a synonym for that error:

```
type NoFunctionsAllowed c
  = TypeError (Text "Cannot " :◇: ShowType c :◇: " functions")
```

which we can later use in the definition of the instance for functions:

```
instance NoFunctionsAllowed Show ⇒ Show (a → b)
```

In that way, we ensure that error messages are all built from the same textual template, instead of each specific instance accidentally having a slightly different wording. This facility opens the way to error customization which can be reused and manipulated: by ensuring that our customizations are expressed as *Constraint*s.

3 Hinting at Solutions

In the first technique we present we annotate constraints with messages which ought to be reported when the constraint leads to an inconsistency or is left undischarged. Using this mechanism, the type signature of the *atop* function from `diagrams` is annotated as shown in Figure 3. We remark that this type signature is plain Haskell (with some GHC extensions enabled). The *InconsistentHint* annotations are predicates which influence the way type error messages are reported.

If we now try to call the function with values of the wrong type, the custom text is attached as a hint to the standard error:

```
example = atop True 'c'
```

```
 * Couldn't match type 'QDiagram b v n m' with type 'Bool'
 * In the expression: atop True 'c'
 * Hint: argument #1 to 'atop' must be a diagram
```

The second argument to *InconsistentHint* must be of kind *ErrorMessage*, the same kind as that of *TypeError*. Therefore, our hint is able to show the types involved in the inconsistency. We could refine the third hint to include the names of the offending back-ends:

```
(b1 ~ b2) `InconsistentHint`
  (Text "the diagrams use different back-ends "
   :◇: ShowType b1 :◇: Text " and " :◇: ShowType b2)
```

We stress that hints are attached to a specific constraint in a specific type signature, achieving the *context-dependency* we were aiming for. These hints do not need to appear in the original DSL, we can create a wrapper library after the fact. For example, we can make *atop* point to the original `diagrams` version:

```
import qualified Diagrams.Combinators

atop :: (...) ⇒ d1 → d2 → d1   -- Previous annotated signature
atop = Diagrams.Combinators.atop
```

Furthermore, by simply type checking this definition, the compiler ensures that the annotated type signature is compatible with the

$$
\begin{aligned}
\mathit{atop} :: ((d_1 \sim \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1)\ `\mathit{InconsistentHint}`\ (\mathit{Text}\ \texttt{"argument \#1 to 'atop' must be a diagram"}),\\
(d_2 \sim \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2)\ `\mathit{InconsistentHint}`\ (\mathit{Text}\ \texttt{"argument \#2 to 'atop' must be a diagram"}),\\
(b_1 \sim b_2)\ \ `\mathit{InconsistentHint}`\ (\mathit{Text}\ \texttt{"the diagrams must use the same back-end"}),\\
(v_1 \sim v_2)\ \ `\mathit{InconsistentHint}`\ (\mathit{Text}\ \texttt{"the diagrams must live in the same vector space"}),\\
(n_1 \sim n_2)\ \ `\mathit{InconsistentHint}`\ (\mathit{Text}\ \texttt{"the diagrams must use the same numeric field"}),\\
(m_1 \sim m_2)\ `\mathit{InconsistentHint}`\ (\mathit{Text}\ \texttt{"the diagrams use different annotations"}),\\
\mathit{OrderedField}\ n_1, \mathit{Metric}\ v_1, \mathit{Semigroup}\ m_1) \Rightarrow d_1 \to d_2 \to d_1
\end{aligned}
$$

Figure 3. Annotated type signature for *atop* with hints

original type. Checking this soundness property requires additional work in other error customization proposals (Heeren et al. 2003a; Serrano and Hage 2017), while in our case it follows from the rewriting rules of hint combinators.

For left-undischarged errors, we provide *LeftUndischargedHint*, allowing us to point to the derivation facilities in `persistent`, as shown in Figure 4. Now, if the type of the value we try to insert in the database is not declared as an entity in Persistent terms, the user obtains a suggestion on how to fix the problem. Using this same idea, we can also improve the user experience for type classes which can be automatically derived, such as *Eq* and *Ord*.

3.1 Implementation

The implementation of hints inside GHC is divided into two parts: the user-facing library part and some modifications to the type checker itself. The former part consists of definitions of new constraint combinators:

class $c \Rightarrow \mathit{InconsistentHint}\ c\ (\mathit{msg} :: \mathit{ErrorMessage})$
instance $c \Rightarrow \mathit{InconsistentHint}\ c\ \mathit{msg}$

class $c \Rightarrow \mathit{LeftUndischargedHint}\ c\ (\mathit{msg} :: \mathit{ErrorMessage})$
instance $c \Rightarrow \mathit{LeftUndischargedHint}\ c\ \mathit{msg}$

These instances imply that to solve either c `` `InconsistentHint` `` msg or c `` `LeftUndischargedHint` `` msg the compiler must solve c itself. Note that this definition, parametric in the constraint c, would not have been possible before the introduction of the *Constraint* kind.[2]

Using type classes to recall the hints poses in principle a performance problem, since dictionaries have to be created and passed around. We can solve this problem by asking the compiler to *inline* the definition of the hint-annotated version.

```
{-# INLINE atop #-}
```

Now, after type checking, each call to *atop* is replaced by its body, *Diagrams.Combinators.atop*, making the use of the annotated version essentially for free from a performance perspective.

To make sure our annotations are taken into account, we need to change the solving process inside the compiler to retain information about our hints. In the OutsideIn(X) formalism, we modify the syntax of constraints. Now simple constraints are not merely conjunctions of primitive constraints, but each of these primitive constraints is optionally annotated with hints. The grammar of hints is just an embedding of GHC's *ErrorMessage* kind.

[2] Actually, in order to compile this code we make use of yet another language extension introduced in GHC 8, namely *UndecidableSuperClasses*.

$$
\begin{array}{lrcl}
\text{Simple constraints} & Q & ::= & \varepsilon \mid Q_1, Q_2 \mid P \mid Q^{h_\perp / h_\mathcal{U}} \\
\text{Hints} & h & ::= & - \quad \text{(No hint)} \\
 & & \mid & \mathit{Text}\ \texttt{"t"} \mid \mathit{ShowType}\ \tau \mid h_1 :\diamond: h_2
\end{array}
$$

Hints are updated by the solver whenever a qualifier for our specials classes *LeftUndischargedHint* and *InconsistentHint* are encountered. Errors may only appear with constraints we aim to *solve*, not from those coming from the antecedent of an implication. Thus, we only need to do special handling when the type class predicates arise in a wanted position. In terms of our rewriting judgment:

$$
\begin{array}{lcl}
Q & \vdash & \langle \varphi\ ;\ Q_a\ ; (Q\ `\mathit{InconsistentHint}`\ h_\perp)^{h_\perp / h_\mathcal{U}}, Q_w \rangle \\
 & \hookrightarrow & \langle \varphi ; Q_a ; P^{h'_\perp / h_\mathcal{U}}, Q_w \rangle \\
Q & \vdash & \langle \varphi\ ;\ Q_a\ ; (Q\ `\mathit{LeftUndischargedHint}`\ h_\mathcal{U})^{h_\perp / h_\mathcal{U}}, Q_w \rangle \\
 & \hookrightarrow & \langle \varphi ; Q_a ; P^{h_\perp / h'_\mathcal{U}}, Q_w \rangle
\end{array}
$$

Every rewriting rule in the solver must be updated to propagate hints, otherwise hints attached to constraints that only indirectly turn into an inconsistency will not be reported. As an example, this is the new rule for splitting type equalities:

$$
\begin{array}{lcl}
Q & \vdash & \langle \varphi ; Q_a ; (\mathsf{T}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{T}\ \rho_1\ \ldots\ \rho_n)^{h_\perp / h_\mathcal{U}}, Q_w \rangle \\
 & \hookrightarrow & \langle \varphi ; Q_a ; (\tau_1 \sim \rho_1)^{h_\perp / h_\mathcal{U}}, \ldots, (\tau_n \sim \rho_n)^{h_\perp / h_\mathcal{U}}, Q_w \rangle
\end{array}
$$

In general, this propagation strategy becomes problematic when more than one constraint is involved in a rewriting step (Serrano and Hage 2016a), since the solver may have to merge different hints.

In the GHC type checker there is the notion of *work item*, which is the constraint considered at each point in time. A work item goes through different phases, including canonicalization and reaction with constraints in the so-called inert set. We made the design decision of propagating always the hints from the work item from any new constraints derived from an interaction with other constraints. This change is also the least invasive for the GHC code base, and works well in practice.

In the PhD thesis of Wazny (2006) we find a proposal similar to ours for attaching custom errors to constraints. The main difference is that we use forward propagation: hints are updated as the solving process progresses, whereas in (Wazny 2006) annotations are obtained by backwards reasoning from the offending inconsistency. This difference is mainly due to the way in which each system decides which errors to report: in Chameleon (Stuckey et al. 2006) after an inconsistency is found the system checks for minimal unsatisfiable subsets of constraints to be blamed. In contrast, GHC does not perform such a step, but rather keeps information while solving in order to produce a message. A similar conclusion can be drawn for Scala's *@implicitNotFound* (Scala Team Scala Team).

```
insertUnique :: (MonadIO m, PersistUniqueWrite backend,
                 PersistEntity record `LeftUndischargedHint` (
                   Text "Data type '" :◇: ShowType record :◇: Text "' is not declared as entity."
                   :$$: Text "Hint: entity definition can be automatically derived."
                   :$$: Text "Check the documentation at http://www.yesodweb.com/...")
              ⇒ record → ReaderT backend m (Maybe (Key record))
```

Figure 4. Annotated type signature for *insertUnique*

3.2 Reuse and Abstraction

In § 2.1, we showed how by means of a type synonym of kind *Constraint* we can reuse messages in *TypeError*. The same technique can be applied to our hints. One common use case is exporting a synonym which comes with the default annotation, instead of the bare type class. For example, the authors of `persistent` could re-define *PersistEntity* as follows:

```
type PersistEntity r = PersistEntity' r `LeftUndischargedHint`
  (Text "Data type '" :◇: ShowType r :◇: ...)
```

where we assume the old *PersistEntity* from the Persistent library has been renamed to *PersistEntity′*. In that way, every time an expression mentions the *PersistEntity* type class in a type signature, the hint comes attached to it and will be shown to the DSL user if the constraint is left undischarged.

We can go one step further by *abstracting* over common patterns in DSL type error diagnosis. As usual in functional programming, this abstraction is described by a function; type families are the tool for type-level functions in Haskell.

Our annotated type signature for *atop* at the beginning of the section has some clear duplication. First, for both arguments we check that they have the correct head constructor, *QDiagram*. Then, for every argument to *QDiagram*, we check for equality between that component, with a similar *InconsistentHint* for each case. We can remove the apparent duplication by introducing some combinators, resulting in the cleaner signature given in Figure 5.

The definitions of *ShapeHints* and *ArgsHints* are given in Figure 6. In both cases we traverse a list of tuples containing the elements which should be equal and information needed to produce the hint if they are not. In the case of *ShapeHints* we use a counter which is incremented at every step to be able to inform which argument did not have the right shape. For the hints over arguments, we also require a name for the whole structure; such type-level string literals have the kind *Symbol* in GHC.

4 Controlling the Solving Order

The technique of adding hints to constraints as described in § 3 is simple and straightforward to implement. Unfortunately, it does not act in a predictable way due to the unspecified order of constraint solving. In this section we describe the problems and a new constraint combinator, *IfNot*, which solves these problems.

Let us look at a concrete example in which we obtain a surprising hint for our *atop* example. Say we want to combine two diagrams d_1::*QDiagram SVG V2 Double* () and d_2::*QDiagram SVG V3 Double* (). The difference between the types lies in the vector space component, so we expect the hint to refer to this fact. The first step in type checking *atop* d_1 d_2 is gathering the necessary constraints. After some simplifications, the constraint set contains the equalities given below; we identify each one by the number on top of the $\sim$ symbol. Furthermore, each constraint is annotated with an inconsistency hint coming from the *atop* signature, which we refrain from showing here for the sake of conciseness.

$$\textit{QDiagram SVG V2 Double}\ () \overset{1}{\sim} \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1,$$
$$\textit{QDiagram SVG V3 Double}\ () \overset{2}{\sim} \textit{QDiagram}\ b_2\ v_2\ n_2\ m_2,$$
$$b_1 \overset{3}{\sim} b_2, v_1 \overset{4}{\sim} v_2, n_1 \overset{5}{\sim} n_2, m_1 \overset{6}{\sim} m_2$$

Since the constraint solver is allowed to consider equalities in any order, it might well perform the four last substitutions first. We are then left with the following two constraints:

$$\textit{QDiagram SVG V2 Double}\ () \overset{1}{\sim} \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1,$$
$$\textit{QDiagram SVG V3 Double}\ () \overset{2}{\sim} \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1$$

From the first constraint the solver now discovers that $v_1 \sim V2$, and applies this fact in the second constraint:

$$\textit{QDiagram SVG V3 Double}\ () \overset{2}{\sim} \textit{QDiagram}\ b_1\ V2\ n_1\ m_1$$

The problem to report is that *V2* does not match *V3*. But the hint associated with the second constraint is `argument #2 to 'atop' must be a diagram`. Reporting that hint as part of the error message is completely misleading for the user: the second argument is in fact a diagram!

Our solution is to give some control to the DSL author over the order in which constraints are considered. If we ensure that the $d \sim \textit{QDiagram}\ b\ v\ n\ m$ constraints are solved before the others, the misleading hint will never arise. Techniques for custom ordering of constraints are available in the Helium Haskell compiler (Hage and Heeren 2009; Heeren et al. 2003a). The challenge is to integrate custom ordering with the GHC solver, without losing the facilities for reuse and abstraction dicussed in § 3.

Our solution is to introduce a new combinator:

```
IfNot (c :: Constraint) (fail :: Constraint) (ok :: Constraint)
```

For each constraint *c* wrapped in this combinator, we specify both how to continue when the constraint is inconsistent – the *fail* branch – and how to continue when that is not known to be the case – the *ok* branch. This is an important point: we also take the *ok* branch if at that point we do not know yet whether the constraint is consistent or not – for example, an equality $\alpha \sim \beta$ may turn out to be inconsistent or not depending on the values we later find for both type variables. By nesting applications of *IfNot* we dictate the order in which constraints are checked. For example, Figure 7 gives part of the type signature of *atop* where constraints are solved in sequentially from top to bottom. As we shall see later, the semantics of *IfNot* ensures that $b_1 \sim b_2$ is only considered by the solver if both $d_1 \sim \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1$ and $d_2 \sim \textit{QDiagram}\ b_2\ v_2\ n_2\ m_2$

$$
\begin{aligned}
&\mathit{atop} :: (\mathit{ShapeHints}\ [(d_1, \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1, \texttt{"diagram"}), (d_2, \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2, \texttt{"diagram"})],\\
&\quad \mathit{ArgsHints}\ \texttt{"diagrams"}\ [(b_1, b_2, \texttt{"back-ends"}), \quad (v_1, v_2, \texttt{"vector spaces"})\\
&\qquad\qquad\qquad , (n_1, n_2, \texttt{"numeric fields"}), (m_1, m_2, \texttt{"annotations"})]\\
&\quad \mathit{OrderedField}\ n_1, \mathit{Metric}\ v_1, \mathit{Semigroup}\ m_1) \Rightarrow d_1 \to d_2 \to d_1
\end{aligned}
$$

Figure 5. Annotated type signature for *atop* with hints, second version

$$
\begin{aligned}
&\textbf{type family}\ \mathit{ShapeHints}\ (xs :: [(*, *, \mathit{Symbol})]) :: \mathit{Constraint}\ \textbf{where}\\
&\quad \mathit{ShapeHints}\ xs = \mathit{SH'}\ 1\ xs\\
&\textbf{type family}\ \mathit{SH'}\ (n :: \mathit{Nat})\ (xs :: [(*, *, \mathit{Symbol})]) :: \mathit{Constraint}\ \textbf{where}\\
&\quad \mathit{SH'}\ n\ [\,] \qquad\qquad = ()\\
&\quad \mathit{SH'}\ n\ ((x1, x2, e) : xs) = (\mathit{SH'}\ (n+1)\ xs, x1 \sim x2\ \text{`}\mathit{InconsistentHint}\text{`}\ (\mathit{Text}\ \texttt{"arg \#"} :\diamond: \mathit{ShowType}\ n :\diamond: \mathit{Text}\ \texttt{" must be a "} :\diamond: \mathit{Text}\ e))\\
&\textbf{type family}\ \mathit{ArgsHints}\ (n :: \mathit{Symbol})\ (xs :: [(*, *, \mathit{Symbol})]) :: \mathit{Constraint}\ \textbf{where}\\
&\quad \mathit{ArgsHints}\ n\ [\,] \qquad\qquad = ()\\
&\quad \mathit{ArgsHints}\ n\ ((x1, x2, e) : xs) = (\mathit{ArgsHints}\ n\ xs, x1 \sim x2\ \text{`}\mathit{InconsistentHint}\text{`}\ (\mathit{Text}\ n :\diamond: \mathit{Text}\ \texttt{" use different "} :\diamond: \mathit{Text}\ e))
\end{aligned}
$$

Figure 6. Common type error patterns using type families

are consistent with the typing. Knowing this, we can safely point to the difference in back-ends as cause for the error.

Another difference between *IfNot* and the hints from § 3 is that in the former case we can use *TypeError* to signal a problem. The fact that we can use any constraint as *fail* argument to *IfNot* gives us additional flexibility which shall be important to implement some of the type error patterns in § 5. The only restriction is that this *fail* argument must ultimately result in an inconsistency for the whole system to be sound – we discuss how to statically enforce this invariant in Appendix A.

There is one caveat. *IfNot* imposes an ordering between constraints gathered from the same type signature. A custom ordering is also needed between constraints coming from different expressions. For example, when type checking *atop* d_1 d_2, the constraints inside the *IfNot* should only come into play after all the constraints from both d_1 and d_2 have already been considered. Otherwise we run the risk that constraints from *atop* infect those from d_1 and d_2, as was the case in the first example in this section.

Techniques to describe such relations include specialized type rules (Heeren et al. 2003a; Serrano and Hage 2016b) and priorities for those systems based on Constraint Handling Rules (Koninck et al. 2007) such as Chameleon (Stuckey et al. 2006). Modifying the entire gathering process to accommodate these constructs in GHC is undesirable, since it means changing a big chunk of important code in the compiler. Fortunately, we do not really need complete control over the ordering. The most common use case is to prioritize constraints coming from the arguments in an application node over those coming from the function application itself. In our implementation, you can request this treatment with a pragma:

```
{-# CHECK_ARGS_BEFORE_FN atop #-}
```

Now, whenever *atop* d_1 d_2 is type checked, we are sure that the maximal amount of information has been obtained from the arguments before considering the application itself.

4.1 Implementation

The implementation affects both the gathering phase, which must be made aware of the `CHECK_ARGS_BEFORE_FN` pragma, and the solving phase, which has to take constraint priorities into account and implement the rewriting of *IfNot* constraints.

The modified syntax of constraints now includes optional priorities of the form $Q\ @\ n$. These priorities are generated by the new judgment given in Figure 8. The main change is the addition of a new numeric input n, which syntactically appears after the environment Γ, and which represents the *current generation priority*. This priority is used when creating new constraints, and updated when traversing an application if the head was annotated with the corresponding pragma. As part of this modification, we moved from a rule for application with only one argument to multi-application. This is consistent with how GHC treats applications.

Clearly, the constraint solver must be made to obey these priorities. This entails two different things. First, it should only perform a rewriting step involving a constraint with priority n if no rewriting step can be taken with a constraint with priority larger than n. Note that it does not mean that all constraints with priority m disappear before considering any constraint at $m-1$: high priority constraints, like $\mathit{Eq}\ [\alpha]$, might get stuck, meaning that rewriting can only proceed after additional information has been found, like an assignment to α. When a priority gets stuck it is possible for constraints with lower priority to be considered.

Furthermore, whenever two constraints of the same priority can be rewritten, the one that is not an *IfNot* should take precedence. This design choice ensures that the inconsistency check can take into consideration as much information as possible. For example, if we have $\alpha \overset{1}{\sim} \mathit{Bool}$ and $\alpha \overset{2}{\sim} \mathit{Int}$, with the same priority, but only the second with a custom message, we prefer the compiler to consider first the equality with *Bool*, so that the error is found in $\mathit{Bool} \overset{2}{\sim} \mathit{Int}$ and the custom error message reported.

Second, the new constraints generated by a rewriting step inherit the priority of their parent. Interaction between two constraints does not pose a problem: the children are assigned the smallest of the two priorities.

The last part of the implementation consists of the custom rewriting of the new constraints, given in Figure 9. In most cases, $\mathit{IfNot}\ c\ f\ o$ is equivalent to the conjunction of c and o, except for the case in

$$\begin{aligned}
&\mathit{atop} :: \mathit{IfNot}\ (d_1 \sim \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1)\ (\mathit{TypeError}\ \texttt{"Arg. \#1 to 'atop' must be a diagram"})\ (\\
&\quad \mathit{IfNot}\ (d_2 \sim \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2)\ (\mathit{TypeError}\ \texttt{"Arg. \#2 to 'atop' must be a diagram"})\ (\\
&\quad \mathit{IfNot}\ (b_1 \sim b_2)\ (\mathit{TypeError}\ \texttt{"Back-ends do not coincide"})\ (\\
&\quad \mathit{IfNot}\ (v_1 \sim v_2)\ (\mathit{TypeError}\ \texttt{"Vector spaces do not coincide"})\ (\ldots)))) \Rightarrow d_1 \to d_2 \to d_1
\end{aligned}$$

Figure 7. Annotated type signature for *atop* using *IfNot*

$$\frac{\overline{\alpha}\ \text{fresh} \qquad (x : \forall \overline{a}.Q \Rightarrow \tau) \in \Gamma}{\Gamma\ ;\ n \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow [\overline{a \mapsto \alpha}]Q\ @\ n}$$

$$\frac{\alpha\ \text{fresh} \qquad \Gamma, (x : \alpha)\ ;\ n \vdash e : \tau \rightsquigarrow C}{\Gamma\ ;\ n \vdash (\lambda x \to e) : \alpha \to \tau \rightsquigarrow C}$$

$$\frac{\Gamma\ ;\ n \vdash f : \tau_0 \rightsquigarrow C_0 \qquad \alpha\ \text{fresh} \qquad n' = \begin{cases} n+1 & \text{if } f \text{ has a } \texttt{CHECK\_ARGS\_BEFORE\_FN} \\ n & \text{otherwise} \end{cases} \qquad \Gamma\ ;\ n' \vdash e_i : \tau_i \rightsquigarrow C_i}{\begin{aligned}&\Gamma \vdash f\ e_1\ \ldots\ e_n : \alpha \\ &\quad \rightsquigarrow C_0, C_1, \ldots, C_n, \tau_0 \sim (\tau_1 \to \cdots \to \tau_n \to \alpha)\ @\ n\end{aligned}}$$

Figure 8. Constraint generation with priorities

$$\begin{aligned}
Q \quad &\vdash \quad \langle \varphi\ ;\ \mathit{IfNot}\ c\ f\ o\ @\ n, Q_a\ ;\ Q_w \rangle \\
&\hookrightarrow \quad \langle \varphi\ ;\ c\ @\ n, o\ @\ n, Q_a\ ;\ Q_w \rangle \\
Q \quad &\vdash \quad \langle \varphi\ ;\ Q_a\ ;\ \mathit{IfNot}\ c\ f\ o\ @\ n, Q_w \rangle \\
&\hookrightarrow \quad \begin{cases} \langle \varphi\ ;\ Q_a\ ;\ f\ @\ n, Q_w \rangle & \text{if } c \text{ is inconsistent} \\ \langle \varphi\ ;\ Q_a\ ;\ c\ @\ n, o\ @\ n, Q_w \rangle & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 9. Rewriting steps for *IfNot* constraints

which c is inconsistent. This rewriting is sound if the property of f ultimately resulting in an inconsistency is respected:

- If c is inconsistent and we know it at the moment in which we process the *IfNot*, we can replace it by f. The compiler will not continue further because an error is to be reported.
- It might be the case that we only found c to be inconsistent later. But in that case we still have that constraint available because of the second branch of the rewriting.

Detecting inconsistencies while solving. We have already discussed that it is not important that we detect *all* inconsistencies at the moment of rewriting an *IfNot* constraint, since the system remains sound if the inconsistency is found later in the process. But we still have the question of how we detect *any* inconsistency at all: finding this out might require an arbitrary amount of solving.

In our implementation we have decided to only handle the simplest case: type equalities which can be proven to fail even if a later substitution refine the involved types. Type equalities, as discussed in the introduction, are the main source of inconsistencies in a Haskell program. Furthermore, GHC already knows of such a notion: it is called *apartness*. For that reason, the user-facing constraint in our implementation is called:

$$\mathit{WhenApart}\ a\ b\ f\ o \quad \text{representing} \quad \mathit{IfNot}\ (a \sim b)\ f\ o$$

The notion of apartness was introduced alongside closed type families (Eisenberg et al. 2014). In contrast with open type families, whose instances must not overlap, closed type families introduce a list of patterns which should be checked in order. This in turn allows for patterns where the same variable appears more than once (non-linear patterns), as in this definition of type equality:

type family $a :==: b :: \mathit{Bool}$ **where**
$a :==: a = \mathit{True}$
$a :==: b = \mathit{False}$

If we are faced with an application of the type family where one of the arguments is a variable, like $\mathit{Int} :==: \alpha$, it is not sound to reduce the type family: we need to wait until a substitution settles the question. However, if we have an application such as $\mathit{Int} :==: [\alpha]$, we are free to rewrite to *False*. The reason is that *Int* and $[\alpha]$ can never be made to unify, or in other words, are apart.

Ensuring soundness. As we have discussed above, the rewriting of $\mathit{IfNot}\ c\ f\ o$ to f whenever c is found to be inconsistent is only sound if f forms an inconsistent set of constraint itself. Otherwise ill-typed code would be able to pass through by wrapping its type signature in an *IfNot* constraint. In order to prevent this scenario, we need to perform an extra check on the type signatures. This check is fully described in Appendix A, but unfortunately not yet implemented in GHC.

5 Type Error Patterns

The integration of hints as part of GHC's *Constraint* kind opened possibilities for reuse and abstraction in type error diagnosis. Below we discuss a number of scenario's where we can do the same, exploiting the increased power that *IfNot* provides.

5.1 Check in order

The description of *IfNot* constraints has simplicity as one of its advantages: only one new kind of constraint and a new pragma need to be considered by the type checker. However, writing a type signature with nested *IfNot* becomes cumbersome really fast. Our first step is to give DSL authors better syntax: an embedded DSL for describing type error diagnosis for embedded DSLs!

First of all, we need to describe the kind of errors we can rewrite, that is, differentiate between inconsistency and left-undischarged errors. Since, in our implementation, the only inconsistencies we can detect are due to apartness, we bake this fact directly into the data type with the $(:\!\not\sim\!:)$ constructor.

data $\mathit{ConstraintFailure} = \forall\ t.t :\!\not\sim\!: t \mid \mathit{Undischarged\ Constraint}$

This data type definition is promoted (Yorgey et al. 2012) by GHC to a kind definition. In fact, we can only use it as a kind since the *Undischarged* constructor needs a *Constraint* as argument. One remark to make at this point is that the definition of $(:\!\not\sim\!:)$ requires types of the same kind: we model homogeneous inequality.

The next step is adding the possibility to attach a custom message to these error conditions. Whether this error message gets converted to *IfNot* or *LeftUndischargedHint* will depend on the error condition itself. In addition, a constraint can be added without any message using *Check*.

As a first approximation, we could define a type signature with custom type errors by merely a list of *CustomError*s. But then once you find the first inconsistency solving completely halts, which means only one type error can be found, even if there are multiple independent ones. For example, in *atop* we can partition the constraints into three sets so that each set contains mutually independent constraints:

- The first elements of the partition checks that both arguments are of the shape *QDiagram b v n m*. Each type equality refers to completely different variables, so they have no dependency between them.
- Once we know that both arguments are diagrams, we can move on and check the equality of each pair of type arguments. These checks are again independent of each other.
- Finally, we can check whether types obey the corresponding instances. We prefer to do this at the latest moment to prevent error messages coming from under-specified types.

As a result, we move to a list of lists, each list representing one set of independent constraints. The code of *CustomErrors* is given in Figure 10. The crucial point is that when a constraint fails, we still look at the ones in the same partition but not into the next ones.

Using *CustomErrors* in *atop* results in the type signature given in the Introduction in Figure 1. As in the case of hints, we have abstracted non-coincidence of type parameters:

```
type DoNotCoincide what a b
  = a :≁: b :⇛: Text what :◇: Text " do not coincide: "
                :◇: ShowType a :◇: Text " vs. " :◇: ShowType b
```

One difference between this embedding of type errors and the previous one with hints is that an abstraction like *DoNotCoincide* gets assigned kind *CustomError* which is different from *Constraint*.

5.2 Siblings

Another common pattern in custom type error diagnosis is reporting functions which are easily confused with one another, called *siblings* in the literature (Hage and Heeren 2006; Heeren et al. 2003a). For example, a simplified version of `diagrams` may contain the following two functions to create a color, both with and without alpha:

```
rgb  :: Int → Int → Int       → Color
rgba :: Int → Int → Int → Int → Color
```

Now if the user of the DSL writes *rgb* 0 255 0 90, we want the reported error message to point to the fact that *rgb* and *rgba* are often confused. But only when the change is actually a fix. The type signature which we can assign to *rgb* to get this behavior is:

```
rgb :: IfNot     (fn ~ Int → Int → Int       → Color)   -- 1
        (IfNot (fn ~ Int → Int → Int → Int → Color)     -- 2
          (fn ~ Int → Int → Int → Color)                -- 3
          (TypeError (Text "Try to use 'rgba'")))       -- 4
        () ⇒ fn                                         -- 5
```

In this case, the whole type of the function is wrapped in an apartness check (1). If we find out that the type for *rgb* is compatible with the typing environment, there is nothing left to do. This is the reason why the second branch (5) of the top-level *IfNot* is simply (). If this is not the case, we try to see whether *fn* is compatible with the type of *rgba* instead in (2). If this is the case (4), we then produce the custom type error hinting at the replacement. If we can assign neither type to *fn*, then we prefer to get the original error message, as declared in (3); there is no mention of the use of siblings if the suggestion does not fix the program.

There is one small caveat with this solution, though. Checking whether the type of *fn* is compatible with the one of *rgba* should be done at the very last possible moment. Otherwise, we run the risk of unifying *fn* with *Int* → *Int* → *Int* → *Int* → *Color*, only to find later that this choice was not right. But at this later point in time we have already lost the error message pertaining to *rgba*. If we have more than one sibling, the order in which we check them does not matter, given that the check is done at the end.

We already have discussed a mechanism in the compiler to indicate when a constraint should be considered: priorities. Thus, we just need to introduce a way to declare that a given constraint should be considered as late as possible. In our implementation, such a declaration is done via another type class:

```
class    c ⇒ ScheduleAtTheEnd (c :: Constraint)
instance c ⇒ ScheduleAtTheEnd c
```

with the special rule that *ScheduleAtTheEnd c* is rewritten to *c*, as the instance declaration indicates, but this later constraint is assigned the lowest priority in the system. In our case we have settled on -1, since gathering may only produce constraints with non-negative priorities. By wrapping our checks for *fn*, we ensure that this choice does not influence further solving, as desired.

The final implementation of the *Sibling* constraint is given in Figure 11. Apart from scheduling the check of the replacement type at the end, the definition also includes an *extra* argument to allow additional constraints to be checked whenever the original type is accepted – the part marked as (5) in the *rgb* type signature.

Our focus in this paper is type error diagnosis for embedded DSLs, but siblings are also useful for general Haskell programming. For example, it is quite common for beginners to forget the difference between these two *Applicative* functions:[3]

```
(⟨$⟩) :: Applicative f ⇒ (a → b) → f a → f b
(⟨$)  :: Applicative f ⇒ a       → f b → f a
```

The solver can be informed now of this fact by means of *Sibling*:

```
(⟨$⟩) :: Sibling "(<$>)" (Applicative f) ((a → b) → f a → f b)
                 "(<$)"                  (a       → f b → f a)
                 fn ⇒ fn
```

Given $f :: Char \to Bool \to Int$, if the user tries $f\ \langle\$\rangle\ [1 :: Int]\ \langle*\rangle$ "a" $\langle*\rangle\ [True]$, the resulting error message reads:

```
* Type error in '(<$>)', do you mean '(<$)'
* In the first argument of '(<*>)',
namely 'f <$> [1 :: Int]' ...
```

5.3 Alternatives and conversions

The ideas underlying the *Sibling* construction can also be used to handle another common source of errors in embedded DSLs. In many of them basic types are wrapped in a custom type in order to differentiate usages and increase type-safety. For example, in

[3] On the other hand, we could also say that (⟨$⟩) and (⟨$) are part of an embedded DSL for "programming with applicative functors".

```
data CustomErrorsPath = Ok | Fail
type family CustomErrors (css :: [[CustomError]]) :: Constraint where
  CustomErrors [ ]        = ()
  CustomErrors (cs : css) = CustomErrors' Ok cs css
type family CustomErrors' p cs r :: Constraint where
  CustomErrors' Ok   [ ]                          r = CustomErrors r
  CustomErrors' Fail [ ]                          r = ()
  CustomErrors' p ((a :≁: b            :⇛: msg) : cs) r = IfNot (a ~ b) (TypeError msg, CustomErrors' Fail cs [ ]) (CustomErrors' p cs r)
  CustomErrors' p ((Undischarged c :⇛: msg) : cs) r = (c `LeftUndischargedHint` msg, CustomErrors' p cs r)
  CustomErrors' p (Check c                 : cs) r = (c,                              CustomErrors' p cs r)
```

Figure 10. Initial implementation of *CustomErrors*

```
type Sibling nameOk extra tyOk nameWrong tyWrong fn
  = IfNot (fn ~ tyOk)
      (ScheduleAtTheEnd (IfNot (fn ~ tyWrong)
        (fn ~ tyOk)
        (TypeError (Text "Type error in '" :◇: Text nameOk :◇: Text "', do you mean '" :◇: Text nameWrong :◇: Text "'")))
      extra
```

Figure 11. Implementation of *Sibling*

`persistent` primary keys are represented by a type *Key e* for a given entity type *e*. Users of the DSL, however, might be tempted to use basic types instead; it can be very helpful then to point them to the right function to perform the conversion.

The `diagrams` library also follows this pattern: points in a two-dimensional space, represented by the type *P2 a*, are distinguished from vectors, represented by *V2 a*. Operations are only defined for those concepts where it makes sense. For example, we can only obtain the perpendicular of a vector, not of a point:

```
perp :: Num a ⇒ V2 a → V2 a
```

It is possible that a user tries to compile *perp* (1, 2), since tuple notation is the usual one for vectors in mathematics. The solution in this case is to first call *r2*, which turns the pair into *V2 a*. Let us introduce a new combinator to describe alternatives in the signature of *perp* and suggest the conversion in the right case, as given in Figure 12, including a fallback message. The error message for the expression *perp* (1, 2) is now more useful:

```
* Expecting a 2D vector but got a tuple.
Use 'r2' to turn the tuple into a vector.
```

This does not automatically mean that the stated conversion is the one intended by the user – another fix in this case is writing *perp* (*V2* 1 2). But, as in the case of siblings, the message is only reported if introducing the conversion leads to a well-typed program.

In order to describe this new combinator we need to refine the promoted *CustomError* data type:

```
data CustomError = ConstraintFailure
                 | (:⇛?:) ([CustomErrorAlternative], ErrorMessage)
                 | Check Constraint
data CustomErrorAlternative = Constraint :⇛!: ErrorMessage
```

Now a failed constraint does not immediately turn into an error message, but first checks a list of alternatives, each with a different message associated to it. These alternatives turn into nested appplications of *IfNot* via the modified *CustomErrors* type family given in Figure 13. The names :⇛?: and :⇛!: are chosen to suggest that whereas in the first case you are still unsure of the type, when you produce the message in the alternatives you know more information about it. The original :⇛: combinator is still available, but now as a synonym for an application of :⇛?: without alternatives:

```
type cs :⇛: msg = cs :⇛?: ([ ], msg)
```

There is some subtlety related to how these alternatives work that must be taken into consideration by DSL authors. These alternatives should be regarded as different ways to fix the program which are tried in order. But as a result of taking one path, some type variables might be unified. In that respect, they are close relatives of siblings as presented in the previous section.

In our example with *perp'*, if we take the fail branch of *IfNot* (*v* ~ *V2 a*) it means that we have enough information to distinguish the top-level constructor of *v*, *but nothing else.* If now the first alternative is *v* ~ (*Int*, *Int*) and all we knew about *v* is that it is of the form (*a*, *b*), taking this path leads to unifying both *a* and *b* with *Int*. In some cases this might not be the intended behaviour; in order to ensure that alternatives do not influence other parts of the program they must be of the form $a \sim T\ b_1 \ldots b_n$:⇛!: *msg*, where b_1 to b_n are fresh variables.

The examples in this section give convincing reasons that the set of basic combinators in this paper are indeed a good choice. Many common patterns in customizable type error diagnosis are expressible using this language, even some which needed special support inside the compiler in other approaches to the problem.

```
perp :: CustomErrors [
  [ v :∝: V2 a :⇛?: ([ v ~ (a, a) :⇛!: Text "Expecting a 2D vector but got a tuple. Use 'r2' to turn a tuple into a vector."]
                    , Text "Expected a 2D vector, but got " :⋄: ShowType v)],
  [ Check (Num a)]] ⇒ v → v
```

Figure 12. Annotated type signature for *perp*

```
type family CustomErrors' p cs r :: Constraint where
  ...
  CustomErrors' p ((a :∝: b :⇛?: (alts, msg)) : cs) r
    = IfNot (a ~ b) (ScheduleAtTheEnd (CustomErrorsAlt alts msg (CustomErrors' Fail cs [ ]))) (CustomErrors' p cs r)
  ...
type family CustomErrorsAlt alts fail rest :: Constraint where
  CustomErrorsAlt [ ] fail rest = (TypeError fail, rest)
  CustomErrorsAlt (((a ~ b) :⇛!: msg) : cs) fail rest = IfNot (a ~ b) (CustomErrorsAlt cs fail rest) (TypeError msg, rest)
```

Figure 13. Extension of *CustomErrors* to support alternatives

6 Related Work

There is plenty of work related to improvements for general type error diagnosis. Approaches include graph representations (Hage and Heeren 2006; Zhang et al. 2015), counter-factual typing (Chen and Erwig 2014), type error slicing (Rahli et al. 2015; Stuckey et al. 2006), and interactive type debuggers (Chitil 2001; Stuckey et al. 2003; Tsushima and Asai 2013). These ideas can be combined with type error diagnosis for embedded DSLs. In the realm of customizable type error diagnosis we find several approaches:

Post-processing. The support for custom error messages in Idris (Christiansen 2014), a dependently typed language with a syntax similar to Haskell's, is based on reflection over error messages reified as a normal Idris data type. If desired, the default error message can be replaced by a custom message.

An instrumented version of the Scala compiler geared towards type feedback is described in (Plociniczak et al. 2014). Starting with low-level type checker events, a derivation tree is built on demand. Custom compiler plug-ins are able to inspect this tree and generate domain specific error messages.

The main disadvantage of the post-processing approach is that there is no influence on the actual type inference and checking process. In other words, the compiler still emits the same error messages, which we just "dress up" with domain terms. On the other hand, this leads to a simpler model for error customization, whereas modifying the solving process sometimes leads to surprising results.

Interacting with the type checker. Whereas post-processing takes place in a phase different from type checking itself, other techniques integrate with this part of the compiler, modifying its behavior. We have already mentioned GHC's *TypeError* (Diatchki 2015) and Scala's *@implicitNotFound* (Scala Team Scala Team) as inspiration for the work presented in this paper.

The Helium Haskell compiler (Heeren et al. 2003b) introduces the notion of *specialized type rules* (Heeren et al. 2003a) as a way to influence the constraints generated by an expression. Constraints might be annotated with custom error messages and/or prioritized during solving (Hage and Heeren 2009). Later, specialized type rules have been extended (Serrano and Hage 2016b) to consider not only purely syntactical information but also some type information.

Our work is heavily influenced by Helium, but with two main differences. The first is lack of support for matching on different syntactical forms: custom error messages can only be attached to function signatures and type class predicates, and priorities can only be modified between functions and arguments, and with the special *ScheduleAtTheEnd*. The resulting system is therefore less powerful, but still covers many common cases.

The second difference is our deep integration with GHC's type system, in particular with the *Constraint* kind. The specialized type rules of Helium are not part of the Haskell language. In other words, Helium is an external DSL for describing type error diagnosis, while our solution is itself an embedded DSL. As a result, we have all the facitilities of GHC's type-level programming available to us to support abstraction and reuse.

Our work does not consider ways to influence the search procedure of the solver, only the order. But DSLs may have additional invariants, e.g., within a given domain it may be impossible for a type to be an instance of two type classes A and B at the same time. The constraint solver can employ such information and mark the constraint set $A\ t, B\ t$ as inconsistent, in addition to the default inconsistency checking. Known techniques in this area include type class directives (Heeren and Hage 2005), the description of Haskell's type checking as CHR solving (Stuckey et al. 2006) and instance chains (Morris and Jones 2010). These mechanisms are orthogonal to our work, although interaction between them may give rise to more powerful customization. It should be noted that several of these mechanisms can already be encoded using recent GHC extensions (Kiselyov et al. 2004; Serrano et al. 2015). Recent versions of GHC also support type checker plug-ins (Gundry 2015), which have been used to accommodate domain specific solving rules such as those for units of measure.

Language modifications. Another approach to customization is changing the underlying language itself. The programming environment offered by Racket is well-known for its focus on students (Marceau et al. 2011), offering increasingly complex language levels.

7 Conclusion and Future Work

We presented a set of combinators to describe customizations to the type checking process, including both domain-specific error messages and priorities among constraints. These combinators are integrated in GHC's constraint system, which can be programmed, leading to abstraction facilities which are new in this area, as far as we know. In addition, libraries do not need custom type error diagnosis built-in from the outset. Rather, error diagnosis can be added gradually at a later stage by wrapping the desired functions. This provides DSL authors with a good upgrade path to provide better diagnosis for libraries which are already in use.

Although the focus of the paper is GHC, a specific dialect of the Haskell language, this research shows that constraint reflection, that is, being able to manipulate typing constraints within the language itself, provides a fertile ground for better type error diagnosis. We only need a few combinators to attach custom hints and messages, but we make heavy use of the *Constraint* kind and type families to get reuse and abstraction. This should help compiler writers to choose which compiler facilities to support. In particular, it is interesting to consider how much of our work can be translated to other statically-typed languages, especially those already based on constraints (Pottier and Rémy 2005; Swift Team 2016).

There are two main directions of future work. The first one is the interaction of general abstractions, such as *Functor*, *Applicative* or *Monad*, with the use of a specific instance in a piece of code. For example, while writing a parser, programmers often use the *Applicative* operators, but it would be helpful to phrase the error messages using terms related to the domain of parsing. In the current implementation, error messages are attached to type signatures, which live in the general type class.

The second direction is defining new combinators which do not replace default messages with custom ones, but rather serve as an explanation of the solving process. The *Eq* type class in Haskell base libraries provides an example. If $Eq\ [Person]$ is left undischarged, the message reports that $[Person]$ might be given an instance automatically. However, this is not the best error, since there is already an instance $Eq\ a \Rightarrow Eq\ [a]$. Instead, we prefer to explain that *Eq Person* is needed because of the instance, and that we can give an instance for *Person* if the corresponding **deriving** is added to the data type declaration.

References

Max Bolingbroke. 2011. Constraint Kinds for GHC. (2011). http://blog.omega-prime.co.uk/?p=127 Blog post.

Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, 583–594.

Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors *(ICFP '01)*. ACM, 193–204.

David Raymond Christiansen. 2014. Reflect on Your Mistakes! Lightweight Domain-Specific Error Messages. *Presented at TFP 2014* (2014).

Iavor Diatchki. 2015. Custom Type Errors. (2015). Available at https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations *(POPL '14)*. ACM, 671–683.

Adam Gundry. 2015. A Typechecker Plugin for Units of Measure: Domain-specific Constraint Solving in GHC Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell (Haskell 2015)*. ACM, 11–22.

Jurriaan Hage. 2014. *DOMain Specific Type Error Diagnosis (DOMSTED)*. Technical Report UU-CS-2014-019. Department of Information and Computing Sciences, Utrecht University.

Jurriaan Hage and Bastiaan Heeren. 2006. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages, 18th International Symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers (Lecture Notes in Computer Science)*, Zoltán Horváth, Viktória Zsók, and Andrew Butterfield (Eds.), Vol. 4449. 199–216.

Jurriaan Hage and Bastiaan Heeren. 2009. Strategies for Solving Constraints in Type and Effect Systems. *Electron. Notes Theor. Comput. Sci.* 236 (April 2009), 163–183.

Bastiaan Heeren and Jurriaan Hage. 2005. Type Class Directives. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages (PADL'05)*. 253–267.

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003a. Scripting the Type Inference Process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, 3–13.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003b. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, 62–71.

Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4es, Article 196 (Dec. 1996).

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, 96–107.

Leslie Koninck, Tom Schrijvers, and Bart Demoen. 2007. User-definable Rule Priorities for CHR *(PPDP '07)*. ACM, 25–36.

Hendrik Vincent Koops, José Pedro Magalhães, and W. Bas De Haas. 2013. A Functional Approach to Automatic Melody Harmonisation. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*. ACM, 47–58.

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, 3–18.

Simon Marlow and others. 2010. Haskell 2010 Language Report. (2010). https://www.haskell.org/onlinereport/haskell2010/.

J. Garrett Morris and Mark P. Jones. 2010. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 375–386.

Hubert Plociniczak, Heather Miller, and Martin Odersky. 2014. Improving Human-Compiler Interaction Through Customizable Type Feedback.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cristal.inria.fr/attapl/

Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electr. Notes Theor. Comp. Sci.* 312 (2015), 197–213.

Scala Team. Docs for `scala.annotation.implicitNotFound`. (????).

Alejandro Serrano and Jurriaan Hage. 2016a. *Context-Dependent Type Error Diagnosis for Functional Languages*. Technical Report UU-CS-2016-011. Department of Information and Computing Sciences, Utrecht University.

Alejandro Serrano and Jurriaan Hage. 2016b. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Programming Languages and Systems, ESOP 2016*. 672–698.

Alejandro Serrano and Jurriaan Hage. 2017. Lightweight Soundness for Towers of Language Extensions. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2017)*. ACM, New York, NY, USA, 23–34.

Alejandro Serrano, Jurriaan Hage, and Patrick Bahr. 2015. Type Families with Class, Type Classes with Family. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*.

Michael Snoyman. 2012. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive Type Debugging in Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, 72–83.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2006. Type Processing by Constraint Reasoning. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Lecture Notes in Computer Science, Vol. 4279. 1–25.

Swift Team. 2016. Type Checker Design and Implementation. (2016). https://github.com/apple/swift/blob/master/docs/TypeChecker.rst

Kanae Tsushima and Kenichi Asai. 2013. *An Embedded Type Debugger*. 190–206.

Markus Voelter. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (2011), 333–412.

Jeremy Wazny. 2006. *Type inference and type error diagnosis for Hindley/Milner with extensions*. Ph.D. Dissertation. University of Melbourne, Australia.

Brent A. Yorgey. 2012. Monoids: Theme and Variations. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, 105–116.

Brent A. Yorgey. 2016. User manual for `diagrams`. (2016).

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion *(TLDI '12)*. ACM, 53–66.

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class *(PLDI 2015)*. ACM, 12–21.

A Soundness check

The check is described as a pair of judgments in Figure 14. Given a type signature $\forall \overline{a}.Q \Rightarrow \tau$, we need to check whether $Q \vdash \epsilon \Vdash Q$ ok. The rules of this judgment traverse the entire tree of constraints, keeping a log of the constraints which have already been visited and are now part of the environment Q. The only non-trivial rule is the one for *IfNot c f o*. In that case, we need to switch to the $Q \vdash Q \Vdash f$ inconsistent judgment for the second argument, whose goal is to check that f is guaranteed to lead to an inconsistency. Primitive constraints are given to the constraint rewriting engine, whose operation must finish with a $\bot$ in the set of residual constraints. A set of constraints is inconsistent whenever at least one of them is inconsistent; in the case of an *IfNot c f o* constraint both branches f and o must lead to an inconsistency.

In the check we assume that all type families have been expanded. This is enough for most use cases, but breaks when type families use recursion. In that case we need to do fixpoint iteration over the call graph, taking into consideration that under a given environment more than one branch of the type family could be taken.

B The path library, annotated

The path library provides operations to manipulate file paths in a safer way than the usual string-based manipulation. The extra type safety comes from using a specialized data type *Path*, which is parametrized by the base location of the path, which can be absolute or relative, and the type of path it represents, either a directory or a file. For example, you may only extract the file name of a path representing a file,

filename :: *Path b File* → *Path Rel File*

and may only concatenate two paths if the first one is a directory and the second one is relative:

(</>) :: *Path b Dir* → *Path Rel t* → *Path b t*

In this section we define type signatures introducing domain-specific terms in all the basic operations of the path library. This serves as an example of how an existing library can be wrapped, and also as a real world usage of our techniques.

Almost every operation imposes some requirement on either the base or the type of path, so we introduce several synonyms in Figure 15. We do so for all possible combinations of base location and type. In addition, some operations expect the bases of two paths to coincide, which we check by means of *EqualBase*.

Another common mistake in using the library is giving a string directly as an argument, without prior conversion to *Path*. Following the idea of conversions and alternatives introduced in the paper, we can define custom error diagnosis in *CheckIsPath* suggesting parsing the string to get a well-typed program.

Finally we can wrap the operations from the *Path* module. The result is given in Figure 16. The original type signatures are given as comments below the error-annotated one.

B.1 Examples of error messages

The following session of the GHC interpreter shows our customized error diagnosis in use. The first example involves using a string as a path without conversion:

```
> filename "type-errors/Example.hs"
<interactive>:1:1: error:
* You need an explicit conversion from string to Path
  using one of the parse* functions.
* In the expression: filename "type-errors/Example.hs"
```

Let us define four (non-existing) files with all combinations of base and type:

```
> let pRelDir :: Path Rel Dir = undefined
> let pAbsDir :: Path Abs Dir = undefined
> let pRelFile :: Path Rel File = undefined
> let pAbsFile :: Path Abs File = undefined
```

We can check that *filename* and *dirname* only accept the corresponding types:

```
> filename pRelDir
<interactive>:2:1: error:
* The given path type 'Dir' does not represent a file.
* In the expression: filename pRelDir

> dirname pAbsFile
<interactive>:3:1: error:
* The given path type 'File' does not represent a dir.
* In the expression: dirname pAbsFile
```

or that *isParentOf* requires bases to coincide and gives a suggestion if given a string:

```
> isParentOf pRelDir pAbsFile
<interactive>:4:1: error:
* The path bases 'Rel' and 'Abs' should coincide:
  they have to be either both relative or both absolute.
* In the expression: isParentOf pRelDir pAbsFile

> isParentOf "type-errors" pAbsFile
<interactive>:5:1: error:
* You need a explicit conversion from string to Path
  using one of the parse* functions.
* In the expression: isParentOf "type-errors" pAbsFile
```

Finally, we can check how the layered implementation of *CustomErrors* gives back two error messages when the arguments to (</>) do not have the right types:

```
> pAbsFile </> pAbsDir
<interactive>:6:1: error:
* The given path type 'File' does not represent a dir.
* In the expression: pAbsFile </> pAbsDir
<interactive>:6:1: error:
* The given path base 'Abs' is not relative.
* In the expression: pAbsFile </> pAbsDir
```

$$\frac{\mathcal{Q} \vdash \langle id\ ;\ \epsilon\ ;\ Q, P\rangle \hookrightarrow \langle \varphi\ ;\ \epsilon\ ;\ Q'\rangle \qquad \bot \in Q'}{\mathcal{Q} \vdash Q \Vdash P \text{ inconsistent}}$$

$$\frac{\mathcal{Q} \vdash Q \Vdash f \text{ inconsistent} \qquad \mathcal{Q} \vdash Q, c \Vdash o \text{ inconsistent}}{\mathcal{Q} \vdash Q \Vdash \mathit{IfNot}\ c\ f\ o \text{ inconsistent}}$$

$$\frac{\mathcal{Q} \vdash Q, Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n \Vdash Q_i \text{ inconsistent for some } i}{\mathcal{Q} \vdash Q \Vdash Q_1, \ldots, Q_n \text{ inconsistent}}$$

$$\frac{}{\mathcal{Q} \vdash Q \Vdash P \text{ ok}}$$

$$\frac{\mathcal{Q} \vdash Q \Vdash f \text{ inconsistent} \qquad \mathcal{Q} \vdash Q, c \Vdash o \text{ ok}}{\mathcal{Q} \vdash Q \Vdash \mathit{IfNot}\ c\ f\ o \text{ ok}}$$

$$\frac{\begin{array}{c}\mathcal{Q} \vdash Q, Q_2, \ldots, Q_n \Vdash Q_1 \text{ ok} \\ \vdots \\ \mathcal{Q} \vdash Q, Q_1, \ldots, Q_{n-1} \Vdash Q_n \text{ ok}\end{array}}{\mathcal{Q} \vdash Q \Vdash Q_1, \ldots, Q_n \text{ ok}}$$

Figure 14. Judgments checking soundness of *IfNot* constraints

```
type CheckRel  x = x :~: Rel
  :⇛: Text "The given path base '" :◇: ShowType x :◇: Text "' is not relative."
type CheckAbs x = x :~: Abs
  :⇛: Text "The given path base '" :◇: ShowType x :◇: Text "' is not absolute."
type CheckDir  x = x :~: Dir
  :⇛: Text "The given path type '" :◇: ShowType x :◇: Text "' does not represent a dir."
type CheckFile x = x :~: File
  :⇛: Text "The given path type '" :◇: ShowType x :◇: Text "' does not represent a file."
type EqualBase x y = x :~: y
  :⇛: Text "The bases '" :◇: ShowType x :◇: Text "' and '" :◇: ShowType y :◇: Text "' should coincide:"
    :$$: Text "they have to be either both relative or both absolute."
type CheckIsPath x b t
  = x :~: Path b t :⇛?: ([x ~ String :⇛!: Text "You need a explicit conversion from string to Path"
                                    :$$: Text "using one of the parse* functions."],
                         Text "The given expression is not a path.")
```

Figure 15. Synonyms for shared error messages

```
import Path hiding ((</>), stripDir, isParentOf, parent, dirname, filename, fileExtension, setFileExtension)
import qualified Path

(</>) :: CustomErrors [[CheckIsPath p1 b1 t1, CheckIsPath p2 b2 t2],
                       [CheckDir t1, CheckRel b2],
                       [Check (r ~ Path b1 t2)]]
      => p1 -> p2 -> r
-- (</>) :: Path b Dir -> Path Rel t -> Path b t
(</>) = (Path. </>)

stripDir :: CustomErrors [[CheckIsPath p1 b1 t1, CheckIsPath p2 b2 t2],
                          [CheckDir t1, EqualBase b1 b2],
                          [Check (r ~ m (Path Rel t2)), Check (MonadThrow m)]]
         => p1 -> p2 -> r
-- stripDir :: MonadThrow m => Path b Dir -> Path b t -> m (Path Rel t)
stripDir = Path.stripDir

isParentOf :: CustomErrors [[CheckIsPath p1 b1 t1, CheckIsPath p2 b2 t2],
                            [CheckDir t1, EqualBase b1 b2]]
           => p1 -> p2 -> Bool
-- isParentOf :: Path b Dir -> Path b t -> Bool
isParentOf = Path.isParentOf

parent :: CustomErrors [[CheckIsPath p b t], [CheckAbs b], [Check (r ~ Path Abs Dir)]]
       => p -> r
-- parent :: Path Abs t -> Path Abs Dir
parent = Path.parent

dirname :: CustomErrors [[CheckIsPath p b t], [CheckDir t], [Check (r ~ Path Rel Dir)]]
        => p -> r
-- dirname :: Path b Dir -> Path Rel Dir
dirname = Path.dirname

filename :: CustomErrors [[CheckIsPath p b t], [CheckFile t], [Check (r ~ Path Rel File)]]
         => p -> r
-- filename :: Path b File -> Path Rel File
filename = Path.filename

fileExtension :: CustomErrors [[CheckIsPath p b t], [CheckFile t]])
              => p -> String
-- fileExtension :: Path b File -> String
fileExtension = Path.fileExtension

setFileExtension :: CustomErrors [[CheckIsPath p b t], [CheckFile t],
                                  [Check (r ~ m (Path b File)), Check (MonadThrow m)]]
                 => String -> p -> r
-- setFileExtension :: MonadThrow m => String -> Path b File -> m (Path b File)
setFileExtension = Path.setFileExtension
```

Figure 16. Annotated signatures for the `path` library

Deriving Lenses Using Generics

Csongor Kiss
Imperial College London
csongor.kiss14@imperial.ac.uk

Matthew Pickering
University of Bristol
matthew.pickering@bristol.ac.uk

Toby Shaw
Imperial College London
toby.shaw14@imperial.ac.uk

Abstract

Lenses are an abstraction which simplify working with product types. Their main advantage is that they allow programmers to specify complicated nested modifications compositionally.

Any product datatype readily admits several possible lenses. For a datatype with named fields, there is a lens which targets each of the named fields. Alternatively, we can specify which part of a product to access positionally by specifying an index. We can also access a product through a type index, accessing the field which uniquely has the given type. Finally, in a more complex case, there is a lens which accesses a sub-record of a larger record by ignoring some fields. All these lenses are useful in different situations but it is onerous for the programmer to write the boilerplate for each definition.

As such, we explore how we can derive these lenses using existing intralinguistic features such as generics. Our setting is Haskell and as such we use the `GHC.Generics` library in order to derive lenses from a generic representation. We describe the implementation techniques for each of the aforementioned lenses and discuss the behaviour of the optimiser on this high-level code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Haskell, Datatype-generic programming, Lenses

1. Introduction

Lenses are proving to be of ubiquitous use in modern functional programming. An abstraction which captures the essence of a named record field by the ability to view and set proves useful in the everyday task of manipulating types composed of products.

However, how should we allow the programmer to access the power of lenses in their programs? Should we make them write the definitions by hand? Should we make them use extralinguistic features like Template Haskell? Should we build support into the compiler? To all these questions we answer no! Of course the answer is that we should derive our lenses *generically*.

Our first lens *field* accepts the name of the record field we wish to access as its first argument and produces a lens which focuses on that record field. We can use it with datatypes which contained named fields.

$$\mathbf{data}\ T1 = T1\ \{a :: Int, b :: Int\}$$

$$inc1 :: T1 \to T1$$
$$inc1 = over\ (field\ @\texttt{"a"})\ (+1)$$

Our second lens *position* accesses fields of datatypes *positionally*. We specify a numerical index and a lens is provided for us which accesses that index. This provides a lightweight way to define lenses for datatypes which don't contain named fields. An advantage over less flexible existing approaches.

$$\mathbf{data}\ T2 = T2\ Int\ Bool\ Char$$
$$inc2 :: T2 \to T2$$
$$inc2 = over\ (position\ @1)\ (+1)$$

Our third lens *typed* focuses on a value by its *type*. So long as the field's type is unique within its parent datatype, we can unambiguously access the field simply by providing the type we wish to access.

$$\mathbf{data}\ T3 = T3\ Int\ Bool\ Char$$
$$inc3 :: T3 \to T3$$
$$inc3 = over\ (typed\ @Int)\ (+1)$$

Our final lens *super* allows us to cast between two datatypes which contain the same named fields. This is useful in business applications where we might have similar but different datatypes which accrue information through the processing pipeline. Our lens allows seamless updating and projection of types related in this way.

$$\mathbf{data}\ PersonWithId = PersonWithId$$
$$\quad \{name :: String$$
$$\quad , age :: Int$$
$$\quad , iden :: Int$$
$$\quad \}$$
$$\mathbf{data}\ Person = Person$$
$$\quad \{name :: String$$
$$\quad , age :: Int$$
$$\quad \}$$
$$getPerson :: PersonWithId \to Person$$
$$getPerson = view\ (super\ @PersonWithId)$$
$$updatePerson :: Person \to PersonWithId \to PersonWithId$$
$$updatePerson = set\ (super\ @PersonWithId)$$

In this example, consider that $Person$ is the result of a user filling in a form. We then want to update the record we hold on our database which also contains an identifier.

This triptych of lenses compose together to write elegant, composable functional programs. We define a family of lenses implemented using the unified representation provided by the $Generic$ type class. Programmers can use our lenses for any datatype which provides a compiler-generated instance for $Generic$. Then, if they wish to give more convenient names to certain lenses, they can.

We don't pre-generate all possible lenses and invisibly pollute the namespace as is the case with Template Haskell solutions.

Our contributions are thus,

1. A description of an algorithm for generically deriving lenses for named fields using `GHC.Generics`.
2. A generic "subtyping" lens which can be used to emulate a structural typing discipline in Haskell.
3. A family of generic positional lenses which target unnamed fields in datatypes.
4. A generic lens targeting unnamed fields by their unique type.
5. An evaluation of the performance implications of defining lenses generically.
6. generic-lens: A library released on Hackage which implements this paper.[1]

2. Motivation

Consider the following datatypes used to configure an application:[2]

```
data Credentials
  = Credentials String String

data Host
  = Host String Int

data AppConfiguration
  = AppConfiguration
    { dbCredentials :: Credentials
    , database      :: String
    , host          :: Host
    , sessionId     :: Int
    }

data DbConfiguration
  = DbConfiguration
    { dbCredentials :: Credentials
    , database      :: String
    , sessionId     :: Int
    }
```

A popular style of programming in Haskell is the parameterisation of capabilities of the computational context via ad-hoc overloaded functions using type classes. An example of this technique is the type class $MonadReader\ e\ m$, which describes that any action running in the context of m has access to a read-only environment of type e. In real-world applications, however, e is usually fixed to a concrete type, so that the programmer has total knowledge of the information available in the environment. Using generically defined data accessors, we can achieve a higher level of abstraction by further parameterising the environment, only ever stating certain requirements about it, and satisfying these requirements automatically.

Given two primitive functions

```
dbConnect :: DbConfiguration → IO Connection
saveLogin :: Host → String → Connection → IO ()
```

We can write a function polymorphic in the environment type, by requiring that it is a structural subtype of $DbConfiguration$, has a field of type $Credentials$ and one of type $Host$ with field name "host".

```
connect
  :: ( MonadReader e m
     , MonadIO m
     , Subtype DbConfiguration e
     , HasType Credentials e
     , HasField "host" Host e
     ) ⇒ m ()
connect = do
  config ← view super @DbConfiguration
  connection ← liftIO (dbConnect config)
  username ← view (typed @Credentials ∘ position @1)
  host ← view (field @"host")
  liftIO (saveLogin host username connection)
  return ()
```

First we view the environment as its supertype, $DbConfiguration$, needed to call the $dbConnect$ function. Then we get the $username$ by referring to the first field of the $Credentials$ in our environment. Finally, we get the $host$ field by name. This function can then be instantiated with $AppConfiguration$, but it is not restricted to it – any other type satisfying the required constraints can be used to run the computation.

We argue that this approach provides better composability, because each function can refer to the fields it actually needs, avoiding the problem of "leaky" environments, whereby a function is needlessly passed more information than it actually needs, due to the $MonadReader$ parameterisation. Since the conversion functions don't have to be manually instantiated, it is easier on the programmer as well. Compared to traditional lens approaches, our *typed* lens doesn't rely on the field name, only the uniqueness of the type in the structure, so a renaming of the field doesn't affect existing code.

3. Background

We start by explaining the extensions to Haskell which we will use in order to implement our lenses.

3.1 DataKinds

Our work makes heavy use of Haskell's type-level programming facilities. When writing type-level programs, we still want the benefits of a static type system. To reason about the type of types (usually referred to as *kinds*), we reach for a richer extension of Haskell's kind system, implemented in GHC under the $DataKinds$ compiler extension (Yorgey et al. 2012).

In particular, we use the ability to promote string and numeric literals to the type-level for indexing our *field* and *position* lenses with statically known indices.

The ordinary string literal `"Foo"` in a type context becomes a type of kind $Symbol$.

In addition to these two primitives, it is also possible to take user-defined datatypes and use them at the type-level. In that case, the constructors become promoted to types, and the type itself gets promoted to a kind – the kind of the promoted constructors.

A simple example is the Peano encoding of natural numbers.

```
data Peano = Zero | Succ Peano
```

We can then use ‘$Zero$ :: $Peano$ and ‘$Succ$ :: $Peano \to Peano$ in our type-level programs much like how we would use them at the value-level. The lifted constructor names are prefixed with the ‘ symbol as a means of helping readability.

3.2 GHC.Generics

When programming generically, we write algorithms that abstract over certain aspects of the data we operate on, allowing our program

[1] `http://hackage.haskell.org/package/generic-lens`

[2] Note that these records share selector names, a property normally rejected by Haskell due to naming conflict. The GHC extension $DuplicateRecordFields$ allows this example to compile, however the overloaded selector functions will not be usable.

to be instantiated to multiple concrete cases, without having to rewrite or modify the functionality.

Datatype-generic programming in particular is concerned with one such aspect, the algebraic structure of the data. This means that instead of programming against some concrete datatype, we express our program in terms of shape, and accept all types that are of the "right shape".

As a simple example, consider the following two data declarations:

$$\textbf{data}\ \mathit{MaybeInt} = \mathit{Just}\ \mathit{Int} \mid \mathit{Nothing}$$
$$\textbf{data}\ \mathit{OptionInt} = \mathit{Some}\ \mathit{Int} \mid \mathit{None}$$

There is clearly an isomorphism between these types: in both cases, we have a constructor of one argument, and another constructor that takes no arguments. We need a vocabulary for talking about such shapes in a unified way, and express functions in terms of this language.

The approach adopted by the `GHC.Generics` library starts with the observation that we only need a handful of building blocks to define most standard Haskell datatypes, and it is possible to then define (and indeed generate by the compiler) mappings from and to these building blocks. This approach is similar to the approach proposed in (Hinze 2004).

For example, the components we need in order to define the above types are

1. a constructor that takes one argument
2. a sum of two other types
3. a constructor that takes no arguments

As such, the generic representation for these datatypes is:

$$\mathit{Rec0}\ \mathit{Int} :+: \mathit{U1}$$

That is, we either have an Int, or a unit. $\mathit{Rec0}$ denotes the point where our unfolding ends: the generic representation is shallow, meaning that it is only concerned with the shape of the immediate datatype, and not the types contained within.

A slightly more complicated example is a list of integers

$$\textbf{data}\ \mathit{List} = \mathit{Cons}\ \mathit{Int}\ \mathit{List} \mid \mathit{Nil}$$

Note that in this case, our first constructor contains two types, one of which is List itself. The generic representation in this case is

$$(\mathit{Rec0}\ \mathit{Int} :*: \mathit{Rec0}\ \mathit{List}) :+: \mathit{U1}$$

The complete set of these constructors is:

1. Values: $\mathit{Rec0}\ a$. A value of this type is constructed by feeding a value of type a to the $\mathit{K1}$ constructor.
2. Sum types: $f :+: g$. When a datatype has multiple constructors, we can think of the type as being the sum of these constructors. To produce the sum of f and g, we either inject a f into the left-hand side using the $\mathit{L1}$ constructor, or a g into $\mathit{R1}$, the right-hand side. Since the sum operator is binary, the way larger sums are represented is by nesting binary sums inside each other. A datatype of three arguments could either be $(f :+: g) :+: h$ or $f :+: (g :+: h)$.
3. Nullary constructors: $\mathit{U1}$.
4. Void datatypes (no constructors): $\mathit{V1}$.
5. Product types: $f :*: g$. A constructor that has multiple arguments can be thought of as the product of those arguments: all of the values must be present at the same time. Similarly to sum types, larger products are constructed as binary products of other binary products.
6. Metadata: The $\mathit{M1}$ constructor is used to wrap another generic node (any of the above), and store additional metainformation about the wrapped structure, such as the name of a the constructor.

The way the compiler exposes this representation is through the derivation of an instance for the $\mathit{Generic}$ type class using the $\mathit{DeriveGeneric}$ extension (Magalhaes et al. 2010) in GHC.

$$\begin{aligned}&\textbf{class}\ \mathit{Generic}\ a\ \textbf{where}\\&\quad\textbf{type family}\ \mathit{Rep}\ a :: * \to *\\&\quad\mathit{from} :: a \to \mathit{Rep}\ a\ x\\&\quad\mathit{to} :: \mathit{Rep}\ a\ x \to a\end{aligned}$$

The $\mathit{Generic}$ class has an associated type family Rep which computes the type of its generic representation. This class is used to describe types of no type arguments. There exists a class, $\mathit{Generic1}$, that provides the same functionality for types of kind $* \to *$ (that is, datatypes with one type parameter), sharing much of the building blocks, which explains why in the simple $\mathit{Generic}$ case, we see an extra, unused type variable, x.

3.2.1 GHC 8.0 additions

With the release of GHC 8.0, the above representation has been augmented with type-level metadata. This means that metainformation, such as the name of a constructor, is available statically through type-level programming techniques. Metadata is attached by using the $\mathit{M1}$ constructor which has a phantom type parameter for storing the additional information.

$$\begin{aligned}&\textbf{newtype}\ \mathit{M1}\ i\ (c :: \mathit{Meta})\ (f :: * \to *)\ p\\&\quad= \mathit{M1}\ \{\mathit{unM1} :: f\ p\}\end{aligned}$$

Here, c is phantom type: it has no inhabitant, and is purely present for carrying information about the wrapped generic subtree $f\ p$.

This metadata can be one of three things (somewhat simplified for brevity):

$$\begin{aligned}&\textbf{data}\ \mathit{Meta}\\&\quad= \mathit{MetaData}\ \mathit{Symbol}\\&\quad\mid \mathit{MetaCons}\ \mathit{Symbol}\\&\quad\mid \mathit{MetaSel}\ (\mathit{Maybe}\ \mathit{Symbol})\end{aligned}$$

1. $\mathit{MetaData}$, the metadata for the datatype itself, containing its name as a type-level string literal.
2. $\mathit{MetaCons}$, the metadata for constructors, containing the name of the constructor.
3. $\mathit{MetaSel}$, the metadata for fields in constructors with an optional name. In case of records, it stores the name of the record field.

Meta's constructors can be used in a type context via $\mathit{DataKinds}$, with Meta being promoted to the kind of its constructors.

3.3 Lenses

A lens is a generic programming abstraction which abstracts the product nature of a datatype. There are many different equivalent implementations of lenses, the simplest is a pair of a projection function and updating function which are more commonly known as the *getter* and the *setter*.

$$\textbf{data}\ \mathit{LensC}\ s\ a = \mathit{Lens}\ (s \to a)\ (a \to s \to s)$$

A simple lens focuses on one side of a product. It is usual to denote the lens which focuses on the first half of a tuple by $\mathit{\_1}$.

We can define $\mathit{\_1}$ as follows:

$$\begin{aligned}&\mathit{\_1} :: \mathit{LensC}\ (a, b)\ a\\&\mathit{\_1} = \mathit{Lens}\ \mathit{get}\ \mathit{set}\end{aligned}$$

```
  where
    get (a, _) = a
    set a (_, b) = (a, b)
```

There are many other examples of lenses. Any composite datatype which is defined as the product composition of datatypes gives rise to a lens. As such, lenses are ubiquitous in common programming scenarios but also amenable to being defined generically.

Lenses also provide an additional level of abstraction, by defining a lens we can treat any abstract datatype as if it were a product even if the concrete representation is not defined in such a way.

For example, positive integers can be partitioned into two disjoint sets of even and odd numbers. Thus, an alternative representation is a pair of a boolean denoting parity and an integer. Instead of choosing this representation which makes arithmetic operations more difficult, we instead define a lens which exposes the parity.

```
parity :: LensC Int Bool
parity = Lens get set where
  get n = even n
  set p n =
    if p ≡ even n
      then n
      else case p of
        False → n − 1
        True → n + 1
```

3.3.1 van Laarhoven Lenses

The representation of lenses that we will use is the van Laarhoven representation. We choose this representation as it is the one predominantly used in the Haskell ecosystem due to the `lens` library (Kmett 2017).

A van Laarhoven lens is a function of the following type:

```
type Lens s a = ∀f.Functor f ⇒ (a → f a) → (s → f s)
```

This representation is exactly equivalent to the basic definition we defined above. The isomorphism is witnessed by the following conversion functions.

```
toVLLens :: LensC s a → Lens s a
toVLLens (LensC get set) = λf s → set <$> f (get s)

view :: Lens s t a b → s → a
view l = getConst ∘ l Const

set :: Lens s t a b → b → s → t
set l b = runIdentity ∘ l (const (Identity b))

toCLens :: Lens s a → LensC s a
toCLens vl = LensC (view vl) (set vl)
```

The correspondence between these two representations has been quite extensively studied. (Jaskelioff and O'Connor 2015) (O'Connor 2011)

The choice of representation is not crucial to our work but it is convenient to only have to implement a single function. We could instead apply our work to the profunctor (Pickering et al. 2017) or applicative (Matsuda and Wang 2015) representation.

3.4 Type Applications

Functions with polymorphic types are able to be applied to monomorphic arguments. When the application takes place the polymorphic type variable is instantiated to the type of its argument. For example, the identity function has the polymorphic type $\forall a.a \rightarrow a$ but when applied to a string, the type variable a is implicitly instantiated to $String$.

The purpose of visible type applications (Eisenberg et al. 2016) is to explicitly represent this instantiation as if it were a normal argument to the function. To carry on with our example, we can choose to explicitly represent the instantiation of the type variable a by providing a type argument prefixed by @.

```
id @String "abc" :: String
```

A programmer can optionally choose to provide an argument which explicitly instantiates polymorphic type variables.

The real value of visible type applications comes when it is necessary to instantiate a type variable in order to resolve ambiguity. A prototypical example is the composition:

```
normalise = show ∘ read
```

As $read$ has a polymorphic result type constrained by the $Read$ type class and $show$ has a polymorphic argument, the constraint solver can't resolve this uninstantiated polymorphic type variable which lies in the middle of this composition.

In previous versions of GHC, the idiomatic way to resolve these situations was to use a datatype called $Proxy$ which had a single nullary constructor and a phantom type variable.

```
data Proxy a = Proxy
```

Users would then explicitly add arguments to functions which required explicit instantiation to resolve ambiguity.

With $Proxy$ we would instead define a normalisation function which takes an explicit argument which says which type we want to normalise by.

```
normaliseProxy :: ∀a.(Read a, Show a)
                  ⇒ Proxy a → String → String
normaliseProxy Proxy = show ∘ readP
  where
    readP :: Read a ⇒ String → a
    readP = read
```

This can then be used in a slightly cumbersome way to resolve this ambiguity.

```
normaliseInt :: String → String
normaliseInt = normaliseProxy (Proxy :: Proxy Int)
```

In a language with visible type application, we can resolve this ambiguity in a more lightweight fashion. We first define

```
normaliseVTA :: ∀a.(Read a, Show a) ⇒ String → String
normaliseVTA = show ∘ read @a
```

Then to call $readVTA$ we must explicitly say which type we wish to instantiate a to be, lest the compiler still complains that a is ambiguous.[3]

Calling $normaliseVTA$ for different types is now very ergonomic as we explicitly instantiate a by using a type application.

```
normaliseBool :: String → String
normaliseBool = normaliseVTA @Bool
```

This approach has two benefits, the code is more concise and conforms to the natural style that one might choose to write these definitions.

Notice that $readVTA$ can be called *without* explicitly instantiating a but will result in an unhelpful ambiguity error. Perhaps a future extension will allow programmers to annotate type variables which must be manually instantiated. This is a distinct advantage of the $Proxy$ approach.

This example might appear artificial but $Proxy$ is used extensively with type families. A type family is in general not injective,

[3] In fact, without enabling the language extension `AllowAmbiguousTypes` our definition of $normaliseVTA$ would be rejected as the type variable a is not resolved by either the argument or the result type

meaning that if an argument type is specified in terms of a family, we can't resolve what the type variable should be by the passed argument.

```
type family F a where
  F Int = Char
  F Bool = Char
f :: F a → Char
f _ = 'a'
```

When a user passes a $Char$ as the first argument to f, we can't resolve the value of a as it could be (in general) anything such that $F\ a \sim Char$.

Visible type applications allow us to call f without changing the definition of f by instantiating a to a specific type. In this case, we choose Int.

```
g :: Char
g = f @Int '5'
```

Another approach to resolve ambiguity is injectivity annotations for type families which instruct the constraint solver in the case where the definition is injective (Stolarek et al. 2015).

3.5 Closed Type Families

Closed type families (Eisenberg et al. 2014) allow programmers to specify closed type functions which have potentially overlapping patterns. The clauses of a closed type family are read from top to bottom and as such allow overlapping patterns.

One type family definable in a closed but not open style is the type family Eq which decides whether two types are equal.

```
type family Eq a b where
  Eq a a = True
  Eq a b = False
```

The closed type family Eq illustrates two important features of closed type families. Firstly, a closed type family can be specified by multiple equations and secondly that these equations can overlap.

3.6 Custom Type Errors

A key part to our API is the use of custom type errors. A custom type error is a special built-in type family which has special reduction behaviour and typing rules. $TypeError$ is defined abstractly as an open type type family which maps types of kind $ErrorMessage$ to any type.

```
type family TypeError (msg :: ErrorMessage) :: k
```

$ErrorMessage$ is a simple DSL which can be used to construct error messages. It allows users to concatenate symbols, display types and print literal strings.

Whilst a failure of a type family to match will not cause failure at runtime, reduction can get stuck and leave confusing error messages about unmatchable types. Using $TypeError$ the programmer can instead handle these special cases by explaining *why* the reduction would get stuck by adding a catch-all case with an explanation.

```
type family G a where
  G Int = ()
  G Char = ()
  G a = TypeError
    (Text "G only accepts types Int or Char."
    :$$: Text "You provided an argument: "
    :<>: ShowType a)
```

By including a catch-all as the final clause, the compiler is able to reduce $G\ Bool$ to $TypeError\ ...$ This now unsolvable constraint is collected and handled specially by the compiler to print at the end, resulting in a error message containing domain specific knowledge provided by us.

Since we use statically known indices (type-level naturals and symbols) to generate our lenses, providing an invalid index results in a type error.

For the ill-typed code snippet $view\ (position\ @3)\ (True, \texttt{"hello"})$, by default the error message is rather cryptic.

```
* No instance for (GHasPosition
                    2
                    3
                    (M1
                       S
                       (MetaSel
                          Nothing
                          NoSourceUnpackedness
                          NoSourceStrictness
                          DecidedLazy)
                       (Rec0 [Char]))
                    a0)
```

With a custom $TypeError$, we can provide a much more useful error message.

```
* The type (Bool, [Char]) does
    not contain a field at position 3
* In the second argument of ‘view’,
    namely ‘position @3’
```

4. Implementation

The prototypical use-case for datatype-generic programming is to stamp out a custom version of a function for a given type. Two examples of this would be generic traversals or folds, of which each type admits one canonical definition.

On the other hand, there are potentially many lenses for each datatype, each corresponding to a way we can decompose our definition into a product.

We consider three possible ways to access an individual field in a datatype.

1. By name
2. By index
3. By type

In addition, we describe a way to access multiple fields at once, by identifying a structural subtype relation.

What follows is an outline of the techniques used to implement these lenses in terms of GHC's Generic representation. Since the field lenses share the general logic, we show an in-depth implementation of the nominal case, then point out the main differences in the others.

4.1 Nominal Fields

One common use-case of lenses is accessing named fields in record datatypes. We define a family of lenses *field*, indexed by field names which focus on named fields in datatype. To recall an earlier example, *field* can be used to modify or update a field.

```
data T1 = T1 { a :: Int, b :: Int }
inc1 :: T1 → T1
inc1 = over (field @"a") (+1)
```

The index is a type-level string and must be explicitly provided using type applications. Specialised to $T1$, $field\ @\texttt{"a"}$ is a lens of type $Lens\ T1\ Int$ as the a field contains an Int.

We implement *field* as a method of the a type class.

```
class HasField (field :: Symbol) a s | s field → a where
  field :: Lens s a
```

We will define a single instance of *HasField* which implements *field* using a helper type class *GHasField* which is exactly the same but defined on the generic representation.

HasField is parameterised by three arguments. *field* is the name of the *field* which we wish to access, a is the type of the field we are accessing and s is the type of the container. *field* it not mentioned by any class methods, therefore we must explicitly instantiate using type applications in order to be able to resolve the type class instance. We provide an explicit kind signature for *field*, to denote we expect it to be a type-level string, but omit the signatures of the other fields, whose kinds are by default $*$.

The ordering of type variables is important: visible type applications follow this ordering. It is crucial to our API that we place the *field* type parameter first as it is the only type we must be explicit about. If we had been careless and instead our variables were ordered s, a, *field* then in order to instantiate *field* we would first have to instantiate s and a. We can rely on the compiler to infer these for us, but the resulting verbosity makes the code less pleasant to read.

```
field @_ @_ @"a"
```

Finally, we notice the functional dependency s *field* $\rightarrow a$, which expresses that in a given record, the field name determines the type of the field. This improves type inference.

The single instance for this class is defined in terms of the two helper classes *GHasField* and *ErrorUnless*, which we describe in the next sections.

```
instance
  ( Generic s
  , ErrorUnless field s (HasTotalFieldP field (Rep s))
  , GHasField field (Rep s) a
  ) ⇒ HasField field a s where
  field = repIso ∘ gfield @field
```

Where *repIso* is a lens that views a value as its (isomorphic) generic representation by using the methods from the *Generic* type class.

```
repIso :: Generic a ⇒ Lens a (Rep a x)
repIso f s = to <$> f (from s)
```

4.1.1 *GHasField*

In order to automatically instantiate this class, we introduce an auxiliary type class, which expresses the same property, not directly on the records, but in terms of the generic representation thereof. Everything is the same, except for the kind of the container, which is now $* \rightarrow * \rightarrow *$ – the kind of *Rep* s.

```
class GHasField (field :: Symbol)
                (s :: * → *) a | s field → a where
  gfield :: Lens (s x) a
```

We then define one instance for each type constructor used in the generic representation. Most of the instances are simple unpacking and repacking of the structure. The interesting cases are for sums (:+:) and products (:*:).

The root node of the derived *Generic* representation is the metadata node for the type. This metadata contains information such as the name of the datatype and its location. These are not useful for our purposes: we simply recurse into the structure.

In order to avoid having to pattern match on and reconstruct these metadata nodes, we define a helper lens, *mLens*, focusing on the value inside the node.

```
mLens :: Lens (M1 i c f p) (f p)
mLens f s = M1 <$> f (unM1 s)
```

And one for focusing on the value inside the value node

```
kLens :: Lens (Rec0 a x) a
kLens f s = K1 <$> f (unK1 s)
```

```
instance GHasField field s a ⇒
         GHasField field (D1 meta s) a where
  gfield = mLens ∘ gfield @field
```

Similarly, a constructor's metadata holds no relevant information.

```
instance GHasField field s a ⇒
         GHasField field (C1 meta s) a where
  gfield = mLens ∘ gfield @field
```

We now consider the base case. This is when we reach a selector with a name that matches our *field* in question. We want the value inside this selector to be *Rec0* a, i.e. a value of type a. The implementation of the actual lens is trivial: we peek into the meta-node corresponding to *S1* with *mLens*, then dig further to reach the value inside the *Rec0*'s *K1* constructor, using *kLens*.

```
instance GHasField field
         (S1 (`MetaSel (`Just field)) p f b) (Rec0 a)) a
         where
  gfield = mLens ∘ kLens
```

In the case that we have multiple constructors that are all records and all contain the required field, we can still derive a lawful lens, due to the distributive property of the Cartesian product.

```
instance
  ( GHasField field l a
  , GHasField field r a)
  ⇒ GHasField field (l :+: r) a where
  gfield =
    combine (gfield @field @l) (gfield @field @r)
```

Where *combine* is lens that joins two lenses into a lens of a sum type.

```
combine :: Lens (s x) a
        → Lens (t x) a
        → Lens ((s :+: t) x) a
combine sa _ f (L1 s)
  = fmap (λa → L1 (set sa a s)) (f (view sa s))
combine _ ta f (R1 t)
  = fmap (λa → R1 (set ta a t)) (f (view ta t))
```

We have now defined the necessary instances for all the required types, except for products (:*:). In the other compound case (the sum, :+:), we could easily define the instance by requiring that both subtrees have an instance. However, the situation for products is slightly more complicated. In order to be able to find a field with the right name and type in a product, it is sufficient to have the field in only one side of the branch. (In fact, records guarantee that if there is a matching field, there is exactly one such field.)

However, in Haskell, it is not trivial to express the disjunction of two constraints.

A first attempt at writing the instances might be

```
instance GHasField field f a
         ⇒ GHasField field (f :*: g) a where
instance GHasField field g a
         ⇒ GHasField field (f :*: g) a where
```

This is rejected by the compiler, because the two instance heads are exactly the same and hence fails the duplicate instance check. This is due to the fact that when picking an instance for a type, GHC looks for the most specific instance declaration that matches the type in question, and only when a matching instance is found, the instance constraints are considered.

We can solve the problem of the identical instance heads by introducing a special type class for handling products. This class contains an extra type parameter, used as a flag to denote which side of the product do we expect the field to be present. For the sake of simplicity, let `'True` mean that the field is in the left-hand side, and `'False` otherwise.

```
class GProductHasField field l r a (left :: Bool)
        | left field l r → a where
  gproductField :: Lens‘((l :*: r) x) a
```

Now our two instances from before can be written without clashes.

```
instance GHasField field l a
           ⇒ GProductHasField field l r a‘True where
  gproductField = first ∘ gfield @field
instance GHasField field r a
           ⇒ GProductHasField field l r a‘False where
  gproductField = second ∘ gfield @field
```

Where *first* and *second* are lenses that focus on the first and second elements of a generic product type respectively.

```
first :: Lens ((a :*: b) x) (a x)
first f (a :*: b)
   = fmap (:*:b) (f a)
second :: Lens ((a :*: b) x) (b x)
second f (a :*: b)
   = fmap (a:*:) (f b)
```

Now we need a way to use this special class for products to define the original *GHasField* instance. The attempt below gets rejected, because *left* can't be instantiated from the variables in the instance head.

```
instance GProductHasField field f g a left
           ⇒ GHasField field (f :*: g) a where
```

To make this initial branching decision, we turn to a closed type family.

```
type family Contains (field :: Symbol) f :: Bool where
  Contains field (S1 (‘MetaSel (‘Just field) _ _ _) _)
    = ‘True
  Contains field (l :*: r)
    = Contains field l ∨ Contains field r
  Contains field (l :+: r)
    = Contains field l ∧ Contains field r
  Contains field (C1 _ f)
    = Contains field f
  Contains field (D1 _ f)
    = Contains field f
  Contains field _
    = ‘False
```

Contains accepts the very same structure as the one our generic *GHasField* class operates on, and it determines whether a generic tree contains a field or not. Named selectors whose name matches *field* are accepted. Products are accepted so long as one of the subtrees contain *field*.

Now we can use our type family to make the first decision: look up the field in the left-hand side. If the field is present, our type family reduces to ‘*True*, picking the instance that expects the field to be in the left subtree. Otherwise, we go right.

```
instance (GProductHasField field f g a (Contains field f))
   ⇒ GHasField field (f :*: g) a where
  gfield =
    gproductField @field @_ @_ @_ @(Contains field f)
```

At this point, we have all the necessary instances to access fields in a generic tree of the right shape. We have defined an instance for each type constructor used in generic representations, thus, any usage of the *field* lens will either produce a lens or the user will receive a custom-type error from one of our instances. They should never face a constraint solver error.

4.1.2 *ErrorUnless*

The *ErrorUnless* constraint is used to improve error messages. The constraint verifies that the requested *field* exists in a datatype.

It might seem more natural to include an error clause in the definition of *Contains* which must also compute this information but occurrences of *Contains* may be nested in a deep chain of indirections. In order to provide the most informative error message, we may require additional information which must be threaded through as an argument to each class or type family.

To get around this issue, we realise the error at a higher level, scouting for incompatibilities in a separate pass to instantiating the class method.

This error condition can be expressed using type families, and is calculated in a separate traversal before the *GHasField* instance is evaluated.

```
instance
  (Generic s
  , ErrorUnless field s (HasTotalFieldP field (Rep s))
  , GHasField field (Rep s) a
  ) ⇒ HasField field a s where
```

This strategy works where the error condition fails in the same situations as the instance resolution. If more fail from the error condition then it is overly restrictive, whereas if less fail from it then cryptic type errors will be reachable.

ErrorUnless is a simple type family that reduces to the trivial unit constraint in the success case, and the custom error message otherwise.

```
type family ErrorUnless (field :: Symbol)
                        (s :: Type)
                        (contains :: Bool) :: Constraint
                        where
  ErrorUnless _ _ True
    = ()
  ErrorUnless field s False
    = TypeError
      (Text "The type "
      :<>: ShowType s
      :<>: Text " does not contain a field named "
      :<>: ShowType field
      )
```

4.2 Positional Fields

Not all useful data-types will be records, and therefore will not admit lenses parameterised by field name. The simplest example are tuples, which have a number of unnamed fields. We can apply

similar techniques as for nominal lenses to create positional lenses, used for accessing fields by position.

We next implement *position* as a method of the *HasPosition* class.

```
class HasPosition (index :: Nat) a s | s index → a where
  position :: Lens s a
```

This class holds types which contain a field of a particular position and type. The modification to *HasField* is that it takes a *Nat* rather than a *Symbol*, reflecting its different indexing functionality.

We can then use this generic lens to access an element in a nested tuple.

```
alicebob :: (String, (Bool, String))
alicebob = ("Alice", (True, "Bob"))

t :: Bool
t = view (position @2 ∘ position @1) alicebob
```

Likewise with *GHasField*, we utilise an auxiliary class *GHasPosition*, which works on the generic representation of our type. The argument *offset* to track our position in the traversal, *index* is the position we wish to access.

```
class GHasPosition (offset :: Nat)
  (index :: Nat)
  (s :: * → *)
  a | offset index s → a where
  gposition :: Lens (s x) a
```

The approach for generating instances of this type class follows a similar strategy as for nominal fields.

We first define a type's size by the minimum number of fields it contains. The type family *Size* calculates this on a type's generic representation.

```
type family Size f :: Nat where
  Size (l :*: r)
    = Size l + Size r
  Size (l :+: r)
    = Min (Size l) (Size r)
  Size (D1 meta f)
    = Size f
  Size (C1 meta f)
    = Size f
  Size f
    = 1
```

where *Min* is defined as the minimum of two type-level natural numbers.

We traverse the generic representation, constructing a lens along the path. Since a specific index is required, we must also keep track of our left-to-right position.

When a sum constructor $l :+: r$ is encountered, the instances for both l and r are used, and *combine* is used to join them.

When a product constructor $l :*: r$ is encountered, we decide whether to choose the instance for the left or right subtree based on the size of the left subtree. If *index* < *offset* + Size l, use the left subtree's instance, else use the right's. When traversing to the left, the offset is kept. When traversing to the right, the offset is incremented by the size of the left subtree. This conditional behaviour is implemented using the second class *GProductHasPosition*.

```
class GProductHasPosition (offset :: Nat)
  (i :: Nat)
  l r a
  (left :: Bool) | offset i l r left → a where
  gproductPosition :: Lens ((l :*: r) x) a
```

The desired field is known to be found when a field node is reached and the index matches the offset.

4.3 *typed*

Accessing a field by its type can be done very similarly to the nominal accessor. We apply the same technique for branching at products as before. One slight modification is that we require the type to be unique inside the construction. In order to achieve this, instead of simply checking whether the tree contains the type, we count the number of times it does so. We proceed only if the whole structure contains exactly one occurrence. This is implemented by the *ErrorUnlessOne* and *CountTotalType* type families.

```
instance
  (Generic s
  , ErrorUnlessOne a s (CountTotalType a (Rep s))
  , GHasType (Rep s) a
  ) ⇒ HasType a s where
  typed = repIso ∘ gtyped
```

4.4 *super*

Given two record types A and B, we say that A is a structural supertype of B (or B is a structural subtype of A), if the set of named fields of A is a subset of that of B, with the corresponding fields being at the same type.

We recall the example from section 2.

```
data AppConfiguration
  = AppConfiguration
    { dbCredentials :: Credentials
    , database  :: String
    , host      :: Host
    , sessionId :: Int
    }
data DbConfiguration
  = DbConfiguration
    { dbCredentials :: Credentials
    , database  :: String
    , sessionId :: Int
    }
```

AppConfiguration is a subtype of *DbConfiguration*, because it contains all of *DbConfiguration*'s fields. The field names serve as tags for their associated types, making the subtype relationship canonical, as field names are unique within a record

Notice that the ordering of fields doesn't matter: records can be thought of as ordinary product types, and as such we can freely associate and commute the fields to create isomorphic definitions.

Using these properties, our *AppConfiguration* example can be rearranged to the isomorphic form, whereby the connection between the structural subtype relationship and lenses becomes more apparent:

```
data AppConfiguration' = AppConfiguration'
  { super :: DbConfiguration
  , host  :: Host
  }
```

Now, a lens focusing on the supertype, *DbConfiguration*, becomes just a traditional record field lens. In general, we can always construct a lens that abstract away this notion of rearrangement to the right shape.

As described previously, a lens consists of two components: a getter $s \to a$, and a setter $a \to s \to s$.

The getter, in the context of subtyping, is called upcasting, because it extracts the more general (supertype) structure out of the

more specific (subtype) structure. The setter is a means of plugging the supertype into the subtype, by overwriting the fields in the latter.

These can be formulated as a lens, that focuses on any supertype of a record, and allows modification in terms of that supertype.

The corresponding type class can thus be written as follow.

```
class Subtype sub sup where
  super :: Lens sub sup
```

Wiring this lens up directly turns out to be rather difficult. Instead, we tackle the problem by deriving the getter and setter functions separately.

We begin by defining a type class for upcasting values from the subtype's generic representation to that of the supertype.

```
class GUpcast (sub :: * → *)
              (sup :: * → *) where
  gupcast :: sub x → sup x
```

The "subtype of" relation induces a partial ordering on the set of Haskell types. The following instance follows from the product order $sub \leqslant a \wedge sub \leqslant b \iff sub \leqslant a \times b$. That is to say, if sub is a subtype of both a and b, then it is a subtype of their product, $a :*: b$.

```
instance (GUpcast sub a, GUpcast sub b)
         ⇒ GUpcast sub (a :*: b) where
  gupcast rep = gupcast rep :*: gupcast rep
```

Much like in the previous cases, the metadata nodes for the constructors and the datatype itself don't contain useful information for the purpose of this class. We simply recurse and wrap up the result to the appropriate metadata node. We omit these definitions for brevity.

Our base case: the record with only one field is the supertype of all records that contain that field name with the same type. Upcasting is as simple as selecting the relevant field, essentially dropping all the other ones.

```
instance
  GHasField field sub t
  ⇒ GUpcast sub
    (S1 ('MetaSel ('Just field) p f b) (Rec0 t))
    where
  gupcast r = M1 (K1 (view (gfield @field) r))
```

The setter, which we call $smash$, can be defined similarly. Here, we plug the structurally smaller supertype sup into the subtype sub.

```
class GSmash sub sup where
  gsmash :: sup p → sub p → sub p

instance (GSmash a sup, GSmash b sup)
         ⇒ GSmash (a :*: b) sup where
  gsmash rep (a :*: b) = gsmash rep a :*: gsmash rep b
```

We have two cases: a field that is present in the subtype is either present in the supertype too, or it isn't. When it is, we want to replace the field with that from sup. Otherwise, we keep the existing field from sub.

As before, we create a new type class to represent the two branches of this choice.

```
class GSmashLeaf sub sup (w :: Bool) where
  gsmashLeaf :: sup p → sub p → sub p
```

The case where we have it in sup, we pick out the field using the relevant $GHasField$.

```
instance GHasField field sup t
         ⇒ GSmashLeaf
           (S1 ('MetaSel ('Just field) p f b) (Rec0 t))
           sup 'True where
  gsmashLeaf sup _ = M1 (K1 (view (gfield @field) sup)
```

Otherwise, we keep the existing value.

```
instance GSmashLeaf
           (S1 ('MetaSel ('Just field) p f b) (Rec0 t))
           sup 'False where
  gsmashLeaf _ = id
```

And finally, we make the decision which instance to pick by checking if sup contains the *field* with our previously defined type family, $Contains$.

```
instance
  (leaf ~ (S1 ('MetaSel ('Just field) p f b) t)
  , GSmashLeaf leaf sup (Contains field sup)
  ) ⇒ GSmash (S1 ('MetaSel ('Just field) p f b) t) sup
  where
  gsmash = gsmashLeaf @_ @_ @(HasTotalFieldP field sup)
```

With the getter ($upcast$) and setter ($smash$) now available, we are finally in a position where we can define the $super$ lens:

```
super :: ∀sup sub.Subtype sub sup ⇒ Lens sub sup
super f sub = fmap (flip smash sub) (f (upcast sub))
```

5. Evaluation

When working generically we must always ask whether the abstraction comes at the cost of performance. In this case, it is pleasing that our use of generics is entirely optimised away by the compiler. We consider several examples and provide formatted core dumps to demonstrate that the generic operations are compiled to idiomatic code.

Highly polymorphic code must always be eventually instantiated with a concrete monomorphic site when it is finally consumed in a productive way. At this point of instantiating, we can concretely solve the class constraints and then by performing *inlining* and *specialisation* remove the cost of the abstraction.

There are two crucial reasons why we can be confident without inspecting the core that GHC will produce efficient code.

1. Lenses are not defined recursively, as such definitions can be aggressively inlined.
2. Arguments as passed as types, we statically know them at compile time and hence can be eliminated by the specialiser.

We first recall the two optimisation techniques which will allow our lenses to be optimised efficiently: inlining and specialisation.

5.1 Inlining

Inlining is the most crucial optimisation in the compiler's pipeline as it enables all other optimisations to occur. Inlining is the process of replacing a function's name by its definition. In a pure language like Haskell, this can be done very aggressively as it is always safe.

However, whilst always safe, we must still be careful about when we inline. If we inline too little then we miss optimisation opportunities. If we inline too much then the size of our program becomes very large and takes a long time to compile. The compiler contains a set of balanced heuristics which take into account factors such as the definition's size, whether the call is saturated, whether the function is recursive and so on to decide whether it should inline a definition or not.

5.2 Specialisation

Specialisation is the process of eliminating dictionary passing introduced by type class constraints. When a type class is desugared without optimisation, each class constraint becomes a new dictionary argument to the function, class methods are then looked up in this dictionary.

$pprint :: Show\ a \Rightarrow a \rightarrow String$
$pprint\ n = \texttt{"Val: "} +\!\!+ show\ n$

is desugared to (omitting type arguments) in idealised core:

```
data DictShow a = DictShow { show :: a -> String }

pprint :: DictShow a -> a -> String
pprint ds a = "Val: " ++ show ds a
```

If we later call *pprint* at a known type, for example *Int*, then the compiler will also supply the *Show* dictionary for *Int*.

$printInt :: Int \rightarrow String$
$printInt\ n = pprint\ n$

desugars to

```
dInt :: DictShow Int
dInt = DictShow { show = .. }

printInt :: Int -> String
printInt n = pprint dInt n
```

In this case, we would expect the compiler to inline *pprint* and hence eliminate the dictionary argument but in general a function which accepts a dictionary argument might not be desirable or able to be inlined. But, it is extremely beneficial to eliminate statically known dictionary arguments to remove indirection. This is what the process of specialisation achieves.

The specialiser looks for calls to overloaded functions called at a known type. It then creates a new type specialised definition which doesn't take a dictionary argument and a rewrite rule which rewrites the old version to the new version.

$foo :: Show\ a \Rightarrow a \rightarrow a \rightarrow Bool$
$foo\ x\ y = show\ x \equiv show\ y$

This source definition desugars to:

```
foo = \ @a $dShow x y ->
    eqString (show $dShow x) (show $dShow y)
```

There are now 4 parameters to foo, the first argument is a type (denoted by `@`), the second argument is the dictionary for Show (denoted by the `$d` prefix) and the last two are the arguments `x` and `y` of the original definition.

The class constraint is translated into a dictionary. Each time a class method is used, we must dynamically lookup which definition to use in the supplied class dictionary.

If we know which type a is instantiated with, we can specialise the definition of *foo* and produce much better code.

$qux :: Bool \rightarrow Bool \rightarrow Bool$
$qux = foo\ @Bool$

Using foo at a specific type produces a new definition `foo_sfoo` which is defined as:

```
foo_sfoo :: Bool -> Bool -> Bool
foo_sfoo = foo @Bool $fShowBool
```

Further optimisations then inline foo and then the dictionary selector show which produces the following more direct program.

```
foo_sfoo =
  \ x y ->
    case x of {
      False ->
        case y of {
          False -> foo4;
          True -> foo3
        };
      True ->
        case y of _ {
          False -> foo2;
          True -> foo1
        }
    }
```

5.3 field

Recall that the *field* function creates a lens for a named field. The name is provided by providing a type literal string. We will first inspect how the lens performs whilst getting and setting before more generally considering the specialisation without applying any operators.

$viewfield :: AppConfiguration \rightarrow String$
$viewfield = view\ (field\ @\texttt{"dbCredentials"})$

This definition produces the following optimised core: [4]

```
viewfield =
  \(x :: AppConfiguration) ->
      case x of
        AppConfiguration g1 g2 g3 g4 -> g1
```

As can be seen, there is no overhead from the abstraction – the resulting output is identical to what a handwritten getter function would look like.

We consider this one example in depth as indicative of how the optimiser treats our definitions. *view* is defined as follows:

$view :: Lens\ s\ a \rightarrow s \rightarrow a$
$view\ l = getConst \circ l\ Const$

Thus applying *view* to a lens l will specialise the functor instance which l uses to *Const*.

field is a method of the *HasField* type class, as such when specifying which field we wish to view for which type we can perform specialisation.

Using the flag `-ddump-spec` we can see the specialisation which is created.

```
"SPEC/Viewfield field
        @ "dbCredentials"
        @ Credentials
        @ AppConfiguration"
  forall (tpl_X2Gj ::
            HasField
              "dbCredentials"
              Credentials
              AppConfiguration).

    field
      @ "dbCredentials"
      @ Credentials
      @ AppConfiguration tpl_X2Gj
      = $sfield_s2Bc
```

Inspecting the single instance for *field* we find that it is implemented in terms of *gfield* which is like *field* but for the generic

[4] As with all core examples, it has been manually formatted for clarity.

structure. As such, it is defined by recursion on the structure of generic types with an instance for each different type constructor which is used to construct a generic representation. At compile time this means that

```
gfield
  @"dbCredentials"
  @(Rep AppConfiguration)
  @Credentials
```

can be completely unrolled by repeated recursive specialisation of *gfield*. In each definition where *gfield* is called, it is specialised at a new type and hence a tree of non-recursive definitions are created.

The generic representation of *AppConfiguration* is defined as: [5]

```
instance Generic AppConfiguration where
  type Rep AppConfiguration =
    C1 ('MetaCons "AppConfiguration" PrefixI'True)
      ((S1 ('MetaSel ('Just "dbCredentials"))
           (Rec0 Credentials)
       :*: S1 ('MetaSel ('Just "database"))
           (Rec0 String))
       :*: (S1 ('MetaSel ('Just "host"))
           (Rec0 Host)
       :*: S1 ('MetaSel ('Just "sessionId"))
           (Rec0 Int)))
  from :: AppConfiguration → Rep AppConfiguration
  from (AppConfiguration g1 g2 g3 g4) =
    (g1 :*: g2) :*: (g3 :*: g4)
  to :: Rep AppConfiguration → AppConfiguration
  to ((g1 :*: g2) :*: (g3 :*: g4)) =
    AppConfiguration g1 g2 g3 g4
```

Calling

```
gfield
  @"dbCredentials"
  @(Rep AppConfiguration)
  @Credentials
```

first matches the instance for *M1 C c s* which calls *gfield* on the product constructed by :*: and so on. At each stage we perform specialisation such that ultimately we reach a definition which contains no calls to *gfield* and only calls to *fmap* due to the instance definitions of *gfield*.

```
$sfield_s2Bc =
  \ $dFunctor f ->
    let f1 = fmap $dFunctor $cto_a278 in
    \s ->
      f1 (fmap
        $dFunctor
        lensM1
        (fmap $dFunctor
           lensM1
           (case s of _
              AppConfiguration g1 g2 g3 g4 ->
                let {b1 = g3 :*: g4 } in
                fmap $dFunctor
                 (\ ds1 -> ds1 :*: b1)
                  (fmap $dFunctor
                    (\ ds2 -> ds2 :*: g2)
                    (fmap $dFunctor
                      lensM1
                      (fmap $dFunctor
                        $fGHasFieldfieldK1a1
                          (f g1))))
})))
```

Each argument to *fmap* rebuilds the generic structure of *AppConfiguration*, at the lowest level we apply he *K1* constructor followed by *M1* before reconstructing the product type and then finally applying *M1* twice to get back to the original representation.

This specialisation alone still contains the generic representation. It is not until we choose our functor, by using *view* that we can eliminate the representation. `$sfield_s2Bc` is then specialised further in the application of *view*. By choosing which instance of functor we want to use and hence providing the argument `$dFunctor` to `$sfield_s2Bc` we can select *fmap* at each call site and collapse all the levels together appropriately.

We will see how to collapse these nested *fmap*s without instantiating *f* concretely in section 5.4.

It is important to note that because `GHC.Generics` only encodes one level of the recursive structure rather than a complete unrolling, the conversion functions *to* and *from* are not recursive and can hence be inlined.

We see much the same in the case of setting. This time the functor in question is *Identity*.

```
setfield
  :: Credentials → AppConfiguration → AppConfiguration
setfield = set (field @"dbCredentials")

set :: Lens s a → a → s → a
set l b = runIdentity ∘ l (const (Identity b))
```

The functor instance for *Const* is trivial in the sense it completely ignores its argument. Thus we should also inspect how the compiler optimises *set* which uses the *Identity* functor.

```
setfield =
  \ (b :: Credentials) (wild :: AppConfiguration) ->
      case wild of
        AppConfiguration g1 g2 g3 g4 ->
          AppConfiguration b g2 g3 g4
```

We see again that the produced core is precisely the same as writing the definition by hand. We have achieved abstraction but without the cost.

5.4 The Lens itself

In the previous section the specialised version of the lens looked quite different to the one which we would write by hand. There were many nested calls to *fmap* which rebuilt the structure iteratively.

```
handwritten :: Lens AppConfiguration Credentials
handwritten f (AppConfiguration creds db host sid) =
  fmap (λc → AppConfiguration c db host sid) (f creds)
```

However, we observe that if we apply the identity *fmap f* ∘ *fmap g* = *fmap* (*f* ∘ *g*) in the definition of `$sfield_s2Bc` we can gather together the intermediate definitions and the optimiser will fuse them together by inlining to generate identical core to the handwritten version.

The definition we will be inspecting is:

```
testSpec :: Lens AppConfiguration Credentials
testSpec = field @"dbCredentials"
```

Notice that we have also fixed the fact that the field `"dbCredentials"` is from *AppConfiguration*. This is the same definition as was created by the specialiser in the previous section.

[5] Strictness information and unpackedness information is omitted from the type for brevity. The values have been simplified by omitting the metadata nodes.

In order to do this we need to instantiate f to a suitable functor F such that $\forall f.f\ a{\sim}F\ a$ but also that we can observe by simple compiler transformations that $fmap\ f \circ fmap\ g = fmap\ (f \circ g)$. Combining these two facts together, we can instantiate f to F, allow the compiler to optimise the nested *fmap* calls and finally undo the transformation to recover f.

The functor which allows us to do this is typically known as *Coyoneda*.

$$\textbf{data}\ Coyoneda\ f\ b = \forall a.Coyoneda\ (a \to b)\ (f\ a)$$

$$\textbf{instance}\ Functor\ (Coyoneda\ f)\ \textbf{where}$$
$$\quad fmap\ f\ (Coyoneda\ g\ fa) = Coyoneda\ (f \circ g)\ fa$$

$$inj :: Functor\ f \Rightarrow Coyoneda\ f\ a \to f\ a$$
$$inj\ (Coyoneda\ f\ a) = fmap\ f\ a$$

$$proj :: Functor\ f \Rightarrow f\ a \to Coyoneda\ f\ a$$
$$proj\ fa = Coyoneda\ id\ fa$$

The functor instance for *Coyoneda* accumulates the mapped functions, as such repeatedly mapping functions can be fused together before they are applied in a single *fmap* at the end.

```
  fmap f (fmap g (Coyoneda h fa)
=
  fmap f (Coyoneda (g . h) fa)
=
  Coyoneda (f . g . h) fa
=
  fmap (f . g) (Coyoneda h fa)
```

We call the operation which fuses together multiple *fmap*s in the definition of a lens as *ravel* due to the intuition that we untangling the nested *fmap* calls from the juicy morsels we have caught in our net.

$$ravel$$
$$\quad :: Functor\ f$$
$$\quad \Rightarrow ((a \to Coyoneda\ f\ b)$$
$$\quad \to (s \to Coyoneda\ f\ t))$$
$$\quad \to (a \to f\ b) \to s \to f\ t$$
$$ravel\ coy\ f\ s = inj\ \$\ coy\ (\lambda a \to proj\ (f\ a))\ s$$

$$testSpecFinal$$
$$\quad :: Functor\ f$$
$$\quad \Rightarrow (Credentials \to f\ Credentials)$$
$$\quad \to AppConfiguration \to f\ AppConfiguration$$
$$testSpecFinal = ravel\ (field\ @\texttt{"dbCredentials"})$$

Inspecting the core for *testSpecFinal* we observe that common sub-expression elimination identifies the two definitions as expected.

5.5 super

The *super* lens seems quite a bit more complicated than the previously discussed *field*. *super* must first deconstruct a datatype and then reconstruct a new different datatype, it isn't as clear that the optimiser will be able to perform as well.

We consider the case of using *set* in order to update an *AppConfiguration* by a *DbConfiguration*.

$$setDbConfiguration$$
$$\quad :: DbConfiguration$$
$$\quad \to AppConfiguration$$
$$\quad \to AppConfiguration$$

$$setDbConfiguration =$$
$$\quad set\ (super\ @DbConfiguration)$$

which we compare to a handwritten version which manually performs the copying.

$$setDbConfigurationHandwritten$$
$$\quad :: DbConfiguration$$
$$\quad \to AppConfiguration$$
$$\quad \to AppConfiguration$$

$$setDbConfigurationHandwritten$$
$$\quad (DbConfiguration\ a1\ a2\ a3)$$
$$\quad (AppConfiguration\ \_\ \_\ h3\ \_) =$$
$$\quad AppConfiguration\ a1\ a2\ h3\ a3$$

Inspecting the core we find that our two definitions are *not* equivalent to each other and the reason why points to a further benefit of our abstraction. *setDbConfiguration* creates a lazier definition than *setDbConfigurationHandwritten*. In *setDbConfigurationHandwritten* we needlessly force the first argument to whnf. If we later try to inspect the *host* field then the difference becomes apparent.

$$> host\ (setDbConfigurationHandwritten\ \bot$$
$$\quad (AppConfiguration$$
$$\quad\quad (Credentials\ \texttt{"a"}\ \texttt{"b"})$$
$$\quad\quad \texttt{"db"}$$
$$\quad\quad (Host\ \texttt{"http://google.com"}\ 8080)$$
$$\quad\quad 14))$$
$$\bot$$
$$> host\ (setDbConfiguration\ \bot$$
$$\quad (AppConfiguration$$
$$\quad\quad (Credentials\ \texttt{"a"}\ \texttt{"b"})$$
$$\quad\quad \texttt{"db"}$$
$$\quad\quad (Host\ \texttt{"http://google.com"}\ 8080)$$
$$\quad\quad 14))$$
$$Host\ \texttt{"http://google.com"}\ 8080$$

Instead a semantically equivalent way to write the setter is:

$$setDbConfigurationHandwritten'$$
$$\quad :: DbConfiguration$$
$$\quad \to AppConfiguration$$
$$\quad \to AppConfiguration$$

$$setDbConfigurationHandwritten'$$
$$\quad {\sim}(DbConfiguration\ a1\ a2\ a3)$$
$$\quad (AppConfiguration\ \_\ \_\ h3\ \_) =$$
$$\quad AppConfiguration\ a1\ a2\ h3\ a3$$

Where we use a tilde to indicate that the pattern *DbConfiguration* is irrefutable as it is the only data constructor for *DbConfiguration* and thus we can match upon it lazily.

This definition is indeed identified with *setDbConfiguration* by the compiler.

5.6 upcast

Finally we consider *upcast* which extracts a smaller sub-record from a record. We can use it to extract the fields which define a *DbConfiguration* from a *AppConfiguration* discarding the *host* field.

$$upcastAppConfiguration$$
$$\quad :: AppConfiguration$$
$$\quad \to DbConfiguration$$
$$upcastAppConfiguration = upcast$$

$$upcastHandwritten$$
$$\quad :: AppConfiguration$$
$$\quad \to DbConfiguration$$
$$upcastHandwritten$$
$$\quad (AppConfiguration\ h1\ h2\ \_h3\ h4) =$$
$$\quad DbConfiguration\ h1\ h2\ h4$$

We also inspect the core produced by the high level generic upcast and low-level handwritten version.

```
upcastAppConfiguration =
  λh →
    DbConfiguration
      (case h of
        AppConfiguration g1 g2 g3 g4 → g1)
      (case h of
        AppConfiguration g1 g2 g3 g4 → g2)
      (case h of
        AppConfiguration g1 g2 g3 g4 → g4)
```

We note again that the derived version is lazier than the naive handwritten version. The naive version forces the argument to whnf.

```
isDbConfiguration :: DbConfiguration → Bool
isDbConfiguration (DbConfiguration { }) = True
> isDbConfiguration (upcastAppConfiguration ⊥)
True
> isDbConfiguration (upcastHandwritten ⊥)
⊥
```

In either case, h is only evaluated once.

6. Future Work

There are several possible extensions to this work which would be exciting to explore in future work.

6.1 Prisms

Prisms are the dual to a lens. A prism witnesses the coproduct nature of a datatype and as such we can define similar operations to generically derive prisms as we did for lenses.

The two canonical operations which define a prism are *match* and *build*.

```
match :: Prism s a → s → Maybe a
build :: Prism s a → a → s
```

For a simple coproduct we again might wish to define prisms positionally, nominally or structurally.

```
data T = A Int | B String
> build (con @"A") 5
A 5
> build (con @A) 5
A 5
> build (nth @1) "ABC"
B "ABC"
> build (inj @T) "ABC"
B "ABC"
```

where *con*, *nth* and *inj* are our prisms would operate nominally, positionally and structurally.

6.2 Anonymous Records Libraries

There are a large number of different libraries which provide support for anonymous record types. That is, anonymous products which have named fields. They provide a lightweight way way to provide additional ad-hoc structure to data without defining many different datatypes.

We speculate that by defining a handwritten `GHC.Generics` instance that we could use one of these libraries in order to provide a lightweight method to update a set of record fields.

For example, supposing that we defined an instance for the datatype found in the `Vinyl` library we might able to write: (Sterling 2017)

```
data Person = Person { name :: String, age :: Int }
updatePerson :: Int → Person → Person
updatePerson n = set (super @Person)
                     (SAge =:: n :& RNil)
```

In such a situation, one might ask why we don't use the built-in record update syntax that Haskell provides. The reason is that it is not compositional, whereas programming directly with these lenses is. More complicated updating operations could also be composed more directly by combining together different anonymous records.

6.3 A super pattern synonym

Using pattern synonyms (Pickering et al. 2016), we could reuse our *super* lens using ordinary record syntax. In particular, record pattern synonyms allow us to define virtual field fields which can be use to access and set virtual properties of record datatypes.

```
updateWith
  :: DbConfiguration
  → AppConfiguration
  → AppConfiguration
updateWith db app = app { super = db }
```

Using *super* in this manner might be more convenient for programmers who are not used to programming with lenses.

7. Conclusion

Deriving lenses generically gives the programmer the best of all possible worlds. The frugality to only define whichever lenses they need to use, the confidence that their abstraction will be without cost and the flexibility to four different types of lenses. This expressive and lightweight solution will hopefully inspire other library writers to embrace `GHC.Generics` as a solid basis on which to build their libraries.

References

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *ACM SIGPLAN Notices*, volume 49, pages 671–683. ACM, 2014.

R. A. Eisenberg, S. Weirich, and H. G. Ahmed. Visible type application. In *European Symposium on Programming Languages and Systems*, pages 229–254. Springer Berlin Heidelberg, 2016.

R. Hinze. Generics for the masses. In *Principles of Programming Languages*, volume 39, pages 236–243. ACM Press, 2004. doi: 10.1145/1016848.1016882.

M. Jaskelioff and R. O'Connor. A Representation Theorem for Second-Order Functionals. *Journal of Functional Programming*, 25(e13), 2015. doi: 10.1017/S0956796815000088.

E. Kmett. `lens-4.15.3` library, 2017. URL `https://hackage.haskell.org/package/lens-4.15.3`.

J. P. Magalhaes, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Haskell Symposium*, pages 37–48. ACM Press, 2010.

K. Matsuda and M. Wang. Applicative bidirectional programming with lenses. In *International Conference on Functional Programming*, pages 62–74, 2015. doi: 10.1145/2784731.2784750.

R. O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR*, abs/1103.2841, 2011. doi: arXiv:1103.2841. Presented at WGP 2011.

M. Pickering, G. Érdi, S. Peyton Jones, and R. A. Eisenberg. Pattern Synonyms. In *Haskell Symposium*, pages 80–91. ACM Press, 2016. doi: 10.1145/2976002.2976013.

M. Pickering, J. Gibbons, and N. Wu. Profunctor Optics: Modular Data Accessors. *CoRR*, abs/1703.10857, 2017. URL `http://arxiv.org/abs/1703.10857`.

J. Sterling. `vinyl-0.5.3` library, 2017. URL `https://hackage.haskell.org/package/vinyl-0.5.3`.

J. Stolarek, S. Peyton Jones, and R. A. Eisenberg. Injective type families for Haskell. In *ACM SIGPLAN Notices*, volume 50, pages 118–128. ACM, 2015.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103795. URL `http://doi.acm.org/10.1145/2103786.2103795`.

Scrap Your Reprinter

A Datatype Generic Algorithm for Layout-Preserving Refactoring

Harry Clarke
School of Computing, University of Kent
hc306@kent.ac.uk

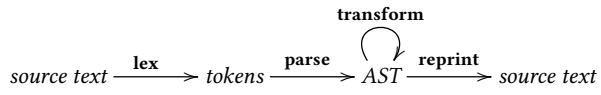
Dominic Orchard
School of Computing University of Kent
d.a.orchard@kent.ac.uk

Abstract

Refactoring tools are extremely useful in software development: they provide automatic source code transformation tools for a variety of laborious tasks like renaming, formatting, standardisation, modernisation, and modularisation. A refactoring tool transforms part of a code base and leaves everything else untouched, including secondary notation like whitespace and comments. We describe a novel datatype generic algorithm for the last step of a refactoring tool – the *reprinter* – which takes a syntax tree, the original source file, and produces an updated source file which preserves secondary notation in the untransformed parts of the code. As a result of being datatype generic, the algorithm does not need changing even if the underlying syntax tree definition changes. We impose only modest preconditions on this underlying data type. The algorithm builds on existing work on datatype generic programming and zippers. This is useful technology for constructing refactoring tools as well as IDEs, interactive theorem provers, and verification and specification tools.

1 Introduction

Refactoring tools act a bit like compilers: they read in source code and convert it into a machine representation upon which some transformations are performed. However, they differ from compilers in that they output textual source code in the language that was originally input. To be of any use, they must preserve the lexical style of all untransformed code, such as comments and whitespace (*secondary notation* or *documentary structure* [13]). Compilers on the other hand are free to discard this information, since it is irrelevant for generating the final binary output. We refer to generation of the output source code in a refactoring tool as the *reprinter*– the last step where an AST is converted back into source text:



As a simple running example, consider an SSA-like language with variables, integer addition in prefix notation, and constants. The following is an example program in our source language:

```
x = +(1,2)
y  =  +(x, 0)
// Calculate z
z  =  +( 1,  +(+(0,x) ,y) )
```

Note that this example deliberately uses varying amount of white space around the assignments and operations.

Imagine a refactoring that removes redundant additions of 0, *i.e.* performs rewrites `+(`e`, 0)` $\rightsquigarrow e$ and `+(0,` e`)` $\rightsquigarrow e$ for some expression e. The refactoring and reprinting should produce the following output source text, preserving whitespace and comments:

```
x = +(1,2)
y  =  x
// Calculate z
z  =  +( 1,  +(x ,y) )
```

An implementation cannot simply pretty print the transformed AST as this would destroy the secondary notation.

One solution is to store as much of the secondary notation as possible in the AST (either via specialised nodes or annotations) to provide a layout-preserving pretty printing (see, *e.g.* [6, 7, 14]). However, this requires a large engineering effort, does not integrate well with many front-end generation tools (for example, most lexers readily throw away additional whitespace), and is extremely difficult when multi-line comments are mixed with multi-line syntactic elements (see discussion by de Jonge *et al.* [3]). Another solution uses *text patching*, where AST changes are converted to edit instructions on the source code (*e.g.*, *diffs*), coupled with heuristics about layout adjustments [3].

We propose a new, simple approach that is language agnostic. Our reprinting algorithm combines an updated AST with the original source text to produce an updated source file. We detail a datatype generic implementation which can be applied to any algebraic data type satisfying a minimal interface for providing source code locations. Genericity means the algorithm does not need reimplementing even if the underlying AST datatypes are changed or extended. Furthermore, our reprinting algorithm has the advantage that an implementer need not write a pretty printer for all parts of their syntax tree, only for those parts which might get refactored, requiring fresh source text generation.

Our implementation is based on the datatype generic facilities provided in GHC (the Glasgow Haskell Compiler) [8, 9] and a datatype generic zipper construct [1] for simplifying the datatype generic traversal.

Reprinting is useful not just in refactoring tools but also in IDEs (for example, with live macro expansions), interactive theorem provers, or specification systems where a tool might automatically generate specifications into an existing code base. Our algorithm is used in the CamFort tool which provides refactoring of Fortran code [11] and several verification features which use the reprinting algorithm to insert inferred specifications into source code [2].

Roadmap We start with an informal overview of the algorithm (Section 1.1). Section 2 provides some background on zippers and

IFL'17, 30th August-1st September 2017, Bristol, UK

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

datatype generic programming. Section 3 outlines the algorithm in detail, including its GHC/Haskell implementation. Section 4 shows that reprinting and parsing form a bidirectional lens. Section 5 concludes with some discussion of the implications of our preconditions and some further work.

Our code is available as a package (see http://hackage.haskell.org/package/reprinter), which includes the examples used here.

1.1 Illustrated example

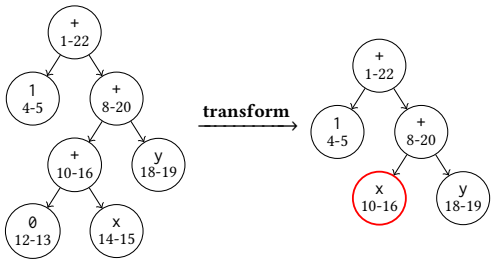
Consider the expression from the last line of the preceding example, which is refactored by removing redundant additions of 0:

```
+( 1,   +(+(0,x) ,y) )  ——transform & reprint——→  +( 1,   +(x ,y) )
```

To make the whitespace preservation here clear, the following shows the column number for the source text before and after transformation and reprinting:

col #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
pre	+	(		1	,			+	(	+	(	0	,	x	)		,	y	)		)
post	+	(		1	,			+	(	x		,	y	)		)					

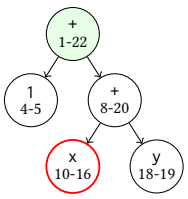
In terms of the abstract syntax trees, the refactoring is represented as follows, where each AST node is annotated with the *source code span*: a pair of column numbers marking the extent of the AST node's origin in the original source text.



The refactored node on the right is lined in red and notably has the original source span preserved. For the full algorithm, this source code span also includes the line numbers, but for simplicity we elide this here since the example is located on a single line.

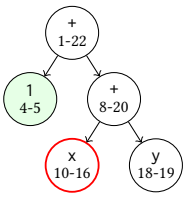
The algorithm performs a depth-first traversal of the AST and applies a pretty printing on any refactored nodes. In the actual implementation (Section 3), the reprinting algorithm is parameterised by a generic function which we call a *reprinting*, which may provide pretty printing for various different types of node.

In the illustration here, we mark the currently visited node in light green. The depth-first traversal of the AST simultaneously "traverses" the input text, which is statefully consumed. At each step we record the state, which comprises the remaining input text and a source code position (called the *cursor*) which signifies our position in the original source text. We also show the partially-built output source code, which grows as we traverse the tree.



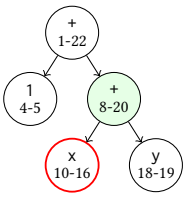
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since the root node is not refactored, the algorithm proceeds to the first child (actions we refer to as **down** and **enter**):



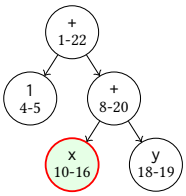
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since the current node is not refactored and it has no children, we proceed to the next sibling (**right** and **enter** actions):



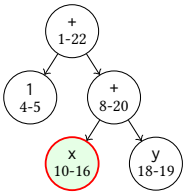
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since this node is not refactored, we proceed to its first child (**down** and **enter** again):



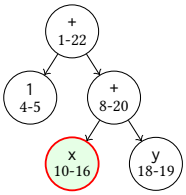
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since this node is marked as refactored, we take the substring of the input source code from the cursor to the lower-bound position of this node (column 10) and add it to the output source code, consuming this part of the input source. The cursor is also updated to the lower bound of the node:



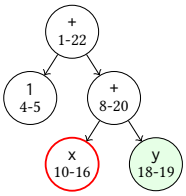
cursor = 10
input = `"+(0,x) ,y) )"`
output = `"+( 1,   +("`

Next, since the node is being refactored, we apply a pretty printing algorithm to generate a fresh piece of source text for this node which is then added to the output text. We also delete the portion of the input source text between the lower and upper bounds of the node, and update the cursor to the upper bound (column 16):



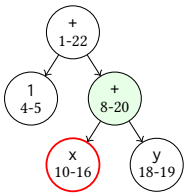
cursor = 16
input = `" ,y) )"`
output = `"+( 1,   +(x"`

This node is now processed so we move to the next right sibling:



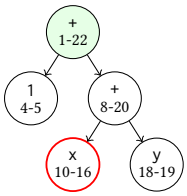
cursor = 16
input = `" ,y) )"`
output = `"+( 1,   +(x"`

This node is not refactored and it has no children or siblings so we return to the parent node:



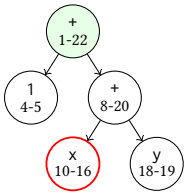
cursor = 16
input = `" ,y) )"`
output = `"+( 1,  +(x"`

This node has no siblings so we return to the parent node:



cursor = 16
input = `" ,y) )"`
output = `"+( 1,  +(x"`

This node has no siblings and no parent. On returning to the root, we append to the output text the remaining input source text:



cursor = 22
input = `""`
output = `"+( 1, +(x ,y) )"`

This concludes the algorithm and we now have the reprinted output, preserving all the whitespace in the non-transformed syntax.

Section 3 gives the implementation of this algorithm in detail. First we provide background on "zippers", the underlying data structure used to implement the tree traversal described above.

2 Background

A zipper is an alternate representation of a datatype that allows subparts of a piece of data to be focussed upon, while preserving the rest of the datatype [5]. The *focus* can then be shifted around the datatype, maintaining the rest of the data structure– the *context* of the focus. The focus is shifted by navigation operations. Our algorithm is defined in terms of a generic zipper construction (Scrap Your Zipper [1]) to provide datatype generic traversal.

2.1 Standard Zippers

Zippers make traversing and transforming a data structure easier, as well as providing more efficient operations, such as constant-time updates. Suppose you had the following list:

```
let list = [1, 2, 3]
```

Without a zipper, a simple recursive function, traversing the list by deconstructing it one constructor at a time, will lose information about previous elements. Zippers retain this information as part of the data type should we need it, either for reconstruction of the original data, or for contextual operations.

A zipper is made up of two parts:

- Focus - This is the part of the data structure that we are currently viewing.
- Context - This represents the rest of our data type not already contained in the focus. To represent this efficiently, nodes are represented by their siblings and their parent, so the tree's direction is reversed within the context, with a path pointing all the way up to the root node.

A list zipper can be defined like so:

```
type ListZipper a = (ListFocus a, ListContext a)

type ListFocus a = [a]
data ListContext a
  = Root | Node a (ListContext a)

toListZipper :: [a] → ListZipper a
toListZipper l = (l, Root)
```

When we first create a zipper, the focus represents the entire data structure (or in our case, the entire list), as we are at the root node, and the context is just *Root*.

Navigation operations shift the focus of the zipper. For the list zipper we have:

```
listDown :: ListZipper a → Maybe (ListZipper a)
listDown (x : xs, ys)    = Just (xs, Node x ys)
listDown ([ ], ys)       = Nothing

listUp :: ListZipper a → Maybe (ListZipper a)
listUp (xs, Node y ys) = Just (y : xs, ys)
listUp (xs, Root)      = Nothing
```

Thus, moving "down" a list navigates into the focus, extending the context with the element from the top of the focus. If the focus is empty, then we return Nothing, *i.e.*, there are no children left to move down into. Moving up unfolds the context, extending the focus. Once Root is reached, we are at the top/beginning of the list, and there is nowhere left for us to go.

Thus, for the list [1, 2, 3], we get the following zipper navigation:

([1, 2, 3], Root)
listUp ⟲ ⟳ *listDown*
([2, 3], Node 1 Root)
listUp ⟲ ⟳ *listDown*
([3], Node 2 (Node 1 Root))
listUp ⟲ ⟳ *listDown*
([ ], Node 3 (Node 2 (Node 1 Root)))

Zippers on trees have a similar structure. Huet [5] shows how a tree can be converted into a tree zipper, where the context contains the parent, and the siblings on the left and right of the focus:

```
data Tree a = Item a | Section [Tree a]

type TreeZipper a = (TreeFocus a, TreeContext a)
type TreeFocus a = Tree a
data TreeContext a
  = TRoot | TNode (TreeContext a) [Tree a] [Tree a]
```

A zipper representation of a datatype can be calculated from the algebraic representation of a datatype via Leibniz differentiation [10]. This implies that zippers can be automatically generated for any datatype. This enables generic zippers.

2.2 Generic zippers

Scrap Your Zipper (SYZ) [1] is a library that allows the programmer to traverse any tree structure with ease by converting a datatype

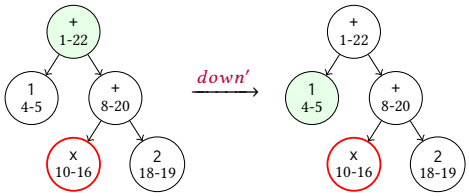
into its zipper representation. The library provides various operations for navigation, insertion, and transformation of zippers. The only requirement is that all parts of the datatype have an instance of Scrap Your Boilerplate's [8] *Data* class, providing datatype generic functionality. This makes it an excellent starting point for generic AST traversal and subsequently, reprinting.

We highlight functions of SYZ in purple to make it clear where they are used in the algorithm.

A function *toZipper* :: Data $a \Rightarrow a \to$ Zipper a maps any Data type into the zipper representation. Our algorithm then only needs two of the zipper navigation operations provided by the SYZ library: *down′* and *right*. The *down′* operation moves the focus of the zipper to the leftmost child of the current node, of type:

```
down′ :: Zipper a → Maybe (Zipper a)
```

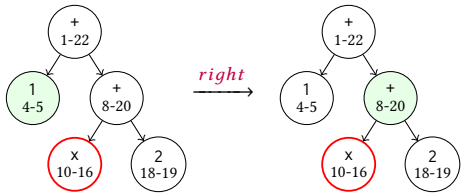
For example, in our illustration the light green highlighted node corresponds to the focus node and *down′* has the following example behaviour (returning a Just value) when the focus is at the root:



We also use *right*, which moves the focus of the zipper to the sibling to the right:

```
right :: Zipper a → Maybe (Zipper a)
```

For example, in terms of our illustration:



The only other additional part of SYZ we use is the application of a generic query to the current focus, provided by the *query* functions:

```
-- Type of a generic query, from Data.Generics.Aliases
-- of Scrap Your Boilerplate
type GenericQ b = ∀c.Data c ⇒ c → b

-- Applies a generic function to a zipper and outputs the result.
query :: GenericQ b → Zipper a → b
```

Thus a function which is defined for all types c satisfying Data to produce a value of type b is applied to the current focus of the zipper to produce a b value.

3 Datatype generic reprinting; implementation

The reprinting algorithm stitches together an AST and a piece of source text into an output source text by a simultaneous traversal of both the AST and input source. The AST is traversed in depth-first order, and the input source is traversed linearly. This relies on the AST containing some information about the origin of its nodes in the original source file, along with some well-formedness conditions of this location information. Note that the datatype's definition for an AST usually spans many mutually recursive datatypes.

Definition 1. An AST is *source coherent* if every refactorable node has a *source span* associated with it, that is a pair of source code positions (*e.g.* line and column number) marking the lower bound and upper bound extent of the node's origin within the original source code. Furthermore, the AST must satisfy the following properties:

1. *Enclosure*: the lower-bound position of a parent is less than or equal to the lower-bound of the first child, and the upper-bound position of a parent is greater than or equal to the upper-bound of the last child.
2. *Sequentiality*: the upper-bound position of a node is less than or equal to the lower-bound position of its right sibling (if there is one).

These two conditions hold for the example here, which is defined in full below; the AST is source coherent. We implicitly restrict ourselves to tree data structures, *i.e.*, data structures with no cycles. The *enclosure* property above goes some way to ruling out cycles, though there may be cycles with the same upper and lower bounds not ruled out by its condition.

We split our description of the algorithm into three parts: the main types (Section 3.1), the core of the algorithm with the top-level function *reprint* and the intermediates *enter*, *enterDown* and *enterRight* (Section 3.2), and finally helpers for building "reprintings" (generic functions parameterising *reprint*) (Section 3.3).

The final section provides an implementation of the introduction code (Section 3.4).

3.1 Types

Let Source be the type of source text, for which we use the efficient ByteString representation:

```
import qualified Data.ByteString.Char8 as B

type Source = B.ByteString
```

Positions in source code are given by pairs of column and line numbers (integers), which are 1-indexed:

```
data Position = Position {posColumn :: Int, posLine :: Int}
  deriving Data

initPosition = Position {posColumn = 1, posLine = 1}
```

The span of a piece of source text can be represented by a pair of positions.

```
type Span = (Position, Position)
```

The top-level function *reprint* has type:

```
reprint :: (Monad m, Data p)
        ⇒ Reprinting m → p → Source → m Source
```

which is a higher-order function parameterised by a "reprinting" function: a generic operation for replacing refactored nodes with fresh output text (*e.g.*, by pretty printing). Reprintings have type:

```
type Reprinting m = ∀b.Typeable b ⇒
  b → m (Maybe (RefactorType, Source, Span))

data RefactorType = Before | After | Replace
```

Thus, a reprinting maps any Typeable type b to Maybe of a triple of the refactoring type, the new source text for this node, and a pair of positions (Span), which represent the lower and upper bound positions of the node. We provide some helper functions for constructing a Reprinting in Section 3.3.

Note that *reprint* and Reprinting are parameterised by some monad *m*. This provides extra power if an application wishes to include some additional side effects as part of the reprint algorithm. For example, additional state can be useful when pretty printing refactored nodes, such as the number of lines deleted or added in order to balance newlines. We show another example in Section 3.4.

3.2 Zipper traversal; core

The zipper traversal part of the algorithm is performed by a trio of functions: *enter*, *enterDown*, and *enterRight*, of type:

```
enter, enterDown, enterRight :: Monad m ⇒
Reprinting m → Zipper p → StateT (Position, Source) m Source
```

where Zipper provides the zipper for some type *p*, provided by the Scrap Your Zipper library [1]. The implementation is shown in Figure 1. Recall that the zipper operations are in *purple*.

The *enterDown* and *enterRight* are respectively used to navigate to a node's children and to its siblings, using the navigation operations of the generic zipper.

The *enter* function provides the main functionality. It firstly applies the generic *reprinting* function (type Reprinting *m*) to the current node, yielding information about whether a reprinting was performed or not. Recall, that the *reprinting* function returns a value of type *m* (Maybe…) thus *lift* in Step 1 raises the monadic computation in *m* to StateT *s m*, and therefore *refactoringInfo* has type Maybe (RefactorType, Source, Span).

The second step matches on this information returned from the reprinting to determine whether to perform some splicing of the input source text or to navigate to the children. If *refactoringInfo* is Nothing then *enterDown* is called to proceed to the children. Otherwise, we have Just (*typ*, *output*, (*lb*, *ub*)) indicating a refactored node with refactoring type *typ*, new output source *output* (*e.g.*, from a pretty printer), and a pair of lower bound (*lb*) and upper bound (*ub*) positions for the node.

The refactoring type controls how much of the input source is used for the output and how much of the input source is discarded:

- Replace - the output source for the current context is the source text up to the lower bound of the node concatenated with the reprinting output. The input source between the lower and upper bounds is discarded.
- After - the output source is the input source up to the node's upper bound, concatenated with the reprinting output.
- Before - the output is the source text up to the lower bound of the node, concatenated with the reprinting output and then concatenated with the input source text between the lower and upper bounds of the node.

In each case, the cursor is updated to be the upper bound of the refactored node.

The third and final step is to navigate to the right sibling, producing the source text *output′*. The result of *enter* is then the concatenation of the output from either the current node or its children (*output*) with the output from the right sibling (*output′*).

The algorithm makes use of the *splitBySpan* function of type:

```
splitBySpan :: Span → Source → (Source, Source)
```

Given a lower bound and upper bound pair of positions, *splitBySpan* splits a Source into a prefix and suffix, where the prefix is of the length of source from the upper bound minus the lower bound. That is, the lower bound position is taken as the start of the parameter source and the source is split into two at the upper bound.

```
enter, enterDown, enterRight :: Monad m ⇒
Reprinting m → Zipper p → StateT (Position, Source) m Source
enter reprinting z = do
  -- Step 1. Apply a refactoring
  refactoringInfo ← lift $ query reprinting z
  -- Step 2. Deal with refactored code, or go to children
  output ←
    case refactoringInfo of
      -- No refactoring, go to children
      Nothing → enterDown reprinting z
      -- A refactoring was applied
      Just (typ, output, (lb, ub)) → do
        (cursor, inp) ← get
        case typ of
          Replace → do
            -- Get source up to start of the refactored node
            let (p0, inp′) = splitBySpan (cursor, lb) inp
            -- Remove source covered by refactoring
            let (_, inp′′) = splitBySpan (lb, ub) inp′
            put (ub, inp′′)
            return $ B.concat [p0, output]
          After → do
            -- Get source up to end of the refactored expr
            let (p0, inp′) = splitBySpan (cursor, ub) inp
            put (ub, inp′)
            return $ B.concat [p0, output]
          Before → do
            -- Get source up to start of the refactored node
            let (p0, inp′) = splitBySpan (cursor, lb) inp
            -- Split source consumed by the refactoring
            let (p1, inp′′) = splitBySpan (lb, ub) inp′
            put (ub, inp′′)
            return $ B.concat [p0, output, p1]
  -- Step 3. Enter the right sibling of the current context
  output′ ← enterRight reprinting z
  -- Append output of current context/children to right sibling
  return $ B.concat [output, output′]
enterDown f z =
  case down′ z of
    Just dz → enter f dz        -- Go to children
    Nothing → return B.empty-- No children
enterRight f z =
  case right z of
    Just rz → enter f rz        -- Go to right sibling
    Nothing → return B.empty-- No right sibling
```

Figure 1. Core traversal of the reprinting algorithm

Top-level The top-level function of the reprint algorithm converts an incoming data type to the datatype generic zipper, and enters into the root node, setting the cursor at the start of the file:

```
reprint :: (Monad m, Data p)
  ⇒ Reprinting m → p → Source → m Source
reprint reprinting tree input
  -- If the input is null then null is returned
  | B.null input = return B.empty
  -- Otherwise go with the normal algorithm
  | otherwise = do
    -- Initial state comprises start cursor and input source
    let state0 = (initPosition, input)
    -- Enter the top-node of a zipper for 'tree'
    let comp = enter reprinting (toZipper tree)
    (out, (_, remaining)) ← runStateT comp state0
    -- Add to the output source the remaining input source
    return $ out `B.append` remaining
```

Note that the final output is the concatenation of the output source from *enter* with the remaining input source text.

3.3 Reprinting parameter functions

The well-formedness conditions on an AST (Definition 1) requires refactored nodes to have a source span. This is captured by the following class, Refactorable:

```
class Refactorable t where
  isRefactored :: t → Maybe RefactorType
  getSpan      :: t → Span
```

That is, refactorable data types provide a span, and also information on whether they have been refactored using the RefactorType (Section 3.1), where Nothing implies a node has not been refactored.

The *reprint* algorithm does not directly enforce the Refactorable constraint since this does not interact well with the datatype generic implementations in GHC/Haskell (see discussion in Section 5.2). Instead, we provide the following builder function for generating a reprinting for a Refactorable type:

```
genReprinting :: (Monad m, Refactorable t, Typeable t)
  ⇒ (t → m Source)
  → t → m (Maybe (RefactorType, Source, Span))
genReprinting f z = do
  case isRefactored z of
    Nothing           → return Nothing
    Just refactorType → do
      output ← f z
      return $ Just (refactorType, output, getSpan z)
```

Given a function f that converts some refactorable type t to some source text, *genReprinting* wraps f and the methods of the Refactorable class to producing a Reprinting-typed function.

A function *catchAll* provides a default generic query which can be used to construct a generic reprinting:

```
catchAll :: Monad m ⇒ a → m (Maybe b)
catchAll _ = return Nothing
```

For example, given a monomorphic function *repr* :: S → Source on some syntax type S which is Refactorable, then a generic reprinting can be defined by

```
reprinting :: Reprinting Identity
reprinting = catchAll `extQ` (genReprinting (return ∘ repr))
```

where *extQ* :: (Typeable *a*, Typeable *b*) ⇒ (*a* → *q*) → (*b* → *q*) → *a* → *q* provides extension of a generic query from the *Scrap Your Boilerplate* library [8]. Here we use the Identity monad, *i.e.*, the reprinting is pure.

3.4 Example

The introduction showed a simple SSA-like language with assignments and integer addition (with prefix operations). We use the following data types to capture its AST:

```
data AST a =
    Seq a (Decl a) (AST a)
  | Nil a
  deriving (Data, Typeable)

data Decl a =
  Decl a Span String (Expr a)
  deriving (Data, Typeable)

data Expr a =
    Plus a Span (Expr a) (Expr a)
  | Var a Span String
  | Const a Span Int
  deriving (Data, Typeable)
```

Note that the data types derive the Data and Typeable classes to provide the datatype generic facilities.[1] These datatypes provide source code spans in each node and an annotation type *a* which can be used for indicating which nodes have been refactored.

We define a simple parser (not included here), which provides the function *parse* :: Source → AST Bool which annotates each with part of the syntax tree with False indicating no refactoring has been performed yet.

We thus have an instance of Refactorable for expressions:

```
instance Refactorable (Expr Bool) where
  isRefactored (Plus True _ _ _) = Just Replace
  isRefactored (Var True _ _)    = Just Replace
  isRefactored (Const True _ _)  = Just Replace
  isRefactored _                 = Nothing

  getSpan (Plus _ s _ _) = s
  getSpan (Var _ s _)    = s
  getSpan (Const _ s _)  = s
```

Given a simple pretty printer for expression (not included here) of type *prettyExpr* :: Expr *a* → Source, we define a reprinter for refactored expressions (but not the full AST data type of declarations) using the *genReprinting* helper:

```
exprReprinter :: Reprinting Identity
exprReprinter = catchAll `extQ` reprintExpr
  where reprintExpr x =
          genReprinting (return ∘ prettyExpr) (x :: Expr Bool)
```

[1] Note that this requires the `DeriveDataTypeable` language extension in GHC.

Given a transformation *refactorZero* :: AST Bool → AST Bool we put all the components together to parse, refactor, and reprint:

```
refactor :: Source → Source
refactor input = runIdentity
  ∘ (λast → reprint exprReprinter ast input)
  ∘ refactorZero
  ∘ parse $ input
```

Note that the input source text is passed to both the parser and the reprinter. We can now run the example as follows:

```
input = B.pack $
  "x = +(1,2)\n"
 ++ "y = +(x,0)\n"
 ++ "// Calculate z\n"
 ++ "z = +( 1, +(+(0,x) ,y) )\n"
output = putStrLn ∘ B.unpack ∘ refactor $ input
```

where *output* prints out the result on stdout:

```
*Main> output
x  =  +(1,2)
y  =  x
// Calculate z
z  =  +( 1,  +(x ,y) )
```

Example using "After" As a final example, we show the use of the After reprinting style as well as a monadic reprinter. Because our example language doesn't allow loops, it's possible for every variable declaration to be calculated before-hand, *i.e.*, all programs are terminating. Using our reprinter we can pass over the AST, calculate variable values as we explore the tree, and comment declarations with the resulting value. For our example, this produces:

```
x = +(1,2) // x = 3
y  =  +(x,0) // y = 3
// Calculate z
z  =  +( 1,  +(+(0,x)  ,y) ) // z = 7
```

We first define an *eval* function that takes an expression, a list of variables and their corresponding values, and returns a Maybe Int wrapped in the State monad:

```
eval :: Expr a → State [(String, Int)] (Maybe Int)
eval (Plus _ _ e1 e2) = do
  e1' ← eval e1
  e2' ← eval e2
  case (e1', e2') of
    (Just v1, Just v2) → return ∘ Just $ v1 + v2
    _                  → return Nothing
eval (Const _ _ i) = return ∘ Just $ i
eval (Var _ _ s) = do
  l ← get
  return $ lookup s l
```

If an unassigned variable is used then Nothing is returned, otherwise Just of the calculated value of the variable is returned.

We then define a reprinting that applies *eval* on Decl pieces of syntax, and returns an After refactoring if a value is calculated, producing a comment after each Decl node:

```
commentPrinter :: Reprinting (State [(String, Int)])
commentPrinter = catchAll `extQ` decl
  where
  decl (Decl _ s v e) = do
    val ← eval (e :: Expr Bool)
    case val of
      Nothing → return Nothing
      Just val → do
        modify ((v, val):)
        let msg = " // " ++ v ++ " = " ++ show val
        return $ Just (After, B.pack msg, s)
```

Note that the State monad is also updated (via *modify*) to record the variable-value assignment of a declaration.

Using *reprint* and *parse*, we now define a Source to Source transformation:

```
refactor2 :: Source → Source
refactor2 input =
    (`evalState` [])
  ∘ (flip (reprint commentPrinter) input)
  ∘ parse $ input
output2 = putStrLn ∘ B.unpack ∘ refactor2 $ input
```

where *output2* prints out the result on stdout:

```
*Main> output2
x = +(1,2) // x = 3
y  =  +(x,0) // y = 3
// Calculate z
z  =  +( 1,  +(+(0,x)  ,y) ) // z = 7
```

4 Reprinting and parsing as a bidirectional lens

A *bidirectional transformation* is a pair of programs converting data from one representation to another, and vice versa, often called the *source* and the *view*. A bidirectional *lens* is a bidirectional transformation capturing the notion of being able to update a source to produce a new source based on changes to a view [12].

Definition 2. A *bidirectional lens* comprises a *source type* S a *view type* V and a pair of **view** and **update** combinators typed [4]:

$$\mathbf{view} : S \to V$$
$$\mathbf{update} : V \times S \to S$$

A *well-behaved lens* satisfies the axioms:

$$\mathbf{view}(\mathbf{update}(v, s)) \equiv v \qquad (\textit{update-view})$$
$$\mathbf{update}(\mathbf{view}(s), s) \equiv s \qquad (\textit{view-update})$$

The first says: *updates to a source with a view should be exactly captured in the source, such that viewing recovers the original view.* The second says: *updating with an unchanged view of a source does not change the source.*

Reprinting and parsing together form a bidirectional lens, which satisfies the (*update-view*) axiom.

Proposition 1 (Reprinting-parsing lens)**.** Let the source type S be the type of source text, and let the view type $V = AST$ – the top-level type of abstract syntax trees – with lens operations **view** = **parse** : $S \to AST$ and **update** = **reprint** : $AST \times S \to S$. This assumes that we have already specialised the reprinting algorithm with some parameter reprinting function. The (*update-view*) axiom of well-behaved lenses then holds:

$$\mathbf{reprint}(\mathbf{parse}(source), source) \equiv source$$

i.e., parsing to an AST and then reprinting this (unmodified) AST with the original source, yields the source. This holds if the parser satisfies the well-formedness condition (Definition 1).

But can reprinting-parsing be a well-behaved lens? The (*view-update*) axiom would be:

$$\textbf{parse}(\textbf{reprint}(ast, source)) \equiv ast$$

This implies that any pretty printing used by the reprinting parameterising the reprinter is the left-inverse of parsing (*i.e.*, pretty printing then parsing is the identity function). This should hold in any reasonable situation: reprinting should form a well-behaved lens with parsing.

In the text-patching approach to reprinting by de Jonge *et al.*, a similar condition to (*update-view*) is introduced called *preservation* and (*view-update*) called *correctness* [3]. They also comment on the connection to lenses.

Note that well-behaved lenses are called *very well-behaved* if an additional property holds: an update followed by a second update is equivalent to just the second update. For reprinting, this would equate to the following axiom, which is unlikely to hold for most reprinters:

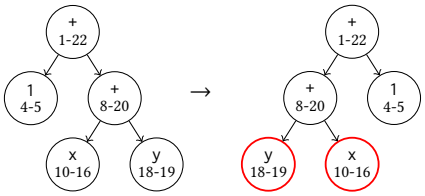
$$\textbf{reprint}(ast_2, \textbf{reprint}(ast_1, source)) \equiv \textbf{reprint}(ast_2, source)$$

This would imply that any changes made to *source* by ast_1 are erase/subsumed by the changes in ast_2. Subsequently, source spans in ast_2 would also need to closely correspond to actual code in *source* and $\textbf{reprint}(ast_1, source)$ simultaneously– which is unlikely.

5 Discussion

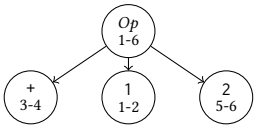
5.1 Considerations about well-formedness

The well-formedness condition (Definition 1) implies that transformations to an AST must take care with source span information. For example, transformations which commute nodes in the tree will almost certainly violate well-formedness, *e.g.*



The solution here is to swap the nodes, but swap the span information back, such that the span information stays in its original location in the tree, ensuring well formedness. There are other more complicated situations which might require more involved span recalculation as part of the transformation.

Another issue occurs if the shape of the AST data structure does not match the lexical shape of code. For example, consider a piece of infix syntax `1 + 2` which is parsed into a ternary tree node:



This parsing violates the well-formedness condition of *sequentiality*.

To solve this, we could include traversal information in the Refactorable class with the parent specifying which order its children should be looked at. Or we could look at the existing span lower bounds of children, and choose the order to view them based on the order they appear in the source text. Adapting our algorithm to deal with this is future work.

5.2 Constrained generic programming

Section 3.3 defined the *genReprinting* function to wrap an output function of type $t \to m$ Source where Refactorable t into a function $t \to m$ (Maybe (RefactorType, Source, Span)), wrapping the methods of Refactorable. However, there is nothing to force the programmer to use *genReprinting* to define a Reprinting. Indeed, the last example of Section 3.4 defined a Reprinting by hand.

An alternate approach would be to define *reprint* directly in terms of Refactorable types, *e.g.*,

```
type Reprinting m =
  ∀b.(Typeable b, Refactorable b) ⇒
    b → m (Maybe (RefactorType, Source, (Position, Position)))
reprint :: (Monad m, Data p, Refactorable p)
  ⇒ Reprinting m → p → Source → m Source
```

The core of the algorithm (*enter*, Figure 1, p. 5) could then be defined in terms of the methods of Refactorable directly. However, the Refactorable constraint must then be pushed into the generic zipper and the datatype generic operations, which are unconstrained. Much of the datatype generic infrastructure for GHC Haskell does not support this *constrained genericity*.

A potential solution is to parameterise datatype generic operations on additional constraint parameters (via GHC's constraint kinds), *e.g.*,

```
type GenericCQ (c :: ∗ → Constraint) r =
    ∀a.(Data a, c a) ⇒ a → r
```

and to define a constrained zipper type, *e.g.*, **data** Zipper $(c :: * \to$ Constraint$)$ $a = \ldots$ which adds the constraint within the intermediate data types of the zipper.

We have done some early exploration and it seems plausible, though such constraints will need to propagated throughout the rest of the libraries. This is further work and would be useful far beyond the topic of this paper.

5.3 Concluding remarks

We have presented a general algorithm that provides core functionality for refactoring tools: outputting source text that preserves secondary notation in untransformed code. The algorithm is relatively short thanks to the GHC Haskell's datatype generic programming facilities. Such an implementation would have been much more complicated 15 years ago.

We have been using a variant of this algorithm for several years and it has proven robust in the context of a real tool (CamFort). In terms of asymptotic performance, the core traversal is $O(n)$. The absolute performance is degraded somewhat by the use of datatype generics which are notoriously slow. Recent work suggests how to improve this considerably via staging [15]. Exploring this, with performance benchmarks, is further work.

Acknowledgements This work is supported by the EPSRC grant EP/M026124/1. We also thank the School of Computing and Faculty of Sciences at the University of Kent for their financial support of the first author. Thanks to Vilem-Benjamin Liepelt who contributed to the implementation of the *splitBySpan* function.

References

[1] Michael D Adams. Scrap Your Zippers: a Generic Zipper for Heterogeneous Types. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 13–24. ACM, 2010.

[2] Mistral Contrastin, Matthew Danish, Dominic Orchard, and Andrew Rice. Lightning talk: Supporting Software Sustainability with Lightweight Specifications. In *Proceedings of the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4), University of Manchester, Manchester, UK, September 12-Âŋ14*, volume 1686. CEUR Workshop Proceedings, 2016.

[3] Maartje de Jonge and Eelco Visser. An algorithm for layout preservation in refactoring transformations. *SLE*, 11:40–59, 2011.

[4] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[5] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[6] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, and Tamás Kozsik. Layout preserving parser for refactoring in erlang. *Acta Electrotechnica et Informatica*, 9(3):54–63, 2009.

[7] Jan Kort and Ralf Lammel. Parse-tree annotations meet re-engineering concerns. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 161–170. IEEE, 2003.

[8] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

[9] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ACM SIGPLAN Notices*, volume 40, pages 204–215. ACM, 2005.

[10] Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.

[11] Dominic Orchard and Andrew Rice. Upgrading Fortran Source Code using Automatic Refactoring. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, pages 29–32, 2013.

[12] B.C Pierce. The weird world of bi-directional programming, 2006. ETAPS invited talk, slides available from http://www.cis.upenn.edu/âĹijbcpierce/papers/lenses-etapsslides.pdf.

[13] Michael L Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 131–141. IEEE, 2001.

[14] Mark GJ van den Brand and Jurgen J Vinju. Rewriting with Layout. In *Proceedings of RULE*, 2000.

[15] Jeremy Yallop. Staging generic programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 85–96. ACM, 2016.

A Distributed Dynamic Architecture for Task Oriented Programming

Arjan Oortgiese
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL
a.oortgiese@student.ru.nl

John van Groningen
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL
johnvg@cs.ru.nl

Peter Achten
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL
peter88@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL
rinus@cs.ru.nl

ABSTRACT

Task Oriented Programming (TOP) is a special flavor of functional programming for real-world application domains in which people and automated systems collaborate to achieve a common goal. The original iTasks framework, which implements TOP, uses a single server multi-client architecture. This is not suited for truly distributed application domains, such as deployed by the Dutch coast guard. In this paper we show how to turn this architecture into a distributed, dynamic, architecture. This is done in an elegant way, by building on the core concepts of TOP and iTasks.

KEYWORDS

Functional Programming; Pure Functional Languages; Web Programming; Distributed Applications; Task-Oriented Programming; Client-Server Architecture; Platform Independent Code Generation; Clean; Haskell

ACM Reference format:
Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of The 29th symposium on Implementation and Application of Functional Languages, Bristol, United Kingdom, October 2017 (IFL 2017),* 13 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Task-Oriented Programming (TOP) is a special flavor of functional programming. TOP focuses on application domains in which people collaborate in a distributed manner to accomplish a certain goal. Typical examples of such applications can be found at the Dutch coast guard. The Dutch coast guard monitors vessel traffic on the North Sea, which is one of the busiest waterways in the world. One monitoring duty is to analyze the sailing route and behavior of a vessel, using information from the past, to determine whether further action is required. This requires the coordination of computation and human decisions how to proceed. Another Dutch coast guard duty is performing Search and Rescue actions [12]. Here teams collaborate to rescue people, and therefore everybody constantly needs to be informed about the latest state of affairs of all involved teams. These kind of systems are dynamic: circumstances change, communication is temporarily impossible, the situation requires down- or up-scaling, new information is available and needs to be taken into account, and so on. This is not limited to the Dutch coast guard: one can think of the government, police, and hospitals.

The iTasks framework [1, 15] implements TOP as an Embedded Domain Specific Language, using the pure, lazy, strongly typed, generic, functional programming language Clean as host language. In Clean one can define type driven generic functions [13]. This feature is used to enable TOP in iTasks: for any first-order data type that arises when modeling the application domain, code is generated to deal with (de-)serialization, rendering GUIs, transfering data between tasks and shared data sources, and so on. TOP has a small core [16]. There are only three core concepts in TOP: *tasks*, *editors*, and *shared data sources*, and a monadic, applicative combinator library featuring two core task combinators: *step* and *parallel*. A task is a type-parameterized abstraction that represents any piece of work performed by (a combination of) people and computerized systems that may occur within the application domain. Tasks that interact with end-users are called editors. The task uses its type-parameter to expose its current progress via its task value that changes over time. It is essential to understand that this abstracts from what is really going on within the task, after all, it might be a human performing the task or a computerized system or any combination. Tasks are observable: any other task can look at the task values of the tasks that it depends on and use this information to decide if and how it affects them. This dependency is defined via (a combination of) the task combinators (tightly) and

IFL 2017, October 2017, Bristol, United Kingdom

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

shared data sources (loosely). The step combinator coordinates a dynamic number of conditional tasks to observe a task and decide with which conditional task to proceed, if applicable. The parallel combinator coordinates a dynamic number of sibling tasks that can observe each other's task value. Shared data sources, finally, make the information sources at organizations accessible for tasks. Shared data sources can also be created dynamically.

Consequences of the above design and implementation decisions are that TOP programs are highly *modular*. The abstraction imposed by the task value hides the implementation of a task. This allows replacement of any task by another that has the same type and policy of task values. A human processed task can be transfered to another human processor in case of illness, holidays, or work-overload. Task value changes abstract from the fine-grained granularity of typical event-based systems: if a task processor (human or machine) is temporarily failing, it can always 'catch up' with a different task value change. These are important requirements when modeling real-world domains such as the ones described above. TOP offers these tools to create robust and realistic applications.

The current iTasks framework implements TOP as a single server, multi-user, web-based application. It works with any modern browser and thus works on mobile devices such as tablets or smart phones. This is fine under circumstances in which a reliable, sufficiently fast, internet connection is always available, and in which all shared data sources can be stored and maintained on the server. However, these circumstances are not to be expected for all application domains. The current architecture shares the following drawbacks with any single-server based web application: (1) The central server is a bottleneck when too many clients need to be served. (2) When the connection with the central server is lost, the client cannot inform the server about the progress tasks make, while the server cannot inform the client about the progress made by others. The end-user has to wait until the browser is reconnected to the internet. (3) To execute computations in the browser, we have to compile to JavaScript. Although the Clean to JavaScript code generator is state-of-the-art, JavaScript executes about ten times slower than native machine code generated by the Clean compiler. For CPU intensive applications JavaScript is unacceptably slow. (4) For security reasons, browsers do not allow access to operating system facilities, the file system, and many sensors which can be found on modern devices such as smart phones and tablets.

In this paper we show how one can elegantly turn the current iTasks architecture into a distributed one that eliminate these restrictions and unlock the full potential of TOP for the above mentioned application domains. The contributions of this paper are:

- We generalize the single-server architecture to one that permits a dynamic topology of arbitrary many 'multi-servers' on a wide range of devices. In this topology of devices, arbitrary tasks can be pushed from one device to the other, and arbitrary tasks can be pulled. This bypasses the single server bottleneck and it allows the proper evaluation of tasks on many devices. The coordination of these 'multi-servers' is done via domain controllers and local controllers. Domain controllers partition the end-users and their tasks. Local controllers run on client-devices and thus allow working offline (for a while) and reducing the workload of the domain controller from which they retrieve tasks.
- In order to bypass the limitations of JavaScript running in a client browser, we additionally compile Clean code for ARM processors. This unlocks more efficient code on the client device, as well as the opportunity to use local device capabilities, such as camera systems and sensor data. The latter is achieved by means of an Android App that can run any task on its device.
- Shared data sources are re-implemented such that their value can be inspected locally and remotely. With these new shared data sources, proxies to any remote task can be created to implement remote, observable, tasks.

The source code of the distributed version is available at https://gitlab.science.ru.nl/distributed-itasks. It contains the examples and scripts to compile for multiple platforms.

The remainder of this paper is organized as follows. First we give a short overview of the most important TOP concepts in section 2. In section 3 we show what has been done to send any closure to be evaluated remotely from one platform to any other. Section 4 presents the extended iTask architecture that uses domain and local controllers, and remote shared data sources to obtain the desired system. Related work is discussed in section 5. Conclusions are drawn and future work is discussed in section 6.

2 TOP / ITASKS IN A NUTSHELL

In this section we introduce the core concepts of iTasks. Next we give an example of an iTasks specification, showing how a chat application can be defined that works for an arbitrary number of end-users and that abstracts over how and which information is exchanged between the end-users. Hereafter we explain the major components of the current iTask run-time system.

2.1 Core Concepts of TOP / iTasks

2.1.1 Tasks. Tasks are created by functions returning a value of type `:: Task a | iTask a` for some concrete type `:: a`. The type `:: a` can be any Clean type, under the context restriction that an instance of **class** `iTask` exists for `a`. This `iTask` class consists of a number of generic functions [8, 13] which are needed to evaluate tasks on servers and clients, such as functions to serialize, de-serialize, compare values, and create GUI's. These generic functions can be automatically derived for *any* concrete *first order type*. So, the context restriction `| iTask a` is not a real restriction in practice.

Tasks are Clean functions defined in a special recursive way [16]. As a result, in contrast to ordinary functions, a task of type `:: Task a` emits *observable* values `v` of type `:: a` that may *change over time*. The task value reflects the current status of the work. When no meaningful information about the current status can be given, `NoValue` is used (for instance, no work has been done yet, or when it is decided to start all over from scratch). It can be unstable (`Value v Unstable`) when one is still working on a task such that the current value `v` might change in the future. It can also become stable (`Value v Stable`) when the work is done and value `v` no longer changes. All the time the status can be observed by the system and its current value might influence what others can see and do.

2.1.2 Editors. An editor is a basic task of type `:: Task a`. Editors make use of type driven generic functions such that the system is able to automatically generate a user interface in the browser for *any* concrete first order type `a`. With the generated interface end-users can only manipulate values of the required concrete type `a`. Examples of predefined editors are: `enterInformation` to enter a value of type `a`, `updateInformation` to change a given value of type `a`, or `viewInformation` to display such a value. For example, the default rendering of a record type is a browser form for filling in the fields (e.g. when the type is `:: String`, `enterInformation` shows a input field and the observable value of the task is the tekst entered in the input field). The user interface of an editor can always be customized. For instance, for the type `:: Picture` one can define a picture-editor, and for the type `:: GoogleMap` a map-navigator.

2.1.3 Task Combinators. The iTask library offers a rich set of Task Combinators, built on top of only two 'Swiss army knife' combinators: one for *sequencing* and one for the *parallel* case [15].

The sequencing combinator, called *step* and denoted as `>>*`, is a choice combinator. In `ta >>* [OnAction a1 p1 atb1,...,OnAction an pn atbn] :: Task b` the task `ta :: Task a` is being observed. The actions a_i in the list (note that this list can be computed on-the-fly) are converted to buttons on the screen. When button a_i is pressed by the end-user and predicate p_i `v` holds, where `v` is the current value of task `ta`, that task is ended and one continues with task $(atb_i\ v)$ `:: Task b`. There are also simple variants of this sequential operator, like a monadic bind `>>=`, `return`, and `>>|` (Haskell: `>>`).

The `parallel` combinator coordinates a dynamic collection of sibling tasks that can observe each others' task values. There are several derived combinators for frequently occurring, less dynamic, parallel patterns. With the parallel *and* operator, `-&&-`, two tasks `ta` and `tb` can be started in parallel, the resulting task returns a tuple combining the results of both tasks, so `ta -&&- tb :: Task (a,b)`. With the parallel *or* operator, `-||-`, the value of the task which has become stable first is returned, so ta_1 `-||-` ta_2 `:: Task a`. Frequently used variants of this operator are `-||` (only return the current value of the first task, ignore the second), and `||-` (the other way around).

One-and-the-same end-user may work on several parallel tasks at the same time. An important aspect of a parallel task is that it can be assigned to someone else. In `Bob @: ta` the task `ta` is assigned to user `Bob`. In this way an arbitrary number of tasks can be assigned to a specific person or to someone with a specific role. In the default set-up, the tasks somebody can work on are listed in a to-do list, and the end-user can freely choose on which tasks she works.

2.1.4 Shared Data Sources. Although tasks can observe each others' progress with the step combinator, and information can be passed from one task to another, one needs to be able to share arbitrary global data between tasks as well. One can think of global information stored in data sources like memory, files and databases, data like information produced by sensors, the current date and time. For *any* type of data that one wishes to share, *one and the same* abstract interface is provided, called a *Shared Data Source* or *SDS* [5]. So, an SDS can be shared data stored in a file, in a relational database, or temporarily stored in main memory. An SDS is used internally to administrate the users who can login, the current user working on a task, and so on. Like tasks, SDS combinators are provided to create more complex SDS's from simpler ones. In the current iTask architecture, SDS's are only available on the iTasks server and cannot be put on a client.

The following atomic, basic operations are defined: `get`, `set`, `upd`, `watch`. With `get` the current value from the SDS is retrieved; `set` writes a new value to the SDS; `upd` uses a higher-order function to update the current value of the SDS; `watch` turns an SDS into a task that emits unstable task values reflecting the current SDS value.

Editor tasks (section 2.1.2) can be 'connected' with an SDS. For any first order type a user interface can be created, also when the value is stored in an SDS. To view the content of an SDS one can use `viewSharedInformation`. The content can be updated with `updateSharedInformation` where the SDS is updated with every change that is made by the end-user in the user interface shown in the browser. The core parallel task combinator (section 2.1.3) utilizes SDS's to inform the parallel child tasks of the (continuously changing) task values of themselves.

Hence, it is important that whenever someone changes an SDS, all tasks that are currently observing that SDS are automatically informed about the change that is made. For example, if an end-user changes the content of an SDS via `updateSharedInformation`, all other users who look at the value via `viewSharedInformation` will see the changed value as well, albeit a bit later due to the latency of the internet. Every SDS is administrated at run-time in the publish-subscribe system of iTasks where is managed which tasks need to be informed when something is changing.

2.2 A dynamic, customizable, chat application

As a running example we present an iTask application that enables an arbitrary number of people to chat with each other. We define it in a general way so that we can easily modify it later in this paper. The entire specification is given in Figure 1. It is not important to understand all the details because we want to give you an idea about what task definitions look like. We will use it later on to explain what the consequences are for a distributed architecture.

The task function `createChatSession` takes two tasks, `enter` and `update`, as argument. The `enter` task can be *any* task producing a value of arbitrary type `:: a`. With the `update` task this information is turned into some convenient display format `:: b` to be stored in an SDS of type `:: [b]`, used to display the chat history to all participants. First the current end-user is determined (4), `me`, who wants to start a chat session. Next, using the bind combinator `>>=`, it asks this user to select the `others` to chat with (5–6). The `enterMultipleChoiceWithShared` task let the user select one of the elements in the users SDS (`[User]`) using a pull down box. The predefined SDS `users` from which the others are chosen, contains a list of all users currently known in the system. Hereafter, a new SDS of type `:: [b]` is created in shared memory to store the history which is initialised with an empty list `[]` (7). This shared list is passed to the `startChats` task together with the `enter` and `update` task and the list of users participating in the chat session.

The task function `startChats` assigns (using `@:`) to every selected person the `chatWith` task, using the parallel combinator `allTasks` (13–15). When all participants have ended their chat session the chat history stored in `chatStore` is returned at the end of the conversation (15).

```
createChatSession :: (Task a) (User a -> Task b)
                                -> Task [b] | iTask a & iTask b
createChatSession enter update
 =              get currentUser
 >>= \me ->     enterMultipleChoiceWithShared
                                  "select_people" [] users
 >>= \others -> withShared []
                          (startChats enter update [me:others])

startChats :: (Task a) (User a -> Task b) [User] (Shared [b])
                                -> Task [b] | iTask a & iTask b
startChats enter update people chatStore
 =   allTasks [ (person, "chat") @:
                  chatWith person enter update chatStore
              \\ person <- people ] >>| get chatStore

chatWith :: User (Task a) (User a -> Task b) (Shared [b])
                                -> Task () | iTask a & iTask b
chatWith me enter update chatStore
 =   viewSharedInformation ("Chat_History:") [] chatStore
 ||- oneChat
where
  oneChat =   enter
          >>* [ OnAction (Action "Send") (hasValue send)
              , OnAction (Action "Quit") (always (return ())) ]
  send new_msg
    =           update me new_msg
    >>= \new -> upd (\msgs -> msgs ++ [new]) chatStore
    >>|         oneChat
```

Figure 1: The dynamic, customizable, chat application

In `chatWith`, a person sees the messages entered so far, due to `viewSharedInformation` (20). At the same time (`||-`), she can enter a new message with `oneChat` (21). The `oneChat` task uses the higher order task function `enter` to let the user enter a new chat message. Two buttons are created, `Send` and `Quit`. When `Send` is pressed, the newly created message is converted with the `update` task (27), and added to the chat history stored in the SDS `chatStore`, after which another message can be entered due to the recursive call of `oneChat`. When the `chatStore` is updated, all persons see the new message due to their `viewSharedInformation` on this SDS. When `Quit` is pressed, the end user quits her session, returning the stable task value `()`. This is the complete specification of the customizable application.

We can now proceed to create a custom instance (Figure 2). To inform all persons who created what message at what time, we introduce a new type `:: Msg a` that keeps track of this information for messages of type `:: a`. The task function `chatExample1` creates the custom instance. For entering messages, the `enter` task uses the generic `enterInformation` editor task. Note that the kind of messages that can entered is determined by the type of `chatExample1` only. For this example we use `String`. These are converted to `Msg String` with the `update` task function.

Note that all the user interfaces and low-level event handling code is generated from the chat definition above. When a new message is entered by someone, it is stored in the SDS and all users automatically see the updated history.

```
:: Msg a  = {time :: Time, user :: User, message :: a}
derive class iTask Msg

chatExample1 :: Task [Msg String]
chatExample1  = createChatSession enter update

enter :: Task a | iTask a
enter      = enterInformation "Type_in_a_message:" []

update :: User a -> Task (Msg a) | iTask a
update me msg
   =           get currentTime
   >>= \now -> return {time = now, user = me, message = msg}
```

Figure 2: An instance of the generic chat specification

```
:: ChatOptions = Line String | Chat [Msg ChatOptions]
derive class iTask ChatOptions

chatExample2 :: Task [Msg ChatOptions]
chatExample2 = createChatSession enter2 update

enter2 :: Task ChatOptions
enter2
 =          enterChoice "Message_kind" [] ["Text","NewChat"]
  >>= \k -> case k of
              "Text"    -> enter        @ Line
              "NewChat" -> chatExample2 @ Chat
```

Figure 3: Invoking chats recursively

To demonstrate the generic and customizable nature of the specification, we define an alternative version in which, at any time, any person involved in the chat session, can start a new chat session with new people and have the extra conversation added automatically to the ongoing conversation. The specification is shown in Figure 3. We first introduce a new domain type, `ChatOptions`, to reflect the choice every person has and generate the necessary generic instances. We only alter the first higher-order parameter of `createChatSession`. This new task, `enter2`, offers the current end-user a choice between entering a line, as before, resulting in the task `enter` or starting an entirely new conversation, resulting in the recursive task `chatExample2`. The `@` operator turns the result of its task into the proper `ChatOptions` value. The effect of this relatively small change in the specification is that whenever the new conversation has terminated, all of its content is automatically added to the currently ongoing conversation.

2.3 Standard iTasks Run-Time System

The standard iTasks run-time architecture consists of a single server that serves multiple clients. These clients are commonly browsers that end-users use to perform their tasks.

2.3.1 The iTasks Server. The core components of the iTasks server are the following. *The task pool administration* contains a list of all the tasks someone can work on. It can be regarded as a to-do list. When a new parallel task is created for someone to work on, it

is added to this pool together with the task attributes. The set of attributes is not fixed but can be freely defined by the programmer. Common attributes are: the `title` of the task, creation `date` and `time`, `priority`, `deadline`, the `creator` of the task and for whom the task is `created for`. The `created for` attribute can be a specific user, or someone with a specific role. *The task instance administration* keeps track of the task instances of tasks administrated in the task pool someone is actually working on. The relation between the task pool and instance pool is one to at most one. Only one person can work on a certain task at the same time. the owner of a task may change, however (e.g. someone else with the same role) at any time. Local status information, stored at the client, may get lost when someone takes over a task from someone else. *The SDS administration* contains which tasks currently depend (subscribed) on which SDS's. This is used to inform those tasks when an SDS changes. *A web server* is built in, although any standard web server can be used as well. Every action on a browser is propagated to the server as an event and handled by the iTask kernel. The kernel continuously processes events from clients, tasks and SDS's, TCP connections and the like to coordinate the tasks.

The iTasks system contains several plug-in components as extensions, to reduce the number of core components to a bare minimum. The authentication process of users and its administration is an example of such an extension. Also the way tasks are presented in a browser to end-users can be freely defined in a task. A standard way to do this is to present all tasks someone can work on in a dedicated to-do list, such that the end-user can choose where to work on by opening one or more of these tasks.

2.3.2 iTasks Clients. The clients in iTasks are commonly HTML 5 compatible browsers. When a browser makes a connection with the server, initially a small client-side application written in JavaScript is loaded. The client sends client-events such as changes made with an editor or `OnAction` buttons pressed to the iTasks server that processes its responses. The response may result in a dynamic update of (a part of) the HTML code, or a new piece of JavaScript is loaded and executed on the client.

One has to realize that iTask applications are very dynamic: the number of tasks created, their content and user interfaces, the relation with other tasks and SDS's, are all only known at run-time. Hence new JavaScript code is automatically added to the initial client code when needed. It consists of just-in-time compiled Clean code needed for the evaluation of a specific closure on the client. For that purpose Clean is compiled twice at compile time: once to native code (Intel) x86-64 code for Windows, Mac, and Linux platforms, and once to SAPL code [10]. SAPL is a core functional language that is dynamically just-in-time translated to JavaScript code. SAPL has as advantage that from the code one can relatively easily determine which functions are additionally needed for the evaluation of a given closure on a specific client.

This scheme eliminates the need to define browser computations in JavaScript, and use the pure and type safe Clean host language instead. We can define custom editors in Clean (see [4]), e.g. for making drawings in the browser, or working with maps, or for interfacing with JavaScript WebAPIs like getting the current location or access the camera of a client device.

3 DISTRIBUTED EVALUATION OF CLEAN CODE

As stated above, iTask programs are compiled to native (Intel) code and to JavaScript via SAPL. To unlock TOP for ARM-based platforms, such as Raspberry Pi and Android, we need to generate code for ARM processors as well. To realize distributed evaluation of Clean functions, two additional facilities are needed. First, the corresponding code needs to be available on the remote processors in the network. Second, any Clean application on any platform must be able to create any closure in a symbolic format such that it can be serialized, shipped, de-serialized, and remotely executed on any of the other platforms.

3.1 Distributing Code

If we ship a function for evaluation to another machine, we have to ensure that the corresponding code needed for the evaluation of that particular function is indeed available on the remote machine.

ABC code is the intermediate code of the Clean compiler that is compiled in a later phase to machine specific code. One option is to ship the ABC code and let the remote machine perform the last phase of the compilation for its platform. We have decided to generate all images for all platforms at once when the application is being developed because it is easier in this way to guarantee that the corresponding generated images have been made from the same source code using the same compiler, yet different code generators and linkers. As said before, the Clean compiler is very fast, so the developer is not hampered by this. To give an idea: on a smart phone like the Samsung Galaxy Note 4 the Clean compiler compiles itself from scratch in 11 seconds.

Next we have to ensure that the required code is available on the (remote) machine before we ship a closure for evaluation to it. It is important to realize that statically it is undecidable what kind of closures are constructed. Because we are dealing with a lazy language, a closure might contain unevaluated function calls. So, one cannot know on forehand which code is needed for the evaluation of a closure. Code can be distributed in different ways, *eagerly* or *lazily*. When shipped eagerly, all code, the complete image, is stored on the target machine in advance. With the image any possible closure received can be evaluated. Eager code distribution has as disadvantage that one perhaps does not want to show all code on the remote machine for security reasons. With the lazy shipment method only the code which is really needed for the evaluation of the closure is shipped. It has as disadvantage that one needs to be able to dynamically extend the running application (dynamic linking). For each closure one has to determine which code, not present yet, needs to be shipped and added to the application.

Currently, for x86-64 (Intel) and ARM processors the eager code distribution approach is used. One has to ensure manually (download or upload) that the image is indeed available on the remote machine. For browsers, code is shipped lazily to the browser on demand using push technology. It should be noted that for Windows a dynamic lazy linking facility already exists, via Clean Dynamics [19]. We intend to add a lazy shipping and linking option for the other platforms and operating systems as well.

Of course, no matter how code is distributed, proper security measures have to be taken to ensure that the code can be trusted, as usual. This is beyond the scope of this paper.

3.2 Distributed Evaluation of Functions in Native Code

To send a closure for evaluation from one platform to some other platform means that we have to be able to serialize *any* function or expression at *any* time for *any* platform. The serialized function needs to have a platform independent representation such that it can be unpacked at any of the other platforms to be evaluated remotely. Of course, the result of the evaluation has to be sent back in the same way as well.

Using native code has the drawback that serialization and deserialization of closures and its resulting value becomes platform *dependent*. The run-time architecture on a platform may differ in all details: in code, address locations, data representations, stack lay-out, and heap lay-out. At run-time therefore a closure has to be reconstructed symbolically from the actual content of the stacks and the heaps, serialized, and on the other machine the information has to be stored in the right way such that evaluation can be done as usual. Closures can refer to data types and functions which are located at a different address in the image of the other platform. So, when reconstructing a closure we have to know where to find the information at run-time and replace concrete memory addresses by the corresponding symbolic function and constructor names. Since all code is generated from the same Clean source, these symbolic names are known in all code variants and can therefore be translated back to the proper run-time format needed for the evaluation of the closure at another platform.

For every platform we have implemented (de-)serialization operations that work for any closure of any type. These operations use the symbol table of the executable to look-up the memory addresses of the descriptors. One can obtain the symbolic names from the symbol table in the corresponding object/executable format. There are several object/executable formats we have to deal with, e.g. ELF for Linux (and Android), Mach-O for MacOS and PE/COFF for Windows. For every of these object formats we are able to obtain the required symbols and reconstruct the function call in such a way that it can be evaluated on the platform.

When generating code, one can choose to generate code for 32-bit or 64-bit machines. We provide the ability to *mix* these formats, and translate the data of 32-bit machines to a 64-bit format and vice versa. Of course, one has to be careful because down scaling a 64 bit integer to a 32 bit integer can create incorrect numbers, and causes a run-time exception. Furthermore, in this mixed setting we currently do not support the hardly used unboxed reals and unboxed arrays. This is future work.

4 DISTRIBUTED ITASKS

In this section we explain the architecture of the distributed iTasks system. Instead of having one central server, we support an arbitrary number of iTask servers, called *controllers* from now on to avoid confusion, since these new iTask servers can run on clients as well as on servers. Actually, an arbitrary network topology of controllers can be made which may change dynamically over time. All controllers are programmed in Clean, and can therefore run on any platform, i.e. any Intel or ARM processor. We assume that all devices on which controllers are running have all the necessary code at their disposal and are able to evaluate any closure shipped to them for evaluation (see Section 3). Hence we focus on the issues related to the distribution of iTask controllers and iTask tasks.

In the new setting we have two types of controllers in the network, *domain controllers* and *local controllers*. We first look at a topology where we only have distributed domain controllers (Section 4.1) and explain what the consequences are for iTask applications in this setting and how the distribution of controllers affects the implementation. To be able to evaluate tasks in a distributed fashion, we have to deal with the fact that an SDS can now be located on some other machine instead of the same central server (Section 4.2). The evaluation of a task on another device differs from the evaluation of an ordinary Clean function because the corresponding task value has to be made observable by the shipping controller (Section 4.3). We show how to reduce the workload of a serving device by splitting up the task coordination over several serving controllers (Section 4.4). We introduce local iTask controllers (Section 4.5). They allow to work offline on a tablet or other device. We show examples of tasks running in a distributed fashion on a network of (domain and local) controllers running as Android app, or located on a Raspberry Pi (Section 4.6). Finally, we discuss the consequences of having a distributed architecture versus a centralized one (Section 4.7).

4.1 Distributed Domain Controllers

Domain controllers for iTasks are inspired by domain servers for handling e-mail traffic. The iTask setting is more complicated because working on a task can have direct effects for other tasks of other users located anywhere, while answering an e-mail has no global effect at all until the e-mail is actually sent. An iTask network has the following properties.

(1) It is assumed that the locations of all domain controllers (e.g. their IP addresses) are globally known and accessible for all domain controllers in the net. Domain controllers can still be added and removed dynamically. For usability we assume that all the domain controllers have a domain name that can be resolved using DNS.
(2) There is at least *one* domain controller in any iTask network. If there is only one domain controller, the architecture behaves just like the old standard configuration with one central server. The new architecture is a real extension of the old one.
(3) As is the case with an e-mail server, a domain controller serves a group of users who are administrated in that domain and therefore can login into the controller. A domain controller takes care of the authentication process. By default the user and roles administration of iTasks is used that stores users and roles in a SDS. However, any other back-end such as an LDAP server or Single Sign On server [3] can be used. A server has to support two operations. First, at authorisation, given a user name and password, results in a unique user identification, including the roles the user can fulfill. Second, it must provide a list of all currently administrated

users, which can be used to (interactively) select a user to assign a task to.

(4) Tasks can be assigned to a user, or a user with a specific role, as usual. To address a user uniquely over the net, the domain name is added as postfix, just like we do in e-mail addresses. The operator `@.` combines them `user @. domain_name`. If no domain is specified when a task is assigned to someone, by default the domain of the sending user is added as postfix.

(5) All tasks sent to a certain domain are administrated in the task pool administration of both the sending domain as well as in the administration of the receiving domain controller. When a user is logged-in into a domain controller, only relevant tasks can be seen. When a controller fails the controller can be restarted. Information of all users, all tasks, and all tasks instances are constantly stored in persistent memory, no information is lost when a machine on which a controller is running, has stopped execution. When the controller is restarted, a user can continue the work by re-opening the task.

We generalize the chat example of Section 2.2 such that we can chat with any user administrated in any domain known in the network. We only need to alter one line of code in the task function `createChatSession` (Figure 1), all other code of the previous chat example remains the same. The call to `enterMultipleChoiceWithShared` has to be replaced by a call to `selectUsersFromDomain` (Figure 4). It calls the recursive task `select` that accumulates a list of persons selected from the domains. The SDS `domains` (6) contains the names of all

```
selectUsersFromDomain :: Task [User]                                    1
selectUsersFromDomain = select []                                       2
                                                                        3
select :: [User] -> Task [User]                                         4
select acc                                                              5
 =         enterChoiceWithShared "Select_Domain" [] domains             6
 >>= \d -> enterMultipleChoiceWithShared                                7
               ("select_persons") [] (usersOf d)                        8
 >>*       [ OnAction (Action "Add")                                    9
               (hasValue (\new -> select (new ++ acc)))                 10
           , OnAction (Action "Done") (always (return acc)) ]           11
```

Figure 4: Selecting domains and users

known domains. The function `usersOf` (8) returns an SDS containing the names of all users administrated in the given domain. The end-user who wants to start a chat session recursively selects a domain (6), and the users to chat with from that domain (7–8). Next the end-user can decide by pressing `Add` to add more people, or to stop the selection process by pressing `Done`.

Although the iTask specification is changed only in one place, the possibility that there are several (domain) servers has quite some consequences for the implementation. Consider Figure 5 where we assume a small distributed network configuration, consisting of two domain controllers, for the domains `A` and `B`. Assume that `Alice` on domain `A` initiates a chat session with `Dave` on domain `B`. The chat-history SDS of type `[Msg String]` which is displayed to both persons is located on the domain controller of `Alice`. So, the task shipped to `Dave` running on controller `B` needs to have access

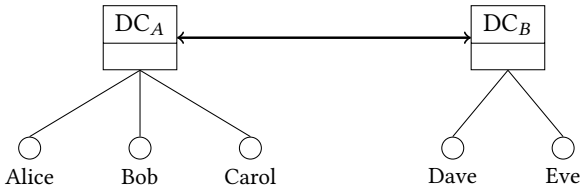


Figure 5: Two mutually connected domain controllers.

to the SDS remotely located on `A`. Moreover, if `Dave` `Stops` the chat session, this will have consequences for the parallel task `allTasks` (Figure 1) located on `A`. Hence, the current value of tasks need to be observable on other controllers.

4.2 Accessing Remote Shared Data Sources

In the original setting, all SDS's are hosted on the same iTasks server. In the distributed setting, a task under evaluation on a certain controller, can create a new SDS which is locally stored on that device. We call this device the *SDS-host*. Future tasks may want to have access to this SDS, also when such a task has been shipped to another controller for evaluation.

At run-time it is known on which device a specific SDS is located. When an SDS is located on the same device as the task accessing it, access can be handled as usual. To enable access to remotely located SDS's, remote versions of the basic SDS operations `get`, `upd`, `set`, and `watch` (Section 2.1.4) are implemented as `r_get`, `r_upd`, `r_set`, and `r_watch`. The programmer does not need to be aware where an SDS is located: at run-time an application of `get`, `upd`, `set`, and `watch` is automatically redirected to a call of `r_get`, `r_upd`, `r_set`, and `r_watch` in the case of a remote SDS. The `r_` versions emulate remote SDS access as a proxy, in such a way that a remote SDS behaves and reacts in the same way as a local SDS.

To access a remote SDS we use the (de-)serialization mechanism (Section 3.2) to send a serialized closure to the SDS-host, and let it de-serialize the closure and apply it to the SDS. This allows us to access any SDS no matter where and how it has been remotely created and stored. It also works for parameterized SDS's [5], SDS projections which allow a task to access a specific part of an SDS enabling a more fine grained and more efficient access to SDS's.

The result of the remotely applied closure is sent back to the requesting controller who is waiting for the response. In this way calls to `r_get`, `r_upd`, and `r_set` at a requesting controller can simply be realized by remotely applying the standard `get`, `upd`, and `set` at the SDS-host. This approach ensures that the operators are applied atomically to the latest value of the SDS, so we do not have to deal with synchronization and version conflicts.

The `r_watch` operation is handled differently, because a straight adaption of the above scheme for this operation results in a lot of network traffic and unnecessarily blocks the requesting controller. Instead we implement a `notify_me` request which evaluates a closure when a given predicate holds at a host, after which the requesting controller is notified asynchronously. We use this notify request facility to implement the `r_watch` task. First we fetch the current value of the remote SDS via `r_get` and store a copy in a new local SDS at the requesting controller. We then locally `watch` if this SDS

copy has been changed. It will only be changed if the SDS-host has notified the requesting controller that the original remote SDS has obtained a value which is different from the copy locally stored. For testing equality we can use the generic equality function which can test equality for any first order type. When we receive a new value from the SDS host it is stored in the SDS copy, which change will trigger the waiting task as usual. This process is repeated as long as the `r_watch` operation remains active.

4.3 Evaluating Tasks Remotely

In section 3 we explained how any Clean function can be serialized, shipped to another processor, de-serialized, and evaluated. However, if we want to evaluate a task function remotely, it is not enough to send over the task function. Also *all* instances of the generic `iTask` functions specifically generated for the task function are required as well. For this purpose we define the following container type `Remote_Task` for shipping a task with its generic functions:

```
:: Remote_Task
   = E.a: Remote_Task (Task a) TaskAttr InstanceNo & iTask a
   | NoTask
```

This existentially quantified algebraic data type can contain *any* task of *any* type (`Task a`) together with its task attributes and task number which is used as unique identification of the task. In this type definition `& iTask a` defines a context restriction: the ADT can only contain tasks of type `Task a` for which also an instance of **class** `iTask a` has been defined. A dictionary containing all the members of the class `Task a` is automatically added to the constructor of the ADT when a value of this type is created. Hence, when such a container is shipped, it will not only contain a task of a certain type, but also all required generic function instances for that type, such that all methods needed to evaluate the task elsewhere are included.

When a message is received that cannot be de-serialized to a proper task as described, the container is not accepted. This can only happen when a container is corrupted somehow due to external communication failures.

When a task is being evaluated, we also might need to inform the sending controller about the current state of the task. The shipped task, like any other task, emits observable task values that change over time (see 2.1.1). If observed by for instance the step task combinator, >>*, this implies that the shipping controller must be able to observe the current value of the remote task.

To support this, we create two special tasks, a proxy task on the sending controller and an evaluating task on the remote controller. The proxy task maintains a local copy of the task value in an SDS. The remote evaluator maintains the current task value and an instance of the task to be evaluated in another SDS. Every time the remote evaluator evaluates the task due to an action of the end-user, it may change the task value. The changed task value is communicated to the sending controller to store it in the SDS of the proxy task for synchronization. In this way other tasks in the sending controller can observe the task value of the remote task.

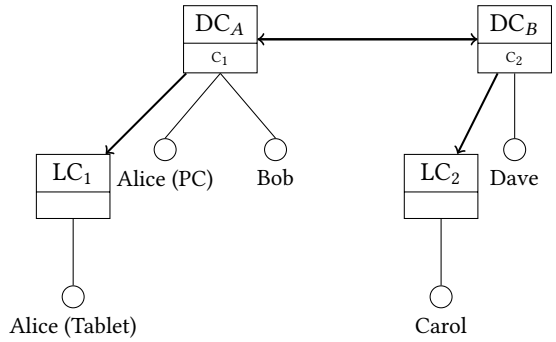


Figure 6: Network with two *domain controllers*, each with a *local controller* attached.

4.4 Distributing Tasks Server-Side

The ability to have multiple controllers can be used to decrease the workload of a domain controller on the server. Another server-side controller can take over part of the work from a *domain controller* to avoid that such a controller becomes a bottleneck.

A controller that wants to take over work from the domain controller can describe the tasks it is willing to take over via a *claim filter*, which is a predicate of type `:: TaskAttributes -> Bool` that defines which tasks are wanted. The claim filter is sent to the domain controller and when a new task is added to its task pool administration, the domain controller first tries to search for a controller that wants to claim the task. The first controller that is found gets the task assigned and then receives the task.

One can freely assign attributes to a new parallel task when a task is created. In this way one can specify for whom a task is intended, or define other demands, such a specific role, processor, or resource. In this way, any algorithm can be realized to spread the workload over servers. For example, the following claim filter claims all tasks of the users with a name starting with a character of the first part of the alphabet.

```
claimUsersAM :: TaskAttributes -> Bool
claimUsersAM attrs
    = let name = readAttr "createdFor" attrs in
      ("a" <= name && name < "n") || ("A" <= name && name < "N")
```

The domain controller can redirect end users after they have logged in to the address (e.g. URL) of the controller that is claiming their tasks. The work and users are divided over the controllers.

4.5 Distributing Tasks to Local Controllers on a Client

One can also have additional controllers on clients. Such a *local controller* might be located anywhere, on a pc, laptop, tablet, or smart-phone. An administrated user can log into a domain controller with a local controller and claim a subset of his or her tasks to be downloaded to the local machine. Claiming tasks to be handled locally is realized in the same way as decribed in section 4.4. The local controller can for example automatically pull all the task for the current active user. Another option is that the user manually chooses the tasks that need to be pulled to the local controller by

opening them. With a local controller one can work on tasks locally without disturbing the server side domain controller one is administrated on. This makes sense when a task requires a lot of work for which no or limited interaction with other end-users or systems is needed. Handling task events can be done faster because the controller is located on the same machine and one does not need to share the controller with other users. As long as no interaction with other end-users or remotely located SDS's is needed, one can also work off-line.

A local controller is an iTask server, which can be used in the following ways:

- A local controller can connect under a user name to its domain controller. Only those tasks intended for the specific end user logged in can be seen and claimed via the claim mechanism. One can subscribe for specific tasks, e.g. that need a camera.
- Any number of local controllers can make a connection to their domain controller. The domain controller has a *local controllers administration* to keep track of the currently connected local controllers. Local controllers can connect or disconnect themselves from the domain controller at any time. If a local controller is temporarily disconnected from the net, then, after re-establishing the connection, the deferred communication continues as if nothing happened.
- A local controller can subscribe to the task pool of another local controller. A whole chain, or a tree, or any topology of local controllers can in principle be made this way.
- The effect of a subscription is that a local controller can subscribe to the task pool of the domain controller. When new tasks are added to the domain controller, the local controller is notified. The notification message contains a wrapper task with a reference to the task in the task pool of this (domain) controller that is used to download the task when one wants to evaluate the tasks on the local controller. This evaluation of a task on the local controller takes place just as described before. So, the task might access remote SDS's and its task value can be observed remotely.
- When a task is downloaded from a (domain) controller, it is stored in the task instance administration which local controller is working on it. This prevents other controllers or end-users to work on the same task. In iTasks only one person or system is allowed to work on a certain task at the same time.
- As in the original iTask system, it is possible to explicitly steal a running task. This can be done by the same user who wants to switch to a different device or by another user with the same role, who has decided to take over the work under evaluation. If the task instance of the task to steal is available, most of the work done so far can be rescued. The task instance can get completely lost when a controller becomes unavailable for some reason. However, the original task to work on is still available and can therefore be restarted from scratch.

By default, the tasks to work on are presented to an end-user for which the tasks are intended in a list, much like incoming e-mails are presented in an e-mail client. The same end-user can log in as many times as desired, directly on the domain controller, or indirectly via a local controller connected to a domain controller. In Figure 6 we show a possible set-up in which end-user `Alice` is logged-in twice. However, `Alice` is not allowed to work on the same task on both machines. If she starts to chat with someone on one of her machines, then this task is blocked on all others. Still, she can steal the task onto the other machine by asking the iTask system to move it. The system warns her that local state information is lost. For the chat example this concerns only the current sentence that is edited in the browser. The chat-history SDS remains unaffected.

4.5.1 Client-Side Configuration Options. There are different ways to connect from a client to a domain controller. Each way has certain advantages and disadvantages.

One can use a browser on any platform and login into the domain controller. This is the simplest way to interact. If too many end-users are connected to a controller, the interaction with a browser might slow down unacceptably, because all events generated by all browsers must be handled by the same domain controller. CPU-intensive computations on the browser can also cause slow-downs due to the use of JavaScript.

Using a private local controller at the client-side with a browser connected to this local controller is another option. It increases the client-side responsiveness because only local network traffic is needed to handle the browser events. Furthermore one obtains the possibility to work offline on tasks, which makes sense for tasks which require a lot of local work. Working offline with a browser can be done by connecting the browser to the local controller as localhost.

In the configurations above, standard browsers are still being used for doing the interaction. Another interesting option we offer is to create an all-in-one Android app as client.

An iTasks Controller as Android app

An Android iTasks app consists of three components:

(1) A local controller that is compiled from the same source code as the other iTask controllers. It is compiled as a shared library instead of an executable, such that it can be used inside an Android application.
(2) Additional functionality as Clean functions which are compiled to native ARM code stored in the app. We no longer need browsers to do client-side calculations, and are therefore not forced to use JavaScript. Clean compiled to native ARM code runs about ten times faster than JavaScript code [9].
    Another advantage is that in the app, if granted upon installation of the app, we have access to *all* hardware devices, resources, file systems, and operating system facilities which is not possible when a standard browser would be used. Any component which has a C interface can be accessed, since Clean offers a C interface. If a component needs to be accessed via a Java library, Java Native Interface (JNI) can be used to call Java methods from C. In this way we can e.g. access the Bluetooth stack or the cameras on the device.
(3) We can optionally include a browser component in the app, and connect it to the local controller. In Android we use the `WebView` component [7] that is part of the Android platform as a browser.

```
:: TextPicture = T String | P Picture                                    1
derive class iTask TextPicture                                           2
                                                                         3
enterPictureChat :: User -> Task (Maybe (Msg TextPicture))               4
enterPictureChat me                                                      5
 =           get device                                                  6
 >>= \dev -> enterInformation "" []                                      7
 >>* [ OnAction (Action "Send" []) (hasValue (newMsg me o T))            8
     , OnAction (Action "Stop" []) (always (return Nothing))             9
     ] ++ if (hasCamera dev)                                             10
     [ OnAction (Action "Picture" []) (mkPict me) ] []                   11
where                                                                    12
 mkPict :: User -> Task (Maybe (Msg TextPicture))                        13
 mkPict me                                                               14
   = takePicture >>= maybe (enterPictureChat me) (newMsg me o P)         15
                                                                         16
chatTextOrPicture :: Task [Msg TextPicture]                              17
chatTextOrPicture = createChatSession enterPictureChat                   18
```

Figure 7: Chat example with support for sending pictures

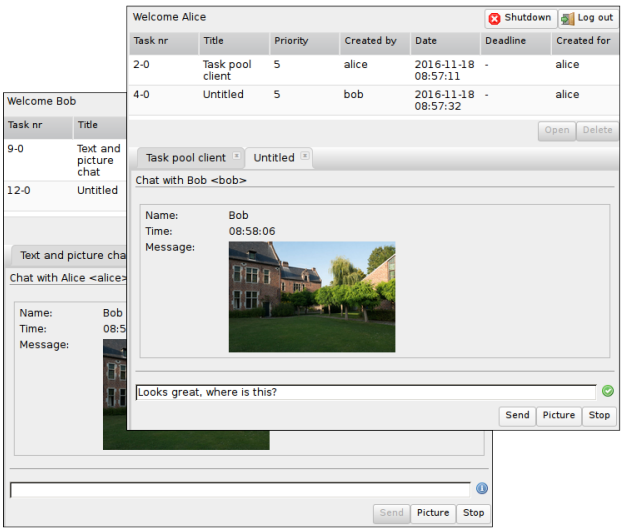


Figure 8: Alice and Bob using the extended chat task

As an example we have added a library, `Device.Camera`, that can be used to create a task with which an end user can take a JPEG picture using the device's camera. The task function returns a `Maybe Base64` that contains the picture encoded as `Base64` string or `Nothing` in the case the end-user canceled the picture (e.g. by pressing the back button). A JPEG picture can be shown to the user using the `viewInformation` editor because there is an editor defined for JPEG images in `iTasks.API.Extensions.Pictures.JPEG` module. The `Device.Location` API allows users to share the location using the location service of Android and can retrieve the current location using GPS. When the device has a public interface we can also make use of a task that connects to a TCP server that is managing the hardware, or a task that calls a web service.

An iTask Controller on a Raspberry Pi
Since a Raspberry Pi uses ARM code, we can also run an iTask controller on a Raspberry Pi. In this way we gain access to the resources available on small IoT devices such as the Raspberry Pi. By providing an interface to its resources, one can simply write tasks to be executed on the Pi. Interaction with the other tasks and SDS's is obtained for free.

4.6 Examples

4.6.1 Extended Chat Example on Android. The first example extends the chat example of Section 2.2 with an option to take a picture. Figure 7 shows the code of the extended `enterChat`, `enterPictureChat`. All other code remains the same.

To enable sending either a text message or a picture, we define the algebraic data type `TextPicture` (1). The characteristics of a device on which a task is executed, is defined in a record of type `DeviceInfo` stored in the `device` SDS. This information is read (6), and used to find out if this device has a camera (10). The task function `enterPictureChat` extends `enterChat` with an action to take a `Picture` (10–11) via the task `mkPict` (13–15). The `takePicture` task activates the camera for taking a picture. If the end-user decides not to take a picture, `enterPictureChat` is called recursively to offer all chat options again. If the end-user has taken a picture `p` then it is returned as `Just (P p)`. The infix operator `o` is function composition. We use the altered `createChatSession` described in section 4.1.

This extended chat example can run on any iTask configuration. Figure 8 shows a screen-shot of Alice and Bob, where Bob has taken a picture with the camera on his Android tablet running an iTask App, and has sent this picture to `Alice`.

4.6.2 Measuring Temperature on a Raspberry Pi. In Figure 9 we show an example of a task that must be executed on a Raspberry Pi that is equipped with a temperature sensor connected to its GPIO pins (for the nitty-gritty Raspberry Pi details how that works we refer to [11]). On the Raspberry Pi, the measured temperature is stored in a file. However, the file system of the Raspberry is not accessible via a Web API. One could create a small service using e.g. TCP to make the information available, or implement an SSH client to obtain the file content. However, in the new distributed iTasks system it suffices to install a local controller on the Raspberry Pi, and ship tasks to it for execution.

The task `showTemperature` creates an in-memory SDS for storing a temperature (5), creates a view on that SDS (6), and starts a parallel task on the Raspberry Pi (7). Every 15 seconds the task reads the temperature from the temperature sensor file, and stores it in the in-memory SDS that is located remotely on the host (14–18). Meanwhile the host continuously displays the current value of the in-memory SDS. Figure 10 shows a screen-shot of the host task. The `viewSharedInformation` task is customized with the `thermometer` function that describes how to display the `Temp` type as a SVG image using the SVG extension [2] of iTask.

4.7 Properties of the Distributed iTask Architecture

With the distributed, dynamic, iTask architecture TOP programs can make effective use of a distributed application domain. This means that it is now possible to distribute tasks over any number of devices, move tasks around over these devices, identify information

```
:: Temp :== Maybe Real

showTemperature :: Task Temp
showTemperature
 = withShared Nothing (\shareTemp ->
         viewSharedInformation title [thermometer] shareTemp
     -|| (myPi @: forever (readTemp interval shareTemp))
   )
where
 title    = "Temperature is:"
 myPi     = Requires "RaspberryPiWithTemperatureSensor"
 interval = {Time | hour = 0, min = 0, sec = 15}

readTemp :: Time (Shared Temp) -> Task Temp
readTemp interval shareTemp
 =            waitForTimer interval
 >>|          readTempFileAndConvert "TemperatureSensorFile"
 >>= \temp -> set temp shareTemp
```

Figure 9: Example showing measuring of the temperature

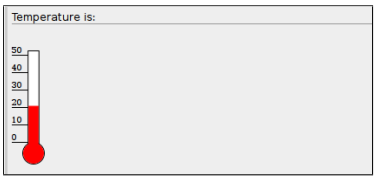


Figure 10: Host showing temperature measured on Raspberry Pi

sources at organizations as shared data sources, and identify work to be performed at organizations as remote tasks. In this section we reflect on the properties of the new implementation and compare them with the properties of the original system [16].

Tasks abstract from the way a task is processed (humans or software systems), and instead emits (lack of) progress via observable task values. In general, if a task value is not changing, this does not imply that nothing is going on in the corresponding task. It is possible to find out when a task has been worked on for the first and last time via the earlier mentioned task attributes (section 2.3.1). Moreover, if you know two subsequent task values, this does not imply that you know what has been done to get to the next task value. Because the TOP event model is defined only in terms of these task values, tasks can be processed on any suitable (human or software system) processor, regardless of its execution speed, without altering the meaning of a program.

Tasks that exhibit lack of progress cannot hamper the progress of the task it is part of, unless this has been defined explicitly. Certainly when human processors are involved, tasks ensure progress by observing its sub tasks and provide a way out of a situation in which no progress is made. In the new architecture, a remote device might become temporarily unavailable. As soon as it is operational, it can resume its work and emit new task values. Again, this does not alter the meaning of a program.

Even though sub tasks might be executed remotely, the two core task combinators, *step* and *parallel*, that coordinate the compositional behavior of these sub tasks always run on a single controller in an atomic way. As a consequence, they will not behave differently from the original system. A key difference of the distributed, dynamic, architecture with the original system is that in the new system one can create arbitrary long latency. When designing a TOP application, the developer always needs to be aware of this. It must be emphasized that this is not new because also in the original system tasks need to be designed in such a way that they can handle situations in which end-users or tasks that refer to external organizations do not respond in a timely manner. This also holds for tasks and shared data sources that have been downloaded on remote devices that fail permanently.

5 RELATED WORK

With the controllers network, we have created a network of processes at different machines and connected them. One can compare this with the processes in Erlang [18]. They support a standard error recovery with the special property that, when a process terminates abnormally, the other processes are notified through the link and can respond. This mechanism even allows layering so that processes can be isolated and restarted when an error occurs. We do not have such a standard error recovery mechanism. A (local) server may terminate anytime due to e.g. a closed or crashed user device or a lost internet connection, hence we are not able to signal such a problem from the machine to the controller. However, controllers are connected to each other, so other controllers can find out that a certain connection is no longer responding. If the disconnected controller returns after a while, all can continue as before. If the controller is lost forever, we have the same ability as in Erlang to restart a task from scratch and assign it to some other controller.

Yinzhou Zhu and Baolin Yin describe an Application-level Web Component Framework for Distributed Workflow Management [20] where there is a notion of *server nodes* and *client nodes*. The clients in this system are browsers only. Task evaluation on a client is not supported. It is not clear to us what happens when a server is going offline due to a failure.

A similar approach of dividing work and working with nodes or peers is the Web Workflow Peers Directory (WWPD) system that offers a peer to peer (P2P) architecture for dynamic workflow management [6]. The system uses a list of all the peers that are available in the WWPD. In this system a peer registers itself and offers its services. The task description language they use is different from ours. The WWPD system uses the Workflow Process Description (WPD), an XML based document containing the task description and their references like URLs. The system does not send over tasks to evaluate remotely. Instead it offers a reference to the place where the task (or system) can be found. The system uses an approach similar to our approach for the task instance pool, where a server manages the task pool and knows how to find all the available clients.

Deriving an executable from a specification in the form of a workflow diagram is described in [14]. The authors derive a distributed version where the work is split in the specification. In that

case the tasks can be distributed. The iTask combinators are more general and do not rely on just splitting tasks.

Using a single source to generate code for clients and servers is also done in the Eliom [17] project that provides a framework for writing web applications. In their language it is explicitly stated in the code which part is intended for a client and which part needs to be executed on a server. In iTasks any function or task can be sent to any other server or client and this decision can be made at run-time. So, we are much more flexible. The Eliom approach has the advantage that it is statically known what can run on a client, which may be important to know for security crucial code. In our current implementation we assume that all code is available on all devices where controllers are running.

6 CONCLUSION AND FUTURE WORK

In this paper we have presented a new distributed architecture for the evaluation of iTask applications and discussed its implementation. The new architecture is a generalization of the old iTask system that had only one central server. In the new architecture we can have multiple iTask servers (controllers) running distributed over a network of platforms, on servers as well as on clients. Tasks can be pushed (assigned to someone somewhere) or pulled (using a subscription mechanism) from one controller to another, regardless of the platform they are running on.

We solved the disadvantages of the original iTasks system: 1) The system is now scalable because we can divide work over multiple server-side controllers. 2) Private client-side controllers allow an end-user to download tasks to work on them off-line. 3) We can generate client-side Android apps running in native code which enables us to create applications that perform CPU-intensive computations, thus avoiding the use of JavaScript. 4) Using an app as client also allows us to make use of any facility the Android platform offers, something which is not allowed in browsers as well.

Clean applications are fast because we generate state-of-the-art native code. The implementation of the distributed platform builds on the new ability to ship, at run-time, any Clean function for evaluation from one platform to another. A symbolic, platform independent serialization of a closure can be constructed given the current state of stacks and heap on one platform, shipped over, and de-serialized to the proper stack and heap representation of the other platform. We currently support 32-bits ARM for Android and Linux (Raspberry Pi) and 32- and 64-bits Intel code for Mac, Windows and Linux systems. In addition we can compile Clean to JavaScript code to run in browsers.

The current implementation requires that the native code images (Intel or ARM) are available on all controlling devices beforehand, either as executable or as app. In the future we want to be able to dynamically extend a running controller with the code needed for the evaluation of a closure. This means that one has to be able to extend a running Clean application with new code while it is running. We already have the infrastructure for Windows, but need to port it for the other platforms. As an alternative option we are also working on an interpreter of the platform independent ABC-code the compiler generates. It will run slower than compiled code, but is easier to port and easier to extend with new code.

The development of the distributed, dynamic, version of the iTask system made us even more enthusiastic about the underlying concepts than we already were. The specification of iTask applications is not affected by the changed architecture and remains elegant and high level. The implementation of the distributed version turned out to be very elegant as well. The major technical hurdle to take was the technological ability to ship closures to distributed Clean applications running on different platforms. Once this was achieved, it was relatively easy to turn the existing iTask system into a distributed, dynamic, version. The new technology enables the system to push tasks to a remote controller, pull tasks that satisfy a certain criteria to a local controller, and accomplish remote access to task values and SDS's. We plan to use the new distributed, dynamic, system for developing real world applications, to start with in the challenging domain of Command and Control.

REFERENCES

[1] Peter Achten, Pieter Koopman, and Rinus Plasmeijer. 2015. An Introduction to Task Oriented Programming. In *Central European Functional Programming School, Revised Selected Papers 5th Summer School, CEFP 2013, Cluj-Napoca, Romania*, Viktória Zsók, Zoltán Horváth, and Lehel Csató (Eds.). Lecture Notes in Computer Science, Vol. 8606. Springer International Publishing, 187–245.

[2] 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 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA, Article 7, 13 pages.

[3] Jan De Clercq. 2002. Single Sign-On Architectures. In *Infrastructure Security: International Conference, InfraSec 2002 Bristol, UK, October 1–3, 2002 Proceedings*, George Davida, Yair Frankel, and Owen Rees (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–58.

[4] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Editlets: Type-based, Client-side Editors for iTasks. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA, Article 6, 13 pages.

[5] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. ACM, New York, NY, USA, Article 9, 11 pages.

[6] Georgios John Fakas and Bill Karakostas. 2004. A peer to peer (P2P) architecture for dynamic workflow management. *Information and Software Technology* 46, 6 (2004), 423–431.

[7] Google. 2016. WebView. https://developer.android.com/reference/android/webkit/WebView.html. (2016). [Online; accessed 29-November-2016].

[8] Ralf Hinze. 2000. A new approach to generic functional programming. In *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL'00, Boston, MA, USA*, Tom Reps (Ed.). ACM Press, 119–132.

[9] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. 2008. *From Interpretation to Compilation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 286–301. DOI: http://dx.doi.org/10.1007/978-3-540-88059-2_8

[10] J. M. Jansen, P. W. M. Koopman, and M. J. Plasmeijer. 2006. Efficient interpretation by transforming data types and patterns to functions. In *Proceedings of the 7th symposium on trends in functional programming, TFP'06*, Henrik Nilsson (Ed.). Nottingham, UK, 157–172.

[11] Matthew Kirk. 2016. Raspberry Pi Temperature Sensor. http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/temperature/. (2016). [Online; accessed 14-November-2016].

[12] Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. 2012. Incidone: A Task-Oriented Incident Coordination Tool. In *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM '12*, Leon Rothkrantz, Jozef Ristvej, and Zeno Franco (Eds.). Vancouver, Canada.

[13] Bas Lijnse and Rinus Plasmeijer. 2011. iTasks 2: iTasks for End-users. In *Lecture Notes in Computer Science*, Marco T. Morazán and Sven-Bodo Scholz (Eds.), Vol. 6041. Springer, Berlin, 36–54.

[14] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelika Kotz Dittrich, and Gerhard Weikum. 1998. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems* 10, 2 (1998), 159–184.

[15] M. J. Plasmeijer, P. M. Achten, and P. W. M. Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th international conference on functional programming, ICFP'07*. ACM Press,

Freiburg, Germany, 141–152.
[16] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*. ACM, Leuven, Belgium, 195–206.
[17] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A core ML language for Tierless Web programming. In *Asian Symposium on Programming Languages and Systems*. Springer, 377–397.
[18] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire.
[19] A. van Weelden. 2007. *Putting types to good use*. Ph.D. Dissertation. Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands.
[20] Yinzhou Zhu and Baolin Yin. 2013. Application-level Web Component Framework for Distributed Workflow Management. *International Journal of u-and e-Service, Science and Technology* 6, 4 (2013), 199–208.

Task Oriented Programming and the Internet of Things

DRAFT

Mart Lubbers
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
mart@martlubbers.net

Pieter Koopman
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
pieter@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
rinus@cs.ru.nl

ABSTRACT

Task Oriented Programming (TOP) is an Embedded Domain Specific Language (EDSL) offered by Clean's iTask library. In iTasks one specifies the tasks, that end-users and systems collaborative have to do to accomplish a certain goal. From such a high-level abstract specification a web-server is generated which coordinates the work thus described for everybody and every system participating. For end-users dedicated web-pages are dynamically generated to guide the work to be done. Any interaction taken place on one machine commonly has direct consequences for the work others participants can see or do.

The iTasks system has originally been designed to guide end-users interacting with the server via their web browsers. But of course, also tasks to be performed by computers or computer applications can be defined. Code can be generated for regular computers like Intel and Arm processors running Windows, MacOS, Unix or Android.

Internet of Things (IoT) technology is emerging rapidly. It has been estimated that there will be around 30 billion IoT devices online in 2020. Many tasks — natural to the iTasks system — could be performed by IoT devices. However, IoT devices commonly consist of very tiny computers with limited memory, e.g. Arduino microcontrollers. It is therefore not possible to compile iTask tasks to code for such tiny machines in a standard way.

In this paper we present a novel way of programming IoT devices in such a way that they can receive tailor made tasks that express IoT functionality in a dynamic way. As a result, it means that the IoT devices are initialized with a byte code interpreter which allows dynamic reprogramming by sending appropriate byte codes. This technique avoids recompilation and thus write cycles on the program memory. Moreover, integration with the iTasks system has been created to allow the programmer to write and interact with the devices following the TOP paradigms and design patterns. As a consequence, it avoids integration problems in IoT by allowing all components to be programmed on a high-level of abstration from one source, hiding detailed knowledge of the underlying systems being used.

IFL'17, August 2017, Bristol, United Kingdom

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

KEYWORDS

Internet of Things; Functional Programming; Pure Functional Languages; Web Programming; Distributed Applications; Task-Oriented Programming; Client-Server Architecture; Platform Independent Code Generation; Clean; Haskell

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2017. Task Oriented Programming and the Internet of Things. In *Proceedings of International Symposium on Implementations and Applications of Functional Languages (IFL'17).* ACM, New York, NY, USA, 16 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Internet of Things

The Internet of Things (IoT) technology is emerging rapidly. It offers myriads of solutions and is transforming the way people interact with technology. The term IoT was coined around 1999 to describe Radio-Frequency Identification (RFID) devices and the communication between them. After a small slumber of the term, it resurfaced recently and has changed definition slightly.

In the current day and age, IoT encompasses all small devices that communicate with each other and — most of all — with the world. It has been estimated that there will be around 30 billion IoT devices online in 2020. Even today, IoT devices are already in everyone's household in the form of smart electricity meters, smart fridges, smartphones, smart watches. These devices are often equipped with sensors, Global Navigation Satellite System (GNSS) modules[1] and actuators [5]. With these new technologies, information can be tracked accurately using little power, bandwidth and money. Moreover, IoT technology is coming into healthcare as well [18]. For example, for a few euros a consumer ready fitness tracker watch can be bought that tracks heartbeat and respiration levels.

The architecture of IoT systems is often divided into layers. A very popular division is the four layer architecture but there are also proponents of a five layer structure. The first layer of the four layer architecture is the sensing layer. This is the layer containing the actual sensing and acting hardware. In — for example — a smart electricity meter, this layer would contains the sensors detecting the current drawn. There are myriads of device available to use in this layer and they can be programmed using a variety of different low level programming languages such as C++ and C, but also higher level languages such as Python and LUA. The second layer of IoT is the networking layer and is responsible for connecting

[1] e.g. the American GPS or the Russian GLONASS.

the first layer with the outer world. In a smart electricity meter, this would be the GSM modem connecting the meter to a server. Existing networking techniques — e.g. WiFi and GSM — are used to convey IoT information but there are also specialized communication techniques devised for IoT such as ZigBee, LoRa and Bluetooth Low Energy. These latter are so called Low Power Low Throughput networks and are characterized by the low bandwidth but also low resources required. The third layer is called the service layer. This layer is responsible for all the servicing and business rules surrounding the application. It provides Application Programming Interfaces (APIs) and interfaces to, and storage of the data. Finally, the fourth layer is the application layer. This final layer provides the applications that the user can use to interact with the IoT devices and their data. In a smart electricity meter, this layer would be the app that can be used to monitor the electricity consumption.

The separation of IoT in layers is a difficulty when developing IoT applications. All layers use different paradigms, languages and architectures which leads to isolated logics for the components. This which makes it difficult to integrate the layers. Moreover, rolling out changes to the system on the device's side is expensive since reprogramming microcontrollers in the field is very expensive. This process is expensive since the devices are in the field and the device's program memory has as fairly limited number of accepted write cycles. Even the changing a few parameters on a device often requires a full reprogram.

1.2 TOP — Task Oriented Programming

The Task Oriented Programming (TOP) paradigm and the corresponding iTasks implementation offer a high abstraction level for real world workflow tasks [16]. These workflow tasks can be described through an Embedded Domain Specific Language (EDSL) hosted in the purely functional programming language Clean. Tasks are the basic building blocks of the language and they resemble actual workflow tasks. For the specification, the system will generate a multi-user web application. This web service can be accessed through a browser and is used to complete these Tasks. Familiar workflow patterns like sequential, parallel and conditional Tasks chaining can be modelled in the language. iTasks has shown to be useful in many fields of operation such as incident management [13]. iTasks is highly type driven and is built on generic functions that generate functionality for the given types. This results in the programmer having to do very little implementation work on details such as user interfaces. It is possible to change the derived functions and adapt them to needs.

1.3 Controlling the IoT with TOP

Tasks in the iTasks system are modelled after real life workflow tasks but the modelling is applied on a high level. Therefore, it is difficult to connect iTasks-Tasks to real world tasks and allow them to interact. A lot of the actual tasks could very well be performed by IoT devices. Nevertheless, adding IoT devices to the current system is difficult — to say the least — as it was not designed to cope with these devices.

In the current system, adapters connecting devices to iTasks — in principle — can be written in two ways.

First, an adapter for a specific device can be written as a SDS[2]. SDSs can interact with the world and thus with hardware, allowing communication with any type of device. However, this requires a tailor-made SDS for every specific device and functionality and does not allow logic to be changed without a reprogram of the client and possibly even the server. Once a device is programmed to serve as an SDS, it has to behave like that forever. Thus, this solution is not suitable for systems that have changing logic.

The second method uses the novel contribution to iTasks by Oortgiese et al. They lifted iTasks from a single server model to a distributed server architecture [14]. As a proof of concept, an android app has been created that runs an entire iTasks core and is able to receive Task from a different server and execute them. While android often runs on small ARM devices, they are a lot more powerful than the average IoT microcontroller. The system is suitable for dynamically sending Tasks but running the entire iTasks core on a microcontroller is not feasible. Even if it would be possible, this technique would still not be suitable because a lot of communication overhead is needed to transfer the Tasks. IoT devices are often connected to the server through Low Power Low Bandwidth which is just not fitted for sending lots of data.

In this paper we present a novel system that bridges the gap between the aforementioned solutions for adding IoT to iTasks. The system consists of updates to the mTask-EDSL [12], a new communication protocol, device application and an iTasks server application. The system supports devices as small as Arduino microcontrollers[3] and operates via the same paradigms and patterns as regular Tasks in the TOP paradigm. Devices in the mTask-system can run small imperative programs written in the EDSL and have access to SDSs. Tasks are sent to the device at runtime, avoiding recompilation and thus write cycles on the program memory. This solution extends the reach of iTasks and allows closer resemblance of Task to actual tasks. Moreover, it tries to solve some integration problems in IoT by allowing all components to be programmed from one source.

1.4 Related Work

Similar research has been conducted on the subject. For example, microcontrollers such as the Arduino can be remotely controlled very directly using the Firmata-protocol[4]. This protocol is designed to allow control of the peripherals — such as sensors and actuators — directly through commands sent via a communication channel such as a serial port. This allows very fine grained control but with the cost of excessive communication overhead since no code is executed on the device itself, only the peripherals are queried. A Haskell implementation of the protocol is also available[5]. The hardware requirements for running a Firmata client are very low. However, the communication requirements are high and therefore it is not suitable for IoT applications that operate through specialized IoT networks which often only support low bandwidth.

[2] Similar as to resources such as time are available in the current iTasks implementation.
[3] "Arduino - Open Source Products for Electronic Projects." http://www.arduino.org/
[4] "firmata/protocol: Documentation of the Firmata protocol." (https://github.com/firmata/protocol). [Accessed: 23-May-2017].
[5] "hArduino by LeventErkok." (https://leventerkok.github.io/hArduino). [Accessed: 23-May-2017].

Clean has a history of interpretation and there is a lot of research happening on the intermediate language SAPL. SAPL is a purely functional intermediate language that is designed to be interpreted. It has interpreters written in C++ [11] and Javascript [6]. Compiler backends exist for and Clean and Haskell which compile the respective code to SAPL [8]. The SAPL language is still a functional language and therefore requires big stacks and heaps to operate and is therefore not directly suitable for devices with little RAM such as the Arduino Uno which only boasts 2*K* of RAM. It might be possible to compile the SAPL code into efficient machine language or C but then the system would lose its dynamic properties since the microcontroller then would have to be reprogrammed every time a new Task is sent to the device.

EDSLs have often been used to generate C code for microcontroller environments. This work uses parts of the existing mTask-EDSL which generates C code to run a TOP-like system on microcontrollers [17] [12]. Again, this requires a reprogramming cycle every time the Task-specification is changed. Hence, the EDSL is used but the backend is not suitable for the purpose of dynamic IoT solutions.

Another EDSL designed to generate low-level high-assurance programs is called Ivory and uses Haskell as a host language [9]. The language uses the Haskell type-system to make unsafe languages type safe. For example, Ivory has been used in the automotive industry to program parts of an autopilot [15] [10]. Ivory's syntax is deeply embedded but the type system is shallowly embedded. This requires several Haskell extensions that offer dependent type constructions. The process of compiling an Ivory program happens in two stages. The embedded code is transformed into an AST that is sent to a chosen backend. The technique used in the novel system using the mTask-EDSL is different, in the new system, the EDSL is transformed directly into functions and there is no intermediate AST. Moreover, Ivory generates static programs and thus it is necessary to reprogram the devices when they need to be repurposed. It would be interesting to explore the possibilities of writing the client software in an EDSL as well.

Not all IoT devices run solely compiled code. The popular ESP8266 powered NodeMCU is able to run interpreted LUA code. Moreover, there is a variation on Python called micropython that is suitable for running on microcontrollers. However, the overhead of the interpreter for such rich languages often results into limitations on the program size. It would not be possible to repurpose a device with IoT because implementing this extensibility in the interpreted language leaves no room for the actual programs. Also, some devices only have 2*K* of ram, which is not enough for this.

2 ITASKS

TOP is a novel programming paradigm implemented as iTasks [1] in the pure lazy functional language Clean [2]. iTasks is an EDSL to model workflow tasks in the broadest sense. A Task is *just* a function that — given some state — returns the observable `TaskValue`. The `TaskValue` of a `Task` can have different states. Not all state transitions are possible as shown in Figure 1. Once a value is stable it can never become unstable again. Stability is often reached by pressing a confirmation button. Tasks yielding a constant value are immediately stable. Combinators are available to combine Tasks in sequence and parallel to resemble real workflow. For all Tasks, a web interface is generated that can be used to complete and observe the Tasks. For a type to be suitable, it must have instances for a collection of generic functions that is captured in the class `iTask`. Basic types have specialization instances for these functions and show an interface accordingly. Derived interfaces can be modified with decoration operators or specializations can be created.

Figure 1: The states of a `TaskValue`

2.1 Shared Data Sources

SDSs are an abstraction over resources that are available in the world or in the iTasks system. For example, shared data can be a file on disk, the system time or a random integer. The actual SDS is just a record containing functions on how to read and write the source. In these functions the `*IWorld` — which in turn contains the real `*World` — is available. Accessing the outside world is required for interacting with it and thus the functions can access files on disk, raw memory, other SDSs and hardware.

The basic atomic operations for SDSs are get, set and update of which the signatures are shown in Listing 1. By default, SDSs are files containing a JSON encoded version of the object and thus are persistent between restarts of the program. The framework provides useful functions to transform, map and combine SDSs using combinators and to watch the value and act upon a change Tasks waiting on an SDS to change are notified when needed which results in low resource usage because Tasks are never constantly inspecting SDS values but are notified.

```
:: RWShared p r w = { ... } // read and write functions
:: Shared r :== RWShared () r r

get ::          (RWShared () r w) -> Task r | iTask r
set :: w        (RWShared () r w) -> Task w | iTask w
upd :: (r -> w) (RWShared () r w) -> Task w | iTask r & iTask w

sharedStore :: String a -> Shared a | JSONEncode{|*|}, JSONDecode{|*|}
```

Listing 1: SDS functions

3 PARAMETRIC LENSES

SDSs can contain complex data structures such as lists, trees and even resources in the outside world. Sometimes, an update action only updates a part of the resource. When this happens, all waiting Tasks looking at the resource are notified of the update. However, it may be the case that Tasks were only looking at parts of the structure that was not updated. To solve this problem, parametric lenses were introduced [7].

Parametric lenses add a type variable to the SDS. This type variable is fixed to the void type (i.e. `()`) in the library functions. When an SDS executes a write operation, it also provides the system with a notification predicate of the type `p -> Bool` where `p` is the parametric lens type. This function then checks whether the Tasks waiting for a notification are eligible to be notified. This allows for the creation of big SDS, and have Tasks only look at parts of the big SDS

Functionality for setting parameters is available in the system. The most important functions are the sdsFocus and the sdsLens function. These functions are listed in Listing 2. sdsFocus allows the programmer to fix a parametric lens value. sdsLens is a kind of mapReadWrite including access to the parametric lens value. This allows the creation of — for example — SDSs that only read and write to parts of the original SDS.

```
:: SDSNotifyPred p      :== p -> Bool
:: SDSLensRead p r rs     = SDSRead   (p -> rs -> MaybeError TaskException r
      )
:: SDSLensWrite p w rs ws = SDSWrite  (p -> rs -> w -> MaybeError
      TaskException (Maybe ws))
:: SDSLensNotify p w rs   = SDSNotify (p -> rs -> w -> SDSNotifyPred p)

sdsFocus :: p (RWShared p r w) -> RWShared p` r w | iTask p
sdsLens :: String (p -> ps) (SDSLensRead p r rs) (SDSLensWrite p w rs ws) (
      SDSLensNotify p w rs) (RWShared ps rs ws) -> RWShared p r w | iTask ps
```

Listing 2: Parametric lens functions

4 EMBEDDED DOMAIN SPECIFIC LANGUAGES

An EDSL is a language embedded in a host language. EDSLs can have one or more backends or views. Commonly used views are pretty printing, compiling, simulating, verifying and proving the program. There are several techniques available for creating EDSLs. They all have their own advantages and disadvantages in terms of extendability, typedness and multiple view support. The main techniques are as follows:

4.1 Deep Embedding

Deep embedding is representing a language as an Algebraic Datatype (ADT) in the host language. Views are functions that do a transformation from, to or within the datatype. As an example, take the expressio EDSL shown in Listing 3.

```
:: DSL
  = LitI  Int      | LitB  Bool
  | Var   String   | Plus  DSL DSL
  | Minus DSL DSL  | And   DSL DSL
  | Eq    DSL
```

Listing 3: An expression deeply EDSL

Deep embedding has the advantage that it is easy to create and views are easy to add. To the downside, the expressions created with this language are not guaranteed type-safe. It is possible to create an expression such as `Plus (LitI 4) (LitB True)` which is illegal. Extending the ADT is simple but extending the views accordingly is difficult because it has to be done for all views.

The lack of type-safety in this method could be tackled by using GADTs [4]. Listing 4 shows the same language but using with GADTs. GADTs are not supported in the current version of Clean and therefore the syntax is hypothetical. However, it has been shown that GADTs can be simulated using bimaps or projection pairs [3]. While GADTs solve the typing problem, the lack of extendability remains.

```
:: DSL a
  =     LitI  Int                -> DSL Int
  |     LitB  Bool               -> DSL Bool
  |∃e: Var   String             -> DSL e
  |     Plus  (DSL Int) (DSL Int)   -> DSL Int
  |     Minus (DSL Int) (DSL Int)   -> DSL Int
  |     And   (DSL Bool) (DSL Bool) -> DSL Bool
  |∃e: Eq (DSL e) (DSL e)          -> DSL Bool &  == e
```

Listing 4: An expression deeply EDSL with GADTs

4.2 Shallow Embedding

In shallowly EDSL the language constructs are expressed as functions in the host language. Subsequently, the view is embedded in the functions. The same language as a shallowly EDSL is shown in Listing 5. Much of the internals of the implementation can be hidden using monads.

The advantage of shallow embedding is t he extendability. It is very easy to add functionality by just adding a function. Compile time checks of the host language guarantee functionality checks and type safety.

The downside of shallow embedding is extensibility of views. It is nearly impossible to add views to a shallowly embedded language. Extra views can be hacked in by decorating the datatype with results of all views. This means that every component executes all views which renders it slow for multiple views and complex to implement.

```
:: Env   = ...           // Some environment
:: DSL a = DSL (Env -> a)

Lit :: a -> DSL a
Lit x = λe -> x

Var :: String -> DSL Int
Var i = λe -> retrEnv e i

Plus :: (DSL Int) (DSL Int) -> DSL Int
Plus x y = λe -> x e + y e

...

Eq :: (DSL a) (DSL a) -> DSL Bool | == a
Eq x y = λe -> x e + y e
```

Listing 5: An expression shallow EDSL

4.3 Class Based Shallow Embedding

Class based shallow embedding has the advantages of both shallow and deep embedding [19]. Here, the language constructs are defined as type classes and a view is a type implementing some of the classes. Listing 6 shows this with the expression EDSL with two views.

Adding views is easy because it just means adding a type implementing some of the classes. Moreover, type safety is guaranteed in class based shallow embedding because the types can contain phantom types and constraints can be enforced by the type signatures of the class functions. Lastly, extensions can be added easily. Existing views do not need to be updated when an extension is added in a new class. If the extension is added in an existing class, all the views implementing the class need to be updated.

```
:: Env   = ...           // Some environment
:: Evaluator a = Evaluator (Env -> a)
:: PrettyPrinter a = PP String

class intArith where
  lit   :: t -> v t            | toString t
  add   :: (v t) (v t) -> (v t) | + t
```

```
  minus :: (v t) (v t) -> (v t) | - t

class boolArith where
  and :: (v Bool) (v Bool) -> (v Bool)
  eq  :: (v t)    (v t)    -> (v Bool) | == t

instance intArith Evaluator where
  lit x   = Evaluator λe->x
  add x y = Evaluator ...

instance intArith PrettyPrinter where
  lit x   = PP $ toString x
  add x y = PP $ x +++ "+" +++ y
  ...
```

Listing 6: An expression class based shallowly EDSL

5 MTASK

The mTask-EDSL — created by Koopman et al. — is a class based shallowly EDSL designed to generate ready-to-compile TOP-like programs for microcontrollers such as the Arduino [12][17]. Another iTasks backend is available for simulation. Since the EDSL uses class based shallow embedding, adding new views is simple. The parts of the EDSL that have not been used in the novel system will not be discussed but the details can be found in the cited literature.

A view for the mTask-EDSL is a type with two free type variables[6] that implements some of the classes given. The types do not have to be present as fields in the view and can, and will most often, be phantom types. Thus, views are of the form: `:: v t r = ...`. The `v` represents the view, the `t` the type and the `r` the role of the expression. Possible roles are expressions, statements, assignables, tasks or functions.

5.1 Expressions

There exist boolean and arithmetic expressions in mTask-EDSL. The class of arithmetic language constructs also contains the function `lit` that lifts a host-language value into the EDSL domain. The classes for this are listed in Listing 7 but some functionality and class constraints are omitted for brevity.

```
class arith v where
  lit          :: t -> v t Expr
  (+.) infixl 6 :: (v t p) (v t q) -> v t Expr          | +, zero t
  (-.) infixl 6 :: (v t p) (v t q) -> v t Expr          | -, zero t
  ...
class boolExpr v where
  Not          :: (v Bool p) -> v Bool Expr             | ...
  (&.) infixr 3 :: (v Bool p) (v Bool q) -> v Bool Expr | ...
  ...
  (==.) infix 4 :: (v t p) (v t q) -> v Bool Expr       | == t
```

Listing 7: Basic classes for expressions

5.2 Control flow

The main control flow operators are the sequence operator and the *if* statement which are both shown in Listing 8. The `IF` class represents the conditional execution whereas the `seq` class provides sequential execution.

[6] kind *->*->*.

```
class IF v where
  IF :: (v Bool p) (v t q) (v s r) -> v () Stmt | ...
  (?) infix 1 :: (v Bool p) (v t q) -> v () Stmt | ...

class seq v where
  (:.) infixr 0 :: (v t p) (v u q) -> v u Stmt | ...
```

Listing 8: Control flow operators

5.3 Input/Output

Values can be assigned to by using the `assign` class under the condition that the role of the lefthand side is `Upd`. Creating expressions with an `Upd` role is possible for variables, GPIO pins and other input/output peripherals. The classes facilitating this are shown in Listing 9. The value of assignable entities can be changed by using the assignment operator `=.` or using the read and write functions.

```
:: DigitalPin = D0 | D1 | D2 | D3 | D4 | D5 | D6 | ...
:: AnalogPin  = A0 | A1 | A2 | A3 | A4 | A5
:: UserLED    = LED1 | LED2 | LED3

class dIO v where dIO :: DigitalPin -> v Bool Upd
class aIO v where aIO :: AnalogPin -> v Int Upd
class analogRead v where
  analogRead   :: AnalogPin -> v Int Expr
  analogWrite  :: AnalogPin (v Int p) -> v Int Expr
class digitalRead v where
  digitalRead  :: DigitalPin -> v Bin Expr
  digitalWrite :: DigitalPin (v Bool p) -> v Int Expr

class assign v where
  (=.) infixr 2 :: (v t Upd) (v t p) -> v t Expr | ...
```

Listing 9: Input/Output classes

A way of storing data in mTask programs is using SDSs. SDSs serve as global variables in mTask and maintain their value across executions and can be used by multiple Tasks to share data. The classes associated with SDSs are listed in Listing 10. The `Main` type is introduced to box an mTask and make it recognizable by the type system by separating programs and decorations such as SDSs. The type signature is complex and uses infix type constructors and therefore, an implementation example is also given.

```
:: In a b = In infix 0 a b
:: Main a = {main :: a}

class sds v where
  sds :: ((v t Upd) -> In t (Main (v c s))) -> (Main (v c s)) | ...

sdsExample :: Main (v Int Stmt)
sdsExample = sds λx.0 In {main= x =. x +. lit 42 }
```

Listing 10: SDSs in mTask

5.4 Class extensions

In the Arduino ecosystem, shields are amply available to plug in and add functionality such as Bluetooth, WiFi, Ethernet, LCD screen, and much more. The functionality for these shields is often housed in a C++ class. Porting this functionality to mTask requires little work and can almost be done by mapping the class functions on a new mTask class. For illustration, Listing 11 and 12 show the class in the mTask EDSL and in the Arduino class.

```
:: LCD = ...

class lcd v where
  begin      :: (v LCD Expr) (v Int p) (v Int q) -> v () Expr
  LCD        :: Int Int [DigitalPin] ((v LCD Expr) -> Main (v b q))
      -> Main (v b q)
  ...
  scrollLeft  :: (v LCD Expr) -> v () Expr
  scrollRight :: (v LCD Expr) -> v () Expr
  ...
```

Listing 11: Adding the LCD to the mTask language

```
class LiquidCrystal {
public:
  void begin(uint8_t cols, uint8_t rows);
  ...
  void scrollDisplayLeft();
  void scrollDisplayRight();
  ...
}
```

Listing 12: Functions from the Arduino LCD library

5.5 Scheduling strategy

The code generation backend of the mTask-system also contains an engine for executing the tasks according to certain rules and execution strategies. mTask-Tasks do not behave like functions but more like iTasks-Tasks. An mTask is queued when either its timer runs out or when it is launched by another mTask. When an mTask is queued it does not block the execution and it will return immediately while the actual code will be executed anytime in the future.

This engine — expressed in pseudocode — is listed as Algorithm 1. All the Task are inspected on their waiting time. When the waiting time has not passed; the delta is subtracted and the Task gets pushed to the end of the queue. When the waiting has surpassed they are executed. When an mTask opts to queue another mTask it can just append it to the queue.

Data: queue queue, **time** t, t_p
$t \leftarrow$ now();
begin
  while *true* **do**
    $t_p \leftarrow t$;
    $t \leftarrow$ now();
    if *notEmpty*(*queue*) **then**
      task $\leftarrow$ queue.pop();
      task.wait $\leftarrow$ *task*.wait $-(t - t_p)$;
      if *task.wait* $> t_0$ **then**
        queue.append(*task*);
      else
        run_task(*task*);
      end
    end
  end
end

Algorithm 1: Engine pseudocode for the C- and iTasks-view

To achieve this in the EDSL a `mtask` class is added that works in a similar fashion as the `sds` class. This class is listed in Listing 13. Tasks can have an argument and always have to specify a delay or waiting time. The type signature of the `mtask` is complex and an example is given for illustration. The code shows a simple system specification containing one Task that increments a value indefinitely every second.

```
class mtask v a where
  task :: (((v delay r) a->v MTask Expr)->In (a->v u p) (Main (v t q))) ->
      Main (v t q) | ...

count = task λcount = (λn.count (lit 1000) (n +. lit 1)) In
  {main = count (lit 1000) (lit 0)}
```

Listing 13: The classes for defining Tasks

5.6 Example

Example mTask-Tasks — using almost all of the functionality — are shown in Listing 14. The `blink` mTask show the classic Arduino blinking LED application that blinks the builtin LED. The `thermostat` expression will enable a digital pin powering a cooling fan when the analog pin representing a temperature sensor is too high using the read write functionality. `thermostat`` shows the same expression but now using the assignment style. The `thermostat` example also shows that it is not necessary to run everything as a Task. The main program code can also just consist of the contents of the root `main` itself. Finally a thermostat example is shown that also displays the temperature on its LCD while regulating the temperature.

```
a0 = aIO A0
d0 = dIO D0

blink = task λblink.(λx.
    IF (x ==. lit True) (ledOn LED0) (ledOff LED0) :.
    blink (lit 1000) (Not x)
  In {main=blink (lit 1000) True}

thermostat = {main = digitalWrite D0 (analogRead A0 >. lit 50) }
thermostat` = {main = d0 =. a0 >. lit 50 }

thermostat`` = task λth.(λlcd.
    d0 =. a0 > lit 50 :.
    print lcd a0      :.
    th (lit 1000) lim   ) In
  LCD 16 12 [] λlcd.{main = th (lit 1000) lim }
```

Listing 14: Example mTask specifications

6 SYSTEM OVERVIEW

A novel system has been built which will be described in the following sections. This system provides a bridge between the gap present in the current iTasks system explained in the introduction regarding integration with IoT. It provides a framework to outsource Tasks to IoT-devices without needing to recompile the code. The Tasks targeted at IoT devices are compiled at runtime to bytecode which is sent to the device for interpretation.

6.1 EDSL for IoT Tasks

Not all Tasks are suitable to run on an IoT-device and therefore an EDSL is used to offer a constrained language that expresses Tasks for the IoT component. The mTask-EDSL provides the language

to create imperative programs suitable to run on microcontrollers. The EDSL's main view is a C code generator who's code compiles to Arduino compatible microcontrollers. The big downside of this approach is the stiffness of the system, once the code has been generated and the microcontroller has been programmed, nothing can be changed to it anymore. IoT-devices often have a limited amount of write cycles on their program memory available and therefore it is very expensive to keep recompiling and reprogramming the chips. To solve this problem, a new view is added for the mTask-EDSL which compiles the expressions to bytecode which is interpetable and thus the need for reprogramming is removed which makes the application more suitable for dynamic environments. The mTask-EDSL is not perfectly suitable and therefore some classes have been added. Moreover, not all of the classes are used.

6.2 Server

The added view for the mTask-EDSL does not result in a self-contained system but compiles the expressions to bytecode representing a single Task. The device and the server communicate using the Leader/Follower principle[7]. Only the server can initiate a connection with a device and only the server can produce and send Tasks to the device. Familiar iTasks concepts such as SDSs are available on the device and are the main use of communication between the client and the server.

7 MTASK EXTENSIONS

The Tasks suitable for a client are called mTask-Task and are written in the mTask-EDSL. Some functionality of the original mTask-EDSL will not be used in this system. Conversely, some functionality needed was not available in the existing EDSL. Due to the nature of class based shallow embedding this obstacle is easy to solve. A type — housing the EDSL — does not have to implement all the available classes. Moreover, classes can be added at will without interfering with the existing views.

7.1 Scheduling

The current mTask engine for devices does not support Tasks in the sense that the C-view does. Tasks used with the C-view are a main program that executes code and launches Tasks. It was also possible to just have a main program. The current mTask-system only supports main programs which are the Tasks in their entirety. However, this results in a problem sincy Tasks can not call other Tasks nor themselves in this way. Therefore, execution strategies have been added that accompany a Task. This strategy can be one of the following three strategies:

The `OneShot` strategy consists of executing the Task only once. In IoT applications, often the status of a peripheral or system has to be queried only once on the request of the user. For example in a thermostat, the temperature is logged every 30 minutes. However, the user might want to know the temperature at that exact moment, and then they can just send a `OneShot` Task probing the temperature. After execution, the Task will be removed from the memory of the client.

[7] Also known as Master/Slave

`OnInterval` is a execution strategy that executes the Task in the given number of milliseconds. This strategy is very useful for logging measurements on an interval. Moreover, the strategy can be (ab)used to simulate recursion. SDSs store global information and is persistent. An added `retrn` instruction can then be used to terminate. Therefore, Task can be crafted that recursively call themselves using a SDS to simulate arguments.

Finally, the `OnInterrupt Int` scheduling method is available that executes a Task when a given interrupt is received. This method is useful to launch a Task on the press on hardware events such as the press of a button. Unfortunately, due to time constraints and focus, this functionality is only built in the protocol, none of the current client implementations support this.

7.2 SDSs

mTask-SDSs on a client are available on the server as in the form of regular iTasks-SDS. However, the same freedom that an SDS has in the iTasks-system is not given for SDS that reside on the client. Not all types are suitable to be located on a client, simply because it needs to be representable on clients and serializable for communication. Moreover, SDSs behave a little different in an mTask device compared to in the iTasks system. In an iTasks system, when the SDS is updated, a broadcast to all watching Tasks in the system is made to notify them of the update. SDSs can update often and the update might not be the final value it will get. Implementing the same functionality on the mTask client would result in a lot of expensive unneeded bandwidth usage. Therefore a device must publish the SDS explicitly to save bandwidth. Note that this means that the SDS value on the device can be different compared to the value of the same SDS on the server.

To add this functionality, the `sds` class could be extended. However, this would result in having to update all existing views that use the `sds` class. Therefore, an extra class is added that contains the extra functionality. Programmers can choose to implement it for existing views in the future but are not obliged to. The publication function has the following signature:

```
class sdspub v where
  pub :: (v t Upd) -> v t Expr | type t
```

Listing 15: The sdspub class

SDSs in the mTask-EDSL are always attached to a `Main` component. Thus, they are not usable in the Task domain. To solve this, the SDSs found in the `Main` object are instantiated as real SDS in the server. This poses a problem of naming because SDSs in the mTask-EDSL are always anonymous at runtime. There is no way of labeling it since it is not a real entity, it is just a function. When SDS is instantiated and communicated with the device, they must be retrievable and identifiable. Internally this identification happens through numeric identifiers, but this is handy for programmers since. Therefore, an added class named `namedsds` is added that provides the exact same functionality as the SDS class but adds a `String` parameter that can later be used to identify an SDS in the bag of instantiated SDSs that result from compilation. The types for this class are shown in Listing 16. Again, an example is added for illustration.

```
class namedsds v where
```

```
  namedsds :: ((v t Upd) -> In (Named t String) (Main (v c s))) -> (Main (v
      c s)) | ...
:: Named a b = Named infix 1 a b

sdsExample :: Main (v Int Stmt)
sdsExample = sds λx.0 Named "xvalue" In
  {main= x =. x +. lit 42 }
```

Listing 16: The namedsds class

8 BYTECODE COMPILATION VIEW

The mTask-Tasks are sent to the device in bytecode and are saved in the memory of the device. To compile the EDSL code to bytecode, a view is added to the mTask-system encapsulated in the type `ByteCode`. As shown in Listing 17, the `ByteCode` view is a boxed Reader Write StateTransformer monad (RWST) that writes bytecode instructions (`BC`) while carrying around a `BCState`. The state is kept between compilations and is unique to a device. The state contains fresh variable names and a register of SDSs that are used.

Types implementing the mTask classes must have two free type variables. Therefore the RWST is wrapped with a constructor and two phantom type variables are added. This means that the programmer has to unbox the `ByteCode` object to be able to make use of the RWST functionality such as return values. Tailor made access functions are used to achieve this with ease. The fresh variable stream in a compiler using a RWST is often put into the *Reader* part of the monad. However, not all code is compiled immediately and later on the fresh variable stream cannot contain variables that were used before. Therefore this information is put in the state which is kept between compilations.

Not all types are suitable for usage in bytecode compiled programs. Every value used in the bytecode view must fit in the `BCValue` type which restricts the content. Most notably, the type must be bytecode encodable. A `BCValue` must be encodable and decodable without losing type or value information. At the moment a simple encoding scheme is used that uses single byte prefixes to detect the type of the value. The devices know these prefixes and can apply the same detection if necessary. Note that `BCValue` uses existentially quantified type variables and therefore it is not possible to derive class instances such as `iTasks`, thus tailor-made instances for these functions have been made.

```
:: ByteCode a p = BC (RWS () [BC] BCState ())
:: BCValue =∃e: BCValue e & mTaskType, TC e
:: BCShare = {sdsi   :: Int, sdsval :: BCValue , sdsname :: String}
:: BCState = {freshl :: Int, freshs :: Int,      sdss    :: [BCShare]}

class toByteCode a :: a -> String
class fromByteCode a :: String -> a
class mTaskType a | toByteCode, fromByteCode, iTask, TC a

instance toByteCode Int, ... , UserLED, BCValue
instance fromByteCode Int, ... , UserLED, BCValue

instance arith ByteCode
...
```

Listing 17: Bytecode view

8.1 Instruction Set

The instruction set is given in Listing 18 and contains the standard instructions for arithmetics, control flow and specialized instructions for peripherals. The instruction set is kept large, but the number of instructions stays under 255 to get as much expressive power while keeping all instruction within one byte.

The interpreter running in the client is a stack machine. The virtual instruction `BCLab` is added to allow for an easy implementation of jumping. However, this is not a real instruction and the labels are resolved to actual program memory addresses in the final step of compilation to save instructions and avoid label lookups at runtime.

```
:: BC = BCNop
  | BCLab Int           | BCPush BCValue      | BCPop
  | BCSdsStore BCShare | BCSdsFetch BCShare | BCSdsPublish BCShare
  | BCNot
  | BCAdd               | BCSub               | ...
  | BCAnd               | BCOr                | ...
  | BCEq                | BCNeq               | ...
  | BCJmp Int           | BCJmpT Int          | BCJmpF Int
  | BCLedOn            | BCLedOff
  | BCAnalogRead Pin    | ...
  | BCReturn
```

Listing 18: Bytecode instruction set

8.2 Arithmetics & Peripherals

Since the `ByteCode` type is just a boxed RWST and therefore access to the whole range of RWST functions is available. However, to use this, the type must be unboxed and boxed again after application. Most of the classes only use helper functions and the implementation is very straightforward. Listing 19 shows some implementations for the classes. Other classes, like `boolExpr` are implemented in the same fashion.

```
tell` (BC l) x = BC $ tell x
op2   (BC x) (BC y) o = BC $ x >>| y >>| tell o
op    (BC x) o = BC $ x >>| tell o

instance arith ByteCode where
  lit x = tell` [BCPush (BCValue x)]
  (+.) x y = op2 x y [BCAdd]
  (-.) x y = op2 x y [BCSub]
  ...

instance userLed ByteCode where
  ledOn l = op l [BCLedOn]
  ...
```

Listing 19: Bytecode view implementation for arithmetic and peripheral classes

8.3 Control Flow

Implementing the sequence operator is very straightforward in the bytecode view. The function just sequences the two RWSTs. The implementation for the *IF* statement speaks for itself in Listing 20. First, all the labels are gathered after which they are placed in the correct order in the bytecode sequence. It can happen that multiple labels appear consecutively in the code. This is not a problem since the labels are resolved to real addresses later on anyway.

```
freshlabel = get
  >>= λst=:{freshl}->put {st & freshl=freshl+1} >>| pure freshl
```

```
instance IF ByteCode where
  IF b t e = BCIfStmt b t e
  (?) b t = BCIfStmt b t noOp

BCIfStmt (BC b) (BC t) (BC e) = BC $
  freshlabel >>= λelse->freshlabel >>= λendif->
  b >>| tell [BCJmpF else] >>|
  t >>| tell [BCJmp endif, BCLab else] >>|
  e >>| tell [BCLab endif]

instance noOp ByteCode where noOp = BC (pure ())
```

Listing 20: Bytecode view for the IF class

The scheduling in the mTask-Task bytecode view is different from the scheduling in the C view and some scheduling strategies let Tasks run indefinitly. To allow interval and interrupt Tasks to terminate, a return instruction is added which just tells the interpreter to stop executing and the Task will be removed from memory. The class for this is shown in Listing 21.

```
class retrn v where
  retrn :: v () Expr

instance retrn ByteCode where
  retrn = tell` [BCReturn]
```

Listing 21: Bytecode view for the return instruction

8.4 Shared Data Sources & Assignment

Fresh SDSs are generated using the state and constructing them involves multiple steps and shown in Listing 22. First, a fresh identifier is grabbed from the state. Secondly, a `BCShare` record is created with that identifier. A `BCSdsFetch` instruction is written and the body is generated to finally add the SDSs to the actual state with the value obtained from the function.

```
freshshare = get >>= λst=:{freshs}->put {st & freshs=freshs+1} >>| pure
      freshs
addSDS sds v s = {s & sdss=[{sds & sdsval=BCValue v}:s.sdss]}

instance sds ByteCode where
  sds f = {main = BC (freshshare
      >>= λsdsi->pure {BCShare|sdsname="",sdsi=sdsi,sdsval=BCValue 0}
      >>= λsds->pure (f (tell` [BCSdsFetch sds]))
      >>= λ(v In bdy)->modify (addSDS sds v)
      >>| unBC (unMain bdy))}

instance sdspub ByteCode where
  pub (BC x) = BC (censor (λ[BCSdsFetch s]->[BCSdsPublish s]) x)
```

Listing 22: Bytecode view for sds

All assignable types compile to an RWST which writes the specific fetch instruction(s). For example, using an SDS always results in an expression of the form `sds x=4 In ...` in which the `x` the RWST is that writes one `BCSdsFetch` instruction with the correctly embedded SDS. Assigning to an analog pin will result in the RWST writing the `BCAnalogRead` instruction. When the operation on the assignable is not a read operation from but an assign operation, the instruction will be rewritten accordingly as shown in Listing 23. This results in a `BCSdsStore` or `BCAnalogWrite` instruction respectively.

```
instance assign ByteCode where
  (=.) (BC v) (BC e) = BC (e >>| censor makeStore v)

makeStore [BCSdsFetch i]    = [BCSdsStore i]
makeStore [BCDigitalRead i] = [BCDigitalWrite i]
makeStore [...]             = [...]
```

Listing 23: Bytecode view implementation for assignment.

8.5 Actual Compilation

All the previous functions are tied together with the `toMessages` function which compiles the bytecode and transforms the Task to a message and is shown in Listing 24. The SDSs that were not already sent to the device are also added as messages to be sent to the device. The compilation process consists of two steps. First, the RWST is executed and then the *Jump* statements that jump to labels are transformed to jump to program memory addresses. The translation of labels to program addresses happens in `computeGotos`. The function consumes the instructions one by one while incrementing the address counter with the length of the instruction except for when it encounters a label because then it will store its location. The generic function `consNum` is used which gives the arity of the constructor. When all instructions are processed, the jump instructions are transformed using the `implGotos` function which translates all the jump instructions. After label translation, the compiler state is inspected to see whether new shares have been instantiated. The compilation concludes with converting the bytecode and SDSs to actual messages ready to send to the client.

```
bclength :: BC -> Int //The number of bytes

computeGotos :: [BC] Int -> ([BC], Map Int Int)
computeGotos [] _ = ([], newMap)
computeGotos [BCLab l:xs] i
# (bc, i) = computeGotos xs i
= (bc, put l i)
computeGotos [x:xs] i
# (bc, i) = computeGotos xs (i + bclength x)
= ([x:bc], i)

toRealByteCode :: (ByteCode a b) BCState -> (String, BCState)
toRealByteCode x s
# (s, bc) = runBC x s
# (bc, gtmap) = computeGotos bc 1
= (concat (map (toString o toByteVal) (map (implGotos gtmap) bc)), s)

implGotos map (BCJmp t) = BCJmp $ fromJust $ get t map
...
implGotos _ i = i

toMessages :: MTaskInterval (Main (ByteCode a b)) BCState -> ([MTaskMSGSend
      ], BCState)
toMessages interval x oldstate
# (bc, newstate) = toRealByteCode (unMain x) oldstate
# newsdss = difference newstate.sdss oldstate.sdss
= ([MTSds sdsi e\\{sdsi,sdsval=e}<-newsdss] ++ [MTTask interval bc],
      newstate)
```

Listing 24: Actual compilation

9 ARCHITECTURE

The system provides a framework of functions with which an iTasks-system can add, change and remove devices at runtime. Moreover, the iTasks-system can send mTask-Tasks — compiled at runtime to bytecode by the mTask-view — to the device. The device runs an interpreter which executes the Task's bytecode following the provided scheduling strategy. Devices added to the system are stored

and get a profile for identification. These profiles are persistent during reboots of the iTasks-system to allow for easy reconnecting with old devices. The way of interacting with mTask-Tasks is analogous to interacting with iTasks-Tasks. This means that programmers can access the SDSs made for a device in the same way as regular SDSs and they can execute, combine and transform mTask-Tasks if they where normal iTasks-Tasks.

9.1 Devices

A device is suitable for the system as a client if it can run the engine. The engine is compiled from one codebase and devices implement (part of) the device specific interface. The shared codebase only uses standard C and no special libraries or tricks are used. Therefore, the code is compilable for almost any device or system. The interface works in a similar fashion as the EDSL. Devices do not have to implement all functionality, this is analogous to the fact that views do not have to implement all type classes in the EDSL. When the device connects with the server for the first time, the specifications of what is available is communicated. Devices are stored in an SDS because they are available throughout sessions and can thus not always be kept in scope.

At the time of writing, four device families are supported and can run the device software. Porting the client software to a new device does not require a lot of work because only the interface needs to be implemented.

The reference implementation is a `POSIX` C version for systems connected via TCP. This port only uses functionality from the standard C library and therefore runs on all UNIX derivatives boasting a sane `libc` implementation.

For microcontrollers, the mbed and ChibiOS platforms are supported via serial communication. This port has been tested on `STM32f7x` series ARM development boards.

Finally, Arduino IDE supported microcontrollers are usable in the system. This does not only include Arduino compatible boards but also other boards capable of running Arduino code such as the ESP8266 powered NodeMCU that is connected via TCP over WiFi. Some Arduino boards only boast *2K* of RAM and require smaller stack sizes but they are still capable of storing a couple of Tasks.

9.2 Client

9.2.1 Engine. The client software is responsible for maintaining the communication and executing the Tasks when scheduled. In practise, this means that the client is in a constant loop, checking communication and executing Tasks. The pseudocode for this is shown in Algorithm 2. The `input_available` function waits for input, but has a timeout set which can be interrupted. The timeout of the function determines the amount of loops per time interval and is a parameter that can be set during compilation for a device.

9.2.2 Storage. Tasks and SDSs are stored on the client not in program memory but in the RAM. Some devices have little memory and therefore memory space is very expensive and needs to be used optimally. Almost all microcontrollers support heaps nowadays, however, the functions for allocating and freeing the memory on the heap are not very space optimal and often leave holes in the heap if allocations are not freed in a last in first out fashion. To overcome this problem, the client will allocate a big memory segment in the

```
Data: list tasks, time tm
begin
    while true do
        if input_available() then
            receive_data();
        end
        tm ← now();
        foreach t ← tasks do
            if is_interrupt(t) and had_interrupt(t) then
                run_task(t);
            else if tm − t.lastrun > t.interval then
                run_task(t);
                if t.interval == 0 then
                    delete_task(t);
                else
                    t.lastrun ← t;
                end
            end
        end
    end
end
```

Algorithm 2: Engine pseudocode

global data block. This block of memory resides under the stack and its size can be set in the interface implementation. This block of memory will be managed in a similar way as the entire memory space of the device is managed. Tasks will grow from the bottom up and SDSs will grow from the top down.

When a Task is received, the program will traverse the memory space from the bottom up, jumping over all Tasks. A Task is stored as the structure followed directly by its bytecode. Therefore it only takes two jumps to determine the size of the Task. When the program arrived at the last Task, this place is returned and the newly received Task can be copied to there. This method is analogously applied for SDSs, however, the SDSs grow from the bottom down.

When a Task or SDS is removed, all the remaining objects in the memory space are reordered in such a way that there are no holes left. In practice this means that if the first received Task is removed, all Tasks received later will have to move back. Obviously, this is quite time intensive but it can not be permitted to leave holes in the memory since the memory space is so limited. With this technique, even the smallest tested microcontrollers with only 2*K* RAM can hold several Tasks and SDSs. Without this technique, the memory space will decrease over time and the client can then not run for very long since holes are evidently created at some point when tasks are sent dynamically.

9.2.3 Interpretation. The execution of a Task is started by running the `run_task` function and always starts with setting the program counter and stack pointer to zero and the bottom respectively. When initialized, the interpreter executes one step at the time while the program counter is smaller than the program length.the code listed in Listing 25. One execution step is basically a switch statement going over all possible bytecode instructions. The implementation of some instructions is shown in Listing 25. The `BCPush` instruction

is a little more complicated in the real code because some decoding will take place as not all BCValues are of the same length and are encoded.

```
#define f16(p) program[pc]*265+program[pc+1]

void run_task(struct task *t){
  uint8_t *program = t->bc;
  int plen = t->tasklength, pc = 0, sp = 0;
  while(pc < plen){
    switch(program[pc++]){
    case BCNOP:
      break;
    case BCPUSH:
      stack[sp++] = pc++ //Simplified
      break;
    case BCSDSSTORE:
      sds_store(f16(pc), stack[--sp]);
      pc+=2;
      break;
    case BCADD:
      stack[sp-2] = stack[sp-2] + stack[sp-1];
      sp -= 1;
      break;
    case BCJMPT:
      pc = stack[--sp] ? program[pc]-1 : pc+1;
      break;
    // ...
```

Listing 25: Rough code outline for interpretation

9.3 iTasks

The server part of the system is written in iTasks which contains functions for managing. This includes functionality for adding, removing and updating devices and functions for sending and removing Tasks and SDSs to devices. Furthermore, an interactive web application has been created that provides an interactive management console for these manageming tasks. This interface provides functionality to show Tasks and SDSs and their according status. It also provides the user with a library of example mTask-Tasks that can be sent interactively to the device.

9.4 Device Storage

Everything that a device encompasses is stored in the MTaskDevice record type which is in turn stored in an SDS. The MTaskDevice definition is shown in Listing 26 accompanied with the required definitions. Devices added to the system must be reachable asynchronously and from all scopes. Devices have a unique channel SDS that is enough to retrieve device data at all times.

```
:: Channels :== ([MTaskMSGRecv], [MTaskMSGSend], Bool)
:: MTaskDeviceSpec = ... // Explained in a later section
:: MTaskMSGRecv = ...    // Message format, explained in a later section
:: MTaskMSGSend = ...    // Also explained in a later section
:: MTaskResource
  = TCPDevice TCPSettings
  | SerialDevice TTYSettings
  | ...
:: MTaskDevice =
  { deviceTask     :: Maybe TaskId,  deviceError :: Maybe String
  , deviceChannels :: String,        deviceName  :: String
  , deviceState    :: BCState,       deviceTasks :: [MTaskTask]
  , deviceResource :: MTaskResource, deviceSpec  :: Maybe MTaskDeviceSpec
  , deviceShares   :: [MTaskShare]
  }

channels :: MTaskDevice -> Shared Channels

class MTaskDuplex a where
  synFun :: a (Shared Channels) -> Task ()
```

Listing 26: Device type

The deviceResource component of the record must implement the MTaskDuplex interface that provides a function that launches a Task used for synchronizing the channels. The deviceChannels field can be used to get the memory SDS containing the channels. This field does not contain the channels itself because they update often. The field is used to get a memory SDS containing the actual channel data when calling the channels function. The deviceTask stores the Task-id for this Task when active so that it can be checked upon. This top-level task has the duty to report exceptions and errors as they are thrown by setting the deviceError field. All communication goes via these channels. To send a message to the device, the system just puts it in the channels. Messages sent from the client to the server are also placed in there. In the case of the TCP device type, the Task is just a simple wrapper around the existing tcpconnect function in iTasks.

Besides all the communication information, the record also keeps track of the Tasks currently on the device, the compiler state and the according SDSs. Finally, it stores the specification of the device that is received when connecting. The definitions of the message format are explained in the following section.

9.5 SDSs

SDSs on the device can be accessed by both the device and the server. While it would be possible to only store the SDSs on the device, this would require a lot of communication since every read operation will then result in sending messages to-and-fro the device. Thus, the Task requesting the shared information can just be provided with the cached value. As stated, the device has to explicitly publish an update which implies that the server and the client can get out of sync. However, this is by design and well documented. In the current system, an SDS can only reside on a single device.

There are several possible approaches for storing SDSs on the server each with their own level of control. A possible way is to — in the device record — add a list of references to iTasks-SDSs that represent the SDS on the device. The problem with this is the fact that an SDS can become an orphan. The SDS is still accessible even when the device is long gone. There is no way of knowing whether the SDS is unreachable because of the device being gone, or the SDS itself is gone on the device. Accessing the SDS happens by calling the get, set and upd functions directory on the actual SDS.

Another approach would be to have reference to an SDS containing a table of SDS values per device. This approach suffers the same orphan problem as before. Accessing a single SDS on the device happens by calling the get, set and upd functions on the actual table SDS with an applied mapReadWrite. Using parametric lenses can circumvent the problem of watchers getting notified for other shares that are written. Error handling is better than the previously mentioned approach because an SDS can know whether the SDS has really gone because it will not be available anymore in the table. It still does not know whether the device is still available.

Finally, all devices containing all of their SDSs in a table could be stored in a single big SDS. While the `mapReadWrite` functions require a bit more logic, they can determine the source of the error and act upon it. Also, the parametric lenses must contain more logic. A downside of this approach is that updating a single SDS requires an update of the entire object. In practise, this is not a real issue since almost all information can be reused and SDS residing on devices are often not updated with a very high frequency.

9.6 Parametric Lenses

The type for the parametric lens of the SDS containing all devices is `Maybe (MTaskDevice, Int)`. There are several levels of abstraction that have to be introduced. First, the SDS responsible for storing the entire list of devices is called the global SDS. Secondly, an SDS can focus on a single device, such SDSs are called local SDSs. Finally, an SDS can focus on a single SDS on a single device. These SDSs are called share SDSs. Using parametric lenses, the notifications can be directed to only the watchers interested. Moreover, using parametric lenses, the SDS can know whether it is updating a single SDS on a single device and synchronize the value with the actual device. This means that when writing to a share SDS the update is also transformed to messages that are put in the channels of the corresponding device to also notify the device of the update. The SDS is tailor-made and uses an actual standard SDS that writes to a file or memory as the storage. The tailor-made read and write functions are only used to detect whether it is required to send an update to the actual device.

Listing 27 shows the implementation of the big SDS of which all other SDSs are derived. The following paragraphs illustrate how this is achieved for the global SDS local SDS and the share SDS. In the big SDS, reading the value is just a matter of reading the standard SDS that serves as the actual storage of the SDS. The derived shares will filter the output read accordingly. Writing the share requires some extra work because it might be possible that an actual device has to be notified. First, the actual storage of the SDS is written. If the parameter was `Nothing` — the global SDS — the write operation is done. If the parameter was `Just (d, -1)` — a local SDS — nothing has to be done as well. The final case is the special case, when the parameter is `Just (d, i)`, this means that the SDS was focussed on device `d` and SDS `i` and thus it needs to write to it. First it locates the device in the list, followed by the location of the share to check whether is still exists. Finally the actual update messages are added to the device channels.

All of the methods share the same `SDSNotifyPred p` which is a function `p -> Bool` and determines for the given `p` whether a notification is required. The predicate function has the `p` of the writer curried in and can determine whether the second argument — the reader — needs to be notified. In practice, the reader only needs to be notified when the parameters are exactly the same.

```
($<) :: a (f a) -> (f b)
($<) a fb = fmap (const a) fb

deviceStore :: RWShared (Maybe (MTaskDevice, Int)) [MTaskDevice] [
      MTaskDevice]
deviceStore = SDSSource {SDSSource | name="deviceStore", read=realRead,
      write=realWrite}
where
  realRead :: (Maybe (MTaskDevice,Int)) *IWorld -> (MaybeError TaskException
       [MTaskDevice], *IWorld)
  realRead p iw = read realDeviceStore iw

  realWrite :: (Maybe (MTaskDevice,Int)) [MTaskDevice] *IWorld -> (
      MaybeError TaskException (SDSNotifyPred (Maybe (MTaskDevice,Int))), *
      IWorld)
  realWrite mi w iw
  # (merr, iw) = write w realDeviceStore iw
  | isError merr || isNothing mi = (merr $> gEq{|*|} mi, iw)
  # (Just (dev, ident)) = mi
  | ident == -1 = (merr $> gEq{|*|} mi, iw)
  = case find ((==)dev) w of
    Nothing = (Error $ exception "Device lost", iw)
    Just {deviceShares} = case find (λd->d.identifier == ident) deviceShares
       of
      Nothing = (Error $ exception "Share lost", iw)
      Just s = case sendMessagesIW [MTUpd ident s.MTaskShare.value] dev iw
      of
        (Error e, iw) = (Error e, iw)
        (Ok _, iw) = (Ok $ gEq{|*|} mi, iw)

  realDeviceStore :: Shared [MTaskDevice]
  realDeviceStore = sharedStore "mTaskDevices" []
```

Listing 27: Device SDS

9.6.1 Global SDSs. Accessing the global SDS is just a matter of focussing the `deviceStore` to `Nothing`. In this way, Tasks watching the SDS will only be notified if a device is added or removed. The actual code is as follows:

```
deviceStoreNP :: Shared [MTaskDevice]
deviceStoreNP = sdsFocus Nothing deviceStore
```

Listing 28: Global SDS

9.6.2 Local SDSs. Accessing a single device can be done using the `deviceShare` function. Since device comparison is shallow, the device that is given is allowed to be an old version. The identification of devices is solely done on the name of the channels and is unique throughout the system. This type of SDS will only be notified if the device itself changed. It will not be notified when only a single SDS on the device changes. The implementation is as follows:

```
deviceShare :: MTaskDevice -> Shared MTaskDevice
deviceShare d = mapReadWriteError
  ( λds->case find ((==)d) ds of
    Nothing = exception "Device lost"
    Just d = Ok d)
  , λw ds->case splitWith ((==)d) ds of
    ([], _) = Error $ exception "Device lost"
    ([_:_], ds) = Ok $ Just [w:ds])
  $ sdsFocus (Just (d, -1)) deviceStore
```

Listing 29: Local SDS

9.6.3 Local-SDS specific SDSs. A single SDS on a single device can be accessed using the `shareShare` function. This function focusses the global SDS on a single SDS from a single device. It can use old share references in the same fashion as the local SDS only treating it as references. It uses the `mapReadWrite` functions to serve the correct part of the information. When a Task writes to this SDS, the global SDS will know this through the parameter and propagate the value to the device. The implementation is similar to the local SDS but it includes a lens to the actual share.

9.7 Communication

The communication from the server to the client and vice versa is just a character stream containing encoded mTask messages. The synFun belonging to the device is responsible for sending the content in the left channel and putting received messages in the right channel. Moreover, the boolean flag in the channel type should be set to True when the connection is terminated. The type holding the messages is shown in Listing 30. Detailed explanation about the message types and according actions will be given in the following subsections.

```
:: MTaskId :== Int
:: MSDSId :== Int
:: MTaskFreeBytes :== Int
:: MTaskMSGRecv
  = MTTaskAck MTaskId MTaskFreeBytes | MTTaskDelAck MTaskId
  | MTSDSAck MSDSId                  | MTSDSDelAck MSDSId
  | MTPub MSDSId BCValue             | MTMessage String
  | MTDevSpec MTaskDeviceSpec        | MTEmpty

:: MTaskMSGSend
  = MTTask MTaskInterval String | MTTaskDel MTaskId
  | MTShutdown                  | MTSds MSDSId BCValue
  | MTUpd MSDSId BCValue        | MTSpec

:: MTaskInterval = OneShot | OnInterval Int | OnInterrupt Int
```

Listing 30: Available messages

9.8 Device Specification

The server stores a description for every device available in a record type. From the macro settings in the client — in the interface file— a profile is created that describes the specification of the device. When the connection between the server and a client is established, the server will send a request for specification. The client serializes its specification and send it to the server so that the server knows what the client is capable of. The exact specification is shown in Listing 31 and stores the peripheral availability, the memory available for storing Tasks and SDSs and the size of the stack. Not all peripheral flags are shown for brevity.

```
:: MTaskDeviceSpec =
  { haveLed     :: Bool, haveLCD     :: Bool, have...
  , bytesMemory :: Int,  stackSize   :: Int
  , aPins       :: Int,  dPins       :: Int
  }
```

Listing 31: Device specification for mTask-Tasks

The code on the device generates the specification. When a device does not have a specific peripheral, the code will also not be on the device. In the interface file, the code for peripherals is always guarded by macros. Thus, if the peripheral is not there, the macro is set accordingly and the code will not be included. The macro flags are serialized to a bytestring that is then parsed by the server.

9.9 Add a device

A device can be added by filling in the MTaskDevice record as much as possible and running the connectDevice function. This function grabs and clears the channels, starts the synchronization Task (synFun), makes sure the errors are handled when needed and runs a processing function in parallel to react on the incoming messages. Moreover, it sends a specification request to the device in question to determine the details of the device and updates the record to contain the top-level Task-id. All device functionality heavily depends on the specific deviceShare function that generates an SDS for a specific device. This allows giving an old device record to the function and still update the latest instance. Listing 32 shows the connection functions and necessary signatures.

```
// Processes the messages
process :: MTaskDevice (Shared Channels) -> Task ()

connectDevice :: MTaskDevice -> Task MTaskDevice
connectDevice device = set ([], [], False) ch
  >>| appendTopLevelTask 'DM'.newMap True
  ( process device ch -||- catchAll (getSynFun device.deviceData ch) errHdl)
  >>= λtid->upd (λd->{d&deviceTask=Just tid,deviceError=Nothing}) (
      deviceShare device)
  >>| set (r,[MTSpec],ss) ch
  >>| treturn device
where
  errHdl e = upd (λd->{d & deviceTask=Nothing, deviceError=Just e}) (
      deviceShare device) @! ()
  ch = channels device
```

Listing 32: Connect a device

9.10 Tasks & SDSs

When a Task is sent to the device it is added to the device record without an identifier. The actual identifier is added to the record when the acknowledgement of the Task by the device is received.

The function for sending a Task to the device and the signatures are shown in Listing 33. First the Task is compiled into messages. The new SDSs that were generated during compilation are merged with the existing device's SDSs. Furthermore the messages are placed in the channel SDS of the device. This will result in sending the actual SDS specification and Task specifications to the device. A Task record is created with the identifier −1 to denote a Task not yet acknowledged. Finally the device itself is updated with the new state and with the new Task. After waiting for the acknowledgement the device is updated again and the Task returns.

```
//Construct functions
makeTask :: String Int -> Task MTaskTask
makeShare :: String Int BCValue -> MTaskShare

sendTaskToDevice :: String (Main (ByteCode a Stmt)) (MTaskDevice,
      MTaskInterval) -> Task (MTaskTask, [MTaskShare])
sendTaskToDevice wta mTask (device, timeout)
# (msgs, newState=:{sdss}) = toMessages timeout mTask device.deviceState
# shares = [makeShare wta "" sdsi sdsval
  \\{sdsi,sdsval}<-sdss, (MTSds sdsi` _)<-msgs | sdsi == sdsi`]
= updateShares device ((++) shares)
  >>| sendMessages msgs device
  >>| makeTask wta -1
  >>= λt->upd (addTaskUpState newState t) (deviceShare device)
  >>| wait "Waiting for task to be acked" (taskAcked t) (deviceShare device)
  >>| treturn (t, shares)
where
  addTaskUpState :: BCState MTaskTask MTaskDevice -> MTaskDevice
  addTaskUpState st task device = {MTaskDevice | device & deviceState=st,
      deviceTasks=[task:device.deviceTasks]}
  taskAcked t d = maybe True (λt->t.ident <> -1) $ find ((==) t)
      d.deviceTasks
```

Listing 33: Sending a Task to a device

9.11 Integration

The system's management is done through the interface of a single Task called `mTaskManager`. To manage the system, a couple of different functionalities are necessary and are launched. The interface allows the user to add and inspect devices interactively, send tasks to them and inspect SDSs. Moreover, this panel allows the user to reconnect with a device after a restart of the server application.

9.12 Example

The management functionality of the system can be skipped and systems can be used without needing interaction. Often, systems built with support for mTask follow the same design pattern. First the devices are created — with or without the interaction of the user — and they are then connected. When all devices are registered, the mTask-Tasks can be sent and iTasks-Tasks can be started to monitor the output. When everything is finished, the devices are removed and the system is shut down.

The thermostat is a classic example program for microcontrollers illustrating interactions between peripherals. The following program shows a system containing two devices. The first device — the sensor — contains a temperature sensor that measures the room temperature. The second device — the actor — contains a heater, connected to the digital pin `D5`. Moreover, this device contains an LED to indicate whether the heater is on. The following code shows an implementation for this. Note that a little bit of type twiddling is required to fully use the result from the SDS. This approach is still type safe due to the type safety of `Dynamic`s.

```
thermos :: Task ()
thermos =           makeDevice "nodeM" nodeMCU >>= connectDevice
  >>= λnod->      makeDevice "stm32" stm32   >>= connectDevice
  >>= λstm->      sendTaskToDevice "sensor" sensing (nod, OnInterval 1000)
  >>= λ(st, [t])->sendTaskToDevice "actor" acting (stm, OnInterval 1000)
    (λ(BCValue s)->set (BCValue $ dynI (dynamic s) > 0) (shareShare nod a))
  >>* [OnAction (Action "Shutdown") $ always $ deleteDevice nod
      >>| deleteDevice stm >>| shutDown 0]
where
  dynI :: Dynamic -> Int
  dynI (a :: Int) = a

  sensing = sds λx=0 In {main= x =. analogRead A0 :. pub x}
  acting = sds λcool=False In {main=
      IF cool (ledOn LED1) (ledOff LED1) :.
      digitalWrite D5 cool}
  nodeMCU = makeDevice "NodeMCU" (TCPDevice {host="10.0.0.1", port=8888})
  stm32   = makeDevice "Stm32" (SerialDevice {devicePath="/dev/ttyUSB0",...)
```

Listing 34: Thermostat example

9.13 Lifting mTask-Tasks to iTasks-Tasks

With the given functions, it is possible to just lift an mTask-Task to an iTasks-Task. The function is called with a name, mTask, device and interval specification and it will return a Task that finishes if and only if the mTask has returned and yields the shares used.

```
liftmTask :: String (Main (ByteCode () Stmt)) (MTaskDevice, MTaskInterval)
      -> Task [MTaskShare]
liftmTask wta mTask c=:(dev, _)= sendTaskToDevice wta mTask c
  >>= λ(t, shs)->wait "Waiting for mTask to return" (taskRemoved t) (
      deviceShare dev)
  >>| viewInformation "Done!" [] ()
  >>| treturn shs
where
  taskRemoved t d = isNothing $ find (λt1->t1.ident==t.ident) d.deviceTasks
```

Listing 35: Lifting mTask-Tasks to iTasks

10 DISCUSSION

The novel system is functional but still a crude prototype and a proof of concept. The system shows potential but improvements and extensions for the system are amply available in several fields of study.

10.1 Simulation

An additional simulation view to the mTask-EDSL could be added that works in the same way as the existing C-backed simulation. It simulates the bytecode interpretation. Moreover, it would also be possible to let the simulator function as a real device, thus handling all communication through the existing SDS-based systems. At the moment the POSIX-client is the reference client and contains debugging code. Adding a simulation view to the system allows for easy interactive debugging. However, it might not be easy to devise a simulation tool that accurately simulates the mTask system on some levels. The execution strategy can be simulated but timing and peripheral input/output are more difficult to simulate properly.

10.2 Optimization

Multitasking on the client: True multitasking could be added to the client software. This allows mTask-Tasks to run truly parallel. All mTask-Tasks get slices of execution time and will each have their own interpreter state instead of a single system-wide state which is reset after an mTask finishes. This does require separate stacks for each Task and therefore increases the system requirements of the client software. However, it could be implemented as a compile-time option and exchanged during the handshake so that the server knows the multithreading capabilities of the client. Multithreading allows Tasks to be truly interruptible by other Tasks. Furthermore, this allows for more fine-grained timing control of Tasks.

Optimizing the interpreter: Due to time constraints and focus, hardly any work has been done in the interpreter. The current interpreter is a no nonsense stack machine. A lot of improvements can be done in this part. For example, precomputed *gotos* can improve jumping to the correct part of the code corresponding to the correct instruction. Moreover, the stack currently consists of 16-bit values. All operations work on 16-bit values and this simplifies the interpreter implementation. A memory improvement can be made by converting the stack to 8-bit values. This does pose some problems since an equality instruction must work on single-byte booleans *and* two-byte integers. Adding specialized instructions per word size could overcome this problem.

10.3 Resources

Resource analysis: Resource analysis during compilation can be useful to determine if an mTask-Task is suitable for a specific device. If the device does not contain the correct peripherals — such as an LCD — then the mTask-Task should be rejected and feedback to the user must be given. It might even be possible to do this statically on the type level. The current system does not have any of this built-in.

Sending a Task that uses the LCD to a device not containing one will result in the device just skipping the LCD related instructions.

Extended resource analysis: The previous idea could be extended to the analysis of stack size and possibly communication bandwidth. With this functionality ever more reliable fail-over systems can be designed. When the system knows precise bounds it can allocate more Tasks on a device whilst staying within safe memory bounds. The resource allocation can be done at runtime within the backend itself or a general backend can be devised that can calculate the resources needed for a given mTask. A specific mTask cannot have multiple views at the same time due to the restrictions of class based shallow embedding. It might even be possible to encode the resource allocation in the type system itself using forms of dependant types.

10.4 Functionality

Add more combinators: More Task-combinators — already existing in the iTasks-system — could be added to the mTask-system to allow for more fine-grained control flow between mTask-Tasks. In this way the new system follows the TOP paradigm even more and makes programming mTask-Tasks for TOP-programmers more seamless. Some of the combinators require previously mentioned extension such as the parallel combinator. Others might be achieved using simple syntactic transformations.

Launch Tasks from a Task: Currently the C-view allows Tasks to launch other Tasks. In the current system this type of logic has to take place on the server side. Adding this functionality to the bytecode-view allows greater flexibility, easier programming and less communication resources. Adding this type of scheduling requires modifications to the client software and extensions to the communication protocol since relations between Tasks also need to be encoded and communicated. A similar technique as used with SDSs has to be used to overcome the scoping problem.

The SDS functionality in the current system is bare. There is no easy way of reusing an SDS for another Task on the same device or on another device. Such functionality can be implemented in a crude way by tying the SDSs together in the iTasks environment. However, this will result in a slow updating system. Functionality for reusing shares from a device should be added. This requires rethinking the storage because some typedness is lost when the SDS is stored after compilation. A possibility would be to use runtime typing with `Dynamic`s or the encoding technique currently used for `BCValue`s. Using SDSs for multiple Tasks within one device is solved when the previous point is implemented.

Another way of improving on SDS handling is to separate SDSs from devices. In this implementation, the SDS not only needs to know on which device it is, but also which internal device SDS id it has. A pro of this technique is that the SDS can be shared between Tasks that are not defined in the same scope because they are separated. A con of this implementation is that the mechanisms for implementing Tasks have to be more complex, they have to keep track of the devices containing or sharing an SDS. Moreover, when the SDS is updated, all attached devices must be updated which requires some extra work.

10.5 Robustness

Reconnect with lost devices: The robustness of the system can be greatly improved. Devices that lose connection are not well supported in the current system. The device will stop functioning and has to be emptied for a reconnect. Tasks residing on a device that disconnected should be kept on the server to allow a swift reconnect and restoration of the Tasks. This holds the same for the client software. The client drops all existing Tasks on a shutdown request. An extra specialization of the shutdown could be added that drops the connection but keeps the Tasks in memory. During the downtime the Tasks can still be executed but publications need to be delayed. If the same server connects to the client the delayed publications can be sent anyways.

Reverse Task sending: Furthermore, devices could send their current Taskss back to the server to synchronize it. This allows interchanging servers without interrupting the client. Allowing the client to send Tasks to the server is something to handle with care because it can easily cause high bandwidth usage.

11 CONCLUSION

This thesis introduces a novel system for adding IoT functionality to the TOP implementation iTasks. A new view for the existing mTask-EDSL has been created which compiles the program into bytecode that can be interpreted by a client. Clients have been written for several microcontrollers and consumer architectures which can be connected through various means of communication such as serial port, wifi and wired network communication. The bytecode on the devices is interpreted using a stack machine and provides the programmer with interfaces to the peripherals. The semantics for mTask-Tasks try to resemble the semantics as close as possible.

The host language has a proven efficient compiler and code generator. The compilation is linear in the amount of instructions generated and is therefore also scalable. Moreover, compiling Tasks is fast because it is nothing more than running some functions native to the host language and there is no intermediate AST.

The dynamic nature of the client allows the microcontroller to be programmed once and used many times. The program memory of microcontrollers often guarantees around 10.00 write or upload cycles and therefore existing techniques such as generating C code are not suitable for dynamic Task environments. The dynamic nature also allows the programmer to design fail-over mechanisms. When a device is assigned a Task but another device suddenly becomes unusable, the iTasks system can reassign a new mTask-Task to another device that is also suitable for running the Task without needing to recompile the code. It also showed that adding peripherals is not a time consuming task and does not even requires recompilation of clients not having the peripheral.

The new functionality extends the reach of iTasks by adding IoT functionality and allowing devices to run mTask-Tasks. With this extension, a programmer can create an entire IoT system from one source that reaches all layers of the IoT architecture. However, this does not limit the applications and makes them static. Components can be updated individually without causing integration problems. Devices can be repurposed just by sending new Tasks to it. Most

importantly, it gives an insight in the possibilities of adding IoT to TOP programs.

REFERENCES

[1] Peter Achten, Pieter Koopman, and Rinus Plasmeijer. 2015. An Introduction to Task Oriented Programming. In *Central European Functional Programming School*. Springer, 187–245.

[2] T. H. Brus, Marko CJD van Eekelen, M. O. Van Leer, and Marinus J. Plasmeijer. 1987. Clean – a language for functional graph rewriting. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 364–384.

[3] James Cheney and Ralf Hinze. 2002. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 90–104.

[4] James Cheney and Ralf Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.

[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] László Domoszlai, Eddy Bruel, and Jan Martin Jansen. 2011. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae* 3 (2011), 76–98.

[7] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric lenses: change notification for bidirectional lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, 9.

[8] László Domoszlai and Rinus Plasmeijer. 2012. Compiling Haskell to JavaScript through Clean's core. In *Selected papers of 9th Joint Conference on Mathematics and Computer Science (February 2012)*.

[9] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt free ivory. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 189–200.

[10] Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. 2014. Building embedded systems with embedded DSLs. ACM Press, 3–9. https://doi.org/10.1145/2628136.2628146

[11] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. 2007. Efficient Interpretation by Transforming Data Types and Patterns to Functions. *Trends in Functional Programming* 7 (2007), 73.

[12] Pieter Koopman and Rinus Plasmeijer. [n. d.]. Type-Safe Functions and Tasks in a Shallow Embedded DSL for Microprocessors. ([n. d.]).

[13] Bas Lijnse. 2013. *TOP to the rescue: task-oriented programming for incident response applications*. s.n.; UB Nijmegen, S.l.; Nijmegen. OCLC: 833851220.

[14] Arjan Oortgiese. 2017. *A Distributed Server Architecture for Task Oriented Programming*. Master's Thesis. Radboud University, Nijmegen.

[15] Lee Pike, Patrick Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. 2014. Programming languages for high-assurance autonomous vehicles: extended abstract. ACM Press, 1–2. https://doi.org/10.1145/2541568.2541570

[16] 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.

[17] Rinus Plasmeijer and Pieter Koopman. 2016. A Shallow Embedded Type Safe Extendable DSL for the Arduino. In *Trends in Functional Programming*. Lecture Notes in Computer Science, Vol. 9547. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-39110-6.

[18] S. M. Riazul Islam, Daehan Kwak, Md Humaun Kabir, Mahmud Hossain, and Kyung-Sup Kwak. 2015. The Internet of Things for Health Care: A Comprehensive Survey. *IEEE Access* 3 (2015), 678–708. https://doi.org/10.1109/ACCESS.2015.2437951

[19] Josef Svenningsson and Emil Axelsson. 2012. Combining deep and shallow embedding for EDSL. In *International Symposium on Trends in Functional Programming*. Springer, 21–36.

Type Directed Workflow Modelling

Tim Steenvoorden
Department of Software Science
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
t.steenvoorden@cs.ru.nl

Rinus Plasmeijer
Department of Software Science
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
rinus@cs.ru.nl

Abstract

Workflow management systems tend to be complicated, containing many basic elements to design and construct workflows. Also, coming from a workflow specification to an application usable by the end user is a lot of effort. Novel ways to generate this kind of applications, like the functional programming oriented iTasks system, are promising, but complicated to use for workflow designers. We present a design tool aiding in specifying work and data flows in a straight forward and modular way. Our tool discerns itself from others by incorporating the information needed to perform a task, thus combining workflows and dataflows into a single concept we call *typed workflows*. Typed workflows have a formal language representation based on natural language as well as a visual representation inspired by Petri nets. Because all information in our workflows is typed, we are able to generate full blown web applications by using the iTasks system as a backend.

CCS Concepts • **Software and its engineering → Functional languages**; **Software prototyping**; **Flowcharts**; *Software usability*; *Domain specific languages*;

Keywords workflow, dataflow, visual programming, program generation

ACM Reference format:
Tim Steenvoorden and Rinus Plasmeijer. 2017. Type Directed Workflow Modelling. In *Proceedings of The 29th symposium on Implementation and Application of Functional Languages, University of Bristol, UK, August 30 – September 1, 2017 (IFL'17)*, 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Contributions

Our contributions to the world of workflow modelling are the following:

- Extend the modelling of workflows with data needed to perform a particular task.
- Ensure correctness of a workflow by enforcing types on all data available during task execution.
- Support designing typed workflows using a visual structure editor.
- Generate full blown web applications, supporting the designed workflow, out of its specification.
- Ensures correct execution of these applications at runtime due to type information.

The remaining of this paper is structured as follows. First, in section 2, we study some motivating examples of typed workflows. Next we develop a formal language for typed workflows including a type system in section 3. We use some examples from natural language as a starting point. In section 4 we discuss the visual representation of our language and how it can be used to incrementally create typed workflows. We discuss related work in section 5 and conclude in section 6.

2 Examples

See fig. 1 for an example typed workflow, modelling part of a subsidy request. The corresponding code can be found in fig. 2.

3 Typed workflows

Our goals is to provide a domain specific language that is readable and understandable by domain experts. It should use a simple and comprehensible set of operations to construct typed workflows. This set of operations should be useable by workflow designers to quickly change and extend existing typed workflows or create new ones, without learning a (functional) programming language. Also, there should be a one-to-one mapping to a visual representation.

Our starting point will be the way people tend to describe tasks and workflows in natural language, which we discuss in section 3.1. Next, in section 3.2, we define four elementary ways to combine these task into bigger workflows. Finally we describe four elementary operations to interact with data flowing through our language in section 3.3.

3.1 Natural language

In this section, we develop a method to define (parts of) workflows and link this to natural language by building an imperative sentence in English.

Every step in a workflow consists of a *task*. A task usually needs to be executed to solve a particular *problem*. We have already seen some of these problems in the examples

IFL'17, August 30 – September 1, 2017, University of Bristol, UK
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

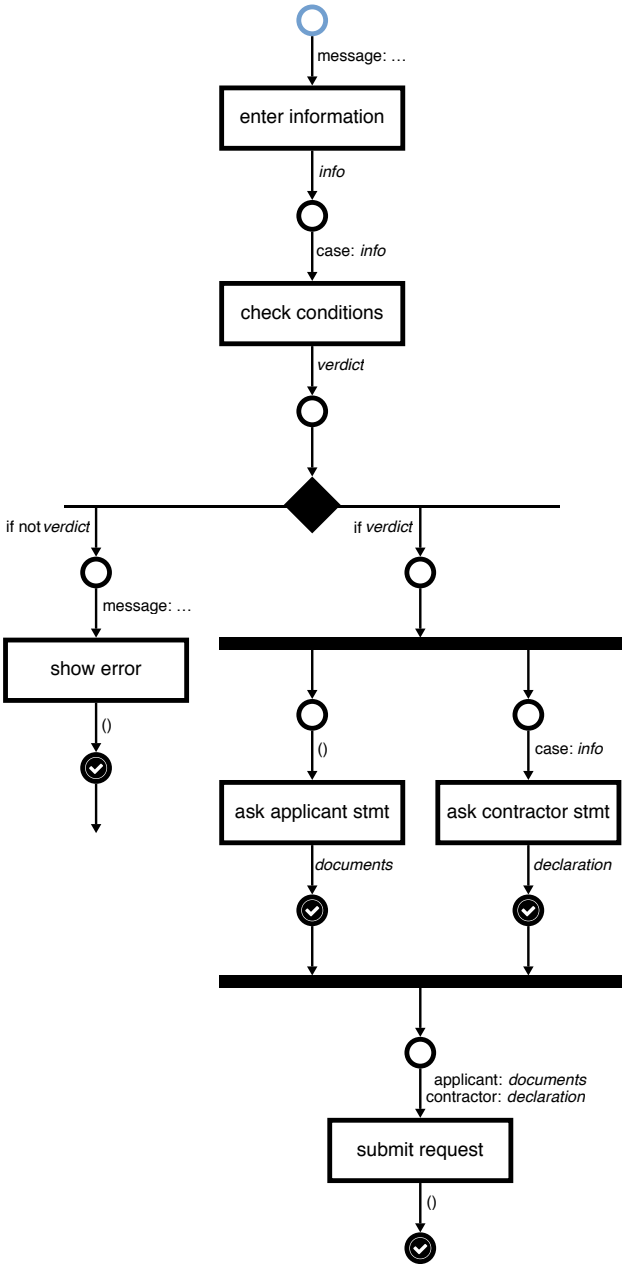


Figure 1. Visualisation of a typed workflow modelling part of a subsidy request using sequencing, parallel composition and internal choice.

extinguish fire
attack enemy
fill in form

Linguistically such a problem consists of a *verb* and a *direct object*. We regard such a problem as the core of a task.

After specifying the problem, the next step is thinking about the things we need to solve it. To extinguish a fire, we may need a water hose. To attack the enemy, we could use some guns and canons. We call these things *resources*. Someone might say (s)he *uses* a resource to solve a problem:

use a water hose to extinguish the fire
use guns and canons to attack the enemy
use nothing to fill in the form

These resources are the *indirect objects* in our English sentence.

Finally, we need to think about the product of the task. This is not just the result of the task, it should be *observable* during the entire execution time of the task. The level of detail exposed by the observable is up to the designer. Attacking the enemy can be observed by just a boolean value, indicating failure or success. The task to fill in a form, however, could be observed by exposing all the details filled in by the user. Yielded observables can be resources for follow up tasks:

use a water hose to extinguish the fire for success
use guns and canons to attack the enemy for success
use nothing to fill in the form for a result form

Note we carefully omit the the word *finished* in the context of tasks. Intermediate products yielded during workflow execution are still allowed to serve as resources. A task further down the chain can make decisions based on these observed values. For example, the head of department could approve a working draft, or the system itself can decide to execute the next task when an (user entered) integer is prime. In the next section we will see how we handle such choices.

```
use {} to enter "Application details" for info then
use details = info to check application for verdict then
if not verdict do
  use {} to show "You cannot apply for this subsidy" for {}
if verdict do
    use {} to ask applicant stmt for documents and
    use details = info to ask contractor stmt
        for declaration then
  use applicant = documents, contractor = declaration
      to submit request for {}
```

Figure 2. Code corresponding to the typed workflow in fig. 1.

3.2 Elementary combinations

Having defined what tasks are, we need ways to combine them into a bigger workflow. We distinguish four different ways to do this. First and foremost tasks can be done one after another, thus in *sequence*, or side-by-side, thus in *parallel*.

Beside this we also need a way of branching and deciding which branch to take. We differentiate between two ways of *choice*. The system can make a choice based on values

it knows. These values could be entered into the system beforehand, or read from a centralised place. The condition is predefined by the designer of the workflow. This kind of choice we call *internal*. On the other hand, the system can wait for any end user to perform an action and instruct the system how to proceed. This happens when, for example, the current user likes to continue to the next task, or an external user decides current work done by somebody else is good enough to proceed. We call this an *external* choice.

3.3 Elementary operations

Our typed workflows operate on *values*. To get these values we can interact with the user, or get them out of a (shared) data store. This brings us to four *elementary operations* we can perform in our typed workflows:

1. let the user *enter* into the system
2. *show* information to the user
3. *read* information from a centralised location, such as a database, filesystem, network etc.
4. *write* information to such a shared location

3.4 Formal language

Our workflow language is based on the assumption that every task, at its core, starts with a *problem*. Such a problem can be solved by using zero or more *resources* and during execution yields zero or more *observables*. These observables can be used as resources by other tasks, performed later in the workflow. In the following section we model this using a domain specific functional language.

A grammar to construct typed workflows based on this assumption is given in fig. 3. It uses above elementary operations and combinations. Elementary combinations are **then** and **and**. Internal and external choices are, respectively, a sequence of **if**-**do** or **on**-**do** expressions, guarding a flow with a condition or an action. Executing an operation can be done using the **use** construct. Operations can be enter or show, read or write, or refer to a predefined task t, which can be build using a **def**. We use a record like syntax to supply zero or more resources to a task, leaving out braces to improve readability, as is done in fig. 2. Actions are uppercased labels, which could be presented to the user as buttons or textual options. Conditions are expressions on resources, where $\odot$ stands for the usual equality and comparison operators.

3.5 Type system

A type system using two basic types Bool and Int, function types and record types is given in fig. 4. Off course they need more explanation, but in short they ensure all yielded observables can be used as a resource by succeeding tasks. In particular, note T-Par exposes the union of all observables where T-If and T-On only expose observables with the same name and type in every branch.

Flows	f, g	::=	f **then** g	– Sequence
		|	f **and** g	– Parallel
		|	$(\textbf{if}\ c\ \textbf{do}\ f)^+$	– Internal choice
		|	$(\textbf{on}\ a\ \textbf{do}\ f)^+$	– External choice
		|	**hole**	– A hole
		|	$\textbf{use}\{\overline{x = r}\}\ \textbf{to}\ o\ \textbf{for}\{\overline{y = z}\}$	
		|	$\textbf{def}\ t\ \textbf{using}\{\overline{x : \tau}\}\ \textbf{yielding}\{\overline{y : \sigma}\}$ $\textbf{as}\ f\ \textbf{in}\ g$	
Operations	o	::=	enter	– Enter value
		|	show	– Show value
		|	read s	– Read from store
		|	write s	– Write to store
		|	t	– Task
Resources	r	::=	x	– Variable
		|	v	– Value
Values	v	::=	$\{\overline{x = r}\}$	– Record
		|	$i \in \mathbb{Z}$	– Integer
		|	$b \in \mathbb{B}$	– Boolean
Conditions	c, d	::=	$c \odot d$	– Operator
		|	r	
Actions	a	::=	Ok	
		|	Cancel	
		|	...	
Types	τ, σ	::=	$\{\overline{x : \tau}\}$	– Records
		|	$\tau \to \sigma$	– Functions
		|	Bool	– Booleans
		|	Int	– Integers
Names	x, y, z, s, t	$\in$	$\mathbb{L}$	– Lowercase identifiers

Figure 3. A language for flows. We use the notation $(x)^+$ for *one or more* occurrences of x and $\overline{y}$ for a *possibly empty* comma separated list $y_1, y_2, \ldots, y_n$ of ys.

We treat records in the same way as sets. So a general syntactic record written as $\{\overline{x = r}\}$ having type $\{\overline{x : \tau}\}$ is short for, for example, a specific record $\{i = 37, b = \text{True}, d = 42\}$, having type $\{b : \text{Bool}, d : \text{Int}, i : \text{Int}\}$. Note the order does not matter here, only the labels used to denote the resources. They also serve as sets containing previous assignment and typing pairs.

4 Visual workflows

4.1 Structured creation

Building typed workflows can be supported by an easy step-by-step refinement mechanism. The start of every typed workflow is a **hole**. Holes can be *filled* with with any concrete type of flow: a predefined task to execute, a parallel combination of two or more branches, or an internal or external choice out of two or more branches. The designer can add or remove branches, which also start as holes. In the case

Resources

T-Int
$$\frac{}{\Gamma \vdash i \in \mathbb{Z} : \mathsf{Int}}$$

T-Bool
$$\frac{}{\Gamma \vdash b \in \mathbb{B} : \mathsf{Bool}}$$

T-Record
$$\frac{\forall x.\ \Gamma \vdash r_x : \tau_x}{\Gamma \vdash \{\overline{x = r}\} : \{\overline{x : \tau}\}}$$

T-Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Conditions

T-Condition
$$\frac{\Gamma \vdash c : \tau_\odot \qquad \Gamma \vdash d : \sigma_\odot}{\Gamma \vdash c \odot d : \rho_\odot}$$

Operations

T-Enter
$$\frac{}{\Gamma \vdash \mathsf{enter} : 1 \to \{\overline{y : \sigma}\}}$$

T-Show
$$\frac{}{\Gamma \vdash \mathsf{show} : \{\overline{x : \tau}\} \to 1}$$

T-Read
$$\frac{}{\Gamma \vdash \mathsf{read}\, s : 1 \to \{\overline{y : \sigma}\}}$$

T-Write
$$\frac{}{\Gamma \vdash \mathsf{write}\, s : \{\overline{x : \tau}\} \to 1}$$

T-Task
$$\frac{t : \{\overline{x : \tau}\} \to \{\overline{y : \sigma}\} \in \Gamma}{\Gamma \vdash t : \{\overline{x : \tau}\} \to \{\overline{y : \sigma}\}}$$

Flows

T-Then
$$\frac{\Gamma \vdash f : \{\overline{x : \tau}\} \qquad \Gamma \cup \{\overline{x : \tau}\} \vdash g : \{\overline{y : \sigma}\}}{\Gamma \vdash f\ \mathbf{then}\ g : \{\overline{x : \tau}\} \cup \{\overline{y : \sigma}\}}$$

T-And
$$\frac{\Gamma \vdash f : \{\overline{x : \tau}\} \qquad \Gamma \vdash g : \{\overline{y : \sigma}\}}{\Gamma \vdash f\ \mathbf{and}\ g : \{\overline{x : \tau}\} \cup \{\overline{y : \sigma}\}}$$

T-If
$$\frac{\begin{array}{ccc}\Gamma \vdash c_1 : \mathsf{Bool} & \cdots & \Gamma \vdash c_n : \mathsf{Bool} \\ \Gamma \vdash f_1 : \{\overline{x_1 : \tau_1}\} & \cdots & \Gamma \vdash f_n : \{\overline{x_n : \tau_n}\}\end{array}}{\Gamma \vdash \mathbf{if}\ c_1\ \mathbf{do}\ f_1 \cdots \mathbf{if}\ c_n\ \mathbf{do}\ f_n : \{\overline{x_1 : \tau_1}\} \cap \cdots \cap \{\overline{x_n : \tau_n}\}}$$

T-On
$$\frac{\begin{array}{ccc}\Gamma \vdash f_1 : \{\overline{x_1 : \tau_1}\} & \cdots & \Gamma \vdash f_n : \{\overline{x_n : \tau_n}\}\end{array}}{\Gamma \vdash \mathbf{on}\ a_1\ \mathbf{do}\ f_1 \cdots \mathbf{on}\ a_n\ \mathbf{do}\ f_n : \{\overline{x_1 : \tau_1}\} \cap \cdots \cap \{\overline{x_n : \tau_n}\}}$$

Definitions

T-Def
$$\frac{\begin{array}{c}\Gamma \cup \{\overline{x : \tau}\} \vdash f : \{\overline{y : \sigma}\}' \qquad \{\overline{y : \sigma}\} \subseteq \{\overline{y : \sigma}\}' \\ \Gamma \cdot t : \{\overline{x : \tau}\} \to \{\overline{y : \sigma}\} \vdash g : \{\overline{z : \rho}\}\end{array}}{\Gamma \vdash \mathbf{def}\ t\ \mathbf{using}\{\overline{x : \tau}\}\ \mathbf{yielding}\{\overline{y : \sigma}\}\ \mathbf{as}\ f\ \mathbf{in}\ g : \{\overline{z : \rho}\}}$$

T-Use
$$\frac{\Gamma \vdash \{\overline{r : \tau}\} \qquad \Gamma \vdash o : \{\overline{x : \tau}\} \to \{\overline{y : \sigma}\}}{\Gamma \vdash \mathbf{use}\{\overline{x = r}\}\ \mathbf{to}\ o\ \mathbf{for}\{\overline{y = z}\} : \{\overline{z : \sigma}\}}$$

Figure 4. Type system for the workflow language specified in fig. 3

the designer fills a hole with a choosing construct, (s)he has to specify conditions or actions when to follow a particular branch.

Using above steps we can direct the designer to create typed workflows in a structured way. Because just a handful of options are available to the designer, it also eases the process of creating a typed workflow. The developing environment aids in producing correct flows by highlighting incorrect use of values flowing through the diagram. Giving these hints to the designer is ongoing work.

5 Related Work

iTasks The iTasks system [Plasmeijer et al. 2012] is a novel way to specify workflows and derive a complete full functional web application out of a task specification. To our own knowledge, this framework is currently the only one using type directed workflow modelling. However, because iTasks is implemented as an EDSL in the functional programming language Clean [Plasmeijer et al. 2011], professionals have a hard time understanding these specifications, jet alone creating one themselves. Although a visualisation library exists,[Stutterheim et al. 2015, 2014] it is only capable to generate graphical representations of the task specification, there is no way to alter the specification in this formalism. At the basis iTasks uses just two combinators: `step` and `parallel`, from which all other combinators are derived. We introduced an expressive but simple subset of combinators to design and adapt existing workflows in spirit of the iTasks framework.

BPMN BPMN makes a distinction between six types of gateways [Model 2011]. Of these gateways two are based on conditions and two on events. The remaining two are parallel composition and a "catch all" complex gateway (only recommended to use when all the others are too simple). We simplify these six *gateways* to three ways of branching: parallel, and internal and external choice. There are no *events*, only actions by the user. All other type of events are modelled by other flows containing internal choices.

In contrast to BPMN models, flows are created in a modular fashion. Essentially every task in a flow is a subprocess specified by another flow, or a basic task interacting with the end user. Therefore, there is just one kind of *activity*: a task.

Most notable difference is off course the addition of *information* needed to complete a task, something BPMN lacks.

Workflow Patterns Van der Aalst and Ter Hofstede [2003] proposes over forty patterns to captures aspects related to control-flow dependencies between various tasks. Most notable are the distinction between a *split* and a *merger*. Flows are always synchronised, both parallel branching and conditional branching are automatically synchronised after all or one of the branches complete. We explicitly decided to make

all branching *deterministic*. Making a choice is *always* done at the branch split by the user (external) or based on values (internal). This gives rise to a simpler model and to an easier implementation while maintaining expressive power.

Instead of differentiating between internal and external choices, Van der Aalst [1998] makes a distinction on the *moment* when a choice is made. This could be right before branching (*explicit*) or by one of the subtasks after branching (*implicit*). The implicit choice is mostly used to model workflows containing some kind of *trigger*, which are special tasks acting as guards in a workflow. Instead of creating a non-deterministic system with tasks acting as triggers, we use *actions* and *external choices* to enable deterministic models of workflows.

6 Future Work

Future investigation of typed workflows entails a correct (operational) semantics of the formal workflow language presented in fig. 3. Also, we would like to investigate the relation of typed workflows to Workflow Nets as proposed by Van der Aalst [1998]. Next to this, the current prototype lacks some important user facing features, like data type declarations, parameter passing and a hinting system. All this should be mitigated in future versions. The prototype should be tested in practice and checked for completeness during modelling.

The biggest challenge however, is extending our tool in such a way an designer as well as a contractor should be comfortable with the view they get on a workflow when they are working on one. The current implementation is mainly directed towards workflow designers, giving options to specify workflows on a high level. A contractor with more programming experience may want fix transitions between tasks and fill in this gaps using his own formalism.

Possible extensions to typed workflows are incorporating a resource analysis as described by Klinik et al. [2017]. Customisation of the generated user interface are not yet possible, although it is possible to do using the current backend (iTasks). User interface customisation can be complicated, and probably should not be exposed to designers, but only to contractors. Incorporating the ideas previously investigated by Achten et al. [2016] are crucial to produce good looking and usable end user applications.

Acknowledgments

The authors likes to thank Markus Klinik for discussing ideas about resources, Johan Jeuring for being open to all ideas, and Peter Achten for long discussions. Above all many thanks to Nico Naus for discussing wonderful ideas during unforeseen events.

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

References

Peter Achten, Jurriën Stutterheim, Bas Lijnse, and Rinus J Plasmeijer. 2016. Towards the Layout of Things. (2016), 1–13.

Markus Klinik, J Hage, J M Jansen, and Rinus J Plasmeijer. 2017. Predicting resource consumption of higher-order workflows. (2017).

B P Model. 2011. *Notation (BPMN) version 2.0.* OMG Specification.

Rinus J Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W M Koopman. 2012. Task-oriented programming in a pure functional language. *PPDP* (2012).

Rinus J Plasmeijer, Marko C J D van Eekelen, and John H G van Groningen. 2011. *Clean Language Report.* Technical Report. Radboud University, Nijmegen.

Jurriën Stutterheim, Peter Achten, and Rinus J Plasmeijer. 2015. Static and Dynamic Visualisations of Monadic Programs. In *Implementation and Application of Functional Programming Languages*. 1–12.

Jurriën Stutterheim, Rinus J Plasmeijer, and Peter Achten. 2014. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In *Trends in Functional Programming*. 1–20.

WMP Van der Aalst. 1998. The application of Petri nets to workflow management. *Journal of circuits* 08, 01 (1998), 21–66.

WMP Van der Aalst and AHM Ter Hofstede. 2003. Workflow patterns. *Distributed and parallel ...* (2003).

Warble: An eDSL for cyber physical systems*

Extended Abstract

Matthew Ahrens
Tufts University
Medford, Massachusetts,USA
matthew.ahrens@tufts.edu

Kathleen Fisher
Tufts University
Medford, Massachusetts,USA
kathleen.fisher@tufts.edu

CCS Concepts •**Computer systems organization** → *Sensors and actuators; Real-time languages;*

Keywords Programming Languages, Functional Languages, Cyber Physical Systems, Sensors and Actuators

ACM Reference format:
Matthew Ahrens and Kathleen Fisher. 2017. Warble: An eDSL for cyber physical systems. In *Proceedings of Implementation and Application of Functional Languages, Bristol, UK, August 2017 (IFL 2017),* 2 pages.
DOI:

Cyber physical systems (CPS) perform logical computation with respect to an expected physical model. To represent logical computation, programs transform data provided by sensors to drive actuators. To ensure sensors and actuators adhere to the physical model, the program periodically checks runtime conditions[4]. When sensors and actuators differ from the physical model, the program takes action to correct their behavior. For example, typical Raspberry Pi and Arduino CPS programs insert delay statements into imperative code to correct hardware call rates [7, 10, 11]. By introducing delay statements after hardware calls, sensors and actuators produce fewer erroneous values and actions, respectively, at the cost of timely precision.

Unfortunately, when delay statements occur infrequently or under complex runtime conditions, debugging becomes difficult because programmers must reconstruct complex program state[8]. When programmers reuse code, abstraction barriers (e.g. functions, objects, etc.) hide delay statements causing synchronization problems the calling program cannot undo. When programmers are uncertain about delay statements, they cannot adapt hardware behavior to expected physical behavior [2]. Therefore, to make hardware calls consistent, programmers only execute delay statements at the top level program rather than in the called abstraction. Manually producing this separation requires programmers to write a lot of glue code.

An effect system or scheduler could alternate program logic and hardware calls with delay statements [13]. With this approach, program logic returns data structures containing runtime conditions, expected delays, logical operations, and hardware calls. For heavily conditional expressions, these programs quickly become unintelligible.

A solution can statically solve for delay instructions when a set of restrictions are enabled:

- Hardware calls occur at constant rates
- Program logic transforms fixed-sized windows of data
- Control flow expressions preserve rates

To support consistent and reusable CPS programs, we have designed the eDSL *Warble*. Warble provides a small expression language and type system to simplify CPS programming. Warble *sources* and *sinks* drive sensors and actuators. These sources and sinks have concrete rates expressed as either constant delay times (e.g. 500 ms) or ranges. *Transforms* relate input rates to output rates when manipulating windows of data. Other control flow expressions preserve rates. This rate preservation echoes the "natural rate" and "throughput ratio" from Bartenstein and Liu's Stream Programs[1]. In contrast to this earlier work, Warble compiles CPS programs to computation tables, which handle data windows, obviating the need for a generic linear constraint solver.

To perform general purpose computation, Warble splices target language code (e.g. Haskell) into source, sink, and transform expressions[12]. Warble leverages the power of the target language in comparison to other CPS eDSLs, such as Kansas Lava, which restricts the target language for more control[5].

The Warble type system defers to the type system of the target language for the spliced code while checking rate types of Warble expression directly. Warble's type checking eliminates programs that drop data, redundantly use data, or call hardware inconsistently. Consequently, Warble programmers do not need timing-related glue code.

To better model physical behavior, Warble opts for concrete rates (e.g. milliseconds) over relative rates (e.g. event order). Other synchronous programming languages choose relative rates to enforce general static guarantees with respect to program logic[9, 10]. Instead, Warble assumes program logic and hardware calls are fixed-time operations.

To illustrate Warble, we present, as an example, a Warble program implementing a self-regulating green house. Warble's type system solves the temperature sensor and light actuator call rates. Warble's compiler produces a computation table with explicit delay statements. This real-time compatible form enables compilation to multiple hardware platforms.

In future work, we will extend Warble's runtime system to support error information for debugging concrete delay violations at runtime, e.g. when program logic exceeds fixed-time. The computation table will accept target expressions other than Haskell, e.g. C, for embedded system deployment. Physical expressions and types will enable a simulator to check expected hardware behavior before deploying on actual hardware. Prior work uses ordinary differential equations, but Warble will provide an expression language similar to the existing one for programmer ease-of-use[3]. Finally, since Warble's programs do not handle state themselves, the semantics can be extended to support distributed CPS systems like sensor networks[6].

*embedded domain specific language

IFL 2017, Bristol, UK
2017. . . .$15.00
DOI:

References

[1] Thomas W. Bartenstein and Yu David Liu. 2014. Rate Types for Stream Programs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 213–232. DOI: http://dx.doi.org/10.1145/2660193.2660225

[2] Kevin Boos, Chien-Liang Fok, Christine Julien, and Miryung Kim. 2012. Brace: An Assertion Framework for Debugging Cyber-physical Systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1341–1344. http://dl.acm.org/citation.cfm?id=2337223.2337413

[3] Timothy Bourke and Marc Pouzet. 2013. ZéLus: A Synchronous Language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC '13)*. ACM, New York, NY, USA, 113–118. DOI: http://dx.doi.org/10.1145/2461328.2461348

[4] Manfred Broy. 2013. Challenges in Modeling Cyber-physical Systems. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks (IPSN '13)*. ACM, New York, NY, USA, 5–6. DOI: http://dx.doi.org/10.1145/2461381.2461385

[5] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. 2011. Types and Type Families for Hardware Simulation and Synthesis: The Internals and Externals of Kansas Lava. In *Proceedings of the 11th International Conference on Trends in Functional Programming (TFP'10)*. Springer-Verlag, Berlin, Heidelberg, 118–133. http://dl.acm.org/citation.cfm?id=2035141.2035149

[6] Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. 2015. The Remote Monad Design Pattern. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 59–70. DOI: http://dx.doi.org/10.1145/2804302.2804311

[7] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. DOI: http://dx.doi.org/10.1145/1592761.1592779

[8] Guido Salvaneschi and Mira Mezini. 2016. Debugging Reactive Programming with Reactive Inspector. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 728–730. DOI: http://dx.doi.org/10.1145/2889160.2893174

[9] Francisco Sant' Anna, Roberto Ierusalimschy, and Noemi Rodriguez. 2015. Structured Synchronous Reactive Programming with CéU. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, USA, 29–40. DOI: http://dx.doi.org/10.1145/2724525.2724571

[10] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-scale Embedded Systems. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 36–44. DOI: http://dx.doi.org/10.1145/2892664.2892670

[11] R. Benjamin Shapiro and Matthew Ahrens. 2016. Beyond Blocks: Syntax and Semantics. *Commun. ACM* 59, 5 (April 2016), 39–41. DOI: http://dx.doi.org/10.1145/2903751

[12] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. DOI: http://dx.doi.org/10.1145/581690.581691

[13] Atze van der Ploeg. 2013. Monadic Functional Reactive Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, New York, NY, USA, 117–128. DOI: http://dx.doi.org/10.1145/2503778.2503783

Multiplayer Web Game Development with Purely Functional Programming

Extended Abstract

Rogan Murley
rogan@roganmurley.com

ABSTRACT

We introduce a multiplayer web game implemented with purely functional programming, and discuss the motivation behind this approach.

We show that the game is written in Haskell and Elm [3, 4]. While familiarity with Haskell is assumed, Elm is introduced as a purely functional reactive programming language that compiles to JavaScript. We compare and contrast our choices with other options, paying particular attention to PureScript and ghcjs [1, 2].

We explore the game's architecture. We look at the server's "Onion Architecture", multi-threading and client-server synchronisation [5].

We review how the vast majority of bugs occur at the boundary between client and server, limiting the utility of a sophisticated type system. We consider ways to bridge this boundary.

We discuss the surprising benefits of applying a purely functional mindset to other disciplines. In particular, we see how immutable deployments can simplify system administration, and how functionally-inspired game design can provide depth without the cost of complexity.

We evaluate the purely functional approach, comparing and contrasting with imperative programming. We consider what could be done to improve purely functional game programming in the general case, specifically discussing laziness and garbage collection.

We describe future work to further the implementation. We discuss the free monad interpreter pattern, and invite feedback on its applicability [6].

CCS CONCEPTS

• **Applied computing → Computer games**; • **Software and its engineering → Functional languages**;

KEYWORDS

Haskell, Elm, functional programming, game development, web applications

ACM Reference format:
Rogan Murley. 2017. Multiplayer Web Game Development with Purely Functional Programming. In *Proceedings of IFL, Bristol, UK, August 2017 (IFL 2017),* 1 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

IFL 2017, August 2017, Bristol, UK

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

REFERENCES

[1] 2017. ghcjs: Haskell to JavaScript compiler, based on GHC. (2017). Retrieved August 20, 2017 from https://github.com/ghcjs/ghcjs
[2] 2017. PureScript: A small strongly typed language that compiles to Javascript. (2017). Retrieved August 20, 2017 from https://github.com/purescript/purescript
[3] Evan Czaplicki. 2012. *Elm: Concurrent FRP for Functional GUIs.* Master's thesis. Harvard University.
[4] Evan Czaplicki. 2017. Elm. (2017). Retrieved August 20, 2017 from http://elm-lang.org/
[5] John A. De Goes. 2016. Modern Functional Programming: Part 2. (2016). Retrieved August 20, 2017 from http://degoes.net/articles/modern-fp-part-2
[6] Wouter Swierstra. 2008. Data types à la carte. (2008).

The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines

Markus Klinik
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
m.klinik@cs.ru.nl

Jan Martin Jansen
Netherlands Defence Academy (NLDA)
Den Helder, The Netherlands
jm.jansen.04@mindef.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

Abstract

In previous work we presented a static analysis for costs of higher-order workflows, where costs are maps from resource types to numbers. In this paper we extend the notion of cost with time. Costs are functions over time, one function for each type of resource a program requires. We present a type and effect system together with an algorithm that yields safe approximations for the cost functions of programs.

CCS Concepts • **Theory of computation → Program analysis**;

Keywords workflow systems, resource modelling, type and effect systems

ACM Reference Format:
Markus Klinik, Jan Martin Jansen, and Rinus Plasmeijer. 2017. The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines. In *Proceedings of The 29th symposium on Implementation and Application of Functional Languages (IFL'17).* ACM, New York, NY, USA, 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In previous work [Klinik et al. 2017] we presented a static analysis for workflows that, given costs for basic tasks, yields a safe approximation of the cost of a whole program. Costs are maps from resource types to numbers, one number for each type of resource a program requires. In this paper we refine the notion of cost by including time, in the sense that each basic task gets a duration, and its cost refers to this duration. The result of the extended analysis is not a sigle number, but a function over time.

1.1 Basic Ideas

We would like to represent the cost of executing tasks by functions over time. As complex tasks are composed of simpler ones, so should be their cost functions. By combining cost functions of simple tasks using operations that correspond to task composition, we obtain cost functions for the complex tasks.

Example 1.1. Consider the two tasks of hosting a brithday party and a wedding, which both require chairs. Let's say the birthday party requires 20 chairs and takes five hours, while the wedding requires ten chairs and takes ten hours. We can visualize these costs over time, called *skylines*, in diagrams. The skylines are shown in Figure 1.

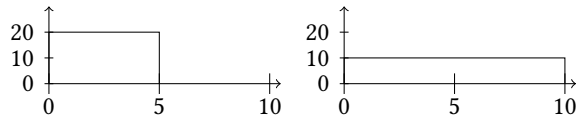


Figure 1. Cost skylines for a birthday and a wedding

Hosting the birthday party first and immediately after it the wedding requires being able to supply 20 chairs for the first five hours and 10 chairs for the hours 5 to 15. The corresponding skyline is shown in Figure 2.

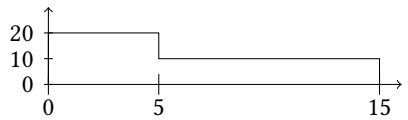


Figure 2. Cost skyline for a birthday *followed by* a wedding

Hosting the birthday party and the wedding at the same time requires 30 chairs for the first five hours, and 10 chairs for the remaining five hours. See Figure 3.

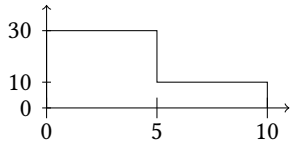


Figure 3. Cost skyline for a birthday *and* a wedding

If, for whatever reason, either the wedding or the birthday takes place, then actually either 20 chairs for five hours or

IFL'17, August 30–September 1, 2017, University of Bristol, UK
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

10 chairs for ten hours are used. In order to be prepared for both situations, a host must calculate with 20 chairs for five hours and ten chairs for another five hours. See Figure 4.

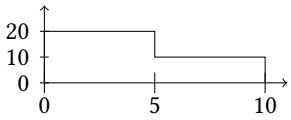


Figure 4. Cost skyline for a birthday *or* a wedding

This example illustrates that for the overall skyline of two tasks, depending on how the tasks are composed, their individual skylines must be combined in different ways. For sequential composition, the skylines must be concatenated. For parallel composition, the skylines must be summed up. An exclusive choice requires the least upper bound of the two skylines. In this example, we considered chairs, which are a reusable resource. Combining skylines of consumable resources works differently, because those can only increase and never decrease. A resource that has been used up stays used up.

1.2 Challenges

The two big challenges with this approach are polymorphism and recursion.

For polymorphic functions, the cost of the result depends on the cost of the arguments, and this dependency has to be encoded in the type. A similar approach to the one we used for costs as single numbers might work: Effect variables in types stand for unknown costs, and types incorporate sets of constraints that relate cost variables. Where constraints talked over single numbers before, they now must talk over whole skylines. The principle approach for solving constraints should work as before, because the domain of skylines is a complete lattice. The details however become more complicated because skylines can grow in height and in length.

The problem with general recursion is, as with all kinds of static analyses, that it can create situations in which no upper bound can be found, because none exists or finding it is undecidable. The analysis has to recognize these situations and give the worst-case overapproximation, in our case infinite cost. For type and effect systems based on constraint solving this is usually done with a widening operator. A widening operator is a heuristic that jumps to infinity immediately when it suspects a loop in the constraints that leads to non-termination in the solver.

The widening operator in the system with single numbers had to deal with infinitely high costs. The widening operator for skylines has to deal with skylines that can become infinitely long, as well as infinitely high. Consider the skyline in Figure 5. There are many ways to approximate it, and choosing the right approximation is one of the goals of this paper.

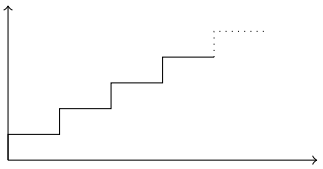


Figure 5. What should be the approximation of this skyline?

The remainder of this paper is organized as follows. In section 2 we formalize the notion of skylines and tasks, and define an operational semantics that constructs skylines during task execution. The main contribution of this paper is in section 3, where we develop a type and effect system that yields a prediction skyline that safely approximates any actual skyline arising from program execution. Section 4 describes an algorithm that implements the type and effect system. Finally, section 5 discusses capabilities and limitations of the method on the basis of example programs.

2 Syntax and Semantics

In this section we define the syntax and operational semantics of a programming language to specify workflows.

We consider two kinds of resources, *consumables* and *reusables*. Consumable resources are used up when a task that requires them is executed. Reusable resources are claimed exclusively during execution of a task and become available again upon completion.

We assume that it is implicitly understood which resources are consumable and which are reusable.

2.1 A Programming Language for Workflows

Our language is a simplified version of Clean and iTasks. It is a small functional programming language with higher-order functions, non-recursive let-bindings and a fixpoint combinator. Tasks and workflows exist as domain-specific constants and combinators in the language.

$$
\begin{aligned}
e ::= {} & b \mid i \mid () \mid x \mid \mathbf{fn}\, x.e \mid \mathbf{fix}\, x.e \mid e_1 e_2 \mid \\
& \mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1 \odot e_2 \mid \\
& \mathbf{use}\ (k, t)\ e \mid \mathbf{return}\ e \mid \\
& e_1\ \&\ e_2 \mid e_1 \gg\!= e_2 \mid e_1 \gg e_2 \\
k ::= {} & n \cdot u \mid n \cdot u + k
\end{aligned}
$$

The general-purpose part of the language has Boolean and integer constants b and i, the unit value (), program variables x, abstraction, application, if-then-else and let-bindings. The symbol $\odot$ stands for the usual binary operators for arithmetic, Boolean connectives and comparison. There is a fixpoint combinator $\mathbf{fix}\, x.e$ that defines recursive expressions. This

variant of the fixpoint combinator is common in languages with call-by-name semantics.

The domain-specific part of the language has a primitive **use** for basic tasks, and task combinators for sequential and parallel composition. All basic tasks are represented by the **use** operator, where k denotes the cost of executing the task, and t its duration. Costs are given in a polynomial-like syntax where n is a natural number and u the unit of a resource. Durations t are given as non-zero natural numbers. For example **use** $(2S+3B, 20)$ 5 may denote a task that uses 2 screwdrivers, 3 bottles of wine, takes 20 minutes to execute, and yields the value 5. Costs, time, and resources are discussed in more detail in Section 2.2.

Expressions of the form **return** e denote tasks that have been executed and return a value e.

There are three combinators for tasks, the parallel combinator (&) and two variants of sequential composition. The regular *bind* operator (≫=) executes its left argument first and passes the resulting value to its right argument as usual. The *sequence* operator (≫) ignores the value of its left argument and yields the value of its right argument. The sequence operator is useful for example programs where the values of tasks do not matter. We include it in our language for convenience, being well aware that it can easily be defined in terms of bind. Since ≫ is a variant of ≫= with identical cost behavior, we ignore it in the formal part of this paper but still use it in examples.

The parallel combinator executes both its arguments simultaneously. In iTasks the result value of parallel composition is a tuple containing the values of both tasks. Our language does not have tuples, a deliberate decision because we want to focus more on side effects than on values. Adding tuples would inflate the various definitions relating to our language while not providing substantial new insights regarding cost analysis. We therefore take the liberty of bending the semantics of the parallel combinator a bit to avoid tuples. Both arguments to (&) and its result are of type *task* ().

2.2 A Domain for Representing Costs Over Time

Definition 2.1. (Extended natural numbers) The extended natural numbers are the natural numbers with infinity: $\overline{\mathbb{N}} = \{0, 1, 2, \ldots, \infty\}$. The only operations we need are addition and comparison, which are extended with ∞ in the obvious way. □

Definition 2.2. (Cost) Let U be a finite set. A *cost over* U is a function $\gamma : U \to \overline{\mathbb{N}}$. Usually, U is understood from the context and we just talk about *costs*. □

The set U contains the units of the resources of interest in a given workflow. For example, let

$$U = \{\text{Screwdrivers}, \text{Chairs}, \text{Potatoes}\}.$$

We can then express that a task uses 5 screwdrivers, 2 chairs, and no potatoes by associating the following cost to it

$$[\text{Screwdrivers} \mapsto 5,\ \text{Chairs} \mapsto 2,\ \text{Potatoes} \mapsto 0].$$

We shall adopt a polynomial-like notation $5S + 2C + 0P$ for this.

Definition 2.3. (Time) For our purposes, a point in time is a natural number. A duration is a non-zero natural number. □

Definition 2.4. (Cost skylines) The cost over time for a single resource is given as a time series $\mathbb{N} \to \overline{\mathbb{N}}$, which we call a *skyline*. Costs involving multiple resources are just multiple skylines, given as a U-indexed family of skylines. □

We compactly encode skylines as lists of tuples $\mathbb{N} \times \overline{\mathbb{N}}$, where the first component denotes a point in time and the second component the cost at that time. The cost at a point not in the list is the cost at the next earlier point in the list, and 0 if there is no such. We assume that skylines encoded that way are always sorted by the time component.

Not all skylines can be encoded as finite lists of that type, but the ones we are interested in can be. Infinitely long tasks can result from infinite loops, and their skylines cannot always be encoded with finite lists. These cases will be treated carefully.

Example 2.5. Consider the skyline $[\langle 1, 2\rangle, \langle 5, 4\rangle, \langle 8, 0\rangle]$. Figure 6 shows a graphical representation of this skyline. If a task has this skyline for a reusable R, it means that the task requires $2R$ from time 1 until 5, $4R$ from time 5 until 8, and after 8 it has released all Rs. In other words, at time 1 the task claims 2 resources, at time 5 it claims 3 more for a total of 5, and at time 8 it releases all resources. □

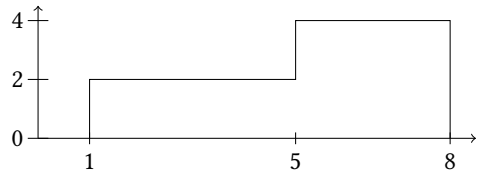


Figure 6. A skyline for cost 2 between 1 and 5, and cost 4 between 5 and 8.

Definition 2.6. (Skyline ordering) Skylines are ordered pointwise. Let r, s be skylines. Then

$$r \sqsubseteq s \text{ iff } r(t) \sqsubseteq s(t) \text{ for all } t \in \mathbb{N}.$$

Definition 2.7. (Merging skylines) The merge of two skylines ($\sqcup$) is defined as their pointwise maximum.

$$(r \sqcup s)(t) = r(t) \sqcup s(t)$$

Proposition 2.8. *The set of all skylines $\mathbb{S}$ together with its ordering $\sqsubseteq$ is a complete lattice.*

The least upper bound of two skylines is their merge. The top element of the lattice is the skyline that rises to infinity at time 0. The bottom element is the skyline that stays at 0 at all times.

2.3 Operational Semantics

We split the operational semantics in two parts. Both are call-by-name small-step structural operational semantics with substitution. The general purpose part, denoted by a normal arrow $\rightarrow$, applies to non-task expressions. The domain specific part of the semantics applies to task expressions. It is denoted by a two-headed arrow $\twoheadrightarrow$ to emphasize its relation with the bind operator $\gg=$.

The general purpose semantics relates expressions using rules of the form $e \rightarrow e'$. The domain specific semantics with rules of the form $\langle s, e\rangle \rightarrow \langle s', e'\rangle$ relates expressions e while constructing a skyline s that records the resources that are used during reduction. The names of the rules of the semantics are prefixed by gs- and ds-, which are to be read as "general purpose step" and "domain specific step".

In order to define the semantics for parallel composition, we add a new syntactic form to the language, called *process pool*. When the parallel composition of two tasks needs to be reduced, a process pool springs into existence and keeps track of the costs of the tasks individually, so that no sharing of resources takes place. Process pools only exist temporarily during reduction and disappear once both tasks have been executed. Process pools in our language hold exactly two tasks.

$$e ::= \ldots \mid pp(e_1 \,\&\, e_2, s_1, s_2)$$

In a process pool, e_1 & e_2 are two tasks in progress of being executed, and s_1 and s_2 are their respective skylines.

The general purpose semantics is given in Figure 7.

$$\text{[gs-fix]} \quad \mathbf{fix}\, x.e \rightarrow e[x \mapsto \mathbf{fix}\, x.e]$$

$$\text{[gs-app-cong]} \ \frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\text{[gs-app-reduce]} \quad (\mathbf{fn}\, x.e_b) e_x \rightarrow e_b[x \mapsto e_x]$$

$$\text{[gs-let]} \quad \mathbf{let}\ x = e_x\ \mathbf{in}\ e_b \rightarrow e_b[x \mapsto e_x]$$

$$\text{[gs-if-cong]} \ \frac{e_c \rightarrow e_c'}{\mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e \rightarrow \mathbf{if}\ e_c'\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e}$$

$$\text{[gs-if-t]} \quad \mathbf{if\ True\ then}\ e_t\ \mathbf{else}\ e_e \rightarrow e_t$$

$$\text{[gs-if-f]} \quad \mathbf{if\ False\ then}\ e_t\ \mathbf{else}\ e_e \rightarrow e_e$$

Figure 7. Small-step semantics for general purpose expressions

The domain specific semantics is given in Figure 8.

TBD

Figure 8. Small-step semantics for task expressions

3 Static Semantics

TBD

4 Implementation

TBD

5 Discussion

TBD

References

Markus Klinik, Jurriaan Hage, Jan Martin Jansen, and Rinus Plasmeijer. 2017. Predicting resource consumption of higher-order workflows. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017*, Ulrik Pagh Schultz and Jeremy Yallop (Eds.). ACM, 99–110. http://doi.acm.org/10.1145/3018882

Probabilistic resource analysis with dependent types.

Christopher Schwaab
University of St Andrews
cjs26@st-andrews.ac.uk

Edwin Brady
University of St Andrews
ecb10@st-andrews.ac.uk

Kevin Hammond
University of St Andrews
kevin@kevinhammond.net

Abstract

Verifying non-functional properties of programs, such as time and/or energy consumption, is often a difficult task. With emerging applications such as the internet-of-things, however, it is becoming increasingly important that devices operate reliably *within known time and energy bounds*. At the same time, processor architectures are becoming more complex. This means that traditional, usually *compositional*, modeling techniques are not effective. To solve this problem, we propose the introduction of fuzzy bounds that will allow us to reason formally about the *probabilistic* consumption of resources, in terms of program inputs. Our approach is type-directed, relating program sources directly to their probabilistic costs. We illustrate our approach with reference to a simple example, showing how to determine that a program's resource consumption is probabilistically within some given bound of the costs as determined by its operational semantics.

CCS Concepts • **Theory of computation → Semantics and reasoning**; • **Computing methodologies** → *Model development and analysis*; Machine learning;

Keywords ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Christopher Schwaab, Edwin Brady, and Kevin Hammond. 2017. Probabilistic resource analysis with dependent types.. In *Proceedings of IFL 2017, Bristol, UK, August 2017 (IFL'17),* 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As processor architectures become more complex, and as compiler optimizations become more sophisticated, so it is increasingly difficult to precisely and accurately determine non-functional software properties, and to verify that required bounds on these behaviors are met. Properties such as the consumption of time and energy (*resource usage*) are often tightly tied to the underlying processor Instruction Set Architecture (ISA). Consider, for example, a simple multiplication, which might be expressed as `y = x*5`. With optimization disabled, on an x86 machine the compiler might be expected to emit an integer multiplication (`imul`) instruction of the form, `imul rcx, rax*5`, where `rax` is the source register and `rcx` is the destination. With optimizations enabled, however, the compiler might produce a single complex instruction, `lea rcx, [rax+rax*4]`. This instruction completely avoids the use of the ALU, using an address calculation to produce the required result. Similarly, on an ARM target architecture, the compiler might use the barrel shifter to evaluate the shift and add in a single instruction `add r1, r1, r1, lsl 2`. Each of these solutions has a distinct time and energy profile. Developing accurate resource usage cost models requires a tight coupling with the processor ISA and internal state. However, it is well-known that only the very simplest current architectures have predictable and context-free timing and energy properties at the instruction level. In order to obtain worst-case bounds, it is typically necessary to make highly pessimistic assumptions about individual instructions and/or to include detailed contextual information, such as the state of the cache and/or pipeline. This often leads to significant over-estimates. Composing these bounds to give overall expression, function or program costs, then magnifies the error. This can yield practically unusable cost estimations in terms of gross estimates of resource usage.

In this paper, we instead propose the use of a well-founded statistical approach to learn the resource usage of a program, and combine this with a symbolic, type-based approach to reason about that resource usage. This yields a probabilistic, automatable resource analysis. Unusually, we directly involve the programmer in determining the cost of their program, exposing non-functional properties as first-class citizens in the form of programmable (dependent) types. We leave it as an open question as to whether these types can be automatically inferred in future. Our approach is loosely coupled with the underlying ISA and compiler through *probability distribution functions* over the observed execution time (and energy etc). This is completely generic in the sense that new ISAs (or processor implementations) and compilers/optimizations can be easily targeted without human intervention and without building a complete instruction-accurate operational semantics, as with e.g. Hume [13] CompCert [19], or van Gastel et al's approach [25]. The only requirement is that we need to collect any additional training data that is necessary to re-learn the distributions of time,

IFL'17, August 2017, Bristol, UK

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

energy etc. behavior. Since we evaluate a program's resource consumption using *probability distribution functions* (PDFs), this has the unfortunate side-effect of introducing potentially unbounded error. In this paper we seek to mitigate this error by developing a pattern for mechanically reasoning about the *accuracy* and *confidence* of a probabilistic cost, so limiting the error within probabilistic bounds.

We start by assigning a probabilistic cost for resource usage to individual program expressions. These are composed in a way that follows the underlying *program shape*, taken to be the expression's *features*. Intuitively, a single operation's shape can be obtained by dropping its arguments and an entire expression's shape can be represented as the flattened tree of operation shapes. Individual expressions are costed using an opaque heuristic $h^\star$. This is programmatically evaluated by observing many expressions of the same shape in different contexts. The resource consumption of a term in each shape is assumed to follow a *sub-Gaussian* distribution, that is a probability distribution with a strong tail decay property. This assumption is the key to allowing us to reason about *potential error*, admitting the application of well established *concentration bounds*.

1.1 Novel Contributions

The main contributions of this paper are:

1. we describe a novel, type-directed, probabilistic approach to resource analysis;
2. we introduce a type-level framework for mechanically reasoning about a program's non-functional costs;
3. we show that our resource analysis is sound wrt an operational semantics for program execution, producing results that are probabilistically within some required bound of the expected value.

While our method makes no explicit assumption about the underlying resource that is being measured, this paper focuses on predicting the running times of programs, as a representative resource of interest. We will illustrate our approach with respect to a simple worked example whose timing bounds can be easily determined.

2 Cost-indexed terms

We begin with a simple language of expressions with conditionals and simple, incremented *for* loops. The cost of a conditional is taken to be the branch with the highest cost. Consider an implementation of the standard factorial function:

$$fact\ n = \text{for}\ 1\ n\ \lambda s.n * s$$

What will be required to give an estimate of the running time this function will require given some value n? Ideally we should have an estimate for:

1. the time that is required to test whether to exit the loop when n is 0;
2. the cost of the multiplication operation; and
3. the overhead of performing the looping.

We will now specify our input language, $\mathcal{L}$, formalize a *measurement* of a program's execution time, and present a term-indexed cost relation that can be used to statically predict that execution time.

$$\begin{aligned}
&GroundTy : \star ::= Nat \mid Bool \\
&Ty : \star ::= GroundTy \mid Arrow\ \overrightarrow{GroundTy}\ GroundTy \\
&Value : Ty \to \star ::= b \in \mathbb{B} \mid n \in \mathbb{N} \mid x \in \mathcal{V} \mid \beta \in Builtin \\
&Builtin : Ty \to \star ::= \beta + \beta \mid \beta * \beta \\
&Ground : Ty \to \star ::= v_t \mid \beta_t \\
&Exp : Ty \to \star ::= v_t \mid \mathcal{G}_t \mid f_{Arrow\ \vec{t}\ t}\ \overrightarrow{Exp_{\vec{t}}} \\
&Stmt : Ty \to \star :: = Exp \\
&\qquad\qquad \mid \text{let}\ Exp\ \lambda x.Stmt \\
&\qquad\qquad \mid \text{for}\ Exp\ Exp\ \lambda x\ n.Stmt \\
&\text{Func} : Ty \to \star ::= f\ \vec{x} = Stmt
\end{aligned}$$

Figure 1. Syntax of the simple Source Language, $\mathcal{L}$.

2.1 Source language, $\mathcal{L}$

We will perform our analysis on type-indexed first-order families of *statements* and *expressions*, as exemplified by the language $\mathcal{L}$ of Figure 1. In the style of Hume [13], we break $\mathcal{L}$ into two layers: *expressions*, whose terms can be given an atomic cost by some heuristic function $h^\star$, and *statements*, the *coordination layer*, whose terms cannot be given a cost by $h^\star$. Statements instead derive their cost *structurally*, by threading together expressions and their constituent costs.

The sub-language of expressions, E, comprises a family of type-indexed values $v \in Value : Ty \to \star$, functions drawn from the set $f \in \mathcal{F}$, variables drawn from the type-indexed family of sets $x \in \mathcal{V} : Ty \to \star$, builtin functions drawn from $\beta \in Builtin : \overrightarrow{Ty} \to \star$, function applications, and *ground normal* expressions $\mathcal{G} \in Ground : GroundTy \to \star$. Ground normal expressions are those with no occurrences of free function names, i.e. they form trees of builtin functions over values. Functions are fully applied and there is a strong distinction between builtin functions that accept arbitrarily complex arguments, and defined functions that are in a normal form, and only accept variables as arguments.

The coordinating sub-language of statements, S, is made up of expressions, let-bindings, non-recursive functions, and loops iterating over natural numbers. To simplify the analysis, we do not currently allow closures to be present in an executing program. This is purely a technical limitation, since defunctionalisation could be used if required, prior to analysis. Furthermore, we restrict ourselves to simple, incremental loops. This not only simplifies the denotation of expressions but, also acts as a necessary aid to the cost

$$
\begin{aligned}
[\![\cdot]\!] &: Ty \to \star \\
[\![Bool]\!] &= \mathbb{B} \\
[\![Nat]\!] &= \mathbb{N} \\
[\![Arrow\ \vec{t}\ t]\!] &= [\![\vec{t_0}]\!] \times \ldots \times [\![\vec{t_n}]\!] \to [\![t]\!]
\end{aligned}
$$

$$
\begin{aligned}
evalFor\ s\ 0\ f &= s_0 \\
evalFor\ s\ (n+1)\ f &= f\ n\ (evalFor\ s\ n\ f)
\end{aligned}
$$

$$
\begin{aligned}
[\![\cdot]\!] &: Stmt_t \to [\![t]\!] \\
[\![x]\!] &= x \\
[\![v]\!] &= v \text{ where } v \in \mathbb{N} \wedge v \in \mathbb{B} \\
[\![e_1 + e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
[\![e_1 * e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
[\![f\ \vec{e}]\!] &= [\![f]\!]\ ([\![\vec{e_0}]\!], \ldots, [\![\vec{e_n}]\!]) \\
[\![\text{let } e\ \lambda x.s]\!] &= [\![s[v/x]]\!] \text{ where } v = [\![e]\!] \\
[\![\text{for } e_0\ e\ \lambda x\ n.s]\!] &= evalFor\ [\![e_0]\!]\ [\![e]\!]\ (\lambda y\ m.eval\ [\![s[x/y, n/m]]\!]) \\
[\![f\ \vec{x} = s]\!] &= \lambda \vec{x}.[\![s]\!]
\end{aligned}
$$

Figure 2. Denotational Semantics for $\mathcal{L}$.

analysis phase. Such loop-count proofs could be extended to structurally recursive functions, automated over e.g. size indexed containers [3]. Many other cases could potentially be automated with the help of a recurrence equation solver such as those found in COSTA [2] or PURRS [4], or by using advanced techniques that have been developed within the worst-case execution time (WCET) community [18, 26]. The denotational semantics of $\mathcal{L}$ is given in Figure 2. This provides a firm foundation for specifying program costs.

2.2 Determining costs for $\mathcal{L}$

Assuming that we are given some heuristic $h^\star$ for estimating the cost of an expression from its *shape* or factorization, a key question is how this can be extended to provide structural costs for the whole programs? In our system, we achieve this by *sampling* individual expression costs and by extending the type of a term with its "observed" cost. Intuitively the (worst-case) cost for a let-binding let $e\ \lambda x.s$ might be calculated as the sum of the cost of evaluating the bound expression, e, plus the cost of binding the result to x, plus the cost of evaluating the body, s, i.e.,

$$
\text{CLet}\ \frac{C = h^\star(\phi(e)) \quad C' = h^\star(\phi(\text{let})) \quad \sigma \vdash s : C_1}{\sigma \vdash \text{let } e\ \lambda x.s : C + C' + C_1}
$$

Unfortunately, a naïve construction of these rules quickly fails when attempting to cost even a simple program such as factorial, above. This program cannot be given a type due to the varying size of the input, which controls the number of times that the inner loop executes and therefore will govern the overall cost bound. This problem can be solved by introducing a *size index* similar to sized types [15] or the dependent type costs introduced by Brady and Hammond [8]. In such an approach, each term not only carries a type, but also carries a corresponding size, thereby giving a static account of the underlying value.

$$
\begin{aligned}
\text{size} &: Ty \to \star \\
\text{size}\ t &= \mathbb{S} \\
\text{size}\ (Arrow\ \vec{t}\ t) &= \vec{\mathbb{S}} \to \mathbb{S}
\end{aligned}
$$

Figure 3. Mapping of Types to Sizes

2.3 Size indices

The sizes of values are introduced by extending function definitions with a special binder that accepts a size index along with a value of this index, similar to the parametric binder used in System F [22]

$$
\text{Func} : Ty \to \star ::= f[\vec{i}]\vec{x} = Stmt
$$

Here, $f[\vec{i}]\vec{x}$ specifies size-application. Although the binder is a term, the introduced variable may only be placed within a type, making it trivially erasable. The syntax of an erased term, s, is given in Figure 1. Intuitively, values will have two "costs":

1. the cost, or size of a value that e.g. bounds the number of loop iterations; and
2. the cost of evaluating a statement/expression at runtime.

Since each term must keep track of its respective size as an index, the denotational semantics must be updated so that it is *size respecting*. The new denotation of each term's type must therefore also account for the size. Intuitively, the size of a ground-type, such as a natural number, is the static loop count; however this leaves the question of the size of an arrow (i.e. some function). Considering again the use as a loop count, if each input has an associated size, then each arrow type will map the size of its inputs to the size of its output. The mapping of types to sizes is given in Figure 3. Reconsidering the semantics of terms, in order to support looping, natural numbers must be size-indexed. They are

$$
\begin{aligned}
[\![\cdot]\!] &: (t : Ty) \to \text{size } t \to \star \\
[\![Bool]\!]\ i &= \mathbb{B} \\
[\![Nat]\!]\ i &= \mathbb{N}_i \\
[\![Arrow\ \vec{t}\ t]\!] f &= (\vec{i} : \vec{\mathbb{S}}) \to [\![\vec{t}_0]\!]\vec{i}_0 \times \ldots \times [\![\vec{t}_n]\!]\vec{i}_n \to [\![t]\!](f\ \vec{i})
\end{aligned}
$$

$$
\begin{aligned}
[\![\cdot]\!] &: Stmt\ t\ i \to [\![t]\!]i \\
[\![e_1 + e_2]\!] &= [\![e_1]\!] \hat{+} [\![e_2]\!] \\
[\![e_1 * e_2]\!] &= [\![e_1]\!] \hat{*} [\![e_2]\!] \\
[\![f[\vec{i}]\vec{e}]\!] &= [\![f]\!]\ ([\![\vec{e}_0]\!], \ldots, [\![\vec{e}_n]\!]) \\
[\![\text{let } e\ \lambda x.s]\!] &= [\![s[v/x]]\!] \text{ where } v = [\![e]\!] \\
[\![\text{for } iter\ e_{i_0}\ n_i\ s]\!] &= elim\ (\lambda j\ m.\mathbb{N}\ (forCost\ iter\ i_0\ j))\ e\ s\ i\ n \\
[\![f\ \vec{x} = s]\!] &= \lambda\vec{x}.[\![s]\!]
\end{aligned}
$$

Figure 4. Size-respecting semantics for $\mathcal{L}$.

$$
\text{CExp}\ \frac{C = h^\star(\phi(e))}{\sigma \vdash e : t\ i\ C}
$$

$$
\text{CLet}\ \frac{C = h^\star(\phi(e)) \quad \sigma \vdash s : t\ i\ C_1}{\sigma \vdash \text{let } e\ \lambda x.s : t\ i\ (C + C_1)}
$$

$$
\text{CCall}\ \frac{\begin{array}{c} f \mapsto s \in \sigma \quad f \notin \text{FV}(s) \\ C_1 = h^\star(\phi(f[\vec{i}]\vec{x})) \quad \vec{x} : \vec{t}\ \vec{i} \\ \sigma \vdash f : (Arrow\ \ \vec{t}\ t), fs, C_2 \end{array}}{\sigma \vdash f[\vec{i}]\vec{x} : t, (fs\ \vec{i}), (C_1 + C_2)}
$$

$$
\text{CFor}\ \frac{\begin{array}{c} C_1 = h^\star(\phi(e_1)) \quad C_2 = h^\star(\phi(e_2)) \\ C' = h^\star(\phi(\text{for})) \\ \sigma \vdash e_1 : t, k \quad \sigma \vdash e_2 : Nat, i \\ \sigma \vdash s : t, j, C \quad \sigma \vdash iter : \mathbb{S} \to \mathbb{S} \to \mathbb{S} \end{array}}{\sigma \vdash \text{for } e_1\ e_2\ s : t, forCost\ iter\ k\ i, (C_1 + C_2 + i(C + C'))}
$$

Figure 5. Probabilistic Cost for Statements in $\mathcal{L}$.

thus equipped with the obvious eliminator:

$$
\begin{aligned}
elim &: (P : \forall j.\mathbb{N}_j \to \star) \\
&\to P\ 0\ 0 \\
&\to (\forall j.m : \mathbb{N}_j \to P\ j\ m \to P\ (j+1)\ (m+1)) \\
&\to \forall i.n : \mathbb{N}_i \to P\ i\ n \\
elim\ P\ P_0\ h\ 0\ 0 &= P_0 \\
elim\ P\ P_0\ h\ (i+1)\ (n+1) &= h\ i\ n\ (elim\ P\ P_0\ h\ i\ n)
\end{aligned}
$$

As intended, this eliminator admits a direct definition for loops on size-indexed naturals. However, what is the resulting size of an iterated value? As noted above, arrows expect not only a value, but also a corresponding size. A loop body therefore uses a corresponding size iterator as follows:

$$
\begin{aligned}
forCost &: (\mathbb{S} \to \mathbb{S} \to \mathbb{S}) \to \mathbb{S} \to \mathbb{S} \to \mathbb{S} \\
forCost\ iter\ i_0\ 0 &= i_0 \\
forCost\ iter\ i_0\ (n+1) &= iter\ n\ (forCost\ f\ i_0\ n)
\end{aligned}
$$

An updated denotational semantics that respects the size indices is given in Figure 4.

Given that the size of a term can be arbitrarily complex, what is expected of the size of the the refined factorial program?

$$
\begin{aligned}
fact &: \forall i : \mathbb{S}.(n : Nat,\ i) \to Nat, factCost\ i \\
fact\ [i]\ n\ &= \text{for } 1\ n\ \lambda s.n * s
\end{aligned}
$$

Consider the fact that in the case of *Nat*, if the size gives a static description of e.g. a loop count, then the size must be a simple type-level *reification* of the variable's value. The size of the result of `fact` is then the factorial of the input size

$$
\begin{aligned}
&factCost : \mathbb{S} \to \mathbb{S} \\
&factCost\ 0 = 1 \\
&factCost\ (i+1) = (i+1) * factCost\ i
\end{aligned}
$$

This naïve equation for the size of the output cannot be easily type-checked within e.g. existing dependently-typed systems, such as Idris [6]. Fortunately, it can be directly expressed as the cost of a loop:

$$
factCost = forCost\ (\lambda i\ n.(i+1) * n)\ 1\ n
$$

As a final simplification, rather than directly updating the size in `fact`, we will require that all functions must be packed with their size update. Refining the definition of Func in this way yields:

$$
\begin{aligned}
&\text{Func} : (t : Ty) \to \text{size } t \to \star ::= \\
&\quad \Sigma(fs : \text{size } (Arrow\ \vec{t}\ t)).f[\vec{i}]\vec{x} = Stmt\ t\ (fs\ \vec{i})
\end{aligned}
$$

Supposing that the cost of the multiplication in the body of `fact`, $n*s$, is C, and that the overhead of performing the loop is C', then trivially the cost of the `fact` function can be given for any $n : Nat\ i$ as $\forall i : \mathbb{S}.i(C + C')$. Our revised typing rules are given in Figure 5. Note that while expressions have *two* types: a size in $\mathbb{S}$ and a base type in *Ty*, each statement has *three* types: a size in $\mathbb{S}$, a base type in *Ty*, and a resource cost in C. Since the cost of an expression can be directly calculated as a function of its shape, this does not need to be tracked by the type-system. The resource cost for a statement, however, allows us to provide a compositional account of its dynamic resource consumption, evaluated as some compiled machine

code. We will now give an example machine-level model, and derive appropriate cost equations from this.

2.4 Machine language

In order to simplify reasoning about execution costs, we now develop an example machine model, M. We will define this as a Static Single Assignment (SSA) language with infinite registers, and assign each instruction an associated probabilistic cost. Machine language syntax notably lacks ϕ nodes—instead a label L in parameters $\vec{x}$ having default values $\vec{v}$ is written as $L(\vec{x}) : \vec{v} : M$. When jumping to such a label, values $\vec{w}$ must be supplied for each parameter in $\vec{x}$ with the instruction goto $L\ \vec{w}$. When initially entering the label—by executing it in sequence—the initial values $\vec{v}$ are used for the arguments $\vec{x}$.

Program execution in M is modeled using a small-step operational semantics, $- \rightsquigarrow -$. We assume that the execution time of a compiled program executing on the underlying machine can be captured by a suitable probability distribution, D. In order to reflect this, the statement language is compiled to a simple call-by-value abstract machine whose syntax is given in Figure 6. The target machine language largely reflects the source statement language, consisting of values, builtins, function application, variable declaration and binding, instruction sequencing, and special instructions for performing timing.

The corresponding operational semantics is shown in Figure 7. As the machine reduces a program, an internal clock tracks the total number of elapsed ticks. Since the machine is probabilistic, reading the internal clock produces a random variable T, drawn from D. The current clock can be (re)started using starttime, and stopped and read using stoptime. The evaluation of machine terms proceeds using an environment σ that maps variable to values and function names to function bodies. The judgment $T, \sigma, m \rightsquigarrow T', \sigma', m'$ reads: in environment σ, the program m atomically evaluates to m' in $T - T'$ ticks, producing a new environment σ'. The relation $\cdot \rightsquigarrow^\star \cdot$ is the reflexive transitive closure of the $\cdot \rightsquigarrow \cdot$ relation.

By capturing and examining the execution times for a large number of example programs, it is possible to *learn* the corresponding probability distribution for the underlying machine. The premise of our technique is to: i) generate a representative set of programs in the style of e.g. `quickcheck` [10]; ii) instrument these programs; iii) evaluate them; and, finally, iv) calculate the parameters of the underlying probability distribution. While this makes it possible to provide a compositional model of an otherwise non-compositional resource, it is also inexact. We would like to recover the ability to reason about the error of a term. However, because the cost value itself is indeterminate, the error can also only reasonably be expected to be indeterminate. Fortunately, one of the advantages of using a generative model is that it can be related to the true underlying or canonical distribution.

$$
\begin{aligned}
M ::= & \text{builtin } \beta\ \vec{M}\ \lambda v.M \\
& \mid \text{call } f\ \vec{v}\ \lambda v.M \\
& \mid \text{let } M\ \lambda x.M \\
& \mid \text{if } v \text{ then } M \text{ else } M \\
& \mid \text{starttime } \lambda v.M \\
& \mid \text{stoptime } \lambda v.M \\
& \mid L(\vec{x} = \vec{v}) :\ M \\
& \mid \text{goto } L\ \vec{v} \\
& \mid \text{ret } v \\
& \mid \text{halt}
\end{aligned}
$$

Figure 6. Syntax of the Machine Language, M.

We would ideally like to know that given a sufficiently large body of samples we can expect our model to be "reasonably close" to the true underlying distribution of the program's execution time (or energy etc. usage). *Markov's inequality theorem* provides a means to formalize the notion that a distribution is "reasonably close" to some well known value

2.5 Reasoning with probabilities

The *Markov inequality theorem* provides a formal means of reasoning about the likelihood that a random variable takes on some specific value, by relating the variable to its expected value.

Theorem 2.1 (Markov Inequality)**.**

$$P(X \geq a) \leq \frac{\mathbb{E}(X)}{a}$$

Note that this places no stipulations on the specific value that we are interested in, a, or on the underlying distribution that is derived from $P(X)$. It therefore gives a highly general, but *loose*, bound. Moreover, although the theorem as stated above doesn't immediately provide a means of relating a random variable to its expected value, this can be resolved by using a suitably chosen value of a. Doing this yields the well-known *Hoeffding inequality theorem*

Theorem 2.2 (Hoeffding Inequality)**.**

$$P(|X - \mathbb{E}[X]| \geq \epsilon) \leq \exp\left(-\frac{2n^2\delta^2}{\sum_{i=1}^{n} X_i}\right)$$

Hoeffding's inequality theorem formalizes the likelihood that a random variable is within ϵ units of its expected value. Here, ϵ is called the *accuracy*, δ captures the the *confidence*, and n is the number of samples that have been seen. Using a well-known rearrangement of variables, the sample size, n, can be determined as a direct function of the accuracy and confidence values.

$$eval : Builtin\ (Arrow\ \vec{t}\ t) \to \vec{Value} \to [\![t]\!]$$
$$eval\ (\cdot + \cdot)\ v_1\ v_2 = v_1 + v_2$$
$$eval\ (\cdot * \cdot)\ v_1\ v_2 = v_1 * v_2$$

$$\text{EBuiltin}\ \frac{}{\exists K \sim D(\theta_b).\ T, \sigma, b\ \vec{v} \rightsquigarrow T + K, \sigma, eval\ b\ \vec{v}}$$

$$\text{ECall}\ \frac{f \mapsto \lambda\vec{x}.m \in \sigma}{\exists K \sim D(\theta_{\text{call}}).\ T, \sigma, \text{call}\ f\ \vec{v}\ k \rightsquigarrow T + K, \sigma, \text{let}\ m[\vec{x}/\vec{v}]\ \lambda x.k\ x}$$

$$\text{ELetRed}\ \frac{T, \sigma, m \rightsquigarrow T', \sigma', m'}{T, \sigma, \text{let}\ m\ \lambda x.k \rightsquigarrow T', \sigma', \text{let}\ m'\ \lambda x.k}$$

$$\text{ELet}\ \frac{}{T, \sigma, \text{let}\ v\ \lambda x.k \rightsquigarrow T, \sigma, k[x/v]}$$

$$\text{EConsequent}\ \frac{}{T, \sigma, \text{if}\ true\ \text{then}\ m_2\ \text{else}\ m_3 \rightsquigarrow T, \sigma, m_2}$$

$$\text{EAlternative}\ \frac{}{T, \sigma, \text{if}\ false\ \text{then}\ m_2\ \text{else}\ m_3 \rightsquigarrow T, \sigma, m_3}$$

$$\text{EStarttime}\ \frac{}{T, \sigma, \text{starttime}\ k \rightsquigarrow T, \sigma, k\ T}$$

$$\text{EStoptime}\ \frac{}{T, \sigma, \text{stoptime}\ k \rightsquigarrow T, \sigma, k\ T}$$

$$\text{ELabel}\ \frac{}{\exists K \sim D(\theta_{label}).\ T, \sigma, l(\vec{x} = \vec{v}) : m \rightsquigarrow T + K, l \mapsto \lambda\vec{x}.m; \sigma, m[\vec{x}/\vec{v}]}$$

$$\text{EGoto}\ \frac{l \mapsto \lambda\vec{x}.m \in \sigma}{T, \exists K \sim D(\theta_{\text{goto}}).\ \sigma, \text{goto}\ l\ \vec{v} \rightsquigarrow T + K, \sigma, m[\vec{x}/\vec{v}]}$$

$$\text{ERet}\ \frac{}{T, \exists K \sim D(\theta_{\text{ret}}).\ \sigma, \text{let ret}\ v\ \lambda x.m \rightsquigarrow T + K, \sigma, m[x/v]}$$

Figure 7. Machine language semantics

Lemma 2.3 (Hoeffding Confidence)**.**

$$n \geq -\frac{1}{2\epsilon^2} \log \frac{\delta}{2}$$

It follows from this lemma that the number of samples that is required to bound the error on the learned model is directly proportional to the accuracy and confidence that are needed.

2.6 Soundness

Now that we are equipped with a means of formally working with probabilistic errors, we would like to derive a general type soundness theorem to guarantee that the cost of an

$$\text{HLet}\ \frac{C_1 = h^\star(\phi(e)) \quad \sigma \vdash s : t, C_2, \epsilon, \rho}{\sigma \vdash \text{let}\ e\ \lambda x.s : t, C_1 + C_2, 2\epsilon, \rho^2}$$

$$\text{HCall}\ \frac{\begin{array}{c} f \mapsto s \in \sigma \quad f \notin \text{FV}(s) \\ C_1 = h^\star(\phi(f[\vec{i}]\vec{x})) \quad \vec{x} : \vec{t}, \vec{i} \\ \sigma \vdash f : Arrow\ \vec{t}\ t, C_2, \epsilon, \rho \end{array}}{\sigma \vdash f[\vec{i}]\vec{x} : t, (C_1 + C_2), 2\epsilon, \rho^2}$$

$$\text{HFor}\ \frac{\begin{array}{c} C_1 = h^\star(\phi(e)) \\ e_0 : Nat,\ i \quad \sigma \vdash s : C_2, \frac{\epsilon}{i}, \sqrt[i]{\rho} \end{array}}{\sigma \vdash \text{for}\ e_0\ e\ s : t, C_1 + iC_2, \epsilon, \rho}$$

Figure 8. Typing rules extended with error bounds of statement costs.

expression as predicted by our analysis is "reasonably close" to some expected value. Intuitively, the compiled machine code representation, m, of a well-typed statement $\sigma \vdash s : t, C$ should *generally* have an execution time that is close to C. Given that the underlying execution time follows a distribution D, we can interpret "generally" to mean the *expected* running time of m, $\mathbb{E}[T]$.

The type-system of costs can then be extended to track how accuracy and confidence are split between sub-computations of statements. For example, a let-binding must first evaluate the expression that is to be bound, and then continue by evaluating the body of the statement. Each step will have an expected cost with some accuracy and within some confidence interval. Assuming mutual independence, the likelihood of both these events occurring with some probability is the *product* of their respective, individual likelihoods. It follows that the accuracy and confidence values for the let-binding will be split between the bound expression and the statement body. In order to support this reasoning about split values, our existing type-system must be further augmented with support for tracking the flow of accuracy and confidence across statements. The updated typing rules are given in Figure 8. Here, a typed program statement $s : t, i, C, \epsilon, \rho$ has type t, size i, cost C, accuracy ϵ, and confidence ρ. Terms that are ascribed with such a type can be given a strong soundness guarantee about their (probabilistic) runtime resource consumption.

Theorem 2.4 (Type Soundness)**.**

$$\begin{array}{l} \forall(\sigma : Env)(s : Stmt\ t)(\epsilon, \rho : \mathbb{R}).\sigma \vdash s : t, i, C, \epsilon, \rho \wedge \\ \forall(P : M\ t \to \star)(k : Value \to M\ t)(\forall(v : Value).P\ (k\ v)) \Rightarrow \\ \exists(m : M\ t).compile\ s\ k = m \wedge \\ \quad (\exists(v : Value)\ T.0, \sigma, m \rightsquigarrow^\star T, \sigma', \text{ret}\ v) \wedge \\ \quad P(|C - \mathbb{E}[T]| \geq \epsilon) \leq \rho) \end{array} \tag{1}$$

The above asserts the fact that a program s which compiles to the machine program m—ascribed cost C, accuracy ϵ, and confidence ρ, by our system—is guaranteed to be within ϵ steps of the expected cost of the compiled program with confidence ρ.

3 Applying the heuristic: $h^\star$

In order to exemplify the use of a probabilistic cost function, it suffices to use a simple Gaussian function to describe the expected behavior of input expressions. To solve for such a function, we generate a variety of samples, observe their execution times, and calculate their mean and standard deviation. This technique appears to be readily extensible to solve for a multinomial Gaussian function, that is intended to represent the execution time of a specific expression in a given context. However, we leave this to future work. Program samples are generated in a fashion similar to the well-known *quickcheck* [10] Haskell framework. One major difference is that, in our approach types can have differing size bounds, whereas *quickcheck* ensures that the size is shared by all types. This simplifies selection from naturally bounded structures, such as contexts that are used to store variables (represented as size-indexed lists). The implementation itself is relatively straightforward: an expression is either: a variable; a constant; a function call; or a ground-normal term. The generator has access to a current context tracking all available variables. Variables are placed by randomly indexing into the current. To generate ground-normal forms only means a single new generator is required to produce builtin functions calls. Recall that because ground-normal forms are effectively trees of builtin functions over values, they are size-bounded by construction as for any other tree.

Only two generators are required to create statements: one to produce let-bindings; and another to produce loops. Because our implementation uses a deep-shallow embedding in Idris [24] with PHOAS [9] style functions, quickcheck provides a standard trick for generating functions; however we have opted for an explicit means of generating function definitions in the input language. To generate a function, first an arbitrary arity n is selected, then because functions are fully applied, a function of a single argument—a vector of size n—is created with a randomly generated body statement. When placing variables, the generator for the body has access to the existing context extended with the n new function inputs.

When calculating $h^\star$, the number of samples that have been observed must be tracked. This is because a proof of guaranteed accuracy and confidence must be provided by the system. Additionally, because the execution time of whole expression trees is being estimated, samples are counted *per shape*. For example, suppose that the runtime of the expression $4 + x * y$ is to be estimated within $0.1\mu s$ accuracy, with 95% confidence. From the Hoeffding bound, it follows that:

$$n \geq \frac{1}{2(0.1)^2} \log \frac{2}{1 - 0.95} \approx 80$$

$h^\star$ must therefore have been calculated from at least 81 samples of normalized expression shape $\{ADD, MUL\}$.

3.1 Example: proving factorial

Given a suitably calculated estimate $h^\star$, it is possible to describe the likely execution time of `fact` for an arbitrary input. Using the typing rules from Figure 8, it is easy to analyze the accuracy of the costs for the factorial of any n.

$$\dfrac{\dfrac{\dfrac{C = h^\star(\phi(n * s) = \{\text{MUL}\})}{\sigma \vdash n * s : C, \frac{1}{3}\epsilon, \sqrt[3]{\rho}} \quad C_0 = h^\star(\phi(\textbf{for}))}{\sigma \vdash \text{for } 1\ n\ (n * s) : i(C_0 + C), \epsilon, \rho}}{\sigma \vdash \begin{array}{l} factCost, fact\ [i]\ (n : Nat, i) = \text{for } 1\ n\ \lambda s.n * s \\ \quad : Arrow\ Nat\ Nat,\ \ factCost\ i, i(C_0 + C), \epsilon, \rho \end{array}}$$

The leaves of the typing derivation tell us what accuracy and confidence are required of $h^\star$.

4 Related Work

Although much previous work has studied how to combine the use of a probabilities with programming languages, relatively little work has been done on leveraging these ideas directly within types, as we have done in this paper. Moreover, the use of probabilistic estimates of program resource consumption has not hitherto been widely studied. The primary focus has been on the use of regression techniques [27] such as LASSO. While these can, in principle, provide better quality results [20], they also significantly complicate reasoning.

IBAL [21] is a probabilistic programming language that aims to provide a means of describing complex probability distributions as compositional components. It extends the ideas of probabilistic graphical models. IBAL allows the programmer to describe a probabilistic process and to perform inference on the model of that process. In contrast, our system exposes a traditional, deterministic programming language which gives constructive guarantees about probabilistic bounds of non-functional effects.

`Uncertain<T>`: A first-order type for uncertain data [5] is a framework that is used to develop generic types of uncertain values `Uncertain<T>`, drawn from values of the underlying type T. Unlike the new work that we describe, this work is primarily concerned with probabilistic programming and runtime values, rather than with statically reasoning about probabilistic properties of non-functional effects. In the `Uncertain<T>` approach, expressions are observed by constructing a Bayesian net, using value flow information. In contrast, our approach assumes that whole expressions can be given an atomic cost. Extending our model to support Bayesian reasoning in e.g. branches, as with `Uncertain<T>` is an interesting area of future work.

A Type Theory for Probabilistic and Bayesian Reasoning [1]. As with our own work, Adams and Jacobs develop a type-theory, COMET, for proving properties of probabilistic data. However, these are, once again, properties of probabilistic runtime values, rather than the non-functional effects that we have studied here. Their work is, moreover, largely directed towards describing probabilistic graphical models, capturing the flow of information through a Bayesian net. In contrast, our work is targeted at using types to capture the potential error of *unknown* probability distribution functions (PDFs).

Resource Bound Certification [11] This work was a strong inspiration for our own. Like our own work, Crary and Weirich develop a type-theory which is used to capture the cost of programs, including their execution times. Unlike our work, however, these costs are assumed to be deterministic. They therefore do not address issues of error as we have done. Crary and Weirich also take a similar approach to ours in handling functions, annotating arrow types with an explicit cost. They additionally allow the programmer to use complex data types, leveraging so called *synthetic dependent types*. We intend to investigate this connection further as part of our future work.

PSI Exact inference for probabilistic programs [12]. This work is closely related to that described in this paper, using the shape of a program to construct an algebraic expression of a probabilistic term. However, like other approaches, it is primarily concerned with reasoning about probabilistic runtime values rather than non-functional effects. Exploiting the PSI solver to reduce the compounding of the error that we observe, would, however, be an interesting area of future work.

Using dependent types to define energy augmented semantics of programs [25] develops a type-directed means of analyzing the energy consumption of programs by modeling power draw over time of the component states. The system tracks the constituent (hardware) component and program states over time, which can be mapped to a precise power draw. Additionally operations may not only produce a value, but change the current state of the system allowing for precise reasoning. Similar to our work, loops are explicitly bounded and the model is parametric to the underlying hardware; however the system presented requires a precise specification of each operation's energy consumption. Additionally there is no means of giving semi-automated estimates for operations consuming unknown amounts of energy.

5 Conclusion

This paper has described a how to automatically determine probabilistic bounds on program execution costs, in terms of time, energy etc., from a source level program, using a new type-based approach. We build on known results from machine learning and statistics, using probability distribution functions that have been obtained from repeated program executions to derive sound accuracy and confidence measures. Base costs and errors are synthesised constructively to yield cost, accuracy and confidence information for complete program statements. For this draft, we have illustrated the use of our approach using a simple factorial function, whose costs are defined in terms of a single iterative construct. In the final version of the paper, we will consider further, more complex, examples, and will also investigate more complex programming language constructs.

A number of avenues would repay further investigation.

Type-level energy and P-State tracking. Current versions of Linux provide an interface to Intel's power consumption subsystem through the running average power limit (RAPL) driver [23]. This exposes a means of evaluating a program's energy consumption, that could be used to build base energy usage probability distributions, as we have done for time. These distributions could be used in a similar way to the time distributions to provide systematic bounds, accuracy and confidence on energy usage at the source level.

Lazy Evaluation. We can, in principle, exploit similar approaches to those taken by Jost et al. [16, 17] for amortised analysis, to embed the costs of lazy evaluation within our types and to track these across composed expressions.

Higher-order functions. $\mathcal{L}$ comprises a simple, first-order language. A reasonable next step would be to include support for higher order functions. Following the work of Crary [11], Jost et al. [16] and Brady [8], igher-order functions could be supported by annotating arrows with not only their cost, but also with their expected bounds.

General Effect Handling. This paper considers only a pure functional language. Extending our work to consider the costs of side-effects, such as simple assignments, should, however, be a relatively small extension. Following the work of Hancock [14], Brady [7], and van Gastel et al [25], side-effects could, for example, be introduced as opaque system calls with axiomatized costs and bounds. Such an approach would, however, greatly increase the required number of training samples since the costs of each distinct call would need to be individually observed.

References

[1] Robin Adams and Bart Jacobs. 2015. A Type Theory for Probabilistic and Bayesian Reasoning. *CoRR* abs/1511.09230 (2015). http://arxiv.org/abs/1511.09230

[2] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2008. *COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode.* Springer Berlin Heidelberg, Berlin, Heidelberg, 113–132. https://doi.org/10.1007/978-3-540-92188-2_5

[3] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015). https://doi.org/10.1017/S095679681500009X

[4] R. Bagnara, A. Zaccagnini, and T. Zolo. 2003. *The Automatic Solution of Recurrence Relations. I. Linear Recurrences of Finite Order with Constant Coefficients.* Quaderno 334. Dipartimento di Matematica, Università di Parma, Italy. Available at http://www.cs.unipr.it/Publications/.

[5] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain: A First-order Type for Uncertain Data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14).* ACM, New York, NY, USA, 51–66. https://doi.org/10.1145/2541940.2541958

[6] Edwin Brady. 2005. *Practical implementation of a dependently typed functional programming language.* Ph.D. Dissertation. Durham University, UK. http://etheses.dur.ac.uk/2800

[7] Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13).* ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

[8] Edwin Brady and Kevin Hammond. 2006. *A Dependently Typed Framework for Static Analysis of Program Execution Costs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 74–90. https://doi.org/10.1007/11964681_5

[9] Adam Chlipala. 2008. Parametric Higher-order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08).* ACM, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00).* ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[11] Karl Crary and Stephnie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00).* ACM, New York, NY, USA, 184–198.

[12] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. *PSI: Exact Symbolic Inference for Probabilistic Programs.* Springer International Publishing, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[13] Kevin Hammond. 2000. Hume: a Concurrent Language with Bounded Time and Space Behaviour. In *Proc. 7th IEEE International Conference on Electronic Control Systems (ICECS 2K), Lebanon.* 407–411.

[14] Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic.* Springer-Verlag, London, UK, UK, 317–331. http://dl.acm.org/citation.cfm?id=647850.737221

[15] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96).* ACM, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

[16] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10).* ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/1706299.1706327

[17] Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *J. Autom. Reason.* 59, 1 (June 2017), 87–120. https://doi.org/10.1007/s10817-016-9398-9

[18] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. 2012. *Symbolic Loop Bound Computation for WCET Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg, 227–242. https://doi.org/10.1007/978-3-642-29709-0_20

[19] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. http://doi.acm.org/10.1145/1538788.1538814

[20] Andrew Y Ng and Michael I Jordan. 2002. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in neural information processing systems.* 841–848.

[21] Avi Pfeffer. 2001. IBAL: A Probabilistic Rational Programming Language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1 (IJCAI'01).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 733–740. http://dl.acm.org/citation.cfm?id=1642090.1642189

[22] John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation.* Springer-Verlag, London, UK, UK, 408–423. http://dl.acm.org/citation.cfm?id=647323.721503

[23] Zhang Rui. [n. d.]. introduce intel rapl driver. https://lkml.org/lkml/2011/5/26/93. ([n. d.]). Accessed: 2017-07-30.

[24] Josef Svenningsson and Emil Axelsson. 2013. *Combining Deep and Shallow Embedding for EDSL.* Springer Berlin Heidelberg, Berlin, Heidelberg, 21–36. https://doi.org/10.1007/978-3-642-40447-4_2

[25] Bernard van Gastel, Rody Kersten, and Marko van Eekelen. 2016. *Using Dependent Types to Define Energy Augmented Semantics of Programs.* Springer International Publishing, Cham, 20–39. https://doi.org/10.1007/978-3-319-46559-3_2

[26] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. https://doi.org/10.1145/1347375.1347389

[27] Xinnian Zheng, Pradeep Ravikumar, Lizy K John, and Andreas Gerstlauer. 2015. Learning-based analytical cross-platform performance prediction. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on.* IEEE, 52–59.

Measuring Energy Usage for Parallel Haskell Programs

Yasir Alguwaifli
School of Computer Science
University of St Andrews
St Andrews, UK
ya8@st-andrews.ac.uk

Kevin Hammond
School of Computer Science
University of St Andrews
St Andrews, UK
kevin@kevinhammond.net

Theodoros Dimopoulos
School of Computer Science
University of St Andrews
St Andrews, UK
td41@st-andrews.ac.uk

Christopher Schwaab
School of Computer Science
University of St Andrews
St Andrews, UK
cjs26@st-andrews.ac.uk

Abstract

In recent years, a number of functional programming languages have started to support non-traditional computing platforms such as System-on-Chip (SoC), where the hardware design puts emphasis on different factors from those of conventional computing platforms. One problem with the deployment of SoCs is the limitation of power sources used to power such systems in a typical environment. Additionally, executing parallel programs on these multicore, low-power systems can produce varying energy consumption scenarios depending on the context. In this paper, we measure and correlate the energy usage of several parallel Haskell programs against execution time and other runtime system (RTS) metrics, produced using GHC. We show how energy usage relates to the number of cores on two real, widely-used multicore architectures: ARM and Intel x86. We give performance results on two machines, one of which has 28 physical cores, and supports hyperthreading. From these results, we construct an energy model and relate this model to the parallel structure of the program source. Given a suitable structural model of parallelism, for example that proposed by Castro *et al* or Schwaab *et al*, this gives a way that can be used to predict the energy usage for a concrete Haskell program running a real multicore system. In the full paper, we will explore how such a model can be constructed, and give results to show that our model has good predictive capability for a variety of parallel Haskell programs and inputs.

Keywords Parallelism, Multicore, Energy Usage, Functional Programming, Haskell, Performance Modelling, Empirical Results

Towards Compiling SAC for the Xeon Phi Knights Corner and Knights Landing Architectures — Draft —

Clemens Grelck
University of Amsterdam
Science Park 904
1098XH Amsterdam
Netherlands
C.Grelck@uva.nl

Nikolaos Sarris
University of Amsterdam
and VU University Amsterdam
De Boelelaan 1105
1081HV Amsterdam, Netherlands
Nikolaos.Sarris@student.uva.nl

ABSTRACT

Xeon Phi is the common brand name of Intel's Many Integrated Core (MIC) architecture. The first publicly available generation, named Knights Corner, and the second generation, Knights Landing, form a middle ground between modestly parallel standard desktop and server architectures and the massively parallel GPGPU architectures.

In this paper we explore the various compilation options for the purely functional data-parallel array programming language SAC (Single Assignment C) for both Knights Corner and Knights Landing. Our particular interest and emphasis lies in doing so with limited or entirely without user knowledge and care about the compilation target architecture. We report on a series of experiments involving two classical benchmarks: matrix multiplication and convolution.

CCS Concepts

•**Computing methodologies** → **Parallel programming languages;**

Keywords

Array processing, Single Assignment C, multithreading, automatic parallelisation, Xeon Phi

1. INTRODUCTION

SAC (Single Assignment C) is a purely functional, data-parallel array language [11, 12, 9] with a C-like syntax (hence the name). SAC features homogeneous, multi-dimensional, immutable arrays and supports both shape- and rank-generic programming: SAC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions). A key motivation for functional array programming is fully compiler-directed parallelization for various architectures starting from exactly the same one source. Currently, the SAC compiler supports general-purpose multi-processor and multi-core systems [8], CUDA-enabled GPGPUs [13], heterogeneous combinations thereof [6], the Amsterdam MicroGrid general-purpose many-core processor [10] or, most recently, clusters of workstations [14].

Xeon Phi is the common brand name of Intel's Many Integrated Core (MIC) architecture. The MIC architecture essentially bridges the gap between standard desktop and server processor architectures, namely Intel's own Xeon product line, and the highly successful domain of general-purpose graphics processing units (GPGPUs) dominated by NVidia. Like GPGPUs the MIC architecture trades sequential performance in favour of the tight integration of a large number of compute cores and thus parallel execution as the default rather than as an add-on. Very much unlike GPGPUs, the Xeon Phi cores are nonetheless universal compute cores, capable of running an adapted, but otherwise fully-fledged Linux operating system. Consequently, we can make use of the standard parallel programming abstractions from shared memory models like OpenMP and Posix Threads to distributed memory models like MPI and indirectly any higher level parallel programming abstraction that is built on top of them. From a software and programming point of view this is arguably the most relevant advantage of the Xeon Phi over GPGPUs that still require a significantly more disruptive programming and performance engineering approach.

The first publicly available generation, coined *Knights Corner*, was released in November 2012 [4]. It has 60 cores running at 1053MHz, each capable of executing 4 threads (pseudo-)simultaneously. The peak performance is advertised as 1011GFLOPS. Like GPGPUs Knights Corner comes as an extension board for the PCI Express bus. Intel released the second and latest Xeon Phi generation, named *Knights Landing*, in June 2016 [1]. It has 64–72 cores running at 1300–1500MHz and like its predecessor it can run 4 threads simultaenously. Depending on the exact model Intel reports the peak performance as between 2662GFLOPS and 3456GFLOPS. Unlike Knights Corner, the current generation Knights Landing is a true host processor architecture and thus avoids the PCI Express bus altogether.

Overall Intel's Many Integrated Core architecture in general and the Xeon Phi product family in particular form

IFL '17 Bristol, United Kingdom

ISBN ??...$15.00

DOI: ??

an interesting middle ground between standard desktop and server processor architectures and GPGPUs. Intel even markets the Xeon Phi as their way forward towards exascale computing. This raises the question how high-level functional languages with a faible for parallel execution can make use of Knights Corner and Knights Landing and what performance they manage to achieve.

The contribution of this paper is to explore various compilation paths for SAC for both Xeon Phi generations and to report on preliminary experimental evaluation of the actual performance achieved.

The remainder of the paper is organised as follows. Section 2 provides some background information on our functional array language SAC while Section 3 introduced the Xeon Phi architecture, namely Knight's Corner and Knight's Landing in more detail. In Section 4 we sketch out the various approaches towards compiling SAC to the Xeon Phi that we investigated. Our preliminary experimental evaluation is discussed in Section 5. Finally, we sketch out related work in Section 6 before we draw conclusions in Section 7.

2. INTRODUCING SAC

As the name "Single Assignment C" suggests, SAC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate adoption in compute-intensive application domains, where imperative concepts prevail. Core SAC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions; details can be found in [11].

Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SAC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation.

On top of this language kernel SAC provides genuine support for processing truly multidimensional and truly stateless/functional arrays using a shape-generic style of programming. Any SAC expression evaluates to an array. Arrays may be passed between functions without restrictions. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, and arrays of any rank, e.g. `int[*]`. The latter include scalars, which we consider rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars. The hierarchy of array types induces a subtype relationship, and SAC supports function overloading with respect to subtyping.

SAC only features a very small set of built-in array operations, whereas all aggregate array operations are specified using WITH-loop array comprehensions:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}:  genarray( shape, default)
```

Here, the keyword `genarray` characterises the WITH-loop as an array comprehension that defines an array of shape *shape*. The default element value is *default*, but we may deviate from this default by defining one or more index partitions between the keywords `with` and `genarray`.

Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to loop variables in FOR-loops. Unlike FOR-loops, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression that is in the scope of *idxvec* and thus may access the current index location. As an example, consider the WITH-loop

```
A = with {
      ([1,1]<=iv<[4,5]): 10*iv[0]+iv[1];
      ([4,0]<=iv<[5,5]): 42;
    }: genarray( [5,5], 99);
```

that defines the 5×5 matrix

$$A = \begin{pmatrix} 99 & 99 & 99 & 99 & 99 \\ 99 & 11 & 12 & 13 & 14 \\ 99 & 21 & 22 & 23 & 24 \\ 99 & 31 & 32 & 33 & 34 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$$

WITH-loops in SAC are extremely versatile. In addition to the dense rectangular index partitions show above SAC supports also strided generators. In addition to the `genarray`-variant used here, SAC features further variants, among others for reduction operations. Furthermore, a single WITH-loop may define multiple arrays or combine multiple array comprehensions with further reduction operations, etc. For a complete, tutorial-style introduction to SAC as a programming language we refer the interested reader to [9].

3. XEON PHI ARCHITECTURE

The Knights Corner and the Knights Landing processor architectures are the beginning of a new product line at Intel, but in fact they continue a line of development that started much earlier. From the beginning the motivation was to counter the ever-growing success of GPGPUs, both with respect to performance and with respect to commercial success, while as far as possible retaining the x86 instruction set architecture.

With the Larrabe microarchitecture (2006–2010) Intel directly targeted highly parallel visual computing applications, but the attempt to directly compete with the then new and rising GPGPUs failed, and the project was terminated before the release of a retail product. In 2009 Intel prototyped the Single-chip Cloud Computer (SCC) [15]. The SCC features 48 P54C cores in 24 tiles of two cores each. The cores communicate via a 2-d mesh network, hence the analogy to cloud computing. One strength of the SCC was the ability to adjust the clock frequency per tile and the voltage per voltage island of 8 cores. Lack of cache coherence, however, made programming the SCC challenging and the lack of tooling made the SCC never rise beyond academic research.

In 2010 Intel announced the first prototype of the Many-core Integrated Architecture (MIC), named Knights Ferry, as a derivative of the aforementioned projects. Knights Ferry consisted of 32 cores, had 2G RAM and its single

board performance exceeded 750 GFlops.

The first architecture we make use of in this paper, Knights Corner, is next in line: Intel's first commercial many-core product with more than 50 cores per chip. Knights Corner comes in the form of a coprocessor that is connected with the Xeon-based host system via PCI Express. A virtual TCP/IP stack is implemented over the bus to provide communication between the host and the coprocessor as a network node. Accordingly, users are able to connect to the chip through a secure shell and start their applications on the Xeon Phi. Furthermore, users are able to build heterogenous applications, parts of which execute on the host and parts on the coprocessor.

The Knights Corner's 60 cores are connected with a bidirectional ring interconnect. Each core has 32KB L1 instruction cache, 32KB L1 data cache and 512KB unified L2 cache memory. All caches are kept coherent with a global-distributed tag directory (TD). Furthermore, the memory controllers and the PCIe client logic provide the interface to the GDDR5 on the coprocessor and the PCIe bus respectively.

In June 2016 at the International Supercomputing Conference of Frankfurt [9], Germany, Intel revealed the second generation of Xeon Phi with the code name Knights Landing,(KNL).

4. COMPILATION APPROACHES

still missing

5. EXPERIMENTAL EVALUATION

still missing

6. RELATED WORK

Among high-level, and in particular among functional programming languages, Intel's Many Integrated Core architecture and the Xeon Phi product family has by far not gained the same popularity as compilation target as GPGPUs. A notable exception is the imperative data parallel programming language Vector Pascal [5] extends ISO Pascal by data-parallel constructs, namely array index ranges. Recent versions of Glasgow Vector Pascal have adopted quite a few array programming concepts pioneered by SaC [3], and so SaC and Vector Pascal share many aspects in the overall approach to data parallel programming. Differences still are in the syntactic look-and-fell: C vs Pascal, but most notably in SaC being a purely functional language, whereas Vector Pascal follows its host language in the imperative paradigm. Among others this means that SaC offers completely automatic resource management for memory, cores, etc, whereas Vector Pascal leaves resource management mostly to the user/programmer.

The Glasgow Vector Pascal compiler was ported to the first generation Xeon Phi, Knights Corner, following a similar strategy as pursued by us [3]. The front end of the compiler is machine-independent, and, hence, the main challenges addressed lie in the backend, same as for SaC. A major difference between the Glasgow Vector Pascal compiler and the SaC compiler is that they compile all the way down to machine assembly, using abstract machine descriptions for automatic compiler backend generation. In contrast we compiler down to C as a general intermediate language and make use of Intel's Xeon Phi enabled C compiler `icc` for the final generation of executable code. The Vector Pascal approach requires explicit incorporation of the advanced vector processing capabilities of the Xeon Phi family into their compiler. In contrast, we can semi-automatically benefit from the auto-vectorisation features of Intel's C compiler as well as the human resources and in-depth hardware information available to Intel's compiler engineers that go way beyond the capabilities of any academic project.

Leaving the realm of high-level programming languages, [7] investigated the Knights Corner architecture from a performance engineering perspective. Based on an experimental investigation much more thorough than our's, they confirmed that it is possible to achieve very good runtime performance, but that at the same time it is way more tricky from a programming perspective to actually do so than the principle approach of using well established parallel programming abstractions suggests. They concluded that the number of applications that might make highly efficient use of Knights Corner may be limited in practice.

7. CONCLUSIONS

still missing

8. ACKNOWLEDGMENTS

Our sincere thanks go to the Advanced School of Computing and Imaging (ASCI) for providing us with access to a Xeon Phi Knights Corner system as part of their DAS-4 distributed research cluster [2]. Moreover, we thank SURFsara, the Netherlands national supercomputing center, for granting us access to one of their Xeon Phi Knights Landings systems.

9. REFERENCES

[1] S. Anthony. Intel unveils 72-core x86 knights landing cpu for exascale supercomputing. Technical report, ExtremeTech, 2013. http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing.

[2] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijsshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[3] M. Chimeh, P. Cockshott, S. Oehler, A. Tousimojarad, and T. Xu. Compiling Vector Pascal to the XeonPhi. *Concurrency and Computation: Practice and Experience*, 27:5060–5075, 2015.

[4] G. Chrysos. Intel xeon phi coprocessor (codename knights corner). In *24th Hot Chips Symposium (HC'12), Stanford, USA*, pages 1–31, 2012.

[5] P. Cockshott. Vector pascal reference manual. *SIGPLAN Notices*, 37(6):59–81, 2002.

[6] M. Diogo and C. Grelck. Towards heterogeneous computing without heterogeneous programming. In K. Hammond and H. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2013.

[7] J. Fang, H. Sips, L. Zhang, C. XU, Y. Che, and A. Varbanescu. Test-Driving Intel Xeon Phi. In *5th*

ACM/SPEC International Conference on Performance Engineering (ICPE'14), Dublin, Ireland, pages 137–148. ACM, 2014.

[8] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[9] C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.

[10] C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and A. Shafarenko. Compiling the Functional Data-Parallel Language SAC for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zürich, Switzerland*, 2009.

[11] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[12] C. Grelck and S.-B. Scholz. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In *2nd Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France*, pages 25–33. ACM Press, 2007.

[13] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.

[14] T. Macht and C. Grelck. SAC goes cluster: From functional array programming to distributed memory array processing. In J. Knoop, editor, *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2015), Pörtschach am Wörthersee, Austria*. Technical University of Vienna, 2015.

[15] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmerâĂŹs view. In *Conference on High Performance Computing Networking, Storage and Analysis (SC'10), New Orleans, USA 2010*. IEEE, 2010.

Transforming Programs into Application Specific Processors

Arjan Boeijink Hendrik Folmer Jan Kuper Marco J.G. Bekooij
University of Twente
{w.a.boeijink,h.h.folmer}@utwente.nl

Abstract

The combination of performance and energy efficiency drives hardware designers to create specialized components for specific applications. Current hardware technology has the space for complex algorithms to be mapped to hardware, however this complexity makes a manual hardware design process no longer tenable. This paper shows that implementing a program as a specialised hardware design can be a systematic process using classic compiler techniques, and yields a processors-like architecture. From programs written in an embedded Haskell DSL with explicit clock cycle boundaries we automatically generate a synthesizable hardware descriptions for application specific processors.

1. Introduction

The past decades we got hardware performance gains first from higher clock frequencies and later from increased parallelism. The energy costs (from data centers to mobile devices) are now the main bottleneck in computing, thus raising clock frequencies is no longer an option. And exploiting more parallelism is getting harder with diminishing returns. This leaves specialisation of hardware designs as the key method to gain efficiency and performance.

The amount of logic gates available on a chip is still increasing, allowing complex algorithms to be directly implemented in hardware. However the costs of producing a high performance chips with the leading edge technology has grown to the point where only few companies have the scale to design new ones. And GPUs, while offering much parallel computation throughput, are not always suitable because of their energy consumption, and have latencies too big for real time applications.

The availability of large reconfigurable hardware chips, known as field-programmable gate arrays (FPGA), provides the opportunity to apply hardware design to many specialised applications. Users of FPGAs range from datacenters requiring energy efficiency and fast response times, to high performance sensor data processing on a flying drone. The complexity of algorithms to be mapped onto FPGAs and the fast changing requirements of specialised applications, makes the manual hardware design process the bottleneck in time and costs.

As functional programmers we were frustrated by the low level nature of common hardware design tools and languages, and how repetitive most of the hardware design process is. Thus we are looking for higher level (and functional) abstractions for describing hardware, and to explore and bridge the gap between software and hardware.

In the first part of this paper we show a systematic transformation process of a functional program into a simple processor using well known techniques and some hardware oriented optimisations. Next we apply this concept to the generation of an application specific processor to a practical useful setting. An embedded Haskell DSL with explicit clock cycle boundaries is introduced, which gives the hardware designer a lot of control over the generated hardware implementation. With a tool to generate CλaSH code from this DSL, we keep the whole hardware design process in Haskell with executable/simulatable code at every stage. As a demonstration application we use the hardware implementation on an FPGA of the signal processing for a laser range finder.

1.1 Background: a Haskell based hardware design process

Functional languages have a long history with hardware design [11]. [2] propose the following hardware design process. To start, one must choose an algorithm to solve a certain problem. While choosing an algorithm it is important to keep in mind how well the target hardware could potentially solve the problem, where FPGAs are suitable for reusing computational structures and consistent data streams. Next, specify the algorithm in executable Haskell code. After which the code can be rewritten in executable CλaSH-code. The functionality can be evaluated/simulated in the Haskell/CλaSH. The CλaSH compiler can automatically generate a hardware architecture which can be synthesized to determine the hardware resource costs. The synthesized architecture consumes a certain amount of hardware resources but the target device has a limited amount of resources available. These resource constrains are in terms of memory, memory bandwidth, logical elements, and digital signal processors (DSPs). There are also constrains on latency and throughput. There are several methods to create a feasible design for the given resource constrains. One could split the algorithm in steps to reduce data access to reduce the memory bandwidth. Finding patterns in the algorithm allows for the reuse of hardware components over time. This reduces both memory and computational area on an FPGA. Breaking long latency computation in parts enables faster clock frequencies which influences latency and throughput. Pipelining and multistep computation requires glue logic/control to operate. During the time-area trade-off one can iterate through all the above mentioned methods.

The nice aspect of this process is that every step the result can be evaluated/simulated because it is executable Haskell/CλaSH code. In the time-area trade-off the impact of every design choice can be evaluated. The downside of this method, especially when iteratively designing a time-area trade-off, is that it comes down to the designers knowledge which technique is necessary to make a feasible design.

Creating glue logic and control to operate the algorithm is error prone and requires the majority of the effort. The automation of this part of the process is the main goal of this paper.

1.2 Hardware in Haskell using CλaSH

Designing hardware architectures for FPGA is traditionally done using low-level hardware description languages (HDLs) like Verilog or VHDL which is cumbersome and prone to error. CλaSH CλaSH [4] is a subset of Haskell that can be compiled to widely supported lower level HDLs. Even though you can use most abstractions that Haskell has to offer, CλaSH is still a structural hardware description language. Being a structural hardware description language CλaSH does not support dynamic data sizes or recursion, so in order to convert Haskell to CλaSH one has to use fixed size vectors instead of lists, specify the size of value data, and use higher-order functions or mealy machine structures instead of recursion.

2. Rewriting a program into a processor

We will first look at the process of turning a program into feasible hardware on a conceptual level. The choices we make depend on basic knowledge about what is not possible and what is efficient in hardware. Each step in the process addresses a hardware problem using familiar transformations from functional languages and compilers. The firsts half of the transformations steps are similar to the ones removing recursion in [13] and [12].

2.1 Example program: binary greatest common divisor

As a running example in this section we use a binary variant of greatest common divisor, that doesn't use any complex arithmetic operation. The use of multiple recursive functions and a data dependent recursion depth makes it non trivial to implement this computation in hardware.

```
binGCD :: Word32 -> Word32 -> Word32
binGCD x 0 = x
binGCD x y = let
    a = dropTrailingZeros x
    b = dropTrailingZeros y
    (s,g) = (min a b, max a b)
  in binGCD s (g - s) <<< countTrZeros (x .|. y)

dropTrailingZeros :: Word32 -> Word32
dropTrailingZeros i = i >>> countTrZeros i

countTrZeros :: Word32 -> Word32
countTrZeros n = if odd n then 0
  else countTrZeros (n >>> 1) + 1
```

The count trailing zeros could been implemented as a primitive operation as a fold over bits.

2.2 Desugaring and flattening the program

The first step is splitting up expressions which are too complex to execute within a single clock cycle. This can be done by introducing let expression like in the conversion to administrative normal form [6]. The question is, which expressions to define as trivial, as for example some arithmetic operations mights be cheap enough to keep nested. For this example we will split up all nested expression, in order to produce a very simple processor.

```
binGCD x y =
  if (y == 0) then x
  else
    let a = dropTrailingZeros x in
    let b = dropTrailingZeros y in
    let g = max a b in
    let s = min a b in
    let d = g - s in
    let r = binGCD s d in
    let o = x .|. y in
    let e = countTrZeros o in
    r <<< e
```

2.3 Explicit sequential execution

The above sequence of let expressions suggests an execution order but does not enforce it. We will make control flow explicit using continuation passing style [1], which is a standard method in compilers for functional languages. With an already flattened program we only have to introduce a continuation for the body of each let expression:

```
binGCD x y k = if (y == 0)
  then cont x k
  else
    dropTrailingZeros x
      (\a -> dropTrailingZeros y
        (\b -> cont (max a b)
          (\g -> cont (min a b)
            (\s -> cont (g - s)
              (\d -> binGCD s d
                (\r -> cont (x .|. y)
                  (\o -> countTrZeros o
                    (\e -> cont (r <<< e) k)
    )))))))
```

Here we use the *cont* function (reversed application) to make continuation application explicit.

2.4 Defunctionalising the continuations

In hardware we can not have higher order functions, thus we have to eliminate the continuations. We can use well known technique of defunctionalisation [9] to transform functions into datatypes.

Explanation to be added here

```
data Cont
  = CA Word32 Word32                          Cont
  | CB Word32 Word32 Word32                   Cont
  | CC Word32 Word32 Word32 Word32            Cont
  | CD Word32 Word32 Word32 Word32 Word32 Cont

binGCD x y          k = if (y == 0)
                    then cont x          k
                    else dropTrZeros x  (CA x y       k)
cont a (CA x y      k) = dropTrZeros y  (CB x y a     k)
cont b (CB x y a    k) = cont (max a b) (CC x y a b   k)
cont g (CC x y a b  k) = cont (min a b) (CD x y a b g k)
```

The steps of continuation passing style and defunctionalisation could be combined into one as shown in 'Calculating Correct Compilers'[5].

2.5 Flattening continuations with a stack

In hardware all data need to be of fixed size, thus we can not have arbitrarily nested continuations. Looking at how the continuations are used, we can see that only one layer of continuation is added/removed at a time. The nested structure behaves like a stack, thus we flatten the continuations in a nonrecursive context datatype and use a list to represent the stack.

```
type Stack = [Context]
data Context
```

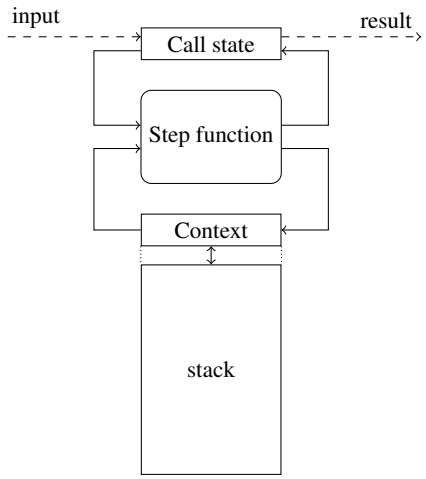


Figure 1: Hardware structure of the stack machine

```
  = CA Word32 Word32
  | CB Word32 Word32 Word32
  | CC Word32 Word32 Word32 Word32
  | CD Word32 Word32 Word32 Word32 Word32
  ...

binGCD x y          cs  = if (y == 0)
                          then cont x          cs
                          else dropZeros x     (CA x y      :cs)
cont a (CA x y    :cs) = dropZeros y     (CB x y a    :cs)
cont b (CB x y a  :cs) = cont (max a b) (CC x y a b  :cs)
cont g (CC x y a b:cs) = cont (min a b) (CD x y a b g:cs)
```

2.6 From mutual recursion to a state machine

The next problem is that we still have (mutual) recursion that is not realisable in hardware. We can combine functions into a single step function by introducing a data type, including the continuation application function.

```
data Call = GCD Word32 Word32 | DropZs Word32
          | CntZs Word32 | Cont Word32

step :: Call -> Stack -> (Call, Stack)
step (GCD x y)          cs = if y == 0
  then (Cont x       , cs                )
  else (DropZs x     , CA x y       : cs)
step (Cont a)          (CA x y       : cs) =
     (DropZs y       , CB x y a     : cs)
step (Cont b)          (CB x y a     : cs) =
     (Cont (max a b), CC x y a b   : cs)
step (Cont g)          (CC x y a b   : cs) =
     (Cont (min a b), CD x y a b g : cs)
```

Figure 1 shows the stack machine structure as hardware. The transformation steps taking until now are similar to the ones for translating recursive functions to hardware in [12] and [13].

2.7 Separating the data stack

Most steps have a lot of data copied from one continuation to another, and stack is very wide because some continuations capture many free variables. This yield inefficient hardware, especially when only a few variables are used every cycle.

To explain the transformation here

```
data State = BinGCD | DropZs | CntZs | Cont
data Context = CA | CB | CC | CD | ...
type CtrlStack  = [Context]
type DataStack = [Word32]

step :: State -> CtrlStack -> DataStack ->
  (State, CtrlStack, DataStack)
step BinGCD     cs       (y:x:ds) = if y == 0
  then (Cont  , cs    ,     x:ds)
  else (DropZs, CA:cs , x:y:x:ds)
step Cont (CA:cs)     (a:y:x:ds) =
  (DropZs, CB:cs,     y:a:y:x:ds)
step Cont (CB:cs)   (b:a:y:x:ds) =
  (Cont,   CC:cs,   g:b:a:y:x:ds) where g = max a b
step Cont (CC:cs) (g:b:a:y:x:ds) =
  (Cont,   CD:cs,  s:g:b:a:y:x:ds) where s = min a b
```

Note that we have to copy the argument to the dropZeros function because they are used again and/or in the wrong order. And for the arguments to the recursive binGCD call we can skip copying as they are in the right order and not used later.

Having two stacks might look like a unnecessary complication from the C-like software stack point of view, but is common in hardware designs based on Forth [8].

2.8 Optimizing control

What happens in every step depends on both the state and top of the continuation stack. We can combine the control state and continuation stack into a control stack by eliminating the Return state. Then only the top of the control stack directly determines what each step does. The steps containing if expressions are more complex than the rest, and can be split up by computing the branch condition in a separate step.

```
data Label = BinGCD | T1 | E1 | CA | CB | CC
   | CDE | DropZs | ... deriving Enum
type ControlStack  = [Label]
type DataStack = [Word32]
type CtrlFun = Label -> CtrlStack ->
  (Label, CtrlStack)

step :: Label -> DataStack -> (CtrlFun, DataStack)
step BinGCD       (y:x:ds) =
  (branch E1 z,     y:x:ds) where z = y == 0
step T1           (y:x:ds) =
  (ret        ,       x:ds)
step E1           (y:x:ds) =
  (call DropZs,   x:y:x:ds)
step CA         (a:y:x:ds) =
  (call DropZs, y:a:y:x:ds)
step CB       (b:a:y:x:ds) =
  (next     , g:b:a:y:x:ds) where g = max a b
step CC     (g:b:a:y:x:ds) =
  (next   , s:g:b:a:y:x:ds) where s = min a b
```

Now we can recognise the top of the control stack as a program counter. Thus a program counter is merely a numeric encoded label, optimized for the common case of continuing at the successive label.

The resulting dual stack machine looks like something you could derive directly by looking at the flattened program from tradition imperative language point of view.

2.9 Splitting into components

To be able to use efficient existing memory components we have to extract the use of the datastack from the step function. This

means splitting each step in parts to separate memory read and write actions. Also splitting off the arithmetic operations makes it easier to reuse hardware for similar (arithmetic) computations.

```
type CtrlFun = Word32 -> Label -> CtrlStack ->
   (Label, CtrlStack)
type Input = DataStack -> Word32
type AluOp = Word32 -> Word32 -> Word32
type StackMod = Word32 -> DataStack -> DataStack
```

```
step :: Label -> (CtrlFun,Input,Input,AluOp,StackMod)
step GCD = (branch E1   ,peek 0,lit 0 , isEq  ,keep)
step T1  = (ret         ,peek 1,lit 0 , pass  ,popNPush 2)
step E1  = (call DropZs,peek 1,lit 0 , pass  ,push)
step CA  = (call DropZs,peek 1,lit 0 , pass  ,push)
step CB  = (next        ,peek 1,peek 0, max   ,push)
step CC  = (next        ,peek 2,peek 1, min   ,push)
```

TODO check if such a description would be synthesizable in CλaSH

2.10 Control by microcode

Each step is now defined as a set of functions to be executed, however functions can not directly exist in hardware. Thus we apply defunctionalisation to each aspect, yielding components controlled by datatypes. The resulting structure looks like a processor controlled by horizontal microcode.

```
data AluOp = Const | Add | Sub | Or | Min | Max |
   ShR | ShL | IsEq | IsOdd
data Input = S Int | I Word32
data StAction = Keep | Push | PopNPush Int
data Ctrl = Call Label |Return |Branch Label |Next
```

```
microcode :: Label -> (Ctrl,Input,Input,AluOp,StAction)
microcode pc = case pc of
  BinGCD -> (Branch E1  , S 0, I 0, IsEq  , Keep)
  T1     -> (Return     , S 1, I 0, Pass  , PopNPush 2)
  E1     -> (Call DropZs, S 1, I 0, Pass  , Push)
  CA     -> (Call DropZs, S 1, I 0, Pass  , Push)
  CB     -> (Next       , S 1, S 0, Max   , Push)
  CC     -> (Next       , S 2, S 1, Min   , Push)
  CDE    -> (Call BinGCD, S 1, S 0, Sub   , Push)
  CFG    -> (Call CntZs , S 5, S 4, Or    , Push)
  CH     -> (Return     , S 1, S 0, ShL   , PopNPush 7)
  DropZs -> (Call CntZs , S 0, I 0, Pass  , Push)
  CI     -> (Return     , S 1, S 0, ShR   , PopNPush 2)
  CntZs  -> (Branch E2  , S 0, I 0, IsOdd , Keep)
  T2     -> (Return     , I 0, I 0, Pass  , PopNPush 1)
  E2     -> (Call CntZs , S 0, I 1, ShR   , Push)
  CK     -> (Return     , S 0, I 1, Add   , PopNPush 2)
```

```
alu :: AluOp -> Word32 -> Word32 -> Word32
alu Pass  x _ = x
alu Add   x y = x + y
alu Sub   x y = x - y
alu Or    x y = x .|. y
alu Min   x y = min x y
alu Max   x y = max x y
alu ShR   x y = x >>> y
alu ShL   x y = x <<< y
alu IsEq  x y = if x == y then 1 else 0
alu IsOdd x _ = if odd x then 1 else 0
```

```
selInput :: DataStack -> Input -> Word32
selInput ds (S i) = ds !! i
selInput _  (I x) = x
```

```
stackMod :: StAction -> Word32 ->
  DataStack -> DataStack
stackMod Keep          = keep
stackMod Push          = push
stackMod (PushAfterPop n) = pushAfterPop n
```

```
ctrl :: Ctrl -> Word32 -> Label -> [Label] ->
  (Label, [Label])
ctrl Next      = next
ctrl Return    = ret
ctrl (Branch e) = branch e
ctrl (Call f  ) = call f
```

2.11 Synthesisable implementation in CλaSH

The main effort of converting into a synthesisable hardware description is in replacing the lists for the control and data stack with memory components and stack pointers. Both stack pointers and the program counter need to be stored in a register. The $microcode$ and alu code from previous step can be directly reused with any change. All components are then connected together in the toplevel of the processor description. This is straightforward except for having to work with Applicative glue logic for Signal's instead of pure values

```
proccesor :: Signal () -> Signal (Label, Word32)
proccesor _ = bundle (pc, z) where
  pc = register def pc'
  (ctrlOp,ia,ib,oper,stOp) = liftB microcode pc

  nPC = liftA succ pc
  (cSP', savePC, pc') =
    liftB5 ctrl ctrlOp z nPC cSP retPC
  cSP = register 0 cSP'
  retPC = asyncRam d64 cSP savePC

  rdA = liftA2 agu dSP ia
  rdB = liftA2 agu dSP ib
  a = asyncRam d128 rdA wrData
  b = asyncRam d128 rdB wrData

  x = liftA2 inputMux ia a
  y = liftA2 inputMux ib b
  z = liftA3 alu oper x y

  dSP = register 0 dSP'
  (dSP', wrData) = liftB3 stackMod stOp dSP z

agu :: Word8 -> Input -> Word8
agu stackSp (S i) = stackSp - i
agu stackSp (I _) = stackSp

ctrl :: Ctrl -> Word32 -> Label -> Word8 -> Label
  -> (Word8, Maybe (Word8, Label), Label)
ctrl Next      _ nPC cSP retPC =
  (cSP  , Nothing          , nPC)
ctrl Return    _ nPC cSP retPC =
  (cSP-1, Nothing          , retPC)
ctrl (Branch e) 0 nPC cSP retPC =
 (cSP  , Nothing          , e)
ctrl (Branch e) _ nPC cSP retPC =
  (cSP  , Nothing          , nPC)
ctrl (Call f)   _ nPC cSP retPC =
  (cSP+1, Just (cSP+1, nPC), f)

stackMod :: StAction -> Word8 -> Word32 ->
```

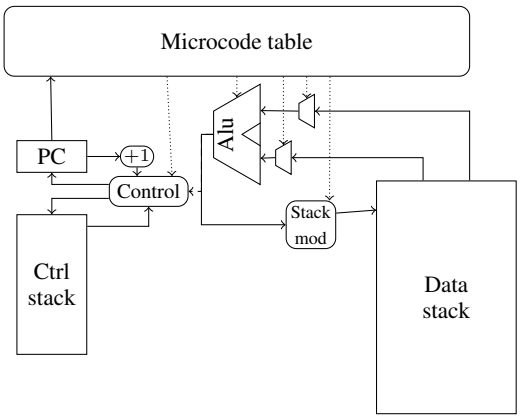


Figure 2: Architecture of derived processor

```
  (Word8, Maybe (Word8, Word32))
stackMod Keep           dSP z =
  (dSP , Nothing)
stackMod Push           dSP z =
  (dSP', Just (dSP', z)) where dSP' = dSP+1
stackMod (PopNPush n) dSP z =
  (dSP', Just (dSP', z)) where dSP' = dSP-n+1
```

Figure 2 shows a schematic of the derived processor architecture

2.12 Differences with general purpose processors

The downside of using horizontal microcode is that it costs a lot of bits to encode each step in the program
also writing microcode by hand is a lot of work
thus almost all processors define an instruction set that provides a slightly higher way of writing programs
having instructions in a writable memory makes a processor reusable for other applications
from the processor point of view an instruction set is just a compressed form of microcode with a hardware component for decoding (decompressing)
secondly an instruction set gives the opportunity of improving the next generation of a processor design without having to change all software

```
instrMem :: [(Label, Instr)]
decode :: Instr->(Input,Input,Oper,StAction,Ctrl)

selInput :: DataStack -> Input -> Word32
alu :: Oper -> Word32 -> Word32 -> Word32
stackMod :: StAction->Word32->DataStack->DataStack
ctrl :: Ctrl -> Word32 -> CtrlStack -> CtrlStack

sim :: DataStack -> [Label] -> Word32
sim ds []       = top ds
sim ds (pc:cs) = sim ds' cs' where
  Just is = P.lookup pc instrMem
  (ia, ib, op, g, f) = decode is
  x = selInput ds ia
  y = selInput ds ib
  z = alu op x y
  ds' = stackMod g z ds
  cs' = ctrl f z (pc : cs)
```

3. A DSL for explicitly clocked programs

When designing application specific hardware components we care a lot about performance and/or efficiency, otherwise we could have used some existing processor. The gains come from doing some specialised computation in less steps (clock cycles) and/or doing more operations in parallel than a normal processor could. While the transformation process described in previous produces a working processor, it is fairly minimalistic and not that efficient. The decision of how much computation to do in each step is a complex optimisation problem that depends on area and clock frequency requirements. Thus we leave it up to the hardware designer to explicitly decide how to group computations into clock cycles. Also the timing with respect to other components can be important, so the order of execution of each cycle needs to be explicit.

3.1 The sequential logic monad

```
binGCD :: Word32 -> Word32 ->
  SeqLogic s Word32 Word32 Word32
binGCD x y = do
  let isZero = y == 0
  if isZero then do
    clock
    return x
  else do
    clock
    a <- call $ dropZeros x
    b <- call $ dropZeros y
    let g = max a b
    clock
    let s = min a b
    clock
    let d = g - s
    r <- call $ binGCD s d
    let o = x .|. y
    e <- call $ countZeros o
return (r <<< e)
```

To be explained

3.2 Expressing control flow

Using functions is essential for code reuse and readability of larger programs, thus multi cycle SeqLogic fragments should be usable as functions. As seen from the transformation process function calls and return form a natural boundary between steps. However, function calls are invisible for the SeqLogic monad and will not get cycle boundaries in simulation. To match the hardware behaviour with simulation we have to wrap every call to a SeqLogic function. However for some smaller functions the added cycle boundaries cause too much overhead. Thus we use the inline wrapper (that does nothing) to make explicit that a function is used without any overhead.

TODO conditional expression/statements and external communication

3.3 Mutable data and explicit memory access

- explicit allocation of writable memory addresses
- indexing in larger memory structures
- using arrays
- loop constructs

3.4 Existing components as coprocessors

Complex arithmetic operations such as divisions, square root and trigonometric functions are usually implemented as separate com-

ponents that take many cycles to execute. A coprocessor can be described as just function in the same EDSL. The coprocessor executes concurrently after forking it from main program, and you can wait for its finished results by joining it. Another reason to use the coprocessor abstraction is making multiple functions execute concurrently, instead of manual interleaving each cycle of those functions.

TODO communication with coprocessor and example

3.5 Vectorisation support

One of the most important ways to achieve higher performance in specialized hardware is to compute a lot things in parallel. However the amount of available parallelism is often larger than what fits on the limited area of the hardware. Thus we have split operations on large vectors into segments which are executed over time.

Work in progress

3.6 Software pipelining

Dependencies between operations can lead to poor utilisation of the available hardware resources. If long dependencies chains exist within a loop, the performance might be improved by (partial) overlapping of the execution of multiple loop iterations. This technique is known as software pipelining and is commonly used in compilers for VLIW processors.

Work in progress

3.7 Cycle accurate simulation

programs are usually part of a larger system
for cosimulation with other component written in CλaSH we support running a $Seqlogic$ program as a signal function
this allows whole system simulation and analysis in early stages of the design process
also finding the best way to split a program into cycles is easier as an incremental process, where at every step the simulation can be run and measured

```
interpretSeqLogic :: (forall s. SeqLogic s i o ())
  -> Signal (Maybe i) -> Signal (Maybe o)
```

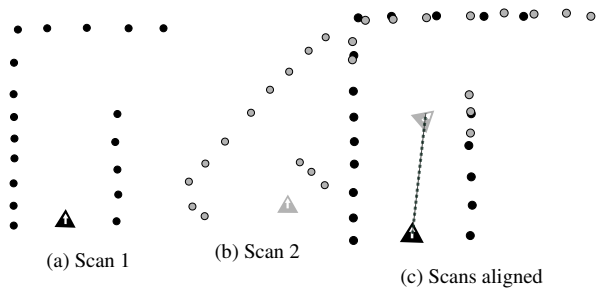
this cycle by cycle simulation is implemented using the operational monad technique, where $clock$ statements suspend the computation
memory is implemented using the lazy ST monad
for coprocessors a list of active ones is maintained and each one is run at every $clock$ or blocking statement in the main program

4. Example application: iterative closest point

To demonstrate the idea's presented in this paper we take the Iterative Closest Point (ICP) as an example case. ICP[10] is an algorithm that minimizes the distance between two point sets. In robotics the algorithm is often used to determine the robot movement based on sensor data. In our case the sensor information comes from an laser-range finder (LRF) which measures the time of flight of a laser beam, and via a scan of the environment retrieves geometrical information. Fig. 3 shows an example of this where robot takes a 2D scan of the environment (3a), moves an unknown distance, and then takes a second scan of the environment (3b), by aligning the two scans the moved distance is calculated.

The basic ICP algorithm consist of the three following steps:

1. Construct correspondences between point set $\vec{p}$ and $\vec{q}$
2. Compute a transformation which minimizes the distances (errors) between the two correspondences



(a) Scan 1 (b) Scan 2 (c) Scans aligned

Figure 3: Proximity scan used to determine robot movement (transformation)

3. Apply the transformation to the set $\vec{p}$ and iterate until convergence

A correspondence is a match between a point from set $\vec{p}$ and its nearest neighbour from set $\vec{q}$. Once the correspondences are found minimizing the error between them is stated in (1). Where $\mathbf{H}$ is a 2D homogeneous transformation matrix, p_i is point i from point set $\vec{p}$, m_j is the closest point (j) from point set $\vec{q}$, n_i is the normal vector to the line between the closest and second closest point in $\vec{q}$. All points are in homogeneous coordinates.

$$\underset{\mathbf{H}}{\operatorname{argmin}} \sum_i ((\mathbf{H}p_i - m_j)n_i) \tag{1}$$

From (1) a linear system, $\mathbf{A}\vec{x} = \vec{b}$, is constructed and solved using QR-decompostion. The system is decomposed into $\mathbf{Q}$ and $\mathbf{R}$ using the Gramm-Schmidt method which is partially shown in (2), where $\mathbf{A} = [\vec{v}_0, \vec{v}_1, \vec{v}_2, \vec{v}_3]$, and $\mathbf{Q} = [\vec{u}_0, \vec{u}_1, \vec{u}_2, \vec{u}_3]$.

$$\begin{aligned}
\vec{u}_0 &= \frac{\vec{y}_0}{|\vec{y}_0|} \Rightarrow \vec{y}_0 = \vec{v}_0 \\
\vec{u}_1 &= \frac{\vec{y}_1}{|\vec{y}_1|} \Rightarrow \vec{y}_1 = \vec{v}_1 - (\vec{v}_1 \bullet \vec{u}_0)\vec{u}_0 \\
\vec{u}_2 &= \frac{\vec{y}_2}{|\vec{y}_2|} \Rightarrow \vec{y}_2 = \vec{v}_2 - (\vec{v}_2 \bullet \vec{u}_0)\vec{u}_0 - (\vec{v}_2 \bullet \vec{u}_1)\vec{u}_1
\end{aligned} \tag{2}$$

The transformation $\vec{x}$ is retrieved by solving the system $\mathbf{R}\vec{x} = \mathbf{Q}^T\vec{b}$ where $\mathbf{R}$ is a 4×4 upper triangular matrix, a more detailed description is given in Chapter 5 and 12 of [2].

4.1 Computing vectors to the closest points

```
icp :: SeqLogic s [Float] [Float] ()
icp = do
  vecPx <- receive
  clock
  vecPy <- receive
  clock
  vecQx <- receive
  clock
  vecQy <- receive
  clock

  vecNx <- alloc (replicate 180 undefined)
  vecNy <- alloc (replicate 180 undefined)
  vecMx <- alloc (replicate 180 undefined)
  vecMy <- alloc (replicate 180 undefined)

  loop 0 upto (180-1) $ \i -> do
    let px = vecPx!!i
    let vecDx = px -. vecQx
    clock
    let py = vecPy!!i
```

```
  let vecDy = py -. vecQy
  clock
  let vecDx2 = vecDx .*. vecDx
  clock
  let vecDy2 = vecDy .*. vecDy
  clock
  let vecSquaredDist = vecDx2 .+. vecDy2
  clock
  let (j,k) = indexSmallestTwo vecSquaredDist
  clock
  let mx0 = vecQx!!j
  let mx1 = vecQx!!k
  clock
  let my0 = vecQy!!j
  let my1 = vecQy!!k
  clock
  let (nx,ny)= findNormVec (px,py) (mx0,my0) (mx1,my1)
  clock
  vecMx?i <~ mx0
  clock
  vecMy?i <~ my0
  clock
  vecNx?i <~ nx
  clock
  vecNy?i <~ ny

v <- allocArr 4
v0 <- peek vecNx
v?0 <~ v0
clock
v1 <- peek vecNy
v?1 <~ v1
clock
let v2'' = vecPx .*. v0
clock
let v2'  = vecPy .*. v1
clock
v?2 <~ v2' .+. v2''
clock
let v3'' = vecPx .*. v1
clock
let v3'  = vecPy .*. v0
clock
v?3 <~ v3'' .-. v3'
clock
vecMx' <- peek vecMx
let b'' = vecMx' .*. v0
clock
vecMy' <- peek vecMy
let b'  = vecMy' .*. v1
clock
let b   = b' .+. b''
```

4.2 Solving a system of equations

```
u <- allocArr 4
call $ qr v u

x <- allocArr 4
linSolv <- start linearSolver
loop 3 downto 0 $ \j -> do
   u_j <- peek (u?j)
   t_j <- inline $ u_j `dotProd` b
   infuse linSolv t_j
   loop 3 downto j $ \k -> do
      v_k <- peek (v?k)
      r_jk <- inline $ u_j `dotProd` v_k
      infuse linSolv r_jk
   x_j <- extract linSolv
   x?j <~ x_j
finish linSolv

transX <- peek (x?0)
clock
transY <- peek (x?1)
clock
cosTh <- peek (x?2)
clock
sinTh <- peek (x?3)
clock
let vecPxCt = cosTh *. vecPx
clock
let vecPySt = sinTh *. vecPy
clock
let vecPxCtPySt = vecPxCt .-. vecPySt
clock
let vecPx' = transX +. vecPxCtPySt
clock
let vecPyCt = cosTh *. vecPy
clock
let vecPxSt = sinTh *. vecPx
clock
let vecPyCtPxSt = vecPyCt .+. vecPxSt
clock
let vecPy' = transY +. vecPyCtPxSt
clock

emit (vecPx')
clock
emit (vecPy')
```

Solving the linear equations of an upper triangular matrix

```
linearSolver :: SeqLogic s Float Float ()
linearSolver = do
  x <- allocArr 4
  loop 3 downto 0 $ \j -> do
     t_j <- receive
     tmp <- alloc t_j
     loop 3 downto (j+1) $ \k -> do
        r_jk <- receive
        x_k <- peek (x?k)
        let rx = r_jk * x_k
        clock
        t <- peek tmp
        tmp <~ t - rx
     r_jj <- receive
     t <- peek tmp
     let x_j = t / r_jj
     x?j <~ x_j
     emit x_j
  return ()
```

QR decomposition using the GramSchmidt process:

```
qr :: Ref s [[Float]] -> Ref s [[Float]] ->
   SeqLogic s [Float] [Float] ()
qr v u = do
  loop 0 upto 3 $ \j -> do
     v_j <- peek (v?j)
     tmp <- alloc v_j
     loop 0 upto (j-1) $ \k -> do
        u_k <- peek (u?k)
        vj_uk <- inline $ v_j `project` u_k
        clock
        t <- peek tmp
        tmp <~ t .-. vj_uk
     t <- peek tmp
     u_j <- inline $ normalize t
     u?j <~ u_j
  return ()

indexSmallestTwo :: [Float] -> (Int, Int)
indexSmallestTwo distances = (j,k) where
  ids = zip distances [0..]
  (xs, ys) = splitAt (div (length distances) 2) ids
  sortedLayer = zipWith sortTwo xs ys
```

```
  (_,j,_,k) = foldl1 combineSmallestPairs sortedLayer

--sort two points, output is (y, iy, z, iz) where y < z
sortTwo :: (Float,i) -> (Float,i) -> (Float,i, Float,i)
sortTwo (a, ia) (x, ix)
  | a < x  = (a, ia, x, ix)
  | otherwise = (x, ix, a, ia)

--find the two closest distances out of two pairs
--NOTE: assume input tuples are sorted (a < b && x < y)
combineSmallestPairs :: (Float, i, Float, i) ->
   (Float, i, Float, i) -> (Float, i, Float, i)
combineSmallestPairs (a, ia, b, ib) (x, ix, y, iy) =
  case (a < x, b < x, a < y) of
    (True, True, _)   -> (a, ia, b, ib)
    (True, False, _)  -> (a, ia, x, ix)
    (False, _, True)  -> (x, ix, a, ia)
    (False, _, False) -> (x, ix, y, iy)
```

Some multi cycle helper functions working with vectors; normalisation, dot product and vector projection:

```
normalize :: [Float] -> SeqLogic s i o [Float]
normalize xs = do
   let sqs = xs .*. xs
   clock
   let n = sum sqs
   clock
   let invsq = invSqrt n
   clock
   return (invsq *. xs)

dotProd :: [Float] -> [Float] -> SeqLogic s i o Float
dotProd xs ys = do
   let zs = xs .*. ys
   clock
   return (sum zs)

project :: [Float] -> [Float] -> SeqLogic s i o [Float]
project vs us = do
   let zs = vs .*. us
   clock
   let n = sum zs
   clock
   return (n *. us)
```

For convenience we define operators for scalar-vector and vector-vector operations:

```
n *. xs = map (n*) xs
n +. xs = map (n+) xs
n -. xs = map (n-) xs
(.*.) = zipWith (*)
(.+.) = zipWith (+)
(.-.) = zipWith (-)


divider :: Number-> Number -> SeqLogic s i o Number
divider x y = do
  let dividend = abs x
  let divisor = pack $ abs y
  -- Create shift register for non-restoring division
  let shReg = alloc (0 :: Number, dividend)
  (_, out) <- loopAccum (41, downto, 0) shReg $ \j tmp -> do
    let shiftReg' = shiftL (pack tmp) 1
    let (r, q) = split# shiftReg'
    let r' = plusMinus r divisor
    let q0 = bitNot $ msb r'
    let (q'', _) =  split# (q::BitVector (IntPart+FracPart))
    let q' = q'' ++# q0
    return (unpack r', unpack q')

  let isNeg = xor (y < 0) (x < 0)
  return $ condNeg isNeg out
```

4.3 Comparison with manual designed hardware

Work in progress

4.4 Optimizing the ICP implementation

- Apply software pipelining to speed up the finding of nearest two points loop.
- Use an ALU output register and computation reordering to limit the data memory usage to a single read port.

Work in progress

5. Generating a processor

- source to source transformation using Haskell-src-exts parser
- input sequential logic edsl, output clash subset of haskell

To be extended

5.1 Resource requirement analysis

The first step is to find for each cycle which variables are used and which are defined and used in another cycle. This determines the amount of read and write ports required on the data memory, and how many inputs and outputs the ALU structure needs. We need to take the type of variables in account, as variables of different size may need to be stored in different memories.

To be extended

5.2 Dealing with control flow

- side effect free if expressions within a cycle just become part of an alu instruction

- problems with delays in the control path
- control unit keeps track of loop counters to support zero overhead loop constructs

To be extended

5.3 Memory structure and addressing

- how to emulate multiport memories with blockram
- local variables are addressed relative to the stack pointer
- explicit allocated memory is addressed absolutely, thus not allowed in recursion functions
- every read and write port gets its own address generation unit (AGU) with access to the loop counters
- if a part of a memory element is written in a program then the memory is split in parts with a writemask to control them

To be extended

5.4 Optimise timing and reuse

- memory with read delay to better match with hardware
- ALU output registers with bypass for higher clocks?
- common subexpression detection on alu expressions

5.5 Generating CλaSH code

- each (co)processor in its own file
- all declarations except for the sequential logic functions are copied

To be extended

5.6 Hardware tradeoffs

The design of processors and mapping of algoritm involves many tradeoffs beyond deciding what happens in which cycle:

- storing data in registers vs. data stack memory
- direct control versus using an instruction memory
- where do constants come from?
- extra delay (pipeline register) before or after the ALU to increase clock frequency

We would like to provide some options in our compiler to control those choices in the generated hardware.

Work in progress

6. Related approaches

Using Haskell EDSLs to generate efficient low level code is common approach to gain both abstraction and performance, Feldspar [3] (used for generating C code for digital signal processing) is a nice example of this. Going from functional program descriptions to physical hardware realisation using Geometry of Synthesis [7] is different in many aspects, although it has similar goals.

To be extended

6.1 High level synthesis

Generates hardware from software written in ordinary programming languages (commonly C(++) or Matlab). We could describe our process as medium level synthesis, because with explicit clock cycle boundaries we do not have to select and schedule operations. While most high level synthesis approaches start from a behavioural description of an algorithm and have to deal with multiple hard optimisation problems.

To be extended

6.2 Customisable processors

Starting with a template of a processor to which specialised ALUs and/or coprocessors to accelerate specific algorithms.

To be extended

7. Conclusions

A processor is a thing that executes programs in many bounded steps. Merely defining the boundaries of steps to execute and their ordering is enough to derive processor like hardware structure from it.

To be extended

7.1 Future work

Implement the transformation process as GHC plugin instead of a fragile source to source compilation. Or even better convince the CλaSH developers to integrate this transformation process in their compiler, as that gives access to more information about the hardware to be generated.

Find suitable higher level abstractions to make the EDSL more functional than the current imperative features.

Would it be possible to extend the work of 'Calculating Correct Compilers' [5] to derive both a compiler and processor?

More to be added

Acknowledgments

The first author conducted this research within the Modern project (647.000.003) supported by NWO.

References

[1] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[2] R. Appel and H. Folmer. Analysis, optimization, and design of a slam solution for an implementation on reconfigurable hardware(fpga) using cλash. Master's thesis, University of Twente, December 2016. URL `http://essay.utwente.nl/71550/`.

[3] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of feldspar. *Implementation and Application of Functional Languages*, pages 121–136, 2011.

[4] C. Baaij. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. PhD thesis, University of Twente, 1 2015. eemcs-eprint-23939.

[5] P. Bahr and G. Hutton. Calculating Correct Compilers. *Journal of Functional Programming*, 25, Sept. 2015.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[7] D. R. Ghica, A. Smith, and S. Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. *ACM SIGPLAN Notices*, 46(9):221–233, 2011.

[8] C. E. LaForest. Second-generation stack computer architecture. B.S. thesis, University of Waterloo, 2007.

[9] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, 11(4):363–397, 1998.

[10] S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001.

[11] M. Sheeran. Hardware Design and Functional Programming: a Perfect Match. *Journal of Universal Computer Science*, 11(7):1135–1158, jul 2005.

[12] I. te Raa. Recursive functional hardware descriptions using cλash. Master's thesis, University of Twente, November 2015. URL `http://essay.utwente.nl/68804/`.

[13] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. Hardware synthesis from a recursive functional language. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*, pages 83–93. IEEE, 2015.

Mapping Functional Languages to GPUs: Memory and Communication Choices

Hans-Nikolai Vießmann
hv15@hw.ac.uk
Heriot-Watt University

Sven-Bodo Scholz
S.Scholz@hw.ac.uk
Heriot-Watt University

Abstract

The side-effect free setting of functional languages inherently comes without a notion of memory cells and assignments. If these notions are needed or desired, extra machinery such as streams, monads or uniqueness typing are required. Consequently, the core of any implementation for functional languages provides mechanisms to manage memory implicitly. Normally, this is either done by means of delayed or non-delayed garbage collection.

When it comes to compiling functional languages for execution on GPUs the situation becomes more complex, at least if a completely implicit notion of memory and hardware are to be maintained. GPU systems are inherently heterogeneous. They always consist of a conventional host system and a GPU component, the device. Each of these have separate memories and data needs to be transferred between them. For a compilation of hardware-agnostic functional programs this opens up a large design space. Programs need to be divided up between host and device, memory needs to be managed on two different systems and data transfers need to be orchestrated accordingly.

Adding to the challenge, GPUs typically offer different interfaces for allocating memory on the host, allocating memory on the device and for sending data from one to the other. These interfaces differ in their runtime performance characteristics, their associated overheads, and their overall behaviours. Some transfers are synchronous, others are asynchronous, yet others are done on demand only. The different memory allocations make use of different host system mechanisms and work differently well together with the different communication strategies. The resulting runtime differences can be as large as a factor of two between different memory transfer techniques.

Previous work shows that one of the key challenges in getting good performance out of GPU systems lies in a mapping that minimises the communication requirements between host and device since data transfers are known to potentially annihilate all performance gains made on GPUs. Despite the aim to reduce transfers between host and device, it turns out that some GPU applications still spend considerable fractions of their wall-clock execution time in such communications. This observation motivates the work presented here.

We look at a given mapping of functional programs onto memory and communication and investigate the effects of using the alternative memory allocators and communication strategies available. As a basis for our experiments we use the existing compiler tool chain for SaC which is capable of generating CUDA code that exposes explicit allocations on host and device as well as the data transfers between them.

We analyse the impact different choices have on several applications and we discuss in detail how these results relate to hand-coded experiments that expose the raw characteristics of the individual interface choices offered by the CUDA API.

It turns out that the wall-clock times even for benchmarks with minimal communication between host and device vary by up to 15%. For more communication intensive programs, larger differences can be observed albeit it turns out that some additional code optimisation is required which actually changes the way garbage collection on host and device are being performed.

IFL'17, Aug. 30 – Sept. 1, 2017, Bristol, UK

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

ACM Reference format:
Hans-Nikolai Vießmann and Sven-Bodo Scholz. 2017. Mapping Functional Languages to GPUs: Memory and Communication Choices. In *Proceedings of 29th Symposium on Implementation and Application of Functional Languages, Bristol, UK, Aug. 30 – Sept. 1, 2017 (IFL'17),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Handlers for Non-Monadic Computations

Ruben P. Pieters
KU Leuven
ruben.pieters@cs.kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@cs.kuleuven.be

Exequiel Rivas
CIFASIS CONICET Universidad Nacional de Rosario, Argentina
rivas@cifasis-conicet.gov.ar

Abstract

Algebraic effects and handlers are a convenient method for structuring monadic effects and for separating the syntax of primitive effectful operations from their interpretation. Unfortunately their scope is somewhat limited as not all side-effects are monadic in nature.

To widen their scope this paper generalises the notion of algebraic effects and handlers from monads to generalised monoids, which notably covers applicative functors and arrows. For this purpose we switch from the conventional approach based on free algebras to one based on free monoids. Moreover, we show how lax monoidal functors allow us to handle, e.g., applicative effects with monadic handlers.

CCS Concepts • **Theory of computation → Functional constructs**; • **Software and its engineering → Functional languages**;

Keywords algebraic effects, effect handlers, generalised monoids, monads, applicative functors, arrows

ACM Reference Format:
Ruben P. Pieters, Tom Schrijvers, and Exequiel Rivas. 2017. Handlers for Non-Monadic Computations. In *Proceedings of Implementation and Application of Functional Languages, University of Bristol, Bristol, August 2017 (IFL'17),* 14 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Monads [10, 13] have for a long time had a monopoly on modelling computational effects in purely functional programming. This changed with the proposal of new classes of side-effects, applicative functors [9] and arrows [4], that capture types of side-effects amenable to static analysis at the cost of the expressiveness of monadic side-effects.

In a separate development algebraic effects and handlers [11] were created as a more convenient formulation of monadic effects and programs. They encode effects as operations given by the signature of an algebraic theory. These signatures are interpreted by an effect handler, which determines their behaviour. An important factor in the current success of the algebraic effects and handlers approach is that it readily integrates with impure functional and imperative languages to enable user-defined effects, more so than the conventional monadic approach.

Yet, while algebraic effects and handlers are a good way to capture monadic effects, they do not cover other classes of effects like applicative functors and arrows. To remedy this situation Lindley [5] has already presented a language design that supports handlers for three different kinds of effects (monadic, applicative or arrow) and a type system that verifies which kind of effect the program uses. What Lindley does not do is extend the category theoretical underpinnings at the basis of Plotkin and Pretnar's original work to the other two kind of effects.

This work aims to close that gap and provide a category-theoretical definition of algebraic effects and handlers for a wider range of effects. For this purpose we leverage the framework of Rivas and Jaskelioff [12] which characterises three kinds of effectful computations in terms of generalised monoids. We use this to replace the conventional monadic approach based on free algebras for syntax and their unique algebra homomorphism for handlers with free monoids for syntax and their unique monoid homomorphism for handlers.

The main advantage of these new types of handlers is the possibility of expressing static analysis with them. To illustrate with a simple example, imagine we have a computation with a sequence of `get`s and `put`s.

```
get()
put(C1)
put(C2)
get()
put(C3)
put(C4)
...
```


IFL'17, August 2017, University of Bristol, Bristol

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

With an applicative handler we could interpret this computation to batch together all `gets` into one and all `puts` into one batched `put`. With an arrow handler we are a bit more restricted, but we could still batch together different `puts` in between `gets`. To create such handlers requires more expressiveness than is possible with monadic handlers, because a monadic computation is in general too expressive to allow this interpretation. For examples of batching remote procedure calls, see the works of Gibbons [2], Gill et al. [3] and Marlow et al. [7].

2 Background

2.1 Notation

Throughout the remainder of the paper we use the following, mostly conventional notation for category theoretical concepts: We use $\mathscr{C}$, $\mathscr{D}$ to denote categories; A, B, X to denote objects in those categories; and a, b, f, g to denote morphisms between those objects. Identity morphisms are usually denoted as the more compact $A : A \to A$ instead of $id_A : A \to A$.

In categories with finite coproducts we use $A + B$ to denote the coproduct of objects A and B, and $[f, g]$ to denote the unique morphism $A + B \to X$ given morphisms $f : A \to X$ and $g : B \to X$.

We use F, G to denote functors and, when the source and target category are important, they are given with $F : \mathscr{C} \to \mathscr{D}$. We also use functor categories $[\mathscr{C}, \mathscr{D}]$ whose objects are functors $F, G : \mathscr{C} \to \mathscr{D}$ and whose morphisms are natural transformations $\alpha, \beta : F \to G$. Of special interests are categories of endofunctors $[\mathscr{C}, \mathscr{C}]$ and profunctors $[\mathscr{C}^{op} \times \mathscr{C}, Set]$.

In Haskell examples, where $\mathscr{C}$ is the category of Haskell's base types and functions between them, we use `~>` for natural transformations and `~~>` for natural transformations for bifunctors. These are defined as:

```
type f ~>  g = forall x.   f x   -> g x
type f ~~> g = forall x y. f x y -> g x y
```

2.2 Algebraic Effects and Handlers

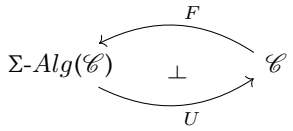
Plotkin and Pretnar's definition of algebraic effects and handlers consists of two parts: the operations, which introduce effects, and the handlers, which eliminate them.

Operations and Signature Algebras Effects are introduced in programs by means of (primitive) *operations*. If A is the object that denotes computations, then operations are morphisms of the form $op_i : P \times A^N \to A$, where N is the *arity* of the operation and P captures any parameters of the operation. The intention is that the operation builds a larger computation out of N given computations and a parameter values.

We combine the operations $\{op_0, \ldots, op_n\}$ that can be used in a program in a Σ-algebra $\langle A, \Sigma A \to A\rangle$ where the domain of computations A is called the *carrier*. Here, Σ is known as the signature; it is the endofunctor defined by the finite coproduct $(P_0 \times -^{N_0}) + \ldots + (P_n \times -^{N_n})$. In essence, a Σ-algebra captures a denotation of the operations.

Free Algebras, Syntax and Handlers For a given signature Σ we can construct a category $\Sigma\text{-}Alg(\mathscr{C})$ whose objects are Σ-algebras with carriers from $\mathscr{C}$ (i.e., all possible denotations of the Σ-operations for all possible domains). The morphisms $f : \langle A, a\rangle \to \langle B, b\rangle$ are algebra homomorphisms; these are actually morphisms $f : A \to B$ in $\mathscr{C}$ such that $f \circ a = b \circ \Sigma f$.

The notions of program syntax and handler arise from the adjunction between the free algebra functor $F : \mathscr{C} \to \Sigma\text{-}Alg(\mathscr{C})$ and the corresponding forgetful functor U.



The forgetful functor U maps an algebra to its underlying carrier, $U(\langle A, a\rangle) = A$, and an algebra morphism to the same morphism in the underlying category, $U(f) = f$. The functor F takes an object A in $\mathscr{C}$ and maps it to the free algebra $F(A) = \langle \Sigma^* A, com_A : \Sigma(\Sigma^* A) \to \Sigma^* A\rangle$. That $F(A)$ is a free algebra means that there is a morphism $var_A : A \to \Sigma^* A$ such that for any other Σ-algebra $\langle B, b : \Sigma B \to B\rangle$ and morphism $f : A \to B$ there is a unique algebra homomorphism $handle_b\, f : \Sigma^* A \to B$ (i.e., with

$$\begin{array}{ccc} \Sigma(\Sigma^* A) & \xrightarrow{\Sigma(handle_b\, f)} & \Sigma B \\ \downarrow{\scriptstyle com_A} & & \downarrow{\scriptstyle b} \\ \Sigma^* A & \xrightarrow{handle_b\, f} & B \end{array} \tag{1}$$

) such that this diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{var_A} & \Sigma^* A \\ & {\scriptstyle f}\searrow & \downarrow{\scriptstyle handle_b\, f} \\ & & B \end{array} \tag{2}$$

Informally we can say that the carrier type $\Sigma^* A$ denotes the uninterpreted (abstract) syntax trees for programs with effectful operations drawn from F and results of type A. The handler $handle_b\, f$ interprets such syntax trees in domain B.

Observe that $\Sigma^* = UF$ is a functor with a monadic structure which is known as the *free monad*.

Haskell Encoding In Haskell the functor Σ^* of the free monad can be defined as

```
data Free f a = Var a | Com (f (Free f a))
```

The handler $handle_b\ f$ can be encoded as `handle f b` where

```
handle :: Functor f => (a -> b) -> (f b -> b)
                    -> (Free f a -> b)
handle f b (Var x)  = f x
handle f b (Com op) = b (fmap (handle f b) op)
```

The Eff Language Algebraic effects and handlers are native in the Eff programming language [1]. The operations from the signature Σ are declared in the form

```
effect op_i : P_i -> N_i
```

and all program expressions with Eff type A (that call zero or more operations) are by default interpreted in the domain $\Sigma^* A$.

An Eff handler is expressed in this form:

```
handler
| val x -> c_v
| op_0 p_0 k_1 ... k_{N_0} -> c_0
  ⋮
| op_n p_n k_0 ... k_{N_n} -> c_n
```

This captures the essential information of the morphism f and the Σ-algebra b as follows. Firstly, f is given in the `val` clause and corresponds to $\lambda x.c_v$. The remaining clauses together define the Σ-algebra

$$b = [\lambda p_0.\lambda k_1.\ \ldots \lambda k_{N_0}.c_0, \ldots, \lambda p_n.\lambda k_1.\ \ldots \lambda k_{N_n}.c_n]$$

3 Handlers for Generalised Monoids

The above free-algebra-based approach necessarily (because of the adjunction) leads to $\Sigma^* A$ being a monad. In order to represent and handle non-monadic effects, we look to free monoids for an alternative construction. These are a good choice because in the case of monads, the free construction happens to coincide with $\Sigma^* A$ and the unique monoid morphism provides an alternative notion of handler.

3.1 Generalised Monoids in Monoidal Categories

Rivas and Jaskelioff [12] present a framework for different kinds of side-effects as (generalised) monoids in different monoidal categories. We reintroduce the relevant definitions relating to monoidal categories here.

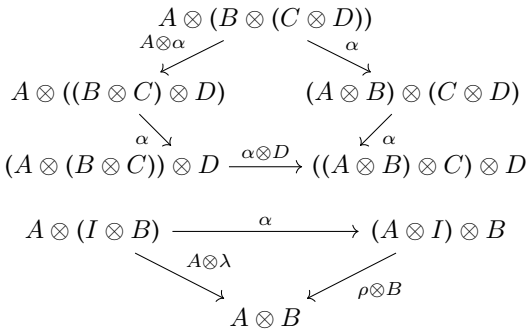
Monoidal Category A monoidal category is a category that has additional stucture which enables it to host a notion of monoids that generalises the usual notion of monoids in the `Set` category.

Definition 3.1 (Monoidal Category)**.** A monoidal category is a tuple $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$, consisting of

a) a category $\mathscr{C}$
b) a bifunctor $\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$
c) a designated object I of $\mathscr{C}$
d) three natural isomorphisms

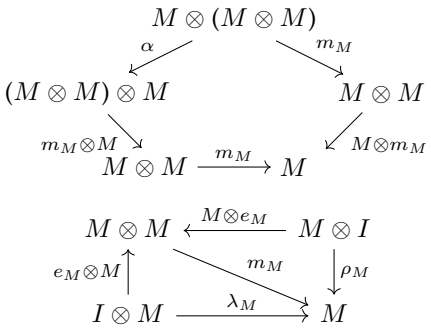
$$\begin{aligned} \alpha_{A,B,C} &: A \otimes (B \otimes C) \to (A \otimes B) \otimes C \\ \lambda_A &: I \otimes A \to A \\ \rho_A &: A \otimes I \to A \end{aligned}$$

such that $\lambda_I = \rho_I$ and with the following commuting diagrams:



Monoid in Monoidal Category A monoid in a monoidal category is a generalisation of monoids in `Set`.

Definition 3.2 (Generalised Monoid)**.** A generalised monoid is a tuple (M, m_M, e_M) where M is an object in a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$. The multiplication $m_M : M \otimes M \to M$ and the unit $e_M : I \to M$ are morphisms in $\mathscr{C}$ such that the following diagrams commute:



Example 3.3. The main examples of monoidal categories we consider are:

1. The category of endofunctors, `End`, with functor composition $\circ$ and the identity functor Id as designated object. This monoidal category is called *strict* since α, λ and ρ are identites. Monoids in this monoidal category are known as monads.

2. The category of endofunctors `End`, now with Day convolution $\star$ as tensor and again with the identity functor Id as designated object. Monoids in this monoidal category are known as applicative functors.
3. The category of profunctors, `Pro`, with profunctor composition and the Hom profunctor as designated object. Monoids in this monoidal category are known as pre-arrows.

Free Monoid Amongst all the monoids in a monoidal category, the free ones are interesting because they give us a notion of syntax and interpretation similar to the free algebra setting.

Definition 3.4 (Free Monoid)**.** Let $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ be a monoidal category. The free monoid on an object A in $\mathscr{C}$ is a monoid (A^*, m_{A^*}, e_{A^*}) together with a morphism $ins : A \to A^*$ such that for any monoid (M, m_M, e_M) and any morphism $f : A \to M$, there exists a unique monoid homomorphism $free\, f : A^* \to M$ that makes the following diagram commute.

$$\begin{array}{ccc} A & \xrightarrow{ins} & A^* \\ & \searrow^{f} & \downarrow{\scriptstyle free\, f} \\ & & M \end{array}$$

The diagrams corresponding to the monoid homomorphism condition are:

$$\begin{array}{ccc} I & \xrightarrow{e_{A^*}} & A^* \\ & \searrow_{e_M} & \downarrow{\scriptstyle free\, f} \\ & & M \end{array} \tag{3}$$

$$\begin{array}{ccc} A^* \otimes A^* & \xrightarrow{free\, f \otimes free\, f} & M \otimes M \\ {\scriptstyle m_{A^*}}\downarrow & & \downarrow{\scriptstyle m_M} \\ A^* & \xrightarrow{free\, f} & M \end{array} \tag{4}$$

The object A^* gives us a notion of program syntax, which in the case of monads and $A = \Sigma$ coincides with that of Section 2.2. Similarly, $free\, f$ gives us a notion of handling this program syntax. However, it is important to note that it differs from $handle_b\, f$.

3.2 Free Monoid Construction

Provided it exists, the initial algebra for the endofunctor $F_A = I + A \otimes -$ is the free monoid on A. This allows us to represent the free monoid as an inductive data type. Moreover, it gives us a new interface to the unique morphism we would like to have as handling construct.

Catamorphism Provided the initial algebra $\langle A^*, in : F_A A^* \to A^* \rangle$ exists, there is a unique algebra homomorphism from the initial algebra to any other algebra $\langle B, b : F_A B \to B \rangle$, called the catamorphism $(\!|b|\!)$. The corresponding algebra homomorphism condition is:

$$\begin{array}{ccc} F_A A^* & \xrightarrow{F_A (\!|b|\!)} & F_A B \\ {\scriptstyle in}\downarrow & & \downarrow{\scriptstyle b} \\ A^* & \xrightarrow{(\!|b|\!)} & B \end{array}$$

We can deconstruct in and b into respectively $in = [in_1 = in \circ inl, in_2 = in \circ inr]$ and $b = [b_1 = b \circ inl, b_2 = b \circ inr]$. Then we can deconstruct the above diagram into two:

$$\begin{array}{ccc} I & = & I \\ {\scriptstyle in_1}\downarrow & & \downarrow{\scriptstyle b_1} \\ A^* & \xrightarrow{(\!|b|\!)} & X \end{array} \tag{5}$$

$$\begin{array}{ccc} A \otimes A^* & \xrightarrow{A \otimes (\!|b|\!)} & A \otimes X \\ {\scriptstyle in_2}\downarrow & & \downarrow{\scriptstyle b_2} \\ A^* & \xrightarrow{(\!|b|\!)} & X \end{array} \tag{6}$$

To show that the initial F_A-algebra gives rise to the free monoid A^*, we show that the carrier of the algebra forms a monoid which is free.

Exponentials In the upcoming paragraphs we make use of *exponentials* which are a way to represent the set of morphisms $\mathscr{C}(X, Y)$ as an object B^A in the category. In particular, we employ a definition specific to monoidal categories, meaning we use the tensor $\otimes$ instead of coproducts with $\times$.

Definition 3.5 (Exponential)**.** Let A be an object of a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$. A right exponential $-^A$ is the right adjoint to $- \otimes A$. That is, the right exponential to A is characterised by an isomorphism

$$\lfloor - \rfloor : \mathscr{C}(X \otimes A, B) \cong \mathscr{C}(X, B^A) : \lceil - \rceil$$

natural in X and B.

The isomorphisms in this adjunction provide a generalized form of currying and uncurrying.

The naturality of $\lceil - \rceil$ in X will be used in proofs later on, so we state it here explicitly:

$$\lceil f \rceil \circ (g \otimes id) = \lceil f \circ g \rceil \tag{7}$$

The counit of the adjunction is called the *evaluation morphism*, and is defined as follows:

$$ev_B = \lceil B^A \rceil : B^A \otimes A \to B \tag{8}$$

A^* is a Monoid We show that (A^*, e_{A^*}, m_{A^*}) is a monoid, Here A^* [1] is the carrier of the initial algebra of F_A. The unit and multiplication of this monoid are defined as:

$$e_{A^*} = in_1 \tag{9}$$

$$m_{A^*} = \lceil (\!| [\![\lfloor \lambda_{A^*} \rfloor, \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \rfloor]\!] |\!) \rceil \tag{10}$$

First we show that e_{A^*} is the left unit of m_{A^*}:

$$m_{A^*} \circ (e_{A^*} \otimes A^*) = \lambda_{A^*} \tag{11}$$

Proof.

$m_{A^*} \circ (e_{A^*} \otimes A^*)$
= (definitions 9 and 10)
$\lceil (\!| [\![\lfloor \lambda_{A^*} \rfloor, \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \rfloor]\!] |\!) \rceil \circ (in_1 \otimes A^*)$
= (naturality of $\lceil - \rceil$)
$\lceil (\!| [\![\lfloor \lambda_{A^*} \rfloor, \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \rfloor]\!] |\!) \circ in_1 \rceil$
= (catamorphism property 5)
$\lceil \lfloor \lambda_{A^*} \rfloor \rceil$
= (isomorphism)
λ_{A^*} □

Next we prove that e_{A^*} is also the right unit of m_{A^*}:

$$m_{A^*} \circ (A^* \otimes e_{A^*}) = \rho_{A^*} \tag{12}$$

Proof.

$m_{A^*} \circ (A^* \otimes e_{A^*})$
= (isomorphism)
$m_{A^*} \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1} \circ \rho_{A^*}$
= (proof see below)
ρ_{A^*}

For the last step in the above proof we show that $m_{A^*} \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1} = id_{A^*}$ by showing both sides are algebra homomorphisms from the initial algebra to itself. Because of the uniqueness of such a homomorphism, they must be equal. This decomposes into two steps, one for in_1 and one for in_2. For in_1 we have:

$m_{A^*} \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1} \circ in_1$
= (ρ^{-1} is a natural transformation)
$m_{A^*} \circ (A^* \otimes e_{A^*}) \circ (in_1 \otimes I) \circ \rho_I^{-1}$
= ($\otimes$ bifunctor)
$m_{A^*} \circ (in_1 \otimes A^*) \circ (I \otimes e_{A^*}) \circ \rho_I^{-1}$
= (Definition 9 of e_{A^*})
$m_{A^*} \circ (e_{A^*} \otimes A^*) \circ (I \otimes e_{A^*}) \circ \rho_I^{-1}$
= (left unit property 11)
$\lambda_{A^*} \circ (I \otimes e_{A^*}) \circ \rho_I^{-1}$
= (λ is a natural transformation)
$e_{A^*} \circ \lambda_I \circ \rho_I^{-1}$
= ($\lambda_I = \rho_I$ as per Definition 3.1)
$e_{A^*} \circ \rho_I \circ \rho_I^{-1}$
= (isomorphism)
e_{A^*}
= (Definition 9 of e_{A^*})
in_1

For in_2 we have:

$m_{A^*} \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1} \circ in_2$
= (definition of m_{A^*} with $b = [\ldots, \ldots]$)
$\lceil (\!| b |\!) \rceil \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1} \circ in_2$
= (ρ^{-1} is a natural transformation)
$\lceil (\!| b |\!) \rceil \circ (A^* \otimes e_{A^*}) \circ (in_2 \otimes I) \circ \rho_{A \otimes A^*}^{-1}$
= ($\otimes$ bifunctor)
$\lceil (\!| b |\!) \rceil \circ (in_2 \otimes A^*) \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (naturality of $\lceil - \rceil$)
$\lceil (\!| b |\!) \circ in_2 \rceil \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (catamorphism property 6)
$\lceil b_2 \circ (A \otimes (\!| b |\!)) \rceil \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (naturality of $\lceil - \rceil$)
$\lceil b_2 \rceil \circ ((A \otimes (\!| b |\!)) \otimes A^*) \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (def. of b_2 and $\lceil \lfloor - \rfloor \rceil$ isomorphism)
$in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \circ ((A \otimes (\!| b |\!)) \otimes A^*)$
$\quad \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (α^{-1} is a natural transformation)
$in_2 \circ (A \otimes ev_{A^*}) \circ (A \otimes ((\!| b |\!) \otimes A^*)) \circ \alpha^{-1}$
$\quad \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= ($\otimes$ bifunctor)
$in_2 \circ (A \otimes (ev_{A^*} \circ ((\!| b |\!) \otimes A^*))) \circ \alpha^{-1}$
$\quad \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (def. of ev_{A^*})
$in_2 \circ (A \otimes (\lceil A^{*A^*} \rceil \circ ((\!| b |\!) \otimes A^*))) \circ \alpha^{-1}$
$\quad \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (naturality of $\lceil - \rceil$)
$in_2 \circ (A \otimes \lceil (\!| b |\!) \rceil) \circ \alpha^{-1} \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (def. of m_{A^*})
$in_2 \circ (A \otimes m_{A^*}) \circ \alpha^{-1} \circ ((A \otimes A^*) \otimes e_{A^*}) \circ \rho_{A \otimes A^*}^{-1}$
= (α^{-1} is a natural transformation)
$in_2 \circ (A \otimes m_{A^*}) \circ (A \otimes (A^* \otimes e_{A^*})) \circ \alpha^{-1} \circ \rho_{A \otimes A^*}^{-1}$
= (property[2] $\alpha^{-1} \circ \rho^{-1} = id \otimes \rho^{-1}$)
$in_2 \circ (A \otimes m_{A^*}) \circ (A \otimes (A^* \otimes e_{A^*})) \circ (A \otimes \rho_{A^*}^{-1})$
= ($\otimes$ bifunctor)
$in_2 \circ (A \otimes (m_{A^*} \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1}))$

Hence we can conclude that $m_{A^*} \circ (A^* \otimes e_{A^*}) \circ \rho_{A^*}^{-1} : A^* \to A^*$ is an F_A-algebra homomorphism. It is trivial to show that $id_{A*} : A^* \to A^*$ is an F_A-algebra homomorphism too. Because $\langle A^*, in \rangle$ is the initial F_A-algebra, there is only one such F_A-algebra homomorphism. Hence both must be equal. □

[1] In this and the upcoming section A^* is the notation for the initial algebra of F_A. We prove that it induces the free monoid, hence the overlapping notation.

[2] see https://ncatlab.org/nlab/show/monoidal+category, do note that the direction of α is inversed on this link

$$m_{A^*} \circ (A^* \otimes m_{A^*}) = m_{A^*} \circ (m_{A^*} \otimes A^*) \circ \alpha \quad (13)$$

Proof. This proof is left as future work. □

Initial Algebra induces Free Monoid Now we show that the carrier of the initial algebra A^* is also the carrier of the free monoid over A with

$$\begin{aligned} ins &= in_2 \circ (A \otimes in_1) \circ \rho_A^{-1} && (14)\\ free\, f &= (\!|[e_M, m_M \circ (f \otimes M)]|\!) && (15)\end{aligned}$$

for any morphism $f : A \to M$.

The first condition for the free monoid is that $free\, f \circ ins = f$.

Proof.

$$\begin{aligned}
& free\, f \circ ins \\
=\ & \quad \text{(defs. of } ins \text{ and } free\, f\text{)} \\
& (\!|[e_M, m_M \circ (f \otimes M)]|\!) \circ in_2 \circ (A \otimes in_1) \circ \rho_A^{-1} \\
=\ & \quad \text{(property of } (\!|-|\!)\text{)} \\
& m_M \circ (f \otimes M) \circ (A \otimes (\!|[e_M, \ldots]|\!)) \circ (A \otimes in_1) \circ \rho_A^{-1} \\
=\ & \quad \text{(bifunctor } \otimes\text{)} \\
& m_M \circ (f \otimes M) \circ (A \otimes ((\!|[e_M, \ldots]|\!) \circ in_1)) \circ \rho_A^{-1} \\
=\ & \quad \text{(property of } (\!|-|\!)\text{)} \\
& m_M \circ (f \otimes M) \circ (A \otimes e_M) \circ \rho_A^{-1} \\
=\ & \quad \text{(bifunctor } \otimes\text{)} \\
& m_M \circ (M \otimes e_M) \circ (f \otimes I) \circ \rho_A^{-1} \\
=\ & \quad \text{(naturality of } \rho^{-1}\text{)} \\
& m_M \circ (M \otimes e_M) \circ \rho_M^{-1} \circ f \\
=\ & \quad \text{(monoid right unit property)} \\
& \rho_M \circ \rho_M^{-1} \circ f \\
=\ & \quad \text{(inverses)} \\
& f \quad \square
\end{aligned}$$

The second condition for the free monoid is that $free\, f$ must be a monoid homomorphism. Hence, diagram 3 must hold:

$$free\, f \circ e_{A^*} = e_M \quad (16)$$

Proof.

$$\begin{aligned}
& free\, f \circ e_{A^*} \\
=\ & \quad \text{(defs. of } free \text{ and } e_{A^*}\text{)} \\
& (\!|[e_M, m_M \circ (f \otimes M)]|\!) \circ in_1 \\
=\ & \quad \text{(property of } (\!|-|\!)\text{)} \\
& e_M \quad \square
\end{aligned}$$

Also, diagram 4 must hold:

$$free\, f \circ m_{A^*} = m_M \circ (free\, f \otimes free\, f) \quad (17)$$

Proof. We prove this property by first showing that both sides, after currying with $\lfloor - \rfloor$, are algebra homomorphisms from the initial algebra to the F_A-algebra $\langle M^{A^*}, [\lfloor free\, f \circ \lambda_{A^*} \rfloor, \lfloor m_M \circ (f \otimes \lceil M^{A^*} \rceil) \circ \alpha^{-1} \rfloor] \rangle$. Due to initiality both sides must then be equal.

We show the algebra homomorphism properties for the left hand side. The in_1 property:

$$\begin{aligned}
& \lfloor free\, f \circ m_{A^*} \rfloor \circ in_1 \\
=\ & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
& \lfloor free\, f \circ m_{A^*} \circ (in_1 \otimes A^*) \rfloor \\
=\ & \quad \text{(def. } m_{A^*} \text{ \& naturality of } \lceil - \rceil\text{)} \\
& \lfloor free\, f \circ \lceil (\!|b|\!) \circ in_1 \rceil \rfloor \\
=\ & \quad \text{(def. } free\, f \text{ \& property of } (\!|-|\!)\text{)} \\
& \lfloor free\, f \circ \lambda_{A^*} \rfloor
\end{aligned}$$

The in_2 property:

$$\begin{aligned}
& \lfloor free\, f \circ m_{A^*} \rfloor \circ in_2 \\
=\ & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
& \lfloor free\, f \circ m_{A^*} \circ (in_2 \otimes A^*) \rfloor \\
=\ & \quad \text{(def. } m_{A^*} \text{ \& naturality of } \lceil - \rceil\text{)} \\
& \lfloor free\, f \circ \lceil (\!|b|\!) \circ in_2 \rceil \rfloor \\
=\ & \quad \text{(property of } (\!|-|\!)\text{)} \\
& \lfloor free\, f \circ \lceil b_2 \circ (A \otimes (\!|b|\!)) \rceil \rfloor \\
=\ & \quad \text{(naturality of } \lceil - \rceil\text{)} \\
& \lfloor free\, f \circ \lceil b_2 \rceil \circ ((A \otimes (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(substitute } b_2 \text{ \& isomorphism)} \\
& \lfloor free\, f \circ in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \circ ((A \otimes (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(def. } free\, f \text{ \& property of } (\!|-|\!) \text{ \& bifunctor } \otimes\text{)} \\
& \lfloor m_M \circ (f \otimes (free\, f \circ ev_{A^*})) \circ \alpha^{-1} \circ ((A \otimes (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(def. } ev_{A^*} \text{ \& naturality } \lceil - \rceil\text{)} \\
& \lfloor m_M \circ (f \otimes \lceil free\, f^{A^*} \rceil) \circ \alpha^{-1} \circ ((A \otimes (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(naturality of } \lceil - \rceil\text{)} \\
& \lfloor m_M \circ (f \otimes (\lceil M^{A^*} \rceil \circ (free\, f^{A^*} \otimes A^*)) \circ \alpha^{-1} \\
& \quad \circ ((A \otimes (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(bifunctor } \otimes\text{)} \\
& \lfloor m_M \circ (f \otimes \lceil M^{A^*} \rceil) \circ (A \otimes (free\, f^{A^*} \otimes A^*)) \circ \alpha^{-1} \\
& \quad \circ ((A \otimes (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(naturality of } \alpha^{-1} \text{ \& bifunctor)} \\
& \lfloor m_M \circ (f \otimes \lceil M^{A^*} \rceil) \circ \alpha^{-1} \circ ((A \otimes (free\, f^{A^*} \circ (\!|b|\!)) \otimes A^*) \rfloor \\
=\ & \quad \text{(naturality of } \lfloor - \rfloor\text{)} \\
& \lfloor m_M \circ (f \otimes \lceil M^{A^*} \rceil) \circ \alpha^{-1} \rfloor \circ (A \otimes (free\, f^{A^*} \circ (\!|b|\!))) \\
=\ & \quad (\!|b|\!) = \lfloor m_{A^*} \rfloor\text{)} \\
& \lfloor m_M \circ (f \otimes \lceil M^{A^*} \rceil) \circ \alpha^{-1} \rfloor \circ (A \otimes (free\, f^{A^*} \circ \lfloor m_{A^*} \rfloor)) \\
=\ & \quad \text{(naturality of } \lfloor - \rfloor\text{)} \\
& \lfloor m_M \circ (f \otimes \lceil M^{A^*} \rceil) \circ \alpha^{-1} \rfloor \circ (A \otimes (\lfloor free\, f \circ m_{A^*} \rfloor))
\end{aligned}$$

We show the algebra homomorphism properties for the right hand side. The in_1 property:

$$\begin{aligned}
& \lfloor m_M \circ (free\, f \otimes free\, f) \rfloor \circ in_1 \\
=\ & \quad \text{(naturality of } \lfloor - \rfloor\text{)} \\
& \lfloor m_M \circ (free\, f \otimes free\, f) \circ (in_1 \otimes A^*) \rfloor
\end{aligned}$$

$$
\begin{aligned}
&= \quad \text{(bifunctor } \otimes\text{)} \\
&\quad \lfloor m_M \circ ((\mathit{free}\, f \circ \mathit{in}_1) \otimes \mathit{free}\, f)\rfloor \\
&= \quad \text{(def. } \mathit{free}\, f \text{ \& property of } (\!|-|\!)\text{)} \\
&\quad \lfloor m_M \circ (e_M \otimes \mathit{free}\, f)\rfloor \\
&= \quad \text{(bifunctor } \otimes \text{ \& monoid left unit property)} \\
&\quad \lfloor \lambda_M \circ (I \otimes \mathit{free}\, f)\rfloor \\
&= \quad \text{(naturality of } \lambda\text{)} \\
&\quad \lfloor \mathit{free}\, f \circ \lambda_{A^*}\rfloor
\end{aligned}
$$

The in_2 property:

$$
\begin{aligned}
&\quad \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor \circ \mathit{in}_2 \\
&= \quad \text{(naturality } \lfloor -\rfloor \text{ \& bifunctor } \otimes\text{)} \\
&\quad \lfloor m_M \circ ((\mathit{free}\, f \circ \mathit{in}_2) \otimes \mathit{free}\, f)\rfloor \\
&= \quad \text{(def. } \mathit{free}\, f \text{ \& property of } (\!|-|\!) \text{ \& bifunctor } \otimes\text{)} \\
&\quad \lfloor m_M \circ (m_M \otimes M) \circ ((f \otimes \mathit{free}\, f) \otimes \mathit{free}\, f)\rfloor \\
&= \quad \text{(monoid associativity property)} \\
&\quad \lfloor m_M \circ (M \otimes m_M) \circ \alpha^{-1} \circ ((f \otimes \mathit{free}\, f) \otimes \mathit{free}\, f)\rfloor \\
&= \quad \text{(naturality of } \alpha^{-1}\text{)} \\
&\quad \lfloor m_M \circ (M \otimes m_M) \circ (f \otimes (\mathit{free}\, f \otimes \mathit{free}\, f)) \circ \alpha^{-1}\rfloor \\
&= \quad \text{(bifunctor } \otimes\text{)} \\
&\quad \lfloor m_M \circ (f \otimes (m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f))) \circ \alpha^{-1}\rfloor \\
&= \quad \text{(isomorphism)} \\
&\quad \lfloor m_M \circ (f \otimes \lceil \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor \rceil) \circ \alpha^{-1}\rfloor \\
&= \quad \text{(naturality of } \lceil -\rceil\text{)} \\
&\quad \lfloor m_M \circ (f \otimes (\lceil M^{A^*}\rceil \circ (\lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor \otimes A^*))) \\
&\qquad \circ \alpha^{-1}\rfloor \\
&= \quad \text{(bifunctor } \otimes \text{ \& naturality of } \alpha^{-1}\text{)} \\
&\quad \lfloor m_M \circ (f \otimes \lceil M^{A^*}\rceil) \circ \alpha^{-1} \\
&\qquad \circ ((A \otimes \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor) \otimes A^*)\rfloor \\
&= \quad \text{(naturality } \lfloor -\rfloor\text{)} \\
&\quad \lfloor m_M \circ (f \otimes \lceil M^{A^*}\rceil) \circ \alpha^{-1}\rfloor \\
&\qquad \circ (A \otimes \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor)
\end{aligned}
$$

We have $\lfloor \mathit{free}\, f \circ m_{A^*}\rfloor = \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor$, and thus $\lceil\lfloor \mathit{free}\, f \circ m_{A^*}\rfloor\rceil = \lceil\lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)\rfloor\rceil$. We remove $\lceil\lfloor -\rfloor\rceil$ to get $\mathit{free}\, f \circ m_{A^*} = m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f)$. □

The last condition is that $\mathit{free}\, f$ must be the unique morphism with the above properties. We show this by assuming another morphism h with these properties exists and then prove that it must be equal to $\mathit{free}\, f$.

$$h \circ \mathit{ins} = f$$
$$h \circ e_{A^*} = e_M$$
$$h \circ m_{A^*} = m_M \circ (h \otimes h)$$

Proof. We prove that h is an algebra homomorphism from $\langle A^*, \mathit{in}\rangle$ to $\langle M, [e_M, m_M \circ (f \otimes M)]\rangle$. We prove the in_1 and in_2 property:

$$
\begin{aligned}
&\quad h \circ \mathit{in}_1 \\
&= \quad \text{(definition } e_{A^*}\text{)} \\
&\quad h \circ e_{A^*} \\
&= \quad \text{(assumed property of } h\text{)} \\
&\quad e_M
\end{aligned}
$$

$$
\begin{aligned}
&\quad h \circ \mathit{in}_2 \\
&= \quad \text{(property } \mathit{in}_2 \text{ and } \mathit{ins}\text{, see appendix)} \\
&\quad h \circ m_{A^*} \circ (\mathit{ins} \otimes A^*) \\
&= \quad \text{(assumed property of } h\text{)} \\
&\quad m_M \circ (h \otimes h) \circ (\mathit{ins} \otimes A^*) \\
&= \quad \text{(bifunctor } \otimes \text{ \& assumed property of } h\text{)} \\
&\quad m_M \circ (f \otimes h) \\
&= \quad \text{(bifunctor } \otimes\text{)} \\
&\quad m_M \circ (f \otimes M) \circ (A \otimes h)
\end{aligned}
$$

Since $\mathit{free}\, f$ is the unique algebra homomorphism to the stated algebra by property of $(\!|-|\!)$, h and $\mathit{free}\, f$ must be equal. □

Example 3.6 (Free Monoids in Haskell)**.** Because we can construct free monoids from initial algebras, we can define them in Haskell as inductive datatypes. The definitions of the three tensors of interest are:

```
data Day f g x where
  Day :: forall y z.
    f y -> g z -> ((y, z) -> x) -> Day f g x
data PCom p q x y where
  PCom :: forall z.
    p x z -> q z y -> PCom p q x y
data Compose f g x =
  Compose {unCompose :: f (g x)}
```

Then the definitions of the free applicative, arrow and monad are respectively:

```
data FreeAp f x =
  Pure x | Rec (Day f (FreeAp f) x)
data FreeAr a x y =
  Hom (x -> y) | Comp ((PCom a (FreeAr a)) x y)
data Free f x =
  Ret x | Con (Compose f (Free f) x)
```

These definitions are isomorphic to the ones presented by Rivas and Jaskelioff [12]. Yet, in order to highlight the similarity between the different free monoids and the functions defined on them, we have not inlined the tensors.

Since they are the carriers of initial algebras, the inductive datatypes come with a $(\!|-|\!)$ morphism.

```
cataAp :: (Functor x)
       => (Id ~> x) -> (Day a x ~> x)
       -> (FreeAp a ~> x)
cataAr :: (Profunctor x)
```

```
        => ((->) ~~> x) -> (PCom f x ~~> x)
        -> (FreeAr a ~~> x)
cata    :: (Functor x)
        => (Id ~> x) -> (Compose a x ~> x)
        -> (Free a ~> x)
```

The morphism *free f* then corresponds to the following functions for each of these.

```
freeAp :: (Applicative m)
       => (a ~> m) -> (FreeAp a ~> m)
freeAr :: (WeakArrow m)
       => (a ~~> m) -> (FreeAr a ~~> m)
free   :: (Monad m)
       => (a ~> m) -> (Free a ~> m)
```

3.3 Expressiveness of Free and $(\!|-|\!)$

Since we have shown that *free* can be expressed in terms of $(\!|-|\!)$, we might expect that $(\!|-|\!)$ is a more expressive handling construct than *free*. This section shows that this is not the case by expressing $(\!|-|\!)$ in terms of *free*.

Monoid of Endomorphisms We need to take a detour and mention that the exponential of an object on itself X^X induces a monoid, called the *monoid of endomorphisms*, with the following unit and multiplication morphisms:

$$e_{X^X} = \lfloor \lambda_X \rfloor \tag{18}$$
$$m_{X^X} = \lfloor ev_X \circ (X^X \otimes ev_X) \circ \alpha^{-1} \rfloor \tag{19}$$

Free Monoid Induces Initial Algebra We construct an F_A-algebra with A^* as carrier and an action *in* defined as follows:

$$in_1 = e_{A^*} \tag{20}$$
$$in_2 = m_{A^*} \circ (ins \otimes A^*) \tag{21}$$
$$in = [in_1 \circ inl, in_2 \circ inr] \tag{22}$$

Now we show that $\langle A^*, in : F_A A^* \to A^* \rangle$ acts as the initial algebra. This means that given another F_A-algebra $\langle X, \phi : F_A X \to X \rangle$, there must exist a unique algebra homomorphism $(\!|\phi|\!) : A^* \to X$. We define $(\!|\phi|\!)$ as follows:

$$(\!|\phi|\!) = ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ free\ \lfloor \phi_2 \rfloor \tag{23}$$

where $[\phi_1 = \phi \circ inl, \phi_2 = \phi \circ inr] = \phi$. Note that $free\ \lfloor \phi_2 \rfloor : A^* \to X^X$ is valid since (X^X, e_{X^X}, m_{X^X}) is a monoid.

We show that $(\!|\phi|\!)$ is an algebra homomorphism separately for in_1 and in_2. First for in_1:

$$(\!|\phi|\!) \circ in_1 = \phi_1$$

Proof.

$(\!|\phi|\!) \circ in_1$
= (def. of in_1 and $(\!|\phi|\!)$)
$ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ free\ \lfloor \phi_2 \rfloor \circ e_{A^*}$
= (*free f* monoid homomorphism, 3)
$ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ e_{X^X}$
= (ρ^{-1} natural transformation)
$ev_X \circ (X^X \otimes \phi_1) \circ (e_{X^X} \otimes I) \circ \rho_I^{-1}$
= ($\otimes$ bifunctor)
$ev_X \circ (e_{X^X} \otimes X) \circ (I \otimes \phi_1) \circ \rho_I^{-1}$
= (def. of ev_X and e_{X^X})
$\lceil X^X \rceil \circ (\lfloor \lambda_X \rfloor \otimes X) \circ (I \otimes \phi_1) \circ \rho_I^{-1}$
= (naturality of $\lceil - \rceil$ & isomorphism)
$\lambda_X \circ (I \otimes \phi_1) \circ \rho_I^{-1}$
= (λ natural transformation)
$\phi_1 \circ \lambda_I \circ \rho_I^{-1}$
= (def. of monoidal category, 3.1)
$\phi_1 \circ \rho_I \circ \rho_I^{-1}$
= (isomorphism)
ϕ_1 □

Next for in_2:

$$(\!|\phi|\!) \circ in_2 = \phi_2 \circ (A \otimes (\!|\phi|\!))$$

Proof.

$(\!|\phi|\!) \circ in_2$
= (def. of in_2 and $(\!|\phi|\!)$)
$ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ free\ \lfloor \phi_2 \rfloor \circ m_{A^*} \circ (ins \otimes A^*)$
= (*free f* monoid homomorphism)
$ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ m_{X^X} \circ$
$(free\ \lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor) \circ (ins \otimes A^*)$
= ($\otimes$ bifunctor & *free f* $\circ$ *ins* = *f*)
$ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ m_{X^X} \circ (\lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor)$
= (ρ^{-1} natural transformation)
$ev_X \circ (X^X \otimes \phi_1) \circ (m_{X^X} \otimes I) \circ \rho^{-1}_{X^X \otimes X^X}$
$\circ (\lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor)$
= ($\otimes$ bifunctor)
$ev_X \circ (m_{X^X} \otimes X) \circ ((X^X \otimes X^X) \otimes \phi_1) \circ \rho^{-1}_{X^X \otimes X^X} \circ$
$(\lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor)$
= (def. ev_X and m_{X^X} & naturality $\lceil - \rceil$ & isomorphism)
$ev_X \circ (X^X \otimes ev_X) \circ \alpha^{-1} \circ ((X^X \otimes X^X) \otimes \phi_1) \circ \rho^{-1}_{X^X \otimes X^X}$
$\circ (\lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor)$
= (α^{-1} natural transformation)
$ev_X \circ (X^X \otimes ev_X) \circ (X^X \otimes (X^X \otimes \phi_1)) \circ \alpha^{-1} \circ \rho^{-1}_{X^X \otimes X^X}$
$\circ (\lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor)$
= (property $\alpha^{-1} \circ \rho^{-1} = id \otimes \rho^{-1}$)
$ev_X \circ (X^X \otimes ev_X) \circ (X^X \otimes (X^X \otimes \phi_1)) \circ (X^X \otimes \rho^{-1}_{X^X})$
$\circ (\lfloor \phi_2 \rfloor \otimes free\ \lfloor \phi_2 \rfloor)$
= ($\otimes$ bifunctor)
$ev_X \circ (X^X \otimes ev_X \circ (X^X \otimes \phi_1) \circ \rho^{-1}_{X^X} \circ free\ \lfloor \phi_2 \rfloor)$
$\circ (\lfloor \phi_2 \rfloor \otimes A^*)$
= (def. of $(\!|\phi|\!)$)
$ev_X \circ (X^X \otimes (\!|\phi|\!)) \circ (\lfloor \phi_2 \rfloor \otimes A^*)$
= ($\otimes$ bifunctor)

$$
\begin{aligned}
& ev_X \circ (\lfloor \phi_2 \rfloor \otimes X^X) \circ (A \otimes (\!|\phi|\!)) \\
= \quad & \text{(def. of } ev_X\text{)} \\
& \lceil X^X \rceil \circ (\lfloor \phi_2 \rfloor \otimes X^X) \circ (A \otimes (\!|\phi|\!)) \\
= \quad & \text{(naturality } \lceil - \rceil \text{ \& isomorphism)} \\
& \phi_2 \circ (A \otimes (\!|\phi|\!)) \quad \square
\end{aligned}
$$

Finally we have to show that $(\!|\phi|\!)$ is unique.

Proof. This proof is left as future work □

Choice of Syntax Because we have shown that the initial algebra of F_A and the free monoid induce each other, we can conclude that we have a choice of syntax for a language with monoidal effects. We can either choose to use the computation introduction rules given by e_{A^*}, m_{A^*} and *ins* or we can choose to use the inductive in_1 and in_2. The same follows for our handling construct: we can choose *free* or $(\!|-|\!)$. In fact, for maximum flexibility, the language can leave the choice up to the end user.

3.4 Handlers in Monoidal Setting

This section will expand on handlers in the monoidal setting and give some additional explanation. Table 1 gives an overview of the different handler approaches. The leftmost column presents the free algebra setting from Section 2.2. The middle and rightmost column give an overview of the free monoid setting in a similar layout for both the *free* and $(\!|-|\!)$ handling construct.

Creating Computations Creating computations in the free algebra setting is done by embedding pure values as a computation (var_A) or applying a signature on top of another computation (com_A). In the free monoid setting we have similar operations, we can create a pure computation by using in_1 or apply a signature on top of another computation with in_2.

A very important difference is the category in which these morphisms are defined. In the free algebra setting these are in a base category $\mathscr{C}$, which we take to mean the base category of our programming language. The choice of category in the free monoid setting isn't one specific category, it can be any monoidal category. The choice is dependent on what type of computations we want to represent. Of course to obtain something meaningful the category will have to have some link with $\mathscr{C}$, e.g. endofunctors on $\mathscr{C}$ or profunctors from $\mathscr{C}^{op}$ to $\mathscr{C}$.

This difference in category means that the result of the handling construct will also be limited to results in that category. We can work around this limitation by using the Continuation Monad[3]. Investigating the use of this technique to achieve an interface in the base category $\mathscr{C}$ is future work.

[3] https://ncatlab.org/nlab/show/continuation+monad

Handling Computations Handling computations is the composition of the handling construct on a computation. This is similar in both settings since this was the aim of this theory. The result of the computation with the given handler will be determined by the handling rules. The rules in Table 1 are given by the inductive syntax since that is likely to be a more familiar setting.

Operations and Signatures Signatures are a vague notion at this level of generality. It involves a coproduct to distinguish the different possible operations. But other than that the extra structure seems dependent on the chosen monoidal category.

We can give the structure of signatures we consider for two categories of interest. For the category of endofunctors we utilize the same signatures as in the free algebra setting, $P_i \times -^{N_i}$. For profunctors we extend the signature to $P_i \times (Q_i \times -)^{N_i \times -}$, where the covariant position is next to Q_i and the contravariant position is next to N_i.

3.5 Handler Constructs in Eff-like Syntax

Let's take a look at how we can apply this as constructs similar to Eff handler constructs, used as an example for the free algebra approach earlier. We will be applying the formula of Table 1 to create the handler syntax and corresponding handling rules. In the handler syntax we will denote types as `x:`T, meaning `x` has type T. Note that due to the categories we are working with we will have functor types FA and profunctor types $F(X, A)$. We will denote natural transformations with the special arrows `~>`$_A$ and natural transformations contravariant in X and covariant in A as `~~>`$_{X,A}$.

To create the handler construct we deconstruct the action $\phi = [\phi_1, \phi_2]$ of the F_A-algebra in the following components:

- a value clause $I \to F$, this is ϕ_1
- ϕ_2 is of the form $A \otimes F \to F$. Since A is a signature consisting of a coproduct $\Sigma_0 + \ldots + \Sigma_n$, we deconstruct it into n signature clauses of the form $\phi_{2i} : \Sigma_i \otimes F \to F$. This requires that we have a distributivity property, which can be characterized by an isomorphism $(\Sigma_0 + \ldots + \Sigma_n) \otimes F \cong \Sigma_0 \otimes F + \ldots \Sigma_n \otimes F$.

Monad For endofunctors our value clause is a natural transformation $Id \to F$, this is equivalent to a natural transformation $A \to FA$ natural in A. For monads our monoidal tensor is functor composition $\circ$. This results in a signature clause of the form $P \times -^{N_i} \circ F \to F$, equivalent to a natural transformation $P \times (FA)^{N_i} \to FA$ natural in A. We interpret the exponential in the base

Table 1. Overview of Handler Approaches

	Free Algebra	Free Monoid (*free*)	Free Monoid ($(\!|-|\!)$)
syntax /computation	$var_A : A \to \Sigma^* A$ $com_A : \Sigma(\Sigma^* A) \to \Sigma^* A$	$in_1 : I \to \Sigma^*$ (20) $in_2 : \Sigma \otimes \Sigma^* \to \Sigma^*$ (21)	$in_1 : I \to \Sigma^*$ $in_2 : \Sigma \otimes \Sigma^* \to \Sigma^*$
handler	Σ-algebra: $\langle B, b : \Sigma B \to B \rangle$ & morphism $f : A \to B$	monoid: (M, e_M, m_M) & morphism $f : \Sigma \to M$	F_Σ-algebra: $\langle F, \phi : I + \Sigma \otimes F \to F \rangle$
handling a computation	$handle_b\ f : \Sigma^* A \to B$	$free\ f : \Sigma^* \to M$	$(\!|\phi|\!) : \Sigma^* \to F$
handling rules	$handle_b\ f \circ var_A = f$ $handle_b\ f \circ com_A = b \circ \Sigma(handle_b\ f)$	$free\ f \circ in_1 = e_M$ $free\ f \circ in_2 = m_M \circ (f \otimes free\ f)$	$(\!|\phi|\!) \circ in_1 = \phi_1$ $(\!|\phi|\!) \circ in_2 = \phi_2 \circ (\Sigma \otimes (\!|\phi|\!))$

category $\mathscr{C}$ as a function type. The above observations result in the following handler syntax:

```
handler
| val x:A ~>_A c_v:FA
| op_i (p_i:P) (k_i:N_i→FA) ~>_A c_i:FA
```

The resulting handling rules are the following (`with h` is omitted in `handle _ with h`):

```
handle (return x) = c_v x
handle (op_i p k) = c_i p (\n -> handle (k n))
```

Applicative In this case we have the same value clause as for monads. The monoidal operator is now the Day convolution $\star$. The signature clause is thus of the form $P \times -^{N_i} \star F \to F$, equivalent to a natural transformation $(\int^{Y,Z} P \times Y^{N_i} \times FZ \times (Y \times Z \to A)) \to FA$ natural in A. We interpret the coend $\int^{Y,Z}$ as an existential quantifier on types Y and Z. The above observations result in the following handler syntax:

```
handler
| val x:A ~>_A c_v:FA
| op_i (p_i:P) (y_i:N_i→Y) (k_i:FZ) (f_i:Y×Z→A)
    ~>_A c_i:FA
```

The resulting handling rules are the following (`with h` is omitted in `handle _ with h`):

```
handle (return x) = c_v x
handle (op_i p y k f) = c_i p y (handle k) f
```

Example 3.7. Imagine a `get` and `put` signature defined as $(I \times -^S) + (S \times -^I)$. Where `get` is $I \times -^S$ and `put` is $S \times -^I$, S is the type of our state value.

We can create a computation where we first get our state and then put the same value we got as result from the get computation, which is inherently a monadic effect.

```
get () (\s -> put s (\_ -> return s))
```

We cannot recreate the same computation with the applicative interface, but we can create a computation which gets the state and eventually returns it after putting a constant `C` in the state.

```
get () (\s -> s) (
  put C (\_ -> ()) (return ()) (\(_, _) -> ())
) (\(s, _) -> s)
```

The above computation is equivalent to the following computation specified with a **do**-notation for applicatives [8].

```
do
  s <- get ()
  put C
  return s
```

We can express static analysis by interpreting to the constant functor Δ_c, where c is the type of the result of our analysis. With the interface for applicative handlers we can create the following handler which counts the amount of put operations:

```
handler
| val x ~> Δ_ℕ 0
| get p y k f ~> k
| put p y k f ~> k +⁴ 1
```

The reason why we have to use the constant functor and can't use the identity functor is because we need to be able to provide a natural transformation which is natural in A for the different operations. When using the identity functor we are only able to provide a result where A is $\mathbb{N}$.

For reference this is the canonical interpreter for running the computation.

```
handler
| val x ~> \s -> (x, s)
| get p y k f ~> \s ->
   let (z, s') = k s in (f (y s) z, s')
| put p y k f ~> \s ->
   let (z, s') = k p in (f (y ()) z, s')
```

Arrow In the arrow case our value clause is a natural transformation $Hom(X, A) \to F(X, A)$ natural in X (contravariantly) and A (covariantly). We interpret

[4] increments the value 'inside' the $\Delta_\mathbb{N}$ functor

the $Hom(X, A)$ profunctor as functions from X to A in our language. The monoidal operator is profunctor composition $\otimes$. This results in a signature clause of $(P \times (Q \times -)^{N_i \times -}) \otimes F(-,-) \to F(-,-)$. This is equivalent to a natural transformation $\int^Z (P \times (Q \times Z)^{N_i \times X} \times F(Z, A)) \to F(X, A)$. The above observations result in the following handler syntax:

```
handler
| val f:X→A ~~>_{X,A} c_v:F(X, A)
| op_i (p_i:P) (f:(N_i×X)→(Q×Z)) (k_i:F(Z, A))
    ~~>_{X,A} c_i:F(X, A)
```

The resulting handling rules are the following (`with h` is omitted in `handle _ with h`):

```
handle (return f) = c_v f
handle (op_i p f k) = c_i p f (handle k)
```

Arrow - $Kleisli(I, -)$ Since the above arrow handler construct is fairly different from the monad and applicative construct, we also illustrate the following simplified but more similar construct. We interpret the computation to the $Kleisli$ profunctor, but with the input parameter set to I. This $Kleisli(I, -) = I \to F-$ functor is isomorphic to $F-$. Simplifying the previous handler construct by doing substitutions and replacing isomorphic signatures gives us:

```
handler
| val f:A ~>_A c_v:FA
| op_i (p_i:P) (f_i:N_i→Q×Z) (k_i:Z→FA)
    ~>_A c_i:FA
```

The resulting handling rules are the following (`with h` is omitted in `handle _ with h`):

```
handle (return x) = c_v x
handle (op_i p f k) = c_i p f (\z -> handle (k z))
```

4 Handler Hierarchy

The free monoid based handlers allows to model different kinds of computation depending on the chosen monoidal category. This allows us to express monadic handlers which allow expressive computations and applicative handlers which allow to express static analysis, but only applies to less expressive computations. Since the monadic handler is able to express less (it is unable to do static analysis) and is able to handle more expressive computations, it seems like there should be a way of reusing this handler for less expressive computations such as applicatives or arrows. This section expands on this idea by leveraging the theory of monoidal functors.

4.1 Monoidal Functors

A monoidal functor allows us to move between monoidal categories while preserving the monoidal structure.[5]

Definition 4.1 (Lax monoidal functor)**.** Let $(\mathscr{C}, \otimes_{\mathscr{C}}, I_{\mathscr{C}}, \alpha^{\mathscr{C}}, \lambda^{\mathscr{C}}, \rho^{\mathscr{C}})$ and $(\mathscr{D}, \otimes_{\mathscr{D}}, I_{\mathscr{D}}, \alpha^{\mathscr{D}}, \lambda^{\mathscr{D}}, \rho^{\mathscr{D}})$ be two monoidal categories. A lax monoidal functor between them is

a) a functor $F : \mathscr{C} \to \mathscr{D}$
b) a morphism $\epsilon : I_{\mathscr{D}} \to F(I_{\mathscr{C}})$
c) a natural transformation $\mu_{A,B} : FA \otimes_{\mathscr{D}} FB \to F(A \otimes_{\mathscr{C}} B)$

with the following commuting diagrams:

$$\begin{array}{ccc} FA \otimes_{\mathscr{D}} (FB \otimes_{\mathscr{D}} FC) & \xrightarrow{\alpha^{\mathscr{D}}} & (FA \otimes_{\mathscr{D}} FB) \otimes_{\mathscr{D}} FC \\ {\scriptstyle FA \otimes_{\mathscr{D}} \mu_{B,C}}\downarrow & & \downarrow{\scriptstyle \mu_{A,B} \otimes_{\mathscr{D}} FC} \\ FA \otimes_{\mathscr{D}} F(B \otimes_{\mathscr{C}} C) & & F(A \otimes_{\mathscr{C}} B) \otimes_{\mathscr{D}} FC \\ {\scriptstyle \mu_{A,(B \otimes_{\mathscr{C}} C)}}\downarrow & & \downarrow{\scriptstyle \mu_{A \otimes_{\mathscr{C}} B,C}} \\ F(A \otimes_{\mathscr{C}} (B \otimes_{\mathscr{C}} C)) & \xrightarrow{F(\alpha^{\mathscr{C}})} & F((A \otimes_{\mathscr{C}} B) \otimes_{\mathscr{C}} C) \end{array}$$

$$\begin{array}{ccc} I_{\mathscr{D}} \otimes_{\mathscr{D}} FA & \xrightarrow{\epsilon \otimes_{\mathscr{D}} FA} & F(I_{\mathscr{C}}) \otimes_{\mathscr{D}} FA \\ {\scriptstyle \lambda^{\mathscr{D}}_{FA}}\downarrow & & \downarrow{\scriptstyle \mu_{I_{\mathscr{C}},A}} \\ FA & \xleftarrow{F(\lambda^{\mathscr{C}}_A)} & F(I_{\mathscr{C}} \otimes_{\mathscr{C}} A) \end{array}$$

$$\begin{array}{ccc} FA \otimes_{\mathscr{D}} I_{\mathscr{D}} & \xrightarrow{FA \otimes_{\mathscr{D}} \epsilon} & FA \otimes_{\mathscr{D}} F(I_{\mathscr{C}}) \\ {\scriptstyle \rho^{\mathscr{D}}_{FA}}\downarrow & & \downarrow{\scriptstyle \mu_{A,I_{\mathscr{C}}}} \\ FA & \xleftarrow{F(\rho^{\mathscr{C}}_A)} & F(A \otimes_{\mathscr{C}} I_{\mathscr{C}}) \end{array}$$

Now the interesting part about the monoidal functor is that given a monoid in $\mathscr{C}$, eg. (A, e_A, m_A), we can create a monoid FA in $\mathscr{D}$. The unit and multiplication of this monoid are defined as follows:

$$e_{FA} = I_{\mathscr{D}} \xrightarrow{\epsilon} F(I_{\mathscr{C}}) \xrightarrow{F(e_A)} FA$$

$$m_{FA} = FA \otimes_{\mathscr{D}} FA \xrightarrow{\mu_{x,y}} F(A \otimes_{\mathscr{C}} A) \xrightarrow{F(m_A)} FA$$

4.2 Conversion of Handlers

To create a conversion of handlers we assume there is a left adjoint to the lax monoidal functor F, let's call it G. This adjunction can be given in terms of an isomorphism $(-)_\ell : \mathscr{C}(GA, B) \cong \mathscr{D}(A, FB) : (-)_r$ natural in A and B.

With this adjunction in place we can describe the conversion in more detail. We will take a handler $h : (GA)^* \to X$ in $\mathscr{C}$ and transform it to a new handler $h' : A^* \to FX$ in $\mathscr{D}$.

We define the following operation

$$convert = (free_{\mathscr{D}}\ (ins_{\mathscr{C}})_\ell)_r : G(A)^* \to (GA)^*$$

[5] https://ncatlab.org/nlab/show/monoidal+functor

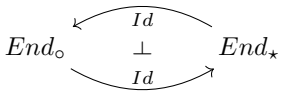
by instantiating $ins_{\mathscr{C}}$ as $GA \to (GA)^*$, and so $(ins_{\mathscr{C}})_\ell$ has the form $A \to F(GA)^*$. $F(GA)^*$ is the object-component of a monoid in $\mathscr{D}$ since F is a monoidal functor. Then we use $free_{\mathscr{D}}$ to obtain a morphism of the form $A^* \to F(GA)*$. As last step $(-)_r$ is used to obtain *convert*.

Then our new handler h' is defined as:

$$h' = (h \circ convert)_\ell : A^* \to FX$$

Example 4.2 (Adjunctions and Handler Conversion in Haskell)**.** The use of the adjunction is based on the following adjunctions related to our monoidal categories. Below we give the Haskell instantiation of these adjunctions and the $(-)_\ell$ and $(-)_r$ isomorphisms.

The adjunction between the identity functor $Id : End \to End$ monoidal functor and itself.

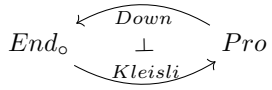



```
data Id1 f x = Id1 { unId1 :: f x }

leftAdjId1 :: (Id1 f ~> g) -> (f ~> Id1 g)
leftAdjId1 h v = Id1 (h (Id1 v))

rightAdjId1 :: (f ~> Id1 g) -> (Id1 f ~> g)
rightAdjId1 h (Id1 v) = unId1 (h v)
```

The adjunction between the *Kleisli* monoidal functor and the *Down* functor.

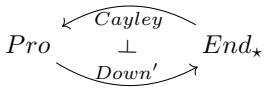



```
data Kleisli m a b = Kleisli
  { unKleisli :: a -> m b }
data Down f x = Down
  { unDown :: f () x }

leftAdjKleisli :: (Profunctor p) => (Down p ~> f)
               -> (p ~~> Kleisli f)
leftAdjKleisli h v = Kleisli (\x ->
  h (Down (dimap (const x) id v)))

rightAdjKleisli :: (p ~~> Kleisli f)
                -> (Down p ~> f)
rightAdjKleisli h (Down v) = unKleisli (h v) ()
```

The adjunction between the $Down'$ monoidal functor and the *Cayley* functor.




```
data Down' f x = Down'
  { unDown' :: forall a. f a (a, x) }
data Cayley f a x = Cayley
  { unCayley :: f (a -> x) }

leftAdjDown' :: (Functor a, StrongProfunctor b)
             => (Cayley a ~~> b) -> (a ~> Down' b)
leftAdjDown' h v = Down' (dimap ((),) swap f)
  where f = first (h (Cayley (fmap const v)))

rightAdjDown' :: (Profunctor b)
              => (a ~> Down' b) -> (Cayley a ~~> b)
rightAdjDown' h (Cayley v) =
  dimap id (\(x, f) -> f x) (unDown' (h v))
```

Following the pattern from 4.2 with each of these, we obtain the following handler conversions, the implementation is the instantiation of the general theory:

```
-- use monad handler on applicative computation
(Functor f) => (Free (Id1 f) ~> g)
-> (FreeAp f ~> Id1 g)
-- use arrow handler on applicative computation
(Functor f, StrongProfunctor g)
=> (FreeAr (Cayley f) ~~> g) -> (FreeAp f ~> Down' g)
-- use monad handler on arrow computation
(Profunctor f) => (Free (Down f) ~> g)
-> (FreeAr f ~~> Kleisli g)
```

Example 4.3 (Conversion of Signatures)**.** In the above examples we can see for the monad handler and applicative computation case the `Id1` adjunction gives us a very clean function equivalent to `(Functor f) => (Free f ~> g) -> (FreeAp f ~> g)`. This however isn't the case for the arrow computations, we have the additional functor component present in the signature.

To give an idea what these converted signatures mean in practice we explore a `Get/Put` signature. It is interesting enough to be a meaningful example while still a relatively simple signature.

If we define a `GetPut` signature as follows:

```
data GetPut s o = Get (s -> o) | Put s o
```

Then `Cayley GetPut` is isomorphic to the following signature:

```
data GetPutCayley s i o
  = GetCayley ((s, i) -> o)
  | PutCayley s (i -> o)
```

We see that the above is an arrow signature, but it is not the signature we would use if we wanted to employ

the full capabilities of arrow computations. With the above signature it is not possible to express that the value of the state we are putting is dependent on the result of previous operations (captured in `i`), at least with the profunctor composition as monoidal operator.

The following `GetPutP` signature does allow us to express this capability.

```
data GetPutP s i o
  = GetP ((s, i) -> o)
  | PutP (i -> (s, o))
```

Both `Down GetPutP` and `Down GetPutCayley` are isomorphic to `GetPut`.

Note that it is possible to create this one-way conversion. While the other direction is not possible.

```
conversion :: GetPutCayley s ~~> GetPutP s
conversion (GetCayley f) = GetP f
conversion (PutCayley s f) = PutP (\x -> (s, f x))
```

This conversion allows us to create a handler `FreeAr (Cayley GetPut) ~~> x` from a handler `FreeAr GetPutP ~~> x`.

Obviously from the isomorphic signatures we can create equivalent conversions as well.

5 Related Work

Algebraic Effects and Handlers There has been a lot of work in the area of algebraic effects and handlers such as by Plotkin and Pretnar [11], Bauer and Pretnar [1] and Lindley et al. [6]. These works have focused mostly on monadic effects. This is due to the fact that the free algebra adjunction induces a monad on the base category $\mathscr{C}$. This work expands on algebraic effects and handlers by deriving handler constructs from a similar idea, but in a setting of free monoids. The theory of free monoids can be instantiated with the various kinds of effects to achieve handlers for monads, arrows and applicatives.

Handlers for Idioms and Arrows Lindley [5] introduces the calculus λ_{flow} which has handler constructs for monadic, applicative and arrow computations. It is a concrete calculus given in terms of typing rules and operational semantics where each of the different computation types (monad, applicative and arrow) have separate handling constructs. We approach the same idea from a category theoretic perspective.

6 Conclusion

This work focuses on the categorical ideas of monoids in monoidal categories to define handlers for effects in these categories. Of notable interest are the monadic, applicative and arrow effects. We define the handler construct as the unique morphism from the free monoid in a similar manner as the handler for free algebras, namely the unique algebra homomorphism. Due to the fact that the initial algebra of F_A induces the free monoid and vice versa, we have a choice in interface of our handling construct. We introduce some possible handler constructs with an Eff-like syntax to show how this theory might be used.

We expand on monoidal functors in relation to handlers defined for the free monoid. We define a conversion of handlers, which allows us to reuse handlers for different types of computations. This is based on the observation of various monoidal adjunctions.

Acknowledgements

This work was partially supported by project GRACeFUL, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 640954.

References

[1] Andrej Bauer and Matija Pretnar. 2012. Programming with Algebraic Effects and Handlers. *CoRR* abs/1203.1539 (2012). http://arxiv.org/abs/1203.1539

[2] Jeremy Gibbons. 2016. Free delivery (functional pearl). In *Proceedings of the 9th International Symposium on Haskell*. ACM, 45–50.

[3] Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. 2015. The Remote Monad Design Pattern. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/2804302.2804311

[4] John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000).

[5] Sam Lindley. 2014. Algebraic Effects and Effect Handlers for Idioms and Arrows. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2633628.2633636

[6] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

[7] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 325–337. https://doi.org/10.1145/2628136.2628144

[8] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's Do-notation into Applicative Operations. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 92–104. https://doi.org/10.1145/2976002.2976007

[9] Conor Mcbride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008),

1–13.
[10] E. Moggi. 1989. *An abstract view of programming languages.* Technical Report ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.
[11] Gordon Plotkin and Matija Pretnar. 2009. *Handlers of Algebraic Effects.* Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7
[12] Exequiel Rivas and Mauro Jaskelioff. 2014. Notions of Computation as Monoids. *CoRR* abs/1406.4823 (2014). http://arxiv.org/abs/1406.4823
[13] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90).* ACM, New York, NY, USA, 61–78.

A property $in_2 = m_{A^*} \circ (ins \otimes A^*)$

Proof.

$m_{A^*} \circ (ins \otimes A^*)$
$=$ (definitions, 10 and 14, $b = [\ldots,\ldots]$)
$\lceil (\!|b|\!) \rceil \circ ((in_2 \circ (A \otimes in_1) \circ \rho_A^{-1}) \otimes A^*)$
$=$ (naturality of $\lceil - \rceil$)
$\lceil (\!|b|\!) \circ in_2 \circ (A \otimes in_1) \circ \rho_A^{-1} \rceil$
$=$ (6 & bifunctor)
$\lceil \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \rfloor \circ (A \otimes (\!|b|\!) \circ in_1) \circ \rho_A^{-1} \rceil$
$=$ (5)
$\lceil \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \rfloor \circ (A \otimes \lfloor \lambda_{A^*} \rfloor) \circ \rho_A^{-1} \rceil$
$=$ (naturality of $\lfloor - \rfloor$)
$\lceil \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ \alpha^{-1} \circ ((A \otimes \lfloor \lambda_{A^*} \rfloor) \otimes A^*) \rfloor \circ \rho_A^{-1} \rceil$
$=$ (α^{-1} is a natural transformation)
$\lceil \lfloor in_2 \circ (A \otimes ev_{A^*}) \circ (A \otimes (\lfloor \lambda_{A^*} \rfloor \otimes A^*)) \circ \alpha^{-1} \rfloor \circ \rho_A^{-1} \rceil$
$=$ ($\otimes$ bifunctor)
$\lceil \lfloor in_2 \circ (A \otimes ev_{A^*} \circ (\lfloor \lambda_{A^*} \rfloor \otimes A^*)) \circ \alpha^{-1} \rfloor \circ \rho_A^{-1} \rceil$
$=$ (definition ev_{A^*})
$\lceil \lfloor in_2 \circ (A \otimes \lceil A^{*^{A^*}} \rceil \circ (\lfloor \lambda_{A^*} \rfloor \otimes A^*)) \circ \alpha^{-1} \rfloor \circ \rho_A^{-1} \rceil$
$=$ (naturality of $\lceil - \rceil$)
$\lceil \lfloor in_2 \circ (A \otimes \lceil \lfloor \lambda_{A^*} \rfloor \rceil) \circ \alpha^{-1} \rfloor \circ \rho_A^{-1} \rceil$
$=$ (isomorphism)
$\lceil \lfloor in_2 \circ (A \otimes \lambda_{A^*}) \circ \alpha^{-1} \rfloor \circ \rho_A^{-1} \rceil$
$=$ (naturality of $\lfloor - \rfloor$)
$\lceil \lfloor in_2 \circ (A \otimes \lambda_{A^*}) \circ \alpha^{-1} \circ (\rho_A^{-1} \otimes A^*) \rfloor \rceil$
$=$ (definition monoidal category, 3.1)
$\lceil \lfloor in_2 \rfloor \rceil$
$=$ (isomorphism)
in_2 □

Purely Functional Federated Learning in Erlang

Work-in-progress paper / extended abstract

Gregor Ulm
Fraunhofer-Chalmers Research
Centre for Industrial Mathematics
Gothenburg, Sweden
gregor.ulm@fcc.chalmers.se

Emil Gustavsson
Fraunhofer-Chalmers Research
Centre for Industrial Mathematics
Gothenburg, Sweden
emil.gustavsson@fcc.chalmers.se

Mats Jirstrand
Fraunhofer-Chalmers Research
Centre for Industrial Mathematics
Gothenburg, Sweden
mats.jirstrand@fcc.chalmers.se

ABSTRACT

Arguably the biggest strength of the functional programming language Erlang is how straightforward it is to implement concurrent and distributed programs with it. Numerical computing, on the other hand, is not necessarily seen as one of its strengths. The recent introduction of Federated Learning, a concept according to which edge devices are leveraged for decentralized machine learning tasks, while a central server only updates and distributes a global model, provided the motivation for exploring how well Erlang was suited to such a use case. We present a framework for Federated Learning in Erlang, written in a purely functional style, and compare two versions of it: one that has been exclusively written in Erlang, and one in which Erlang is relegated to coordinating client processes that rely on performing numerical computations in the programming language C. Initial results are promising, as we learnt that a real-world industrial use case of distributed data analytics can easily be tackled with a system purely written in Erlang. The novelty of our work is that we present the first implementation of a Federated Learning framework in a functional programming language, with the added benefit of being purely functional. In addition, we demonstrate that Erlang can not only be leveraged for message passing but also performs adequately for practical machine learning tasks.

CCS CONCEPTS

• **Computing methodologies** → **Simulation environments**; *Neural networks*; *Distributed programming languages*; *Concurrent programming languages*;

KEYWORDS

Machine learning, Federated learning, Distributed computing, Purely functional programming, Erlang

ACM Reference format:
Gregor Ulm, Emil Gustavsson, and Mats Jirstrand. 2017. Purely Functional Federated Learning in Erlang. In *Proceedings of The 29th symposium on Implementation and Application of Functional Languages, Bristol, United Kingdom, August 30 to September 1, 2017 (IFL'17),* 8 pages.
https://doi.org/

IFL'17, August 30 to September 1, 2017, Bristol, United Kingdom

ACM ISBN …$15.00
https://doi.org/

1 INTRODUCTION

With the explosion of the amount of data gathered by networked devices, more efficient approaches for distributed data processing have to be explored. The reason is that it would be infeasible to transfer all gathered data from edge devices to a central data center, process data, and transfer results back to edge devices via the network. There are several approaches to taming the amount of data received, such as filtering on edge devices, transferring only a representative sample, or performing data processing tasks decentrally. A recently introduced example of distributed data analytics, more concretely distributed machine learning, is Federated Learning [10]. The key idea behind Federated Learning is the distribution of learning tasks to a subset of available devices, performing machine learning tasks locally on data that never leave the edge device, and iteratively updating a centrally maintained model by, for instance, averaging updated models received from edge devices.

In this paper, we present an implementation of a Federated Learning framework implemented in a purely functional style in the programming language Erlang. This work was produced in the context of an industrial research project, with the goal of evaluating approaches for distributed data analysis.

Our contribution consists of the following:

- Creating an open source implementation of a framework for Federated Learning
- Highlighting the feasibility of a purely functional style for the aforementioned framework
- Creating a purely functional implementation of an artificial neural network in Erlang
- Conducting a performance comparison of a Federated Learning implementation exclusively in Erlang with one in which client processes depend on computations being executed in the programming language C
- Demonstrating the suitability of Erlang for decentralized machine learning tasks

The main motivation behind our work was the comparison of an implementation of a Federated Learning framework that was

done exclusively in Erlang with one in which Erlang orchestrates C nodes, both from a runtime and memory consumption perspective.[1]

The remainder of our paper is organized as follows. In Section 2 we introduce the problem description, including a description of Federated Learning and the motivating use case that led us to embark on our research. In Section 3 we describe our implementation in detail, including a, for now cursory, performance comparison. Section 4 gives a brief overview over related work, while Section 5 describes possible future work, before concluding our paper with Section 6.

2 PROBLEM DESCRIPTION

In this section we give a brief overview of Federated Learning (2.1) and present the mathematical foundation of Federated Stochastic Gradient Descent (2.2). We furthermore describe the motivating use case that led us to pursue this line of research (2.3).

2.1 Federated Learning

Federated Learning is a decentralized, i.e. distributed, approach to Machine Learning. The general idea is to perform machine learning tasks on a possibly very large number of edge devices, which process data that is only accessible locally, while a central server only assigns tasks, and updates a central model based on received updates. In more detail, one iteration of Federated Learning consists of the following steps:

(1) Select a subset of clients, i.e. edge devices
(2) Send the current model from the server to each client
(3) For each client, update the provided model based on local data performing iterations of a machine learning algorithm
(4) For each client, send the updated model to the server
(5) Aggregate the client models, for instance by averaging, in order to construct an improved global model

In Fig. 1, Federated Lerning is illustrated. The central server process updates a global model, illustrated by the edge colored in black. Client nodes receive the updated global model, illustrated by the red edges, update it by training it with their local data, and send the resulting new local model back to the central server process, illustrated by the blue edges. Afterwards, the central server computes an update of the global model, and the circle continues.

There are several justifications for Federated Learning. First, there is the bandwidth problem. The amounts of data generated by local devices is too large to be transferred via the network to a central server. This is particularly relevant in a big data context with data that is very high in both volume and velocity. Second, there are economic concerns. Edge devices get more and more powerful. Modern smart phones, for instance, have been compared to (old-generation) supercomputers in our pockets [2], in terms of raw computational power. Also, modern cars have a plethora of special-purpose and general-purpose computational hardware on board. It seems prudent to use as much of the available computational power as possible, instead of only tasking a powerful central server with data analytics tasks. Third, there are data privacy issues, as some jurisdictions have very strict privacy laws. Thus, transmitting data via the network in order to perform central machine learning tasks on the "cloud" is frayed with potential data privacy issues, as summarized by Chen et al. [3], while Tene et al. point out legal issues [14]. Federated Learning is thus an excellent way of sidestepping legal quagmires surrounding data privacy.

[1] At the time of writing, we cannot yet publish the source code of our implementation as part of it is still work-in-progess. We intend to do so as soon as possible. The artefacts we intend to share with the research community include a general skeleton for Federated Learning, a purely functional implementation of an artificial neural network using backpropagation, and a combination of both, i.e. a purely functional simulation of distributed machine learning following the Federated Learning paradigm.

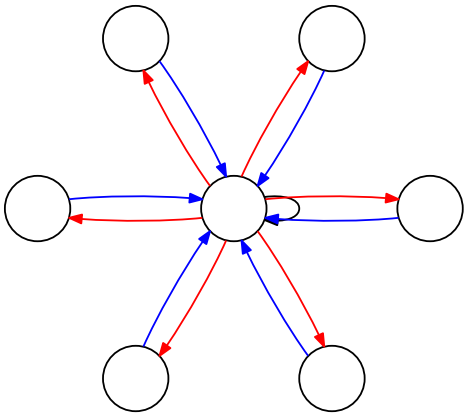


Figure 1: Illustrating Federated Learning

2.2 Federated Stochastic Gradient Descent

Federated Stochastic Gradient Descent (Federated SGD) is based on Stochastic Gradient Descent (SGD). The latter is a well-established method in the field of statistical optimization techniques. In this subsection, we therefore first introduce Stochastic Gradient Descent (2.2.1). This is followed by a presentation of Federated SGD (2.2.2) as well as Federated Averaging (2.2.3). The latter two are based on McMahan et al. [10], albeit our presentation is slightly different.

2.2.1 Stochastic Gradient Descent. The aim of Stochastic Gradient Descent is to minimize an objective function Q that is defined as the following sum:

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w) \tag{1}$$

The goal is to find a value for the parameter w that minimizes the value of Q. The i-indexed function Q in the sum represents the i-th element in the input set while training, i.e. Q_i is the value of the objective function at iteration i.

In order to minimize Q, the following operation is performed in order to update w:

$$w := w - \eta \nabla Q(w) \tag{2}$$

This means that the parameter w gets updated by computing the gradient (∇) of the previous objective function and subtracting the result from the previous parameter value w.

For a data set with i observations, i iterations of this step are performed. Thus, w is updated the following way:

$$w := w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla Q_i(w) \tag{3}$$

The learning rate η is a modifier for adjusting the learning speed. In practice, small positive values in the interval $(0, 1]$ are used. A common starting value is 0.01. A learning rate that is too large may have the opposite of the intended effect as the various updates to w may overshoot a global optimum. A learning rate that is too low, on the other hand, may get trapped in a local optimum. In our code below, η has not been taken into consideration. It is thus implicitly set to 1.

2.2.2 Federated Stochastic Gradient Descent. Federated SGD is an extension of SGD. It takes into account that there are k partitions P of the training data. Consequently, Equation 3 has to be extended the following way.

We need to consider the work performed on each client c. The objective function has to be minimized for each of the j clients.

$$f_j(w) = \frac{1}{|P_k|} \sum_{i=1}^{|P_k|} f_i(w) \tag{4}$$

In total, the objective function is thus:

$$f(w) = \frac{1}{n} \sum_{i=1}^{k} |P_k| f_j(w) \tag{5}$$

2.2.3 Federated Averaging. McMahan et al. [10] suggest averaging performed by the server based on a weighted average of the models that were received from the clients, but they point out that there may be repeated training passes over the same data on the clients, leaving details unspecified. Thus, it is implied that different clients may execute a different number of completed batch training passes. The resulting weighting on the server, however, is not affected by this.

We suggest an alternative approach for federated averaging in Section 3.2 that goes beyond the approach just outlined. It takes the quality of the updated weights that were received from client processes into account.

2.3 Motivating Use Case

The rise of *intelligent vehicles* led to increased data collection. Indeed, modern cars generate gigabytes of data every day. Considering even a moderately sized fleet of just a few thousand cars, collecting data, transferring data to a central server, processing data on a central server and afterwards sending results to each car is infeasible. Instead, our goal was to evaluate approaches for distributed data analysis. One such example is the training of a machine learning model with local data, for instance applied to a computer vision problem.

The real-world setting this relates to is one in which networked vehicles, i.e. connected cars [4], are equipped with on-board units that continuously gather data. Those on-board units are connected to a central server, possibly connected via intermediaries, so called road-side units. This is by no means a merely theoretical scenario. For instance, in recent years there have been significant large-scale experiments, such as Lee et al.'s work in South Korea in 2010 [9].

3 SOLUTION

Given the large amount of data in a realistic setting, an efficient approach for distributed machine learning had to be found. Federated Learning appeared to be a promising approach. The research prototype we are going to present simulates a distributed system in which a central server process interacts with a large number of client processes that perform tasks on data that is only accessible to themselves. Due to its inherent suitability for distributed computing, Erlang was an obvious choice for creating a framework for Federated Learning.

We describe our solution by first discussing the main components of the framework itself, which is a representation of generalized Federated Learning as outlined in Section 2 above. This is followed by a discussion of a purely functional implementation of an artificial neural network (ANN) in Erlang. Subsequently, we briefly describe how the skeleton and the ANN can be combined, including a brief discussion of how the framework can be used with existing C code. Finally, we briefly mention early experimental results.

3.1 The Skeleton of the Framework

The goal of the Federated Learning framework is to illustrate the general principle of implementing a distributed machine learning framework. The code in this section therefore leaves some details unspecified, but the missing details can be filled in easily.[2] We present the main parts of our skeleton, namely the client process and the server process. We make the assumption that there is only one server process, while the number of client processes is arbitrary. In practice, it may depend on the number of available edge devices, for instance all phones that are powered on and are below a certain threshold value of CPU usage, or all cars of a fleet that are currently in operation.

Further below, we present a purely functional implementation of an artificial neural network (3.3), which runs on client processes in our framework. Afterwards, we discuss two approaches of using client processes with the general framework.

The skeleton consists of a client process, which may be instantiated an arbitrary number of times, and a server process. Code Listing 1 shows the client process. The `receive` clause awaits a tuple tagged as an *assignment*, which indicates that the model received with this tuple should be used to process locally available data. Details of that operation are hidden behind the function `train`, which takes the aforementioned model as input. An example of such a model is a specification of the weights of an artificial neural network, or the model parameters of a multiple regression equation. After training with the locally available data has been concluded, the updated model is sent to the server process. Afterwards, the client function is called recursively, awaiting an updated model. This is close to a textbook case of message passing in Erlang, which shows how suitable, in principle, Federated Learning is for being expressed in Erlang.

As Code Listing 1 has shown, training on data is performed on the client, even though the code skeleton leaves it unspecified how

[2] Full access to the source code will be provided in the future.

```
1  client() ->
2
3      receive
4
5      { assignment, Model, Server_Pid } ->
6
7          Val = train(Model),
8          Server_Pid ! { update, self(), Val },
9          client()
10
11     end.
```

Listing 1: The client process

that step is implemented. There are many such client processes, and they all use data that may only be available to themselves, albeit there may be partly overlapping data between clients, for instance if there is overlapping input.

The server process does not perform computationally intensive tasks. The code of the skeleton of our framework related to the server process is presented in Code Listing 2. Its role is to maintain a global model, based on updates received from client processes. Our simulation selects a random subset of all available devices, which are represented by their own client process. This happens in the deliberately undefined function `select_subset`. The implementation of that function depends on the requirements of the problem that needs to be tackled. In a production system, one plausible approach is to keep track of all edge devices and, for instance, submit an assignment to a subset of those based on certain criteria, for instance being alive, or using less than a certain threshold value of computational resources. It is certainly plausible that the subset, in a production system, contains all active client processes. Sending the current model to the selected subset of client processes can be concisely expressed as a *map* over a list, a seen in line 8 of Code Listing 2.

```
1  client() ->
2
3  server(Client_Pids, Model) ->
4
5      Subset = select_subset(Client_Pids, []),
6
7      % send assignment
8      lists:map(fun(X) -> X ! { assignment, Model, self()}
        end, Subset),
9
10     % receive values
11     Vals = [ receive { update, Pid, Val } -> Val end ||
        Pid <- Subset ],
12
13     % update model:
14     Model_ = update_model(Model, Vals, length(Client_Pids
        )),
15
16     server(Client_Pids, Model_).
```

Listing 2: The server process

In the simulation, it is assumed that all devices that receive an assignment will complete it. This is reflected in the list comprehension in line 11 in Code Listing 2, which blocks until the server has received the results of all assignments that were sent to client processes. The resulting list of values, `Vals`, contains the updated model of each client process. In order to further process those models, a conceptually straight-forward approach is to average them and replace the existing model with the model that has been computed that way. However, more sophisticated approaches would be, for instance, computing the weighted average of a model, based on the number of received updates and the number of client processes that did *not* receive the current model. Such an approach should only be seen as a first approximation, though. A more precise approach may be to perform a forward pass on each received model and discard those that do not show an improvement, i.e. a reduced error compared to the previous model, and only average those models that show an improvement compared to the previous global model.

We leave the `update_model` function unspecified, but indicate that it may take the current model, the received value, and the number of clients in the distributed system into account. This would, for instance, suffice to compute a weighted average taking the previous model and the received values into account. After updating the model, the server process calls itself recursively. This concludes the presentation of the skeleton of our framework. Of course, in order to deploy this framework, the functions we left unspecified would need to be implemented, depending on user requirements.[3]

3.2 A Neural Network with Back-Propagation in Erlang

3.2.1 Some Background on Artificial Neural Networks. Artificial Neural Networks (ANN) are a standard method in machine learning for a variety of learning tasks. A prime example is classification based on pattern recognition, for instance image tagging. The general principle is to minimize an objective function that measures the magnitude of an error.

There are normally three steps to deploying an ANN: training, validation, and use in production. First, a labeled data set is used to *train* an ANN, which means that the ANN is iteratively executed until all of the carefully selected training data has been labeled correctly. There is the risk that the ANN has been over-trained and essentially memorized its input. Thus, as a counter measure a labeled validation set is used to ensure that the ANN is able to also correctly process a similar data set. If those two steps have been performed satisfactorily, the ANN is ready to be used for real-world data classification tasks.

The previous description was on a higher level and thus rather abstract. We will therefore also walk through the inner workings of an ANN. Please refer to Fig. 2. That ANN consists of two input neurons, three hidden neurons, and two output neurons. The two input neurons on the left are shaded and have the values 0.25 and 0.70. They are shaded in order to indicate that an ANN is normally not applied to a fixed set of inputs but instead the ANN is applied to each input in a larger data set, thus the input values change as the input is iterated over. The layer of neurons in the middle is the hidden layer, while the layer on the right is the output layer. The labelled edges represent connections between neurons; they are

[3] A fully working implementation of the framework, with a toy example, will be made available by the Fraunhofer-Chalmers Research Centre for Industrial Mathematics in due time.

labelled with their weights. The two edges leading from the output layer nodes to nowhere indicate that those nodes produce output. During the training phase, the output of an ANN is compared to the target value. The resulting difference is the error, which has to be minimized.

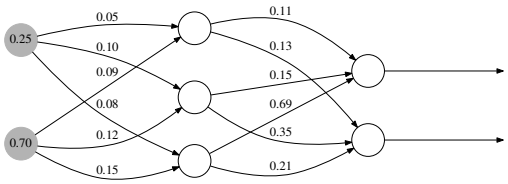


Figure 2: Example of an artificial neural network

There are two sets of labelled edges, one set connecting the input with the hidden layer, and the other connecting the hidden layer with the output layer. Their weights are normally initialized to a small value at the beginning. Via executing an ANN, those weights will be modified, with the goal of minimizing the output error, which is the difference between the target values and the actual values the output layer neurons emit. We will walk through an example shortly. Without wanting to into too much detail, the general principle is to perform one *forward pass*, which makes it possible to determine how close the value of the output neurons is to the target value, and then performing so-called backpropagation, which has the goal of adjusting the weights in order to minimize the error, i.e. the deviation between the target and the output, for subsequent rounds. In a nutshell, ANNs perform an optimization task. After training and validation, an ANN is used for classifying new input.

Using the ANN in Fig. 2 as an example, the first task is to perform a forward pass, first computing the input of each hidden layer neuron by calculating the dot product of the input weights and all edges connecting the input nodes with that hidden layer neutron. For instance, the input of the topmost hidden layer neuron is $0.25 \times 0.05 + 0.70 \times 0.09 = 0.6425$. After applying the *objective function*, which is the function that needs to be minimized, to that value, the input and output for the output layer neurons is computed similarly. Afterwards, the difference between the target and the actual output can be determined. This is followed by a backpropagation pass, in which the weights of the ANN get updated, first the weights for the edges from the output layer to the hidden layer, then the weights for the edges from the hidden to the input layer. The calculation is performed the same way, with the exception that the derivative of the objective function is used when calculating the respective dot products.

Training an ANN is done by processing all elements of the input set by using them as data for the input layer and performing one complete training pass. After each such pass, the weights are retained, as the goal is to perform training on the entire input set.

Our description pertains to a simple example of an ANN. We do not consider typical modifications such as setting a specific learning rate, or performing any kind of adaptive behavior based on previous results. We also use a standard objective function, the sigmoid function. Practitioners may use different objective functions. They may also make use of the many engineering techniques for improving the performance of ANNs, see for instance Orr et al. [11], but for the sake of brevity, we will omit such considerations.

As a final note, we would like to highlight that ANNs can approximate any function [6, 7], which is commonly refered to as the universal approximation theorem. Furthermore, ANNs are widely used in practice. The example described above, consisting of three layers, is a *shallow* ANN. Those are very versatile, albeit somewhat limited. A particularly noteworthy early breakthrough was the successful classification of handwritten digits, which is used by postal services [13]. More recent developments include so-called *deep* neural networks, often in the catch phrase *deep learning*. Those are networks that have multiple hidden neuron layers. One event that recently gained widespread attention was mastery of the game Go [12], for instance, which led to a high-profile victory of the computer against some of the strongest players in the world.

3.2.2 Implementation. In the following, we are going to cover some aspects of our implementation of a basic artificial neural network in Erlang. We will again leave out some implementation details, and instead focus on the big picture. Keep in mind that the code pertaining to the ANN implementation is executed in the client process. Please refer to Code Listing 3, which contains the function function `ann`. The input of the ANN consists of the values for the input neurons (`Input`), the weights for both layers, as well as the target values for the output. As described above, the first step is the computation of the input of the hidden layer (`Hidden_In`). The function `forward`, which is shown in Code Listing 4, computes the sum of the dot of its input. The output of the hidden layer is the result of mapping the objective function to the list `Hidden_In`. The corresponding values for the output layer are computed the exact same way. The list `Delta` contains the difference between the target values and the actual values.

The function `forward` computes the dot product of input values and the weights of outgoing edges. It is called twice by the function `ann`, which corresponds to the two transitions between layers, first from input to hidden layer, afterwards from hidden to output layer. Computing the dot product maps nicely to a functional programming style, as the required computation is the element-wise multiplication of two lists, followed by its summation.

The reverse pass, in this case backprogapation, starts with precomputing the output error, zipped with a *squashing factor*. In our case, the derivative of the objective function is given by $x(1-x)$. The code is shown in Code Listing 5.

The function for the reverse pass (`reverse`) performs backpropagation, which results in adjusted weights for the edges connecting output layer and hidden layer, as well as for the edges connecting the hidden layer with the input layer. The new weights are computed by adding the product of the error and the input to each weight.

Slightly trickier is the computation of the errors of the hidden layer, due to using lists as data structures. The weights for the edges

```
ann({Input, Weights, Targets}) ->

    {W_Input, W_Hidden} = Weights,

    % Forward Pass
    Hidden_In  = forward(Input, W_Input, []),
    Hidden_Out = lists:map(fun(X) -> objective_fun(X) end
     , Hidden_In),

    Output_In  = forward(Hidden_Out, W_Hidden, []),
    Output_Out = lists:map(fun(X) -> objective_fun(X) end
     , Output_In),

  % Target vs Output
    Delta = lists:zipwith(fun(X, Y) -> X - Y end, Targets
     , Output_Out),

    % Reverse pass
    Output_Errors = output_error(Output_Out, Targets),

    % Update weights for output layer
    W_Hidden_     = reverse(
                        Hidden_Out, Output_Errors,
     W_Hidden, []),
    Hidden_Errors = errors_hidden(
                        Hidden_Out, Output_Errors,
     W_Hidden_, []),

    W_Input_ = reverse(Input, Hidden_Errors, W_Input, [])
     ,
    Weights_ = {W_Input_, W_Hidden_},

    { Output_Errors, {Input, Weights_, Targets}}.
```

Listing 3: The core ANN function

```
forward(_    , []      , Acc) -> lists:reverse(Acc);
forward(Input, [W | Ws], Acc) ->

    Val = lists:sum(
            lists:zipwith(
                fun(X, Y) -> X * Y end, Input, W)),

    forward(Input, Ws, [Val | Acc]).
```

Listing 4: Computing the dot product of weights and inputs

```
output_error(Vals, Targets) ->
    lists:zipwith(
        fun(X, Y) -> X * (1.0 - X) * (Y - X) end, Vals,
     Targets).
```

Listing 5: Squashing the output error

```
reverse(_ , []    , []      , Acc) -> lists:reverse(Acc);
reverse(In, [E|Es], [Ws|Wss], Acc) ->

    A = lists:zipwith(
            fun(W, I) -> W + (E * I) end, Ws, In),

    reverse(In, Es, Wss, [A|Acc]).
```

Listing 6: Squashing the output error

connecting the hidden layer with the output layer were specified as a list of lists in which each list contains the incoming weights for one of the output neurons. In the reverse pass, however, we need to traverse the ANN the opposite way, so the edges connecting hidden and output layer need a representation that considers all edges that incoming into each hidden layer node. This is achieved by recursively taking the heads of the lists of lists of the weights mentioned, before performing the error calculation. This is shown in Code Listing 7.

```
errors_hidden([]    , _          , _       , Acc) -> lists:
    reverse(Acc);
errors_hidden([H|Hs], Output_Err, Weights, Acc) ->

    % take first element of output weights for
     backpropagation,
    % results in a list of weights of outgoing edges for
     each hidden
    % layer neuron
    Outgoing = lists:map(fun(X) -> hd(X) end, Weights),

    % remaining weights for next iteration
    Rest = lists:map(fun(X) -> tl(X) end, Weights),

    % compute error for current hidden layer neuron
    TMP  = lists:zipwith(fun(X, E) -> E * X end, Outgoing
     , Output_Err),
    A    = lists:sum(TMP) * H * (1.0 - H),

    errors_hidden(Hs, Output_Err, Rest, [A|Acc]).
```

Listing 7: Calculating the errors of the hidden layer neurons

Lastly, performing training on the entire input, so-called batch training, can be elegantly expressed in a functional style. A related detail of the implementation is shown in Code Listing 8. The arguments of the function `wrap_ann` are, in order, the list of inputs that constitute the training set, the weights, the target values associated with the input data, as well as an accumulator `Errors` that collects the output error for each member of the input set. The weights are constantly updated so that every subsequent invocation of the function `ann` uses the weights of the preceding invocation.

```
wrap_ann([]    , Weights, []    , Errors) ->
  {lists:reverse(Errors), Weights};

wrap_ann([I|Is], Weights, [T|Ts], Errors) ->
    { Error, Weights_ } = ann(I, Weights, T),
    wrap_ann(Is, Weights_, Ts, [Error | Errors]).
```

Listing 8: Batch processing of a list of inputs

3.3 The Combined Framework

3.3.1 Erlang. The parts introduced earlier can be easily combined in order to construct a distributed system for Federated Learning.[4] It boils down to using the skeleton introduced in Section 3.2, adding code for an artificial network to the client process, and adding some further program logic. What has not been covered is, for instance, code for input/output handling. Our assumption is that each client process operates on data that is only locally available,

[4]Of course, the Erlang code shown above is concurrent, but it can be trivially modified to distribute it across multiple devices.

so the client process needs to be adjusted correspondingly so that the available data is processed for batch-training with the ANN. Likewise, the server process needs to process the incoming models from the clients, in order to update the model that is centrally maintained. One possibility would be averaging of the weights of all the models that are sent from the client processes.

3.3.2 Erlang with C Nodes. While the description of our implementation is purely in Erlang, an alternative approach consists of a C implementation of the artificial neural network. From a user perspective, there is no difference. However, internally, the client process communicates with a node that executes an ANN written in C. A system in which C nodes were used was used as a benchmark for the performance of an implementation solely in Erlang.

3.4 Experimental Results

The main purpose of our work was to implement a distributed machine learning application that followed the Federated Lerning concept, with the goal of executing machine learning tasks on edge devices that take proprietary data as input. While the existing implementation that makes use of C nodes meets performance requirements, it seemed consequential to also explore whether an implementation purely in Erlang would be competitive.

While we have not performed detailed measurements yet, initial results show that the implementation purely in Erlang performs well. We have not benchmarked execution time on artificial data sets yet, but one key takeaway is already that memory requirements are significantly lower. This is a particularly important finding as edge devices tend to only have very limited amounts of RAM, so any savings in that regard are beneficial. In short, the existing Erlang implementation is fast enough for our industrial use case, and uses less memory than an implementation that uses C nodes.

As this extended abstract documents work that is still in progress, we cannot yet present data that quantifies the relative performance of the Federated Learning framework written exclusively in Erlang versus the version that only uses Erlang for orchestrating client processes that rely on C nodes for computationally heavy work. Please refer to Section 5 for a description of the remaining experimental work we intend to carry out.

4 RELATED WORK

There has been relatively little work in academia related to using functional programming languages for tackling *practical* machine learning tasks. About a decade ago, Allison explored using Haskell for defining various machine learning and statistical learning models [1]. Yet, that work was of a theoretical nature. Going back even further, Yu and Clack presented a system for polymorphic genetic programming in Haskell [15]. Likewise, this seems to have been work from a theoretical perspective.

5 FUTURE WORK

5.1 Plans for this paper

This draft only highlights preliminary and promising findings that compare a Federated Learning framework purely written in Erlang with one in which the client processes execute C code. Yet, more work needs to be done. As we are not able to share results from processing proprietary data, we intend to perform experimental results on standard machine learning data sets with the dual purpose of directly comparing the performance of both systems. We intend to present these results in a future revision of this paper.

As a first test case, we intend to use the so-called *Iris data set* [5], which contains observational data of the petal length of various iris species, where the task is to correctly classify them. A more ambitious use case is the MNIST handwritten digits database [8]. While it is tempting to work with a more challenging data set, it may not lead to additional insights, as our key interest is in performance comparison between two related systems.

5.2 Plans beyond this paper

Machine learning is a very active research area. In addition, due to the emergence of the Internet of Things, and the firm establishment of distributed computing, there are further avenues to pursue. An obvious example is a performance comparison between a powerful central server, possibly even a GPU-based system, with a large network of edge devices. It would be interesting to experimentally evaluate how much computational power could be harnessed by using edge devices for computations instead.

Another approach is to implement more elaborate machine learning algorithms, for instance more complex neural networks. There is little hope that a collection of edge devices will be able to compete with specialised hardware for deep learning, but, from a practitioner's perspective, there is great value in quantifying what can be achieved with edge devices. Those devices use CPUs that are also found in smart phones. As that area is one of rapid technological advancement, it is certainly worthwhile investigating what they are, or will be, capable of, as those CPUs will only get more powerful.

6 CONCLUSION

Erlang is not a go-to programming language for numerical computations. In fact, using external C libraries is common. In this paper, however, we presented an implementation of a recent machine learning concept in a purely functional style in Erlang. The result was a system that is rather concise due to the expressivity afforded by purely functional programming. We have also indicated that such a system is competitive with one in which the heavy lifting is delegated to C nodes. An important conclusion to draw from this is that Erlang can be used for *practical* distributed machine learning tasks in a real-world scenario. In addition to exhibiting competitive performance, a noteworthy benefit is that the system itself is easier to maintain than a hybrid written in both C and Erlang. As a consequence, we have been encouraged to consider Erlang, due to its surprisingly fast performance, for future industrial research projects in our research lab.

REFERENCES

[1] Lloyd Allison. 2005. Models for machine learning and data mining in functional programming. *Journal of Functional Programming* 15, 1 (2005), 15–32.
[2] Harald Bauer, Yetloong Goh, Sebastian Schlink, and Christopher Thomas. 2012. The supercomputer in your pocket. *McKinsey on Semiconductors* (2012), 14–27.
[3] Deyan Chen and Hong Zhao. 2012. Data security and privacy protection issues in cloud computing. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, Vol. 1. IEEE, 647–651.
[4] Chris Evans-Pughe. 2005. The connected car. *IEE Review* 51, 1 (2005), 42–46.
[5] RA Fisher and Michael Marshall. 1936. Iris data set. *RA Fisher, UC Irvine Machine Learning Repository* (1936).

[6] G Gybenko. 1989. Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems* 2, 4 (1989), 303–314.
[7] Kurt Hornik. 1991. Approximation capabilities of multilayer feedforward networks. *Neural networks* 4, 2 (1991), 251–257.
[8] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010).
[9] Junghoon Lee and Cheolmin Kim. 2010. A Roadside Unit Placement Scheme for Vehicular Telematics Networks.. In *AST/UCMA/ISA/ACN*. Springer, 196–202.
[10] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2016. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* (2016).
[11] Genevieve B Orr and Klaus-Robert Müller. 2003. *Neural networks: tricks of the trade*. Springer.
[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.
[13] Sargur N Srihari and Edward J Kuebert. 1997. Integration of hand-written address interpretation technology into the united states postal service remote computer reader system. In *Document Analysis and Recognition, 1997., Proceedings of the Fourth International Conference on*, Vol. 2. IEEE, 892–896.
[14] Omer Tene and Jules Polonetsky. 2011. Privacy in the age of big data: a time for big decisions. *Stan. L. Rev. Online* 64 (2011), 63.
[15] Tina Yu and Chris Clack. 1998. PolyGP: A polymorphic genetic programming system in Haskell. *Genetic Programming* 98 (1998).

Modeling CPS Systems using Functional Programming *

– Draft paper –

Zsók V., Souza D., Cesar B., Horst M., Alves Almeida D., Zborowski F., Somavila R., Caia L., Alencar A., Costa R, Pinheiro Ph., Souza R.

Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/c.
zsv@inf.elte.hu

Abstract

In nowadays CPS systems there is a need for efficient management of the tasks, input signals and workflows in order to obtain a good behaviour of the smart environment in which they are applied. The coordination of CPS components is crucial for assuring efficient collaboration between constituent elements.

In this paper we propose a model to concert the events of a SmartHouse components with use-cases that can be applied to coordinate events of the cyber-physical system. The model includes among others reading signal components, pooling events, triggering actions, handling events of the components, and managing information flowing on communication channels.

The CPS model is implemented using iTask, a task oriented workflow management tool based on the Clean pure lazy functional language. Functional language environments proved to be highly abstract and easily applicable for modeling. CPS systems are specific distributed systems used to coordinate cyber-physical world entities.

The proposed system modeling type enables the definition and the integration of special components into complex systems using iTask web services. Hence one of the main advantages of the system is that it can be used on any device providing browsers. The system functionalities are defined in an interactive way, where the base components and the relationship between them can be efficiently programmed.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed applications

General Terms functional programming, distributed systems, high-performance computing

Keywords modeling, semantics

1. Introduction

The functional programming domain offers a good programming environment for the newest trends in computer science like cyber physical systems, embedded systems, domain specific programming. The high level abstraction of functional interfaces enables the development of new applications and tools for specific purposes.

Studying relationships between CPS and distributed systems or CPS and embedded systems are important for taking adequate decisions in the designing and modeling steps of the sophisticated CPS systems description.

The analysis covers general terminology used in CPS systems for describing the collaborating computational units controlling physical entities. Relationships with other type of complex systems (embedded systems and distributed systems) are also considered as well

* Supported by ...

IFL 2017, August 30– September 1, 2017, Bristol, UK.

ACM 978-1-nnnn-nnnn-n/yy/mm.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

as executable semantics approach description for the modeling and designing is proposed using iTasks.

General structural description is considered when modeling and designing steps are taken for describing CPS systems by probabilistic semantical approach. The CPS system implementation of the paper covers aspects, features, and approaches from various practical point of views in order to establish a general prototype model of the SmartHouse CPS systems. Several important notions, definitions of CPS systems' world are clarified.

In the CPS system design important questions of semantics of complex systems are addressed from probabilistic and behavioral viewpoints as well. The interoperability with other complex systems is important when establishing main features and definitions of the prototype of specific CPS systems.

The interoperating units, the CPS actors are identified in order to design and model a prototype of a SmartHouse CPS systems using system descriptions and features a priori established. Executable semantics application possibilities for CPS systems are considered, which contributes to the generalization of the CPS prototype model.

The goal of the paper is to present the implemented prototype, which will be defined in terms of the functionalities of the components. Case studies are applied to illustrate the systems' components functionalities, and the connections with other type of distributed systems. The comparisons with distributed systems are important due to the earlier expertise and experience in multi-layered distributed system description with semantics given in executable ways.

The distributed system implemented earlier is based on two-level coordination using the D-Clean and D-Box languages developed with the purpose of skeleton based functional and distributed programming on clusters and grids [10].

1.1 D-Clean language descriptions

The D-Clean primitives offer the advantage of writing distributed and functional applications without knowing the details of the application environment and middleware services and abstracting from the technical aspects. The distributed computation uses a multiparadigm-oriented environment with several different layers. On the highest level D-Clean coordination language primitives are applied in order to define the distribution of the pure functional computation subtask. This distribution is made according to a predefined computational scheme, algorithmic skeleton.

D-Clean consists of high-level language elements for the coordination of the purely functional computational nodes in the designed distributed environment. D-Clean constructs generate D-Box computation boxes connected via buffered communication channels. The dataflows carried by the channels are typed according to the rules of the input-output protocols of the boxes generated for the evaluation of the computations. Using D-Clean the programmer controls the dataflow in the distributed process-network, and indicates how a distributed computation can be generated from the high level skeletons. The D-Box language is intermediate level language based on Petri nets describing the computational nodes and hiding the low-level implementation elements, and enabling direct control over the process-network. The boxes are compiled back into pure Clean functional programs, which communicate via channels according to well-established protocols using the middleware services of the built distributed system.

The two languages were introduced for skeleton based functional and distributed programming and the main purpose of the two languages was to define parallel computation schemes, i.e. skeletons. The actual amount of parallelism depends on many factors such as the order of channel creation, the amount of work on one channel, the speed of the data retrieving and data storage in the channels, the complexity of the computation described in the distributed graph. In a large number of D-Clean examples high speed-up for parallelism was obtained.

The skeleton based functional modeling approach used by the coordination languages can be applied to the prototype CPS systems' behaviour. The CPS systems' prototype case studies implemented aims at describing the collaborating computational units controlling physical entities, the relationships with other type of complex systems (embedded systems and distributed systems), and the design issues for executable semantics applications for CPS systems.

Practical experiences of the coordination of the computational nodes in the distributed program development showed that an automatic tool is required for prior visualization of the computation network abstracting from the real environment. The graphical visualization of the distributed computation by an executable semantics code comprehension tool supports the user when programming in the real distributed system. It depicts the

expected parallelism drawn by the boxes and channels generated using the well-defined high-level skeleton for distributed computation. Leaving out the details of the box and channel generation, it aims at modeling and formulating properties of operational semantics of the D-Clean distributed system [12], details of the executable semantics for D-Clean can be found in [13, 14].

The skeleton based functional modeling approach used by the coordination languages is applied to the nowdays fashionable CPS systems' prototypes as well. Studying the relationships between distributed systems and CPS are benefic for taking the correct essential design and modeling steps.

The CPS systems' analysis covers aspects, approaches, and case studies from various point of views to establish major features of prototype models for CPS systems.

This paper aims similar approach and the goal is to illustrate the appropriateness and applicability of the previously introduced languages" modeling for distributed evaluation of the functional iTasks of the CPS systems" models.

2. General description of modeling CPS systems

The term cyber-physical systems (CPS) was first used to underline the integration of computation in physical processes. In CPS systems, as opposed to the general purpose software systems, the simulation of elapsed time is key element in order to execute tasks in a well-defined critical period, where the correct functioning of the collaborating system units are essential. However, the most important feature of CPS systems is the intrinsic concurrent physical environment they appear in.

These systems are tested for cyber, physical and communication failures. The fault-tolerance is achieved by synchronous update of all sensors and actors, and by self-stabilization algorithms. The failure detectors are reporting permanent failures (rupture of services), distributed consensus failures (synchronization problems), intermittent failures (agent behaviour anomalies). Stabilizers regain the control of erroneous components and are transferring the workload to non-faulty components.

Another important set of studies, applied especially in robotics, are considering the safe flocking problem, where the agents maintain a minimum safe separation by avoiding collisions, form an equally spaced flock, and reach safely together a destination.

Our case studies are covering functionalities of a smart house, a special cyber-physical system were the taskflow of the events has to be managed in order to obtain an optimal behaviour of the intelligent housing units after various sensors are giving the inputs. The iTask system implemented in Clean ([6]) is used to describe the complex interactions between the events and dataflows processed.

2.1 Related works

2.1.1 Designing phases

In CPS systems the key steps are in designing, modeling, analyzing and implementing in an iterative way, by refinement steps taken to adjust, coordinate and control the envisaged co-acting units. Modeling includes the dynamic behaviours, the continuous dynamics building and the establishments of actors. A number of properties can be studied involving causality, memorylessness, linearity, time invariance, and stability with a constant feedback control. Two aspects can be set in CPS systems: a discrete and a continuous one. Our modeling includes the dynamic behaviour, the continuous units building and the establishments of actors.

Different designing phases can be observed in concrete CPS system types. Embedded systems (see the introductory book [4]) play important role in the cyber-physical world with well-defined technologies.

In case of embedded systems in CPS processors are playing the role of micro-controllers with a vide range of possible architectures, where I/O operations are used for signal processing using multitasking and scheduling algorithms.

The embedded computers or units are monitored and controlled by the physical processes inside a complex network. In CPS systems the key steps are in designing, modeling, analyzing and implementing in an iterative way, by refinement steps taken to adjust, coordinate and control the envisaged co-acting units.

The actors are modeled by functions, the properties are encapsulated in states, while the changes are mapped to state transitions. Different type of states enable to model all the operations considered for classes of hybrid systems' transitions.

The compositionality of the models offers a vide range of topology constructions using the basic units established. It can be distinguished side-by-side con-

current, synchronous or asynchronous composition of actors, shared values for concurrent composition, cascade and hierarchical compositions, or reactive models of actors. The structure and the dynamic dataflow make the model controllable in various process models.

The theoretical approach of the analysis and the verification of CPS systems at all levels require invariants and properties study. Establishing equivalences and model refinement is part of the design phase. However, our approach is a practical one, where the simulations of real application provide model checking steps when implementing. Quantitative analysis and control flow simulation are also part of the verifications of the prototype, the feasibility simulations with bounds and limitations analysis are considered in the fault tolerance of CPS system type modeled.

Multitasking and scheduling algorithms are implemented via iTasks, which offers prototyping the microcontrollers as embedded systems' objects of the SmartHouse model. CPS processors are playing important role in the vide range of possible architectures, where I/O operations are signal processing simulated by dedicated iTasks.

2.1.2 Semantical approach of system descriptions

Several semantical approaches are taken in describing the CPS systems with the collaborating units being enumerated from probabilistic point of view. The probabilistic semantics descriptions are present in some very different topics. However, the experiences can be useful to be applied when modeling CPS systems as well. Here we are considering papers describing the probabilistic approach for the semantics of various CPS entities and units.

An interesting probabilistic semantic representation is given in [1], which introduces the notion of semantic network for representing semantic information. Such models can be used in topologies that are organized in bipartite graphs. The probability of information distribution is used over such graphs for semantic representation. The model can be used in case of CPS where probabilistic relationships between the information transferred on dataflows are varying on different contexts.

The semantic web services and semantic interoperabilities among web resources are considered in [2]. Shared repositories representation of knowledge in domains characterized by uncertainty enable service composition opportunities, and provide a semantic mapping under different probabilities. The probability has two roles: in the security and uncertainty of the availability of the information and in the probabilistic assessment and request of the information on web services. Semantic web considerations opens analogies between the web services and the services of CPS system components.

The probabilistic approach is mostly present in fault tolerance analysis of system components. The semantics issues of CPS systems, evaluated according to the applicability of probabilistic semantics of some other fields, offers new ways of CPS systems' model and design, earlier being not covered in literature.

In general the case studies are mostly taking artificial intelligence approach, the above types are mostly related to CPS systems' modeling. Our prototyping style applied in the SartHouse project is a novel approach in the design of CPS systems.

2.1.3 Modeling and describing systems

Modeling CPS systems are considered according to the application area envisaged, where the notions connected to CPS units are specified in terms of the domain where it is used. A complex prototype study and important properties are identified in [9].

The important design phases are identified as: system definition with the description of distributed parameters for coordinating, controlling, identifying, observing and following units. Sensors are important actors in CPS systems. They are used for heterogeneous dynamics sensing with parameter estimations and for remote sensing with well defined policies and protocols. The applied control framework needs to be established together with parameter estimations and optimizations using the probabilistic approach. The communication topology is optimized by dynamic sensors after scaling and building fault-tolerance approximations.

In our CPS design the most important phase is the identification of collaborating components and establishing the principles of taskflows and used dataflows. This includes the design and construction of a very well structured communication topology. We will cover these steps in our model description providing specific information related to each of the main components together with their use cases.

2.1.4 Case studies for CPS systems

Important, typical cases studies of distributed cyber-physical systems (DCPS) can be observed. DCSP com-

puting systems are a collection of individual devices communicating with each other and interacting with their physical environment through sensors and actuators. Examples of such systems include: mobile robots or aerial vehicles used for rescue tasks, automated highway network systems, the electric grids or SmartGrids, management of computers for dynamic voltage and frequency scaling, wireless sensor networks and actors.

DCSP systems are tested for cyber, physical and communication failures. The fault-tolerance is achieved by synchronous update of all sensors and actors, and by self-stabilization algorithms. The failure detectors are reporting permanent failures (rupture of services), distributed consensus failures (synchronization problems), intermittent failures (agent behaviour anomalies). Stabilizers regain the control of erroneous components and are transferring the workload to non-faulty components.

The distributed cellular traffic control case study analyzes the safety and the progress properties, stabilizes the routing algorithm of the cells, and reconsiders the routing algorithm towards the targets. The simulation of the self-stabilizing distributed traffic control protocol for the partitioned planes has each partition controlling the motion of all entities within that partition.

The algorithm guarantees the entities, that did not face crash failures of the software controlling a partition, are progressing to the target (see more in [8]).

The second study, applied especially in robotics, describes the safe flocking problem. The agents form an equally spaced flock and are separated to avoid collisions while they safely reach together a destination.

The modeled DCPS combined with failure detector satisfy self-stabilization property. A reduced form of safety, when a single failure occurs, is simulated where a strong flock reaches a destination by failure-free execution without causing their neighbors to follow or diverge. The fault-tolerant DCPS is coordinated in a way that the collapse of the system is avoided.

Our case studies include tests of the major properties of the system, where the safe collaboration among the components are crucial in prototyping.

2.2 iTask overview

The `iTask` system is a combinator library implemented in *Clean* [5] for specifying dynamic data dependent workflows in a very flexible way, see [6]. The specified workflows are executed as a multi-user web-application. An intensive research about client-side `iTask` processing is reported in [3].

3. Description of the SmartHouse system

One important area of the CPS systems is the domain of the intelligent housing, where all the functionalities of the house are determined by the collaboration of the built-in sensors. The SmartHouse managing CPS system connects important main components with the necessary subparts in order to monitor all the events and to react accordingly. The SmartHouse CPS application is prototyping the sophisticated collaboration between sensors and smart components in order to trigger the actions needed to obtain the desired optimal behaviour.

In this project events are the most important units to encode and to represent the functionalities. Here we have the `MyEvent` type, created to encapsulate like in the following all the major activities in the SmartHouse:

```
:: MyEvent = {
        mytype :: String,
        message :: String,
        componentId :: String
        }
```

where `mytype` is representing the event type, `message` is a string via which a component stores messages, and `componentId` the identification of the component.

There is a dedicated `Monitor` task, who keeps monitoring all the events happening inside the house (for example a was door opened, the temperature is too high so turn on air conditioning, or it's getting dark outside so switch on the lights). The task `Monitor` can be delegated to another user to fully observe the events.

When an event is triggered in form of a `(type, message, compID)` triple, then a flag is being sent. An example of a generated event between two components might be like in the following:

```
type: Transmit
message: comp2 | turn on the lights
compID: comp1
flag: 1
```

Once a flag is set to 1, the Monitor is allowed to start reading the message and create `Events` in the `EventPool`.

```
cWriteEvent sharedCI
= forever (wait "" (λci → ci.flagIn == 1 ) sharedCI
>>|
get sharedCI>>=λci →
   (upd (λt → t ++ [(ci.inp!!0)]) EventPool)
>>|
```

```
set {compID = ci.compID,inp = (tl ci.inp),
     out = ci.out,flagIn = 0} sharedCI)
```

As the Monitor observes parallel tasks, it also keeps reading any event that comes from a component comp1 in the ConsequencePool.

```
cWatchConsequence sharedCI = forever (get sharedCI
>>=
λci → wait "" (λcPool →
    waitForC cPool ci.compID ) ConsequencePool
>>|
get ConsequencePool >>=λcPool →
set {compID = ci.compID, inp = ci.inp,
    out = ci.out ++ [(getFromPool cPool ci.compID)],
    flagIn = ci.flagIn} sharedCI
>>|
upd (λt → remFromPoolE t
    (getFromPool cPool ci.compID))
    ConsequencePool)
```

Both EventPool and ConsequencePool are shared memories to store distributed data used by the various components of the smart house. Both have essentially the same functioning mechanisms, except for what they represent.

```
EventPool :: Shared [MyEvent]
EventPool = sharedStore "EventPool" []

ConsequencePool :: Shared [MyEvent]
ConsequencePool = sharedStore "ConsequencePool" []
```

When EventPool receives an Event, the Reactor will notice it and will create an Event on ConsequencePool, like the following:

```
type: transmit
message: turn on the lights
compID: comp2
```

The Reactor basically it is a task that watches the EventPool for an event of type T After reading this event, acts as needed, and produces a consequence on the ConsequencePool. Each Reactor reacts only to a certain type and will execute a function specified for it.

```
reactor = enterInformation "Type the evType" []
      >>= λevType = waitTypeSelection evType

waitTypeSelection evType
| evType == "Transmit"
= (viewInformation
    "Waiting for a Event" [] "Working...")
-||
forever (wait "" (λevPool →
    waitForE evPool evType) EventPool
>>|
get EventPool >>=λevPool →
    return (getFromPoolE evPool evType)
>>=
λev → upd (λt → remFromPoolE t ev) EventPool
>>|
get ConsequencePool >>=λcPool →
set [{mytype = ev.mytype, message =
    toString (drop 1 (dropWhile (λx = x ≠ '|')
    (fromString ev.message))) ,
    componentId = toString (takeWhile (λx = x ≠ '|')
    (fromString ev.message))}] ConsequencePool )
| otherwise = (viewInformation "Failure" []
  "The event type didnt match any specified type")
```

Each component has its own Reactor which is always looking for any event on ConsequencePool and once it finds something related to its component it reacts immediately executing the work requested.

References

[1] Costa, P.C., Laskey, K.B., Laskey, K.J.: Probabilistic Ontologies for Efficient Resource Sharing in Semantic Web Services, Proceedings of the Second Workshop on Uncertainty Reasoning for the Semantic Web (URSW 2006) at the Fifth International Semantic Web Conference (ISWC 2006), pp 1–10.

[2] Griffiths, T.L., Steyvers, M.: A probabilistic approach to semantic representation, Proc. of the 24th Annual Conf. of the Cognitive Science Society, 2002, pp 1–6.

[3] Jansen, J. M.: *Functional Web Applications - Implementation and Use of Client Side Interpreters*, Ph.D. Thesis, Radboud University Nijmegen, 2010.

[4] Lee, E. A., Seshia, S. A.: Introduction to Embedded Systems - A Cyber-Physical Systems Approach, LeeSeshia.org, 2011.

[5] Plasmeijer, R. and van Eekelen, M.: *Concurrent Clean language report (version 2.0)*, December 2001. http://www.cs.ru.nl/~clean/.

[6] Plasmeijer, R., Achten, P., Koopman, P.: An Introduction to iTasks: Defining Interactive Work Flows for the Web, In: Horváth Z. et al. (eds.): *Central European Functional Programming School*, CEFP 2007, Second Summer School, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures, LNCS Vol. 5161, Springer-Verlag, 2008, pp. 1–40.

[7] Plasmeijer, R., Achten, P., Koopman, P., Lijnse, B., van Noort, T., van Groningen, J.: iTasks for a Change - Type-Safe Run-Time Change in Dynamically Evolving Workflows, Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '11, January 24-25, 2011, Austin, Texas, USA, ACM, pp. 151–160.

[8] Taylor, T.J.: Fault-tolerant distributed cyber-physical systems: two case studies, MSc Thesis, University of Illinois at Urbana-Champaign, 2010.

[9] Tricaud, C, YangQuan Ch.: Optimal Mobile Sensing and Actuation Policies in Cyber-physical Systems, Springer Verlag, 2012.

[10] Zsók V., Hernyák Z., and Horváth, Z.: Designing Distributed Computational Skeletons in D-Clean and D-Box. In: Horváth Z. (ed.): *Central European Functional Programming School*, CEFP 2005, First Summer School, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures, LNCS Vol. 4164, Springer-Verlag, 2006, pp. 223–256.

[11] Zsók V., Hernyák Z., Horváth, Z.: Improving the Distributed Elementwise Processing Implementation in D-Clean, In: Horváth Z., Kozma L, Zsók V. (eds.): *Proceedings of the 10th Symposium on Programming Languages and Software Tools* SPLST 2007, Dobogókő, Hungary, June 14-16, 2007, Eötvös University Press, 2007, pp. 256–264.

[12] Zsók V., Koopman, P., Plasmeijer, R.: Generic Executable Semantics for *D-Clean*, In: Porkoláb Z. et al. (eds.): *Proceedings of the Third Workshop on Generative Technologies*, WGT 2011, ETAPS 2011, Saarbrücken, Germany, March 27, 2011, ENTCS Vol. 279, No. 3, Elsevier, December 2011, pp. 85–95.

[13] Zsók V., Porkoláb Z.: The Distributed D-Clean Model Revisited by Templates, International Conference on Numerical Analysis and Applied Mathematics ICNAAM 2011 - SCLIT 2011, Halkidiki, Greece, 19-25 September 2011, In: Theodore E. Simos et al. (eds.): *AIP Conference Proceedings*, American Institute of Physics, September 2011, Vol. 1389, pp. 877–880.

[14] Zsók V., Porkoláb Z.: Rapid Prototyping for Distributed D-Clean using C++ Templates, *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, Eötvös Loránd University, Budapest, Hungary, 2012, Vol. 37, pp. 19–46.

Constraint Handling Rules with Scopes

Alejandro Serrano
A.SerranoMena@uu.nl
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

Jurriaan Hage
J.Hage@uu.nl
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

Abstract

Constraint Handling Rules provide a language to describe constraint solvers. However, when type systems including some form of local reasoning, such as higher-rank types or GADTs, we often need some sort constraints which include implications. CHR frameworks do not usually deal with this form of constraints.

In this paper we show how to extend the CHR$^\nabla$ formalism introduced in (Serrano and Hage 2017) to directly manage (certains forms of) constraints with implications within our extended CHR framework. Using this extension we handle GADTs and certain forms of higher-rank polymorphism directly at the constraint level. In addition, we can reason about universally quantified constraints, such as $\forall\, a.\, \mathit{Monoid}\ a$ in Haskell.

ACM Reference format:
Alejandro Serrano and Jurriaan Hage. 2017. Constraint Handling Rules with Scopes. In *Proceedings of 29th Symposium on Implementation and Application of Functional Languages, Bristol, United Kingdom, 30 August – 1 September 2017 (IFL 2017),* 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction to CHR$^\nabla$

Constraint Handling Rules (Frühwirth 2009) provide a language to describe constraint solvers. The great body of work related to CHRs, summarized in (Frühwirth 1998; Sneyers et al. 2010), makes them a great framework to describe algorithms where constraint rewriting is involved.

Type inference for functional languages is one of the areas in which CHRs have been applied successfully. CHRs have been used to improve type error reporting (Stuckey et al. 2006; Wazny 2006), describe and extend the type class machinery in Haskell (Dijkstra et al. 2007; Sulzmann et al. 2007) and generalize the shape of algebraic data types (Sulzmann et al. 2006). However, CHRs are not enough to describe in a concise form the solving needed for some advanced type system features.

In particular, CHRs do not handle more advanced type system features without extra-logical features. In particular, visible type application (Eisenberg et al. 2016) or higher-rank types (Peyton Jones et al. 2007) demand *instantiation* of polymorphic types to be *delayed* as much as possible. We perform this delay we describing the instantiation as a constraint $\tau_1 \leqslant \tau_2$. For example, when *id* is used an an expression, we assign a new variable β to the expression and recall that $\forall\, a.\, a \to a \leqslant \beta$.

The question is how to proceed during constraint solving when one of those constraints is found. Usually, a rule for instantiating polymorphic types looks similar to:

$$\mathit{forall}\ (X, T) \leqslant S \iff T' = \mathit{inst}\ (X, T) \mid T' = S$$

This rule turns $\forall\, a.\, a \to a \leqslant \beta$ into $\alpha \to \alpha \leqslant \beta$ for a fresh variable α. When dealing with higher-rank types, the converse situation is also possible:

$$T \leqslant \mathit{forall}\ (X, S) \iff S' = \mathit{skol}\ (X, S) \mid T = S'$$

In this situation we do not introduce new unification variables, but rather *rigid* or *Skolem* constants.

Unfortunately, in both cases *inst* and *skol* are defined outside of the language of CHRs. This has a number of outcomes:

- The binding structure in *forall* is manually maintained; we need to enforce that the new variables or constants introduced by *inst* and *skol* are fresh by means of some internal state.
- We remark that how to adequately handle type variables within CHRs is not yet well understood (Csorba et al. 2012).
- Since *inst* and *skol* are extra-logical, we cannot apply the usual tools for reasoning about CHRs.

In (Serrano and Hage 2017) we propose an extension of CHRs, called CHR$^\nabla$, which solves these problems. The new rules dealing with polymorphic types become:

$$\mathit{forall}\ (T) \leqslant S \iff \exists V.\, T\ V \leqslant S$$
$$T \leqslant \mathit{forall}\ (S) \iff \nabla A.\, T \leqslant S\ A$$

2 CHR$^\nabla$ with Implication

Extending Constraint Handling Rules with λ-tree syntax and ∇ quantification is enough for implementing a simple form of higher-rank types. However, programming languages such as Haskell include the notion of *qualified types*. In that case, when quantifying over a variable it is possible to attach some *constraints*[1] that the variable must satisfy. A classic example are *type class instance* constraints, as in $\mathit{show} :: \forall\, a.\, \mathit{Show}\ a \Rightarrow a \to \mathit{String}$.

We are not directly concerned with solving qualified constraints using CHRs; this problem has been studied previously (Sulzmann et al. 2007). However, when combined with higher-rank types, CHR$^\nabla$ is not sufficient to implement the type system concisely. Take the following constraint:

$$\forall\, a.\, \mathit{Eq}\ a \Rightarrow a \to a \ \leqslant\ \forall\, a.\, \mathit{Ord}\ a \Rightarrow a \to a$$

In the standard Haskell base libraries, *Ord* is a subclass of *Eq*. Thus, the previous instantiation constraint should hold. However, we cannot directly apply the rule for polymorphic types in the right-hand side of an instantiation constraint we described in the previous section, since it does not account for qualifiers. Intuitively,

This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213).

IFL 2017, Bristol, United Kingdom

DOI: 10.1145/nnnnnnn.nnnnnnn

[1] Unfortunately, the same term is used for constraints in polymorphic types and constraints as in CHRs.

$$
\begin{array}{llll}
\text{Extended constraints} & \hat{C} & ::= & C \\
 & & \mid & \hat{C}_1, \ldots, \hat{C}n \\
 & & \mid & \nabla X_1 \ldots X_\ell . \hat{C} \\
 & & \mid & \exists X_1 \ldots X_\ell . \hat{C} \\
 & & \mid & I \supset \hat{C} \\
\text{Antecedent} & I & ::= & C_1, \ldots, C_n \\
\text{Simple constraints} & C & ::= & c(T_1, \ldots, T_n) \quad c \in \mathfrak{C}
\end{array}
$$

Figure 1. Syntax of extended constraints

we want to check that $\forall\, a.\, Eq\ a \Rightarrow a \to a \leqslant b \to b$ for some constant b *assuming* that $Ord\ b$ holds. This is a form of implication:

$$\nabla\, b.\, Ord\ b \supset (\forall\, a.\, Eq\ a \Rightarrow a \to a \leqslant b \to b)$$

Constraints with implications also appear when dealing with Generalized Algebraic Data Types (Sulzmann et al. 2006; Vytiniotis et al. 2011; Xi et al. 2003). In that case variables are quantified only in a particular branch, so existential quantification appears *inside* implications.

In this section we show how to extend CHR$^\nabla$ to deal with implication and quantification in constraints. In this case we do not need to extend the language: all the manipulation happens inside the CHR$^\nabla$ formalism.

2.1 Scoped Constraints

The key point of our approach is to annotate each constraint with a *scope*, which is used to decide which constraints may interact and to make it possible to reconstruct a complex constraint from its constituents. This approach, however, cannot handle arbitrarily nested structures of implications and quantification. In particular, implication and quantification are *not* allowed on the *left-hand* side of an implication. That means that $A \supset (B \supset C)$ is allowed, but $(A \supset B) \supset C$ is not. The formal description of the syntax is given in Figure 1. In § 2.3 we relax some of these restrictions.

Built-in constraints are not considered in this section. The reason is that built-in are opaque to the CHR machinery, and might perform arbitrary changes to the constraint set, like unification across all constraints. This does not make our technique useless: type inference algorithms, for example, can be described without any kind of built-in process (Vytiniotis et al. 2011).

Now we can state our goal more precisely: we want to associate to each extended constraint $\hat{C}$ a set of scoped simple constraints $\{C_i \ @\ \varsigma_i\}$ in a bijective manner. Furthermore, we must modify the existing CHRs so that they obey the restrictions imposed by the scopes. The first step is defining scopes themselves.

Definition 2.1. We denote a *finite sequence* of elements by $\langle e_1, \ldots, e_n \rangle$. We denote the *split* of a sequence ς into two (possibly empty) subsequences ς_1 and ς_2 as $\varsigma = \varsigma_1, \varsigma_2$. In that case we say ς_1 is a *prefix* and ς_2 a *suffix* of ς, respectively. When $\varsigma_2 = \langle \star \rangle$ is a singleton sequence, we write $\varsigma = \varsigma_1, \star$.

Definition 2.2. A *scope* is a sequence of *scope elements* $\langle \diamond_{i_1}, \ldots, \diamond_{i_n} \rangle$ where:

- The symbol $\diamond \in \{\nabla, \exists, \supset^L, \supset^R\}$ denotes the type of nested structure.
- Each identifier i_j is unique in the sequence and taken from a set $\Im$. The set $\Im$ must be endowed with an operation to generate fresh elements. For simplicity, in this paper we take $\Im$ to be the set of natural numbers.

$$\frac{C \text{ simple}}{C \ @\ \varsigma \rightsquigarrow \{C \ @\ \varsigma\}}\ \textsc{simple}$$

$$\frac{\hat{C}_i \ @\ \varsigma \rightsquigarrow C_i}{\hat{C}_1, \ldots, \hat{C}_n \ @\ \varsigma \rightsquigarrow \bigcup C_i}\ \textsc{conj}$$

$$\frac{i \text{ fresh} \qquad \mathrm{a}_1, \ldots, \mathrm{a}_\ell \text{ fresh constants} \qquad \hat{C}[X_j \mapsto \mathrm{a}_j] \ @\ \varsigma, \nabla_i \rightsquigarrow C}{\nabla X_1 \ldots X_\ell . \hat{C} \ @\ \varsigma \rightsquigarrow C}\ \nabla$$

$$\frac{i \text{ fresh} \qquad Y_1, \ldots, Y_\ell \text{ fresh variables} \qquad \hat{C}[X_j \mapsto Y_j] \ @\ \varsigma, \exists_i \rightsquigarrow C}{\exists X_1 \ldots X_\ell . \hat{C} \ @\ \varsigma \rightsquigarrow C}\ \exists$$

$$\frac{i \text{ fresh} \qquad \hat{I} \ @\ \varsigma, \supset_i^L \rightsquigarrow I \qquad \hat{C} \ @\ \varsigma, \supset_i^R \rightsquigarrow C}{\hat{I} \supset \hat{C} \ @\ \varsigma \rightsquigarrow I \cup C}\ \supset$$

Figure 2. Translation from extended to a set of scoped simple constraints

The translation $\hat{C} \ @\ \varsigma \rightsquigarrow C$ from extended constraints to simple constraints annotated with scopes is given in Figure 2. During translation, a current scope ς is pushed down, which will ultimately be reflected as an annotation in rule SIMPLE. In CONJ, that current scope is distributed among constraints.

The three remaining rules extend the current scope to reflect a new nesting level. Note that in the case of implications, the two parts are recursively converted to simple constraints annotated with a different $\supset^L$ and $\supset^R$ symbol, but with the *same* identifier. In that way we keep the connection between antecedent and consequent of the implication.

In this form, the rules in Figure 2 define a function, but not an invertible one. The information we need to recover an extended constraint from a set of scoped simple constraints is given by the sets of variables which have been introduced at each step. The translation judgment can easily be extended to recall that information too; we refrain from doing so for the sake of conciseness.

2.2 Rules with Scoped Constraints

Extended constraints feature implication and quantification, which come with a uniform meaning derived from logic. This suggests that given a set of Constraint Handling Rules working on the language of simple constraints, it is possible to modify them to work on scoped constraints.

Generation rules. The first question is how to handle the generation of new scopes when universal or existential quantification is involved in a rule. An example of such a rule was given in our discussion of higher-rank types:

$$T \leqslant \mathit{forall}\ (\lambda X.\, Q) \iff \nabla\, V.\, T \leqslant Q\ V$$

The scoped variant of that rule starts with a scoped head,

$$(T \leqslant \mathit{forall}\ (\lambda X.Q)) \ @\ \varsigma$$

In the body we need to enlarge ς to account for the new level of nesting

$$\nabla\, V.\, (T \leqslant Q\ V) \ @\ \varsigma, \nabla_i \quad \text{with } i \text{ fresh}$$

The same transformation, modulo change of symbol, is done if existential variables are introduced. When both ∇ and $\exists$ are used in the body of a rule, we need to generate two fresh identifiers instead, one per level.

We mentioned at the beginning of the section that our aim is to extend polymorphic types with qualified constraints. When the polymorphic type is on the left-hand side of an instantiation constraint, we just need to generate the extra constraints as part of the rule, but otherwise the rule is left unchanged:

$$\begin{aligned} &\mathit{forall}\ (\lambda X.\, C_1, ...C_n \Rightarrow Q) \leqslant T\ @\ \varsigma \\ &\quad \Longleftrightarrow\ \exists V.\, C_1\ V\ @\ (\varsigma, \exists_i), ..., C_n\ V\ @\ (\varsigma, \exists_i), \\ &\qquad\qquad Q\ V \leqslant T\ @\ (\varsigma, \exists_i) \end{aligned}$$

In the case of a polymorphic type in the right-hand side, an implication would be introduced. That means that we need to introduce different symbols for antecedent and consequent, sharing a common identifier.

$$\begin{aligned} &T \leqslant \mathit{forall}\ (\lambda X.\, C_1, ..., C_n \Rightarrow Q)\ @\ \varsigma \\ &\quad \Longleftrightarrow\ \nabla V.\, C_1\ V\ @\ (\varsigma, \nabla_i, \supset_j^L), ..., C_n\ V\ @\ (\varsigma, \nabla_i, \supset_j^L), \\ &\qquad\qquad T \leqslant Q\ V\ @\ (\varsigma, \nabla_i, \supset_j^R) \quad \text{with } i, j \text{ fresh} \end{aligned}$$

Note that we have also introduced a ∇ symbol, since this rule introduces a new nominal constant along with the constraints.

The previous transformation relies on an operation generating fresh identifiers. In order to get a pure implementation, we introduce an extra constraint current-id (i) which saves the state of the fresh name generator. Whenever we need a new identifier, we update the generator using the next-id operation we assume for the set $\Im$ of identifiers. When applied in the first rule for higher-rank types, the modified CHR reads:

$$\begin{aligned} &(T \leqslant \mathit{forall}\ (\lambda X.Q))\ @\ \varsigma \setminus \text{current-id}\ (i) \\ &\quad \Longleftrightarrow\ \nabla V.\, (T \leqslant Q\ V)\ @\ (\varsigma, \nabla_i), \\ &\qquad\qquad \text{current-id}\ (\text{next-id}\ (i)) \end{aligned}$$

Flow transformation. Apart from being able to generate new scopes when quantifiers are used, we need to restrict the application of rules to those cases in which they would be allowed when considering full extended constraints. For simplicity, let us first consider the case in which only one constraint appears as a head to be simplified, and no quantification is done.

$$H_1^k, \ldots, H_n^k \setminus H^r \iff G \mid B_1, \ldots, B_m$$

Intuitively, if this rule applies, then all H_i^k are allowed to *interact* with H^r. For example, we may have the H^k at the left-hand side of an implication and H^r at the right-hand side. Or they might all be in the same scope. We formalize this intuition by the notion of *flow* or *nesting* between scopes.

Definition 2.3. We say that a scope ς *flows* to ς', or that ς' is *nested* into ς, denoted $\varsigma \ggg \varsigma'$, if ς' does not include $\supset^L$ elements, and either:

- ς is a prefix of ς', or
- $\varsigma^*, \supset_i^L$ is a prefix of ς and $\varsigma^*, \supset_i^R$ is a prefix of ς'.

Now we can be precise: the rule can be applied only if all the scopes of the H^k constraints flow to the scope of H^r. The resulting body B replaces H^r, and thus B must have the same scope as H^r.

$$\begin{aligned} H_1^k\ @\ \varsigma_1, \ldots, H_n^k\ @\ \varsigma_n \ \setminus\ & H^r\ @\ \varsigma' \\ \Longleftrightarrow\ & G, \varsigma_1 \ggg \varsigma', \ldots, \varsigma_n \ggg \varsigma' \\ \mid\ & B_1\ @\ \varsigma', \ldots, B_m\ @\ \varsigma' \end{aligned} \tag{1}$$

The transformation over rules we have just mentioned may also be seen as the combination of two different rules. The first transformation is plainly an extension of the original CHR rule to account for the extra scoping information, which only works when all constraints live in *in the same scope* ς.

$$\begin{aligned} H_1^k\ @\ \varsigma, \ldots, H_n^k\ @\ \varsigma \ \setminus\ & H^r\ @\ \varsigma \\ \Longleftrightarrow\ & G \mid B_1\ @\ \varsigma, \ldots, B_m\ @\ \varsigma \end{aligned} \tag{2}$$

The second rule allows to copy a constraint to a nested scope:

$$H\ @\ \varsigma \implies \varsigma \ggg \varsigma' \mid H\ @\ \varsigma' \tag{3}$$

Lemma 2.4. *Rule 3 is sound.*

Proof. We look at how the transformations are translated back to extended constraints. We only need to consider four cases depending on how ς is extended to get ς', other cases follow by induction:

- If ς is extended by $\exists$, this translates back to:
$$(\exists V.\, C), Q \rightsquigarrow \exists V.\, (C, Q), Q$$
In other words, the constraint Q is introduced inside the existential. This transformation follows from the rule of logic, as V is not bound inside Q.
- The extension by ∇ follows the exact same pattern as $\exists$:
$$(\nabla V.\, C), Q \rightsquigarrow \nabla V.\, (C, Q), Q$$
- If we extend by $\supset^R$, there are two possible cases depending on the last element of ς. If the last element is $\supset^L$, the transformation is equivalent to:
$$(Q, Q') \supset C \rightsquigarrow (Q, Q') \supset (C, Q')$$
In the other cases, the transformation is different:
$$(Q \supset C), R \rightsquigarrow (Q \supset C, R), R$$
Both transformations are allowed by the rules of implication. The first case is reflexivity of the implication. In the second case, if we can prove R without any assumption for Q, it must be the case that we can also prove it with those extra assumptions.
- We do not need to consider the case in which ς' ends in $\supset^L$, since flowing is then not allowed by the definition of $\ggg$. As an extended constraint, this rule would allow moving constraints to the antecedent of an implication. Note also that we do not allow extended constraints in the left-hand side of an implication, which means that $\supset^L$ may only appear at the end of a scope.

□

Rule 1 can be derived by several applications of Rule 3 followed by one application of Rule 2. Logically speaking, thus, we are fine by including the latter two. Operationally, however, their effect is quite different: Rule 3 leads to a multiplication of copies of every constraint in every possible scope in which it could be applied. Rule 1 is more efficient: it only allows moving a constraint to a nested scope when it takes part in some simpagation.

In the case in which more than one constraint appears in the set H^r, we need to be a bit more careful. In order for the rule to be applicable, there must be a scope ς^* such that all scopes $\{\varsigma_1, \ldots, \varsigma_n\}$ from H^k and H^r may flow to, that is $\varsigma_i \ggg \varsigma^*$ for every i. The constraints generated in the body B should be annotated with the scope ς^*.

$$\begin{array}{lrcl} \text{Antecedent} & I & ::= & C \\ & & \mid & I_1, \ldots, I_n \\ & & \mid & \nabla X_1 \ldots X_\ell . I \end{array}$$

Figure 3. Upgraded syntax for implications

Collapse rules. The approach we presented to generate scopes is a bit naïve, and might lead to too much nesting. Consider for example the extended constraint $\nabla X.(Q_1, \nabla Y.Q_2)$. Once translated, Q_2 is assigned a scope nested into that of Q_1. Because of our flow restriction, that means that Q_2 is not allowed to interact with Q_1 in the outer scope. However, logically speaking, that extended constraint is equivalent to $\nabla X\, Y.\, Q_1, Q_2$, since Y is not free in Q_1. That suggests that we should remove one level of nesting, so Q_1 and Q_2 can interact freely.

More formally, we want to forbid scopes to have two consecutive ∇ or $\exists$ identifiers. A quantifier with an empty set of variables is not useful either, so we want to get rid of these too. One possibility is to have an operation which performs the reduction of scopes. However, scheduling them to happen at the right time is a difficult task. Instead, we prefer to prevent bad scopes from being introduced, by modifying the generation rules given above.

In order to upgrade the generation rules, we introduce an $\mathsf{extend}(\varsigma, \diamond, n, i)$ operation, which given a scope ς, a symbol $\diamond$ quantifying over n variables and an identifier i, returns an enlarged scope satisfying all the restrictions.

$$\mathsf{extend}(\varsigma, \diamond, n, i) = \begin{cases} \varsigma & \text{if } n = 0 \\ \varsigma & \text{if } \varsigma = \varsigma^*, \diamond_j \text{ for some } j \\ \varsigma, \diamond_i & \text{otherwise} \end{cases}$$

If we combine this change with the manual manipulation of generator state, we come to rules such as:

$$\begin{array}{l} (T \leqslant \mathit{forall}\ (\lambda X.Q))\ @\ \varsigma \setminus \mathsf{current\text{-}id}\ (i) \\ \quad \Longleftrightarrow\ \nabla\, V.\, (T \leqslant Q\ V)\ @\ \mathsf{extend}\ (\varsigma, \nabla, 1, \mathsf{next\text{-}id}\ (i)), \\ \qquad \mathsf{current\text{-}id}\ (\mathsf{next\text{-}id}\ (i)) \end{array}$$

This rule says that if the CHR engine is presented with a constraint $T \leqslant \mathit{forall}\ (\lambda X.Q)$, it should replace it by $T \leqslant Q\ V$ where the V is universally quantified. In the case the original $T \leqslant \mathit{forall}\ (\lambda X.Q)$ constraint came from an existential or implication scope, we must create a new universal scope; this is taken care of by extend. For the operation of extend we need access to the fresh variable generator, encoded in the current-id constraint.

2.3 Universally Quantified Antecedents

Scoped constraints as presented up to now impose restrictions on the antecedents of implications. In this section we lift some of them, allowing ∇ quantification on both sides of an implication, not only on the right-hand side.

The rules do not have to undergo any change: the definition of nesting can be kept the same, and the soundness result still holds. Unfortunately, they are not enough for practical uses. Take for example the implication:

$$\nabla X.\, P(X) \supset \nabla Y.\, P(Y)$$

Since each of the sides is a simple α-renaming of the other one, this formula trivially holds. During the operation of the CHR engine, however, we obtain the following scoped constraints:

$$P(\mathrm{x})\ @\ \langle \supset_1^L, \nabla_2 \rangle,\ P(\mathrm{y})\ @\ \langle \supset_1^R, \nabla_3 \rangle$$

Although the first one is allowed to flow into the second one, and thus they can interact, discharging the second constraint cannot take place. The problem is that at each ∇ quantifier different nominal constants have been introduced. These constants are different, so $P(\mathrm{x}) \not\equiv P(\mathrm{y})$.

The solution to this problem lies in the IDπ rule:

$$\frac{\pi(B) \equiv \pi'(B') \qquad \pi, \pi' \text{ permutations of constants}}{\Gamma, B \vdash B'}\ \textsc{id}\pi$$

In our case, the permutation $[\mathrm{x} \mapsto \mathrm{y}, \mathrm{y} \mapsto \mathrm{x}]$ does the job. If by renaming all the nominal constants we can find a common instance of the predicate, the left-hand side version can discharge the right-hand side one.

As a Constraint Handling Rule, the IDπ reads:

$$\begin{array}{l} B\ @\ \varsigma^*, \supset_i^L, \varsigma_1 \setminus B'\ @\ \varsigma^*, \supset_i^R, \varsigma_2 \\ \quad \Longleftrightarrow\ B, B' \text{ permutations over the vars. of } \varsigma_1 \text{ and } \varsigma_2 \\ \qquad \mid\ \top \end{array}$$

There are several remarks to be made regarding this rule:

- The general structure says that if B is in the left-hand side of an implication, B' in the right-hand side and there is a permutation taking one to the other, the second one can be removed. In CHR terms, B' is simplified to $\top$.
- The usual flow relation $\ggg$ is too general for this case, since two constraints may flow into one another also in cases where no implication is involved. However, the IDπ rule really needs constraints in an implication hierarchy. The solution is to inline the desired case of the $\ggg$ relation directly.
- The definition of permutation in the usual presentation of the ∇ quantifier may affect any variable. However, in that case the information about nominal constants is scoped over each constraint. In our case, that information is constant. The effect is that we cannot allow *any* nominal constant to be renamed, just the ones introduced *after* the implication. This is the reason the guard specifies that the permutation must be over the variables in the tails of the scope sequences. This last restriction is satisfied in our example: x comes from the ∇_2 subscope, y from ∇_3, so they can be swapped.

This small extension to CHR$^\nabla$ with implications is not merely theoretical, but also has some practical uses. In particular, we can extend Haskell's class language to account for universally quantified contexts, of the form $\forall\ a.\ C\ a.$ For example, the *Alternative* class is defined in the base library:

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  (⟨|⟩) :: f a → f a → f a
```

where *empty* and $(\langle|\rangle)$ form a monoid for each *a*. The Haskell class declaration, however, does not show any relation between *Alternative* and *Monoid*. Even more, the names from *Monoid* are not reused in *Alternative*, so you need to know when to use $(\langle|\rangle)$ in contrast to *mappend*. Disregarding the *Applicative* superclass, a more direct definition is:

```
class ∀ a. Monoid (f a) ⇒ Alternative f
```

Building on the translation of Haskell's type classes as CHRs from (Sulzmann et al. 2007), the more direct definition of *Alternative* gives rise to two different rules.

$$\begin{array}{l} \mathit{Alternative}\ F \Rightarrow \nabla\, A.\ \mathit{Monoid}\ (\mathit{con}\ (F, [A])) \\ \mathit{Alternative}\ F \setminus \mathit{Monoid}\ (\mathit{con}\ (F, [X])) \iff \top \end{array}$$

The reader may wonder why the second rule is needed instead of just reading $\forall$ as ∇. The reason is that we want the semantics of $\forall$ in this context to coincide with its *extensional* reading, not simply the intensional one embodied by ∇. Thus, a constraint *Alternative f* can be used also to discharge a constraint *Monoid* $(f\ a)$ for a given *instantiation* of *a*, not simply for the constant case.

Imagine now that you need to provide a transformation for *Alternative* values of the type $\forall\, f\ a.\ \mathit{Alternative}\ f \Rightarrow f\ a \to f\ a$ and you decide to use a function $\forall\ m.\ \mathit{Monoid}\ m \Rightarrow m \to m$. The generated constraint is:

$$\begin{array}{l} \mathit{forall}\ (\lambda M.\ \mathit{Monoid}\ M \Rightarrow \mathit{fn}\ (M, M)) \\ \quad \leqslant \mathit{forall}\ (\lambda F.\ \mathit{forall}\ (\lambda A. \\ \qquad \mathit{Alternative}\ F \Rightarrow \mathit{fn}(\mathit{con}(F, [A]), \mathit{con}(F, [A])))) \end{array}$$

which is then simplifed into scoped constraints equivalent to:

$$\begin{array}{l} \nabla F.\ \nabla A.\ \mathit{Alternative}\ F \\ \quad \supset\ \exists M.\ \mathit{Monoid}\ M, \\ \qquad \mathit{fn}\ (M, M) = \mathit{fn}(\mathit{con}(F, [A]), \mathit{con}(F, [A])) \end{array}$$

From the last constraint, we get $M = \mathit{con}\ (F, [A])$. Then, we need to discharge *Monoid con* $(F, [A])$. Since we know that *Alternative*[], we can discharge it.

Universally quantified constraints may also appear in types. For example, you may ask for a value of the type $\forall\, f\ b.\ (\forall\ a.\ \mathit{Semigroup}\ (f\ a)) \Rightarrow f\ b \to f\ b$, a relaxation of the *Alternative* pre-condition. If we provide a value of type $\forall\ g\ c.\ \mathit{Alternative}\ g \Rightarrow g\ c \to g\ c$, we end up having to solve the constraint:

$$\mathit{Alternative}\ G\ \supset\ \nabla\, A.\ \mathit{Semigroup}\ (\mathit{con}\ (G, [A]))$$

In the CHR engine, these constraints are represented as *Alternative* g and *Semigroup* $(\mathit{con}\ (\mathrm{g}, [\mathrm{a}]))$ for some nominal constants g and a. From the left-hand side, we propagate the constraint *Monoid* $(\mathit{con}\ (\mathrm{g}, [\mathrm{b}]))$ for a new nominal constant b. Since every *Monoid* is also a *Semigroup*, a second propagation takes place, leading to *Semigroup* $(\mathit{con}\ (\mathrm{g}, [\mathrm{b}]))$. Finally, we can use the CHR version of the IDπ rule and discharge the version of *Semigrouop* instance with a with the one of a, as desired.

3 Implementation

We have implemented the type checker featured in this paper; it is available at https://git.science.uu.nl/f100183/quique. The solver is written using CHRs: the only difference is that ∇ quantification is not explicit, but rather inferred from the scope information. That is, if a new nominal constant is introduced, it is assumed to as nested as the constraint where it first appears.

4 Related Work

Implication in CHRs. In order to type check programs with Extended Algebraic Data Types, implication checking for CHRs has been developed (Sulzmann et al. 2006). The main difference is that (Sulzmann et al. 2006) introduces an external driver for solving, whereas we transform the constraints and rules themselves by adding scope information, and keep the solver untouched.

Another approach (Schrijvers et al. 2006) for implication checking in CHRs is based on the fact that proving $A \supset B$ is equivalent to proving that $A \leftrightarrow (A \wedge B)$. The CHR engine itself does not need to be modified, its trace is inspected afterwards to determine whether $A \wedge B$ has been ultimately proven when assuming A. In our case, no post-processing is needed.

Quantified class constraints. We are definitely not the first ones to speak about quantification in type class constraints. Class contexts featuring both universal quantification and implication have been found useful in the field of generic programming (Hinze and Jones 2001). In some cases, a translation to Haskell 98 is possible (Trifonov 2003), but such approach requires non-local changes to the code. Our approach, more limited in expressivity, only entails changes in the class definition itself.

With the advent of *ConstraintKinds* for GHC (Bolingbroke 2011), it has been possible to encode some form of universal quantification, as witnessed by the constraints package. The main idea is to have un-exported types taking the role of Skolem variables. Since our code cannot see what is the type being manipulated, it must be the case that a constraint holds for every type. This reading is closer to the extensional style for universal quantification, although the "Skolem types" are quite similar to our nominal constants, except that un-exported Skolem types cannot be checked for inequality.

References

Max Bolingbroke. 2011. Constraint Kinds for GHC. (2011). http://blog.omega-prime.co.uk/?p=127 Blog post.

János Csorba, Zsolt Zombori, and Péter Szeredi. 2012. Pros and Cons of Using CHR for Type Inference. (2012).

Atze Dijkstra, Gerrit van den Geest, Bastiaan Heeren, and S. Doaitse Swierstra. 2007. *Modelling Scoped Instances with Constraint Handling Rules.* Technical Report. Department of Information and Computing Sciences, Utrecht University.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems, ESOP 2016.*

Thom Frühwirth. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming* 37, 1–3 (1998), 95–138.

Thom Frühwirth. 2009. *Constraint Handling Rules* (1st ed.). Cambridge University Press.

Ralf Hinze and Simon Peyton Jones. 2001. Derivable Type Classes. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 5 – 35.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82.

Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. 2006. Automatic Implication Checking for CHR Constraints. *Electron. Notes Theor. Comput. Sci.* 147, 1 (Jan. 2006), 93–111.

Alejandro Serrano and Jurriaan Hage. 2017. Constraint Handling Rules with Binders, Patterns and Generic Quantification. In *ICLP 2017, Melbourne, Australia, August 28 - September 1, 2017.*

Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie Koninck. 2010. As time goes by: Constraint Handling Rules. *TPLP* 10, 1 (2010), 1–47.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2006. Type Processing by Constraint Reasoning. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Lecture Notes in Computer Science, Vol. 4279. 1–25.

Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. 2007. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.* 17, 1 (2007), 83–129.

Martin Sulzmann, Jeremy Wazny, and Peter J. Stuckey. 2006. A Framework for Extended Algebraic Data Types *(FLOPS'06).* 47–64.

Valery Trifonov. 2003. Simulating Quantified Class Constraints *(Haskell '03).* ACM, 98–102.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (2011), 333–412.

Jeremy Wazny. 2006. *Type inference and type error diagnosis for Hindley/Milner with extensions.* Ph.D. Dissertation. University of Melbourne, Australia.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors *(POPL '03).* ACM, 224–235.

Speculative Strictness of Array Comprehensions

— 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

Robert Stewart
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
r.stewart@hw.ac.uk

Hans-Nikolai Viessmann
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
hv15@hw.ac.uk

Abstract

This extended abstract presents a functional language that is capable to switch evaluation strategy of certain expressions from strict to non-strict. In particular, we study this technique in application to array comprehensions. Array comprehensions offer richer expressive power under non-strict semantics, however strict evaluation usually offers better performance. For the cases when it is impossible to analyse statically whether a program under strict semantics evaluates a value, we propose to chose a strict mode by default and then recover at runtime if the initial assumption was wrong. Given that the overheads of such switching are minimal, the proposed technique can be seen as a universal evaluation strategy for array comprehensions.

ACM Reference Format:
Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Stewart, and Hans-Nikolai Viessmann. 2017. Speculative Strictness of Array Comprehensions. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A large number of high-performance problems can be naturally formulated using operations on arrays. As demonstrated by the examples of SaC, DpH, Futhark, *etc.*, array operations can be expressed within functional languages, offering lots of opportunities for program optimisations and code generation.

To facilitate program analysis, array-based languages try to bring all the array operations to some normal form. Usually such a normal form is based around the concept of map/reduce combinators. We refer to such a normal form as *array comprehension.*

PL'17, January 01–03, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Many experiences suggest that strictly evaluated array expressions deliver better performance than non-strict. Therefore, a lot of compilers attempt to enforce maximal strictness when optimising high-performance problems.

Consider array comprehension based on the *imap* (index map) construct:

```
imap 5 λi.i = [0,1,2,3,4]
```

it has two arguments: the length of the resulting array and the mapping function from indexes to values. The function is called for the range of indexes from 0 to 5 (including 0, but excluding 5), and results of its application are put together at the corresponding array indices forming an array value. The above expression evaluates to an immutable array of 5 elements. We assume that all arrays are one-dimensional. Array indexing starts with 0.

Conditionals within the *imap* function, make it possible to partition the *imap* index-space:

```
imap 5 λi.if i < 2 then 1 else 2 = [1,1,2,2,2]
```

Given that semantics of *imap* is strict, expressing typical Haskell-like memoizing recursive definitions:

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

becomes very tricky.

Ideally, we would like to write something like:

```
letrec fibs = imap 5
              λi.if i = 0 then 1
                 else if i = 1 then 1
                 else fibs.(i-1) + fibs.(i-2)
in fibs
```

However, when selecting from *fibs*, the strict evaluation strategy would require the entire array to be fully evaluated, before the element could be selected. Such a requirement makes the above expression to diverge[1].

Evaluation of recursively-defined *imap*s require a non-strict mode. At the same time, evaluating all the *imap*s non-strictly can be harmful for performance. In the example

[1] Expression does not reduce to a value.

above, a straight-forward static analysis will be able to figure out that the *imap* will diverge under strict evaluation. For simple non-recursive *imap*s is is easy to proof that they can be evaluated strictly. However, any such analysis is not universal, as some of the values may be statically unavailable:

```
letrec a = imap n
           λi.if i < 10 then 1
              else a.(i mod 10)
in a
```

depending on the value of n, the above expression can or cannot be evaluated strictly.

We could take a conservative approach, prescribing non-strict evaluation for the above example because there is a possibility that it can have self references. Instead, we investigate a speculative approach, when we start evaluating all the *imap*s strictly and observe self references at runtime.

Further we present a draft of semantics for the language that captures such an idea.

2 Proposed Approach

Consider a minimalistic array-based functional language with the following syntax:

e	::=	$0, 1, \ldots$	(natural numbers)
	|	x	(variables)
	|	$\lambda x.e$	(abstractions)
	|	$e\ e$	(applications)
	|	$if\ e\ then\ e\ else\ e$	(conditionals)
	|	$letrec\ x = e\ in\ e$	(recursive let)
	|	$e + e, \ldots$	(built-in binary)
	|	$imap\ e\ e$	(index map)
	|	$e.e$	(selections)

For simplicity, the only scalar values we support are natural numbers. Abstractions and applications are standard; conditionals treat 0 as *false* and any other number as *true*; *letrec* is a recursive let binding; primitive operations are built-in and are defined on scalars in a usual way. Key operations of interest are *imap* that constructs immutable one-dimensional arrays and selection operation that makes it possible to access individual array elements.

Considering previous discussion, we want this language to be strict. To give a meaning to programs we will use *natural semantics* [1]. We use the following values:

$$v ::= v_u \mid [v_u, \ldots, v_u] \qquad v_u ::= c \mid [\![\lambda x.e, \rho]\!]$$

where v_u can be a scalar constant or a function closure; $[v_u, \ldots, v_u]$ is a homogeneous array of numbers or functions; ρ is the environment. To make sharing more visible, instead of binding variables to values in the environment, we bind variables to *pointers*. Pointer-value bindings are kept in a storage, commonly denoted with S in this paper.

$$\rho ::= \cdot \mid \rho, x \mapsto p \qquad S ::= \cdot \mid p \mapsto v$$

Environment and storage look-ups happen *right to left* and are denoted as $\rho(x)$ and $S(p)$ respectively. Judgements take the following form:

$$S; \rho \vdash e \Downarrow S'; p$$

This means that within the storage S and the environment ρ we can show that e reduces to the storage S' and the pointer p. We would also use a shortcut notation:

$$S; \rho \vdash e \Downarrow S'; p \Rightarrow v \qquad \text{for} \qquad S; \rho \vdash e \Downarrow S'; p \wedge S'(p) = v$$

The core rules are:

$$\textsc{Const}\ \frac{S_1 = S, p \mapsto c}{S; \rho \vdash c \Downarrow S_1; p} \qquad \textsc{Var}\ \frac{x \in \rho \qquad \rho(x) \in S}{S; \rho \vdash x \vdash S; \rho(x)}$$

$$\textsc{Prf}\ \frac{\oplus \in \{+, -, \ldots\} \qquad S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow v_1 \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \Rightarrow v_2 \qquad S_3 = S_2, p \mapsto v_1 \text{ sem } (\oplus)\, v_2}{S; \rho \vdash e_1 \oplus e_2 \Downarrow S_3; p}$$

$$\textsc{Abs}\ \frac{S_1 = S, p \mapsto [\![\lambda x.e, \rho]\!]}{S; \rho \vdash \lambda x.e \Downarrow S_1; p} \qquad \textsc{App}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow [\![\lambda x.e', \rho']\!] \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \qquad S_2; \rho', x \mapsto p_2 \vdash e' \Downarrow S_3; p_3}{S; \rho \vdash e_1 e_2 \Downarrow S_3; p_3}$$

$$\textsc{If-true}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow n \in \mathbb{N}, n \neq 0 \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2}{S; \rho \vdash if\ e_1\ then\ e_2\ else\ e_3 \Downarrow S_2; p_2}$$

$$\textsc{If-false}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow n \qquad n = 0 \qquad S_1; \rho \vdash e_3 \Downarrow S_2; p_3}{\rho \vdash if\ e_1\ then\ e_2\ else\ e_3 \Downarrow S_2; p_3}$$

Here, 'sem $(\oplus)$' corresponds to the meaning of the binary operation. The rules for *imap* and selections are:

$$\textsc{Imap}\ \frac{S; \rho \vdash e_1 \Downarrow S'; p_n \Rightarrow n \in \mathbb{N} \qquad S'; \rho \vdash e_2 \Downarrow S_0; p_f \qquad \forall k \in \{0, \ldots, n-1\} : S_k; \rho, f \mapsto p_f \vdash f\ k \Downarrow S_{k+1}; p_k \Rightarrow v_k}{S; \rho \vdash imap\ e_1\ e_2 \Downarrow S_n, p \mapsto [v_0, \ldots, v_{n-1}]; p}$$

$$\textsc{Sel}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow [v_0, \ldots, v_{n-1}] \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \Rightarrow k \in \mathbb{N}, k < n}{S; \rho \vdash e_1.e_2 \Downarrow S_2, p \mapsto v_k}$$

The *imap* rule evaluates the length of the array and the mapping function, then it applies the mapping function to indices $\{0, \ldots, n-1\}$ and combines results into a vector. Selection rule extracts the k-th element from the array. Array indexing is zero-based.

The rules we have defined so far are sufficient to proof evaluation of non-recursive *imap*s. For example, we can show that

$$\cdot; \cdot \vdash \textbf{imap}\ 5\ \lambda i.\, i \Downarrow S; p \Rightarrow [0, 1, 2, 3, 4]$$

An interpreter for the presented language can be straightforwardly derived from the presented rules.

The rule for the *letrec* given in [1] can be transliterated in our notation as follows:

$$\textsc{Letrec-Kahn}\quad \frac{S, p \mapsto v; \rho, x \mapsto p \vdash e_1 \Downarrow S_1; p \Rightarrow v \qquad S_1; \rho, x \mapsto v \vdash e_2 \Downarrow S_2; p_2}{S; \rho \vdash \mathit{letrec}\, x = e_1\ \mathit{in}\ e_2 \Downarrow S_2; p_2}$$

Before evaluating e_1 we need to guess the value that e_1 evaluates to. This means that the entire mechanics of recursive evaluation is not exposed. This makes the rule concise, but the previous claim that the interpreter is straightforwardly deducible from the rules does not hold anymore. However, this rule makes it possible to prove that recursive *imap*s evaluate to arrays; for example:

```
·;· ⊢ letrec a = imap 5
                      λi.if i = 0 then 0
                         else a.(i−1)+1
      in a
⇓ S;p ⇒ [0, 1, 2, 3, 4]
```

Let us define a more explicit version of the letrec rule, yet preserving the above property. We start with a typical approach to letrec evaluation used in many strict languages. This can be denoted as follows:

$$\textsc{Letrec}\quad \frac{S_1 = S, p \mapsto \bot \qquad \rho_1 = \rho, x \mapsto p \qquad S_1; \rho_1 \vdash e_1 \Downarrow S_2; p_2 \qquad S_3 = S_2[p_2/p] \qquad S_3; \rho, x \mapsto p_2 \vdash e_2 \Downarrow S_4; p_4}{S; \rho \vdash \mathit{letrec}\, x = e_1\ \mathit{in}\ e_2 \Downarrow S_4; p_4}$$

The $\bot$ value ensures that if a letrec variable will be accessed during the computation of e_1, as for example in the expression $\mathit{letrec}\, x = x\ \mathit{in}\, x$, the evaluation will fail. $S[p_2/p]$ denotes substitution of the $x \mapsto p$ bindings inside of the enclosed environments with $x \mapsto p_2$, where x is any legal variable name. Such a rule makes it possible to create circular references that we need. Consider an example of evaluating the program $\mathit{letrec}\, f = \lambda x.f\ x\ \mathit{in}\, f$:

$\cdot;\cdot$	$\mathit{letrec}\, f = \lambda x.f\ x\ \mathit{in}\, f$	Letrec
$S = p \mapsto \bot; \rho = f \mapsto p$	$\lambda x.f\ x$	Abs
$S_1 = S, p_1 \mapsto [\![\lambda x.f\ x, f \mapsto p]\!]; \cdot$	$\mathit{letrec}\, f = p_1\ \mathit{in}\, f$	$S_2 = S_1[p_1/p]$
$S_2 = S, p_1 \mapsto [\![\lambda x.f\ x, f \mapsto p_1]\!]; f \mapsto p_1$	f	Var
$S_2; \cdot$	p_1	□

As it can be seen, the closure with the recursive function has the environment that correctly reference the closure. Unfortunately, this rule does not treat recursive *imap*s properly. The reason for this is inability to evaluate selections into partially evaluated *imap*s.

To regain this ability, we can evaluate the *imap* lazily. One way of doing this is to introduce a new value for an *imap* closure:

$$[\![\mathit{imap}\, p_n\ p_f, \{\}]\!]$$

which contains the *imap*, where both arguments are pre-evaluated and an index-value mapping to store a partial result. The partial result will be updated each time we make a selection into an *imap* closure. The rules to create such a closure and evaluate selections into it are:

$$\textsc{Imap-Lazy}\quad \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \qquad S_1\rho \vdash e_2 \Downarrow S_2; p_2 \qquad S_3 = S_2, p \mapsto [\![\mathit{imap}\, p_1\ p_2, \{\}]\!]}{S; \rho \vdash \mathit{imap}\, e_1\ e_2 \Downarrow S_3; p}$$

$$\textsc{Sel-Lazy-Imap-1}\quad \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow [\![\mathit{imap}\, p_1'\ p_2', M]\!] \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \Rightarrow m \in \mathbb{N}, m < S_1(p_1') \qquad m \in M}{\rho \vdash e_1.e_2 \Downarrow S_2, p \mapsto M(m); p}$$

$$\textsc{Sel-Lazy-Imap-2}\quad \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow [\![\mathit{imap}\, p_1'\ p_2', M]\!] \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \Rightarrow m \in \mathbb{N}, m < S_1(p_1') \qquad m \notin M \qquad S_2; \rho, f \mapsto p_2' \vdash f\ m \Downarrow S_3; p_3 \Rightarrow v \qquad S_4 = S_3 \oplus_{p_2} [\![\mathit{imap}\, p_1'\ p_2', M \cup \{m \mapsto v\}]\!]}{\rho \vdash e_1.e_2 \Downarrow S_4, p \mapsto v; p}$$

The Imap-Lazy rule evaluates *imap* arguments and creates a closure, with an empty partial result. The Sel-Lazy-Imap-1 rule asserts that the value at the given index can be found within the partial result M of the *imap* closure. In this case we look-up the value in M. The Sel-Lazy-Imap-2 rule asserts that the value is not within the mapping M, in which case we apply the *imap* function at the given index and we memoize the evaluated result. To do the memoization we update the storage using $S \oplus_p v$ operation that replaces $p \mapsto \_$ mapping with $p \mapsto v$ in S.

These extensions make it possible to handle recursive *imap*s, even if some of the elements are not defined:

```
letrec a = imap 5
                  λi.if i = 0 then a.i
                     else 1
in a
```

That is, all the selections into *a* will deliver a result, except if we select at index 0. The price for this is the necessity to maintain and update partial results within *imap* closures.

At this point, we have two sets of rules for *imap* and selections. How do we chose which one to use, assuming that we prefer strict evaluation over the non-strict one.

Our approach is to speculatively evaluate *imap*s strictly, and detect selections into $\bot$ values and then recover. We introduce a new value called *Rec*. It will be returned on selections into $\bot$ and captured by *imap*s.

We extend the *imap* and selection rules as follows:

$$\textsc{Sel-1}\quad \frac{S; \rho \vdash e_1 \Downarrow S_1; p \mapsto \bot}{S; \rho \vdash e_1.e_2 \Downarrow S, p \mapsto \mathit{Rec}; p}$$

IMAP-1

$$\frac{S;\rho \vdash e_1 \Downarrow S';p_n \Rightarrow n \in \mathbb{N} \qquad S';\rho \vdash e_2 \Downarrow S_0;p_f \quad \forall k \in \{0,\ldots,n{-}1\} : S_k;\rho,f \mapsto p_f \vdash f\ k \Downarrow S_{k+1};p_k \Rightarrow v_k \quad \exists k \in \{0,\ldots,n{-}1\} : v_k = Rec}{S;\rho \vdash imap\ e_1\ e_2 \Downarrow S_0,p \mapsto [\![imap\ p_n\ p_f, \{\}]\!]}$$

Finally, we update all the rules to propagate the *Rec* value:

$$\frac{S;\rho \vdash e \Downarrow S_1;p \Rightarrow Rec}{S;\rho \vdash E[e] \Downarrow S_1;p}$$

where $E[e]$ denotes a term that has e as a sub-expression which has to be evaluated before the overall term E, *e.g.* an application $E = (e_1\ e_2)$ requires e_1 to be evaluated before we can proceed with E.

Optimisations After such a semantic-switching mechanism is in place, we investigate further optimisations that improve performance of the compiled programs. First of all, if the *imap* finds an iteration that evaluates to *Rec*, currently all the non-*Rec* values are abandoned, and a new *imap* closure is created. This can be improved by gathering non-*Rec* values into a partial result. How exactly we detect such a *Rec* value at runtime, specifically in the context of parallel execution, deserves a proper explanation.

Secondly, the *imap* rule prescribes an order in which *imap* index space is traversed. For non-recursive *imap*s it can be easily shown that the order can be arbitrary, and even that all the iterations can be evaluated concurrently. This is used extensively by code-generators when targeting parallel architectures. In case of recursive specifications the choice of the order may result in the code that does not need to evaluate recursive *imap*s lazily. For example:

```
letrec a = imap 5
                λi.if i = 0 then 0
                   else a.(i-1)+1
in a
```

can be compiled to the following C code:

```
int a[5];
for (size_t i = 0; i < 5; i++)
    a[i] = i == 0 ? 0 : a[i-1]+1;
```

In some cases the chosen order would even allow for some form of parallelism.

Thirdly, inn this abstract we have presented the language that supports one-dimensional arrays only. The same ideas can be straight-forwardly applied for multi-dimensional arrays.

Finally, the number of implementation-related questions needs to be answered.

In the full paper we are going to address the above issues and provide a proof-of-concept implementation of the proposed ideas.

References

[1] G. Kahn. 1987. Natural semantics. In *STACS 87*, FranzJ. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Lecture Notes in Computer Science, Vol. 247. Springer Berlin Heidelberg, 22–39. https://doi.org/10.1007/BFb0039592