Proceedings of the 30th Symposium on Implementation and Application of Functional Languages

IFL 2018

Lowell, MA, USA
5th - 7th September 2018

Edited by

Matteo Cimini and Jay McCarthy

Preface

This volume contains the proceedings of IFL 2018 (30th Symposium on Implementation and Application of Functional Languages), which was held in Lowell, Massachusetts, USA, in the period 5th-7th September 2018. IFL 2018 was organized with the help of the University of Massachusetts Lowell.

IFL symposia aim at bringing together researchers actively engaged in the implementation and application of functional and function-based programming languages. IFL 2018 has been 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.

In response to the call for papers, the Program Committee has received 13 full paper submissions: 7 full paper pre-symposium submissions, and 6 full paper post-symposium submissions. We are thankful to all authors who submitted a paper to IFL 2018. After careful reviewing and discussions, the Program Committee have selected 11 full paper submissions for publication. We thank our Program Committee members and external reviewers for their excellent work.

The IFL 2018 program comprised the presentations of full and draft papers. The program was greatly enriched by the invited keynote speech of Adam Chlipala, and the invited keynote speech of Arjun Guha. Furthermore, the IFL 2018 program highly enjoyed a Haskell mini-course organized by Galois, Inc., which has been delivered by David Thrane Christiansen and José Calderón. We are thankful to all our speakers, tutorial lecturers, and to Galois, Inc., and we also thank the IFL Steering Committee for its invaluable help. Finally, we would like to thank the ACM staff for working with us towards the publication of this volume.

February 2019

Matteo Cimini
Jay McCarthy

Organization

Chairs

Matteo Cimini	University of Massachusetts Lowell, US
Jay McCarthy	University of Massachusetts Lowell, US

Publicity Chair

Jurriaan Hage	Utrecht University, NL

Steering Committee Chair

Rinus Plasmeijer	Radboud University Nijmegen, NL

Steering Committee

Peter Achten	Radboud University Nijmegen, NL
Andrew Butterfield	University of Dublin & Trinity College, IE
Olaf Chitil	University of Kent, UK
Matteo Cimini	University of Massachusetts Lowell, US
Olivier Danvy	Aarhus University, DK
Andy Gill	University of Kansas, US
Clemens Grelck	University of Amsterdam, NL
Jurriaan Hage	Utrecht University, NL
Kevin Hammond	University of St Andrews, UK
Ralf Hinze	Oxford University, UK
Zoltan Horvath	Eötvös Lorand University, Budapest, HU
Frank Huch	University of Kiel, DE
Pieter Koopman	Radboud University Nijmegen, NL
Ralf Lammel	University of Koblenz, DE
Rita Loogen	University of Marburg, DE
Jay McCarthy	University of Massachusetts Lowell, US
Marco T. Morazan	Seton Hall University, US
Ricardo Pena	Complutense University of Madrid, ES
Rinus Plasmeijer	Radboud University Nijmegen, NL
Sven-Bodo Scholz	University of Hertfordshire, DE
Tom Schrijvers	Katholieke Universiteit Leuven, BE
Sam Tobin-Hochstadt	Northeastern University, US
Nicholas Wu	University of Bristol, UK

Program Committee

Arthur Charguéraud	Inria, FR
Ben Delaware	Purdue University, US
Christos Dimoulas	Northwestern University, US
David Darais	University of Vermont, US
Dominic Orchard	University of Kent, UK
Ekaterina Komendantskaya	Heriot-Watt University, UK
J. Garrett Morris	University of Kansas, US
Heather Miller	Carnegie Mellon University, US
Jeremy Yallop	University of Cambridge, UK
Keiko Nakata	SAP SE, DE
Laura M. Castro	University of A Coruña, ES
Magnus O. Myreen	Chalmers University of Technology, SE
Peter Achten	Radboud University Nijmegen, NL
Natalia Chechina	Bournemouth University, UK
Peter-Michael Osera	Grinnell College, US
Richard Eisenberg	Bryn Mawr College, US
Trevor L. McDonell	Utrecht University, NL
Yukiyoshi Kameyama	University of Tsukuba, JP

External Reviewers

Jérémie Detrey
Paul Zimmermann

Table of Contents

Pure Functional Epidemics

An Agent-Based Approach

Jonathan Thaler
Thorsten Altenkirch
Peer-Olaf Siebers
jonathan.thaler@nottingham.ac.uk
thorsten.altenkirch@nottingham.ac.uk
peer-olaf.siebers@nottingham.ac.uk
University of Nottingham
Nottingham, United Kingdom

ABSTRACT

Agent-Based Simulation (ABS) is a methodology in which a system is simulated in a bottom-up approach by modelling the micro interactions of its constituting parts, called agents, out of which the global system behaviour emerges.

So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology, which simulates the spreading of an infectious disease through a population, we demonstrate how to use pure Functional Reactive Programming to implement ABS. With our approach we can guarantee the reproducibility of the simulation at compile time and rule out specific classes of run-time bugs, something that is not possible with traditional object-oriented languages. Also, we found that the representation in a purely functional format is conceptually quite elegant and opens the way to formally reason about ABS.

CCS CONCEPTS

• **Computing methodologies → Agent / discrete models**; **Simulation languages**;

KEYWORDS

Functional Reactive Programming, Monadic Stream Functions, Agent-Based Simulation

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Pure Functional Epidemics: An Agent-Based Approach. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310372

1 INTRODUCTION

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al [15] in which the authors claim "[..] object-oriented programming to be a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]" (p. 179). This work established the metaphor in the ABS community, that *agents map naturally to objects* [31], which still holds up today.

In this paper we challenge this metaphor and explore ways of approaching ABS in a pure (lack of implicit side-effects) functional way using Haskell. By doing this we expect to leverage the benefits of pure functional programming [21]: higher expressivity through declarative code, being polymorph and explicit about side-effects through monads, more robust and less susceptible to bugs due to explicit data flow and lack of implicit side-effects.

As use case we introduce the SIR model of epidemiology with which one can simulate epidemics, that is the spreading of an infectious disease through a population, in a realistic way.

Over the course of three steps, we derive all necessary concepts required for a full agent-based implementation. We start with a Functional Reactive Programming (FRP) [46] solution using Yampa [20] to introduce most of the general concepts and then make the transition to Monadic Stream Functions (MSF) [34] which allow us to add more advanced concepts of ABS to pure functional programming.

The aim of this paper is to show how ABS can be implemented in *pure* Haskell and what the benefits and drawbacks are. By doing this we give the reader a good understanding of what ABS is, what the challenges are when implementing it and how we solve these in our approach.

The contributions of this paper are:

- We present an approach to ABS using *declarative* analysis with FRP in which we systematically introduce the concepts of ABS to *pure* functional programming in a step-by-step approach. Also this work presents a new field of application to FRP as to the best of our knowledge the application of FRP to ABS (on a technical level) has not been addressed before. The result of using FRP allows expressing continuous time-semantics in a very clear, compositional and declarative way, abstracting away the low-level details of time-stepping and progress of time within an agent.
- Our approach can guarantee reproducibility already at compile time, which means that repeated runs of the simulation with the same initial conditions will always result in the

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310372

same dynamics, something highly desirable in simulation in general. This can only be achieved through purity, which guarantees the absence of implicit side-effects, which allows to rule out non-deterministic influences at compile time through the strong static type system, something not possible with traditional object-oriented approaches. Further, through purity and the strong static type system, we can rule out important classes of run-time bugs e.g. related to dynamic typing, and the lack of implicit data-dependencies which are common in traditional imperative object-oriented approaches.
- Using pure functional programming, we can enforce the correct semantics of agent execution through types where we demonstrate that this allows us to have both, sequential monadic behaviour, and agents acting *conceptually* at the same time in lock-step, something not possible using traditional object-oriented approaches.

In Section 2 we define Agent-Based Simulation, introduce Functional Reactive Programming, Arrowized programming and Monadic Stream Functions, because our approach builds heavily on these concepts. In Section 3 we introduce the SIR model of epidemiology as an example model to explain the concepts of ABS. The heart of the paper is Section 4 in which we derive the concepts of a pure functional approach to ABS in three steps, using the SIR model. Section 5 discusses related work. Finally, we draw conclusions and discuss issues in Section 6 and point to further research in Section 7.

2 BACKGROUND

2.1 Agent-Based Simulation

Agent-Based Simulation is a methodology to model and simulate a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is known. Those parts, called agents, are modelled and simulated, out of which then the aggregate global behaviour of the whole system emerges.

So, the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in some neighbourhood by exchange of messages.

We informally assume the following about our agents [26, 39, 47]:

- They are uniquely addressable entities with some internal state over which they have full, exclusive control.
- They are pro-active, which means they can initiate actions on their own e.g. change their internal state, send messages, create new agents, terminate themselves.
- They are situated in an environment and can interact with it.
- They can interact with other agents situated in the same environment by means of messaging.

Epstein [14] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. They exhibit the following properties:

- Linearity & Non-Linearity - actions of agents can lead to non-linear behaviour of the system.
- Time - agents act over time, which is also the source of their pro-activity.
- States - agents encapsulate some state, which can be accessed and changed during the simulation.
- Feedback-Loops - because agents act continuously and their actions influence each other and themselves in subsequent time-steps, feedback-loops are common in ABS.
- Heterogeneity - agents can have properties (age, height, sex,...) where the actual values can vary arbitrarily between agents.
- Interactions - agents can be modelled after interactions with an environment or other agents.
- Spatiality & Networks - agents can be situated within e.g. a spatial (discrete 2D, continuous 3D,...) or complex network environment.

2.2 Functional Reactive Programming

Functional Reactive Programming is a way to implement systems with continuous and discrete time-semantics in pure functional languages. There are many different approaches and implementations but in our approach we use *Arrowized* FRP [22, 23] as implemented in the library Yampa [9, 20, 29].

The central concept in Arrowized FRP is the Signal Function (SF), which can be understood as a *process over time* which maps an input- to an output-signal. A signal can be understood as a value which varies over time. Thus, signal functions have an awareness of the passing of time by having access to Δt which are positive time-steps, the system is sampled with.

$$Signal\ \alpha \approx Time \rightarrow \alpha$$
$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [9, 20, 29].

Event. An event in FRP is an occurrence at a specific point in time, which has no duration e.g. the recovery of an infected agent. Yampa represents events through the *Event* type, which is programmatically equivalent to the *Maybe* type.

Dynamic behaviour. To change the behaviour of a signal function at an occurrence of an event during run-time, (amongst others) the combinator *switch :: SF a (b, Event c) → (c → SF a b) → SF a b* is provided. It takes a signal function, which is run until it generates an event. When this event occurs, the function in the second argument is evaluated, which receives the data of the event and has to return the new signal function, which will then replace the previous one. Note that the semantics of *switch* are that the signal function, into which is switched, is also executed at the time of switching.

Randomness. In ABS, often there is the need to generate stochastic events, which occur based on e.g. an exponential distribution.

Yampa provides the combinator *occasionally :: RandomGen g ⇒ g → Time → b → SF a (Event b)* for this. It takes a random-number generator, a rate and a value the stochastic event will carry. It generates events on average with the given rate. Note that at most one event will be generated and no 'backlog' is kept. This means that when this function is not sampled with a sufficiently high frequency, depending on the rate, it will lose events.

Yampa also provides the combinator *noise :: (RandomGen g, Random b) ⇒ g → SF a b*, which generates a stream of noise by returning a random number in the default range for the type *b*.

Running signal functions. To *purely* run a signal function, Yampa provides the function *embed :: SF a b → (a, [(DTime, Maybe a)]) → [b]*, which allows to run an SF for a given number of steps where in each step one provides the Δt and an input *a*. The function then returns the output of the signal function for each step. Note that the input is optional, indicated by *Maybe*. In the first step at $t = 0$, the initial *a* is applied and whenever the input is *Nothing* in subsequent steps, the most recent *a* is re-used.

2.3 Arrowized programming

Yampa's signal functions are arrows, requiring us to program with arrows. Arrows are a generalisation of monads, which in addition to the already familiar parameterisation over the output type, allow parameterisation over their input type as well [22, 23].

In general, arrows can be understood to be computations that represent processes, which have an input of a specific type, process it and output a new type. This is the reason why Yampa is using arrows to represent their signal functions: the concept of processes, which signal functions are, maps naturally to arrows.

There exists a number of arrow combinators, which allow arrowized programing in a point-free style but due to lack of space we will not discuss them here. Instead we make use of Paterson's do-notation for arrows [32], which makes code more readable as it allows us to program with points.

To show how arrowized programming works, we implement a simple signal function, which calculates the acceleration of a falling mass on its vertical axis as an example [35].

```
fallingMass :: Double -> Double -> SF () Double
fallingMass p0 v0 = proc _ -> do
  v <- arr (+v0) <<< integral -< (-9.8)
  p <- arr (+p0) <<< integral -< v
  returnA -< p
```

To create an arrow, the *proc* keyword is used, which binds a variable after which the *do* of Patersons do-notation [32] follows. Using the signal function *integral :: SF v v* of Yampa, which integrates the input value over time using the rectangle rule, we calculate the current velocity and the position based on the initial position *p0* and velocity *v0*. The <<< is one of the arrow combinators, which composes two arrow computations and *arr* simply lifts a pure function into an arrow. To pass an input to an arrow, -< is used and <- to bind the result of an arrow computation to a variable. Finally to return a value from an arrow, *returnA* is used.

2.4 Monadic Stream Functions

Monadic Stream Functions (MSF) are a generalisation of Yampa's signal functions with additional combinators to control and stack side effects. An MSF is a polymorphic type and an evaluation function, which applies an MSF to an input and returns an output and a continuation, both in a monadic context [34]:

```
newtype MSF m a b = MSF {unMSF :: MSF m a b -> a -> m (b, MSF m a b)}
```

MSFs are also arrows, which means we can apply arrowized programming with Patersons do-notation as well. MSFs are implemented in Dunai, which is available on Hackage. Dunai allows us to apply monadic transformations to every sample by means of combinators like *arrM :: Monad m ⇒ (a → m b) → MSF m a b* and *arrM_ :: Monad m ⇒ m b → MSF m a b*. A part of the library Dunai is BearRiver, a wrapper, which re-implements Yampa on top of Dunai, which enables one to run arbitrary monadic computations in a signal function. BearRiver simply adds a monadic parameter *m* to each SF, which indicates the monadic context this signal function runs in.

To show how arrowized programming with MSFs works, we extend the falling mass example from above to incorporate monads. In this example we assume that in each step we want to accelerate our velocity *v* not by the gravity constant anymore but by a random number in the range of 0 to 9.81. Further we want to count the number of steps it takes us to hit the floor, that is when position *p* is less than or equal 0. Also when hitting the floor we want to print a debug message to the console with the velocity by which the mass has hit the floor and how many steps it took.

We define a corresponding monad stack with *IO* as the innermost Monad, followed by a *RandT* transformer for drawing random-numbers and finally a *StateT* transformer to count the number of steps we compute. We can access the monadic functions using *arrM* in case we need to pass an argument and *_arrM* in case no argument to the monadic function is needed:

```
type FallingMassStack g = StateT Int (RandT g IO)
type FallingMassMSF g   = SF (FallingMassStack g) () Double

fallingMassMSF :: RandomGen g => Double -> Double -> FallingMassMSF g
fallingMassMSF v0 p0 = proc _ -> do
  -- drawing random number for our gravity range
  r <- arrM_ (lift $ lift $ getRandomR (0, 9.81)) -< ()
  v <- arr (+v0) <<< integral -< (-r)
  p <- arr (+p0) <<< integral -< v
  -- count steps
  arrM_ (lift (modify (+1))) -< ()
  if p > 0
    then returnA -< p
    -- we have hit the floor
    else do
      -- get number of steps
      s <- arrM_ (lift get) -< ()
      -- write to console
      arrM (liftIO . putStrLn) -< "hit floor with v " ++ show v ++
                                  " after " ++ show s ++ " steps"
      returnA -< p
```

To run the *fallingMassMSF* function until it hits the floor we proceed as follows:

```
runMSF :: RandomGen g => g -> Int -> FallingMassMSF g -> IO ()
runMSF g s msf = do
  let msfReaderT = unMSF msf ()
      msfStateT  = runReaderT msfReaderT 0.1
      msfRand    = runStateT msfStateT s
      msfIO      = runRandT msfRand g
  (((p, msf'), s'), g') <- msfIO
  when (p > 0) (runMSF g' s' msf')
```

Dunai does not know about time in MSFs, which is exactly what BearRiver builds on top of MSFs. It does so by adding a *ReaderT Double*, which carries the Δt. This is the reason why we need one

Figure 1: States and transitions in the SIR compartment model.

extra lift for accessing *StateT* and *RandT*. Thus *unMSF* returns a computation in the *ReaderT Double* Monad, which we need to peel away using *runReaderT*. This then results in a *StateT Int* computation, which we evaluate by using *runStateT* and the current number of steps as state. This then results in another monadic computation of *RandT* Monad, which we evaluate using *runRandT*. This finally returns an *IO* computation, which we simply evaluate to arrive at the final result.

3 THE SIR MODEL

To explain the concepts of ABS and of our pure functional approach to it, we introduce the SIR model as a motivating example and use-case for our implementation. It is a very well studied and understood compartment model from epidemiology [25], which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [13].

In this model, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact with each other *on average* with a given rate of β per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus these interactions are ignored in this model. This definition gives rise to three compartments with the transitions seen in Figure 1.

This model was also formalized using System Dynamics (SD) [36]. In SD one models a system through differential equations, allowing to conveniently express continuous systems, which change over time, solving them by numerically integrating over time, which gives then rise to the dynamics. We won't go into detail here and provide the dynamics of such a solution for reference purposes, shown in Figure 2.

An Agent-Based approach

The approach of mapping the SIR model to an ABS is to discretize the population and model each person in the population as an individual agent. The transitions between the states are happening due to discrete events caused both by interactions amongst the agents and time-outs. The major advantage of ABS is that it allows to incorporate spatiality as shown in Section 4.3 and simulate heterogenity of population e.g. different sex, age. This is not possible with other simulation methods e.g. SD or Discrete Event Simulation [48].

According to the model, every agent makes *on average* contact with β random other agents per time unit. In ABS we can only contact discrete agents thus we model this by generating a random event on average every $\frac{1}{\beta}$ time units. We need to sample from an

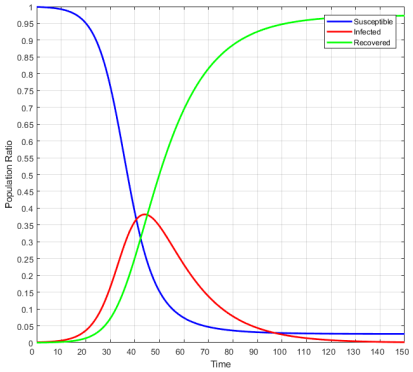


Figure 2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

exponential distribution because the rate is proportional to the size of the population [5]. Note that an agent does not know the other agents' state when making contact with it, thus we need a mechanism in which agents reveal their state in which they are in *at the moment of making contact*. This mechanism is an implementation detail, which we will derive in our implementation steps. For now we only assume that agents can make contact with each other somehow.

4 DERIVING A PURE FUNCTIONAL APPROACH

In [42] two fundamental problems of implementing an ABS from a programming-language agnostic point of view is discussed. The first problem is how agents can be pro-active and the second how interactions and communication between agents can happen. For agents to be pro-active, they must be able to perceive the passing of time, which means there must be a concept of an agent-process, which executes over time. Interactions between agents can be reduced to the problem of how an agent can expose information about its internal state which can be perceived by other agents.

Both problems are strongly related to the semantics of a model and the authors show that it is of fundamental importance to match the update-strategy with the semantics of the model - the order in which agents are updated and actions of agents are visible can make a big difference and need to match the model semantics. The authors identify four different update-strategies, of which the *parallel* update-strategy matches the semantics of the agent-based SIR model due to the underlying roots in the System Dynamics approach. In the parallel update-strategy, the agents act *conceptually* all at the same time in lock-step. This implies that they observe the same environment state during a time-step and actions of an agent are only visible in the next time-step - they are isolated from each other, see Figure 3.

Also, the authors [42] have shown the influence of different deterministic and non-deterministic elements in ABS on the dynamics and how the influence of non-determinism can completely break

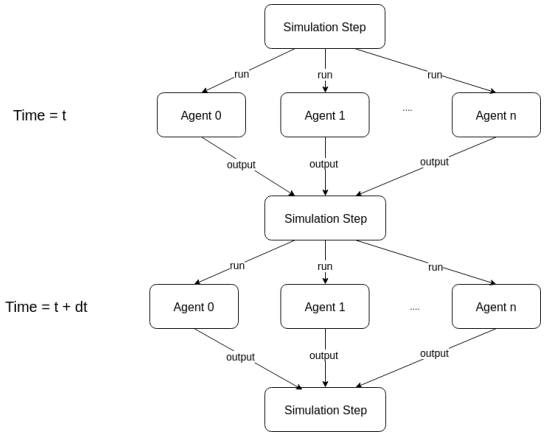


Figure 3: Parallel, lock-step execution of the agents.

them down or result in different dynamics despite same initial conditions. This means that we want to rule out any potential source of non-determinism, which we achieve by keeping our implementation pure. This rules out the use of the IO Monad and thus any potential source of non-determinism under all circumstances because we would lose all compile time guarantees about reproducibility. Still we will make use of the Random Monad, which indeed allows side-effects but the crucial point here is that we restrict side-effects only to this type in a controlled way without allowing general unrestricted effects like in traditional object-oriented approaches in the field.

In the following, we derive a pure functional approach for an ABS of the SIR model in which we pose solutions to the previously mentioned problems. We start out with a first approach in Yampa and show its limitations. Then we generalise it to a more powerful approach, which utilises Monadic Stream Functions, a generalisation of FRP. Finally we add a structured environment, making the example more interesting and showing the real strength of ABS over other simulation methodologies like System Dynamics and Discrete Event Simulation [1].

4.1 Functional Reactive Programming

As described in the Section 2.2, Arrowized FRP [22] is a way to implement systems with continuous and discrete time-semantics where the central concept is the signal function, which can be understood as a process over time, mapping an input- to an output-signal. Technically speaking, a signal function is a continuation which allows to capture state using closures and hides away the Δt, which means that it is never exposed explicitly to the programmer, meaning it cannot be messed with.

As already pointed out, agents need to perceive time, which means that the concept of processes over time is an ideal match for our agents and our system as a whole, thus we will implement them and the whole system as signal functions.

4.1.1 Implementation. We start by defining the SIR states as ADT and our agents as signal functions (SF) which receive the SIR states of all agents form the previous step as input and outputs the current SIR state of the agent. This definition, and the fact that Yampa is not monadic, guarantees already at compile, that the agents are isolated from each other, enforcing the *parallel* lock-step semantics of the model.

```
data SIRState = Susceptible | Infected | Recovered

type SIRAgent = SF [SIRState] SIRState

sirAgent :: RandomGen g => g -> SIRState -> SIRAgent
sirAgent g Susceptible = susceptibleAgent g
sirAgent g Infected    = infectedAgent g
sirAgent _ Recovered   = recoveredAgent
```

Depending on the initial state we return the corresponding behaviour. Note that we are passing a random-number generator instead of running in the Random Monad because signal functions as implemented in Yampa are not capable of being monadic.

We see that the recovered agent ignores the random-number generator because a recovered agent does nothing, stays immune forever and can not get infected again in this model. Thus a recovered agent is a consuming state from which there is no escape, it simply acts as a sink which returns constantly *Recovered*:

```
recoveredAgent :: SIRAgent
recoveredAgent = arr (const Recovered)
```

Next, we implement the behaviour of a susceptible agent. It makes contact *on average* with β other random agents. For every *infected* agent it gets into contact with, it becomes infected with a probability of γ. If an infection happens, it makes the transition to the *Infected* state. To make contact, it gets fed the states of all agents in the system from the previous time-step, so it can draw random contacts - this is one, very naive way of implementing the interactions between agents.

Thus a susceptible agent behaves as susceptible until it becomes infected. Upon infection an *Event* is returned, which results in switching into the *infectedAgent* SF, which causes the agent to behave as an infected agent from that moment on. When an infection event occurs we change the behaviour of an agent using the Yampa combinator *switch*, which is quite elegant and expressive as it makes the change of behaviour at the occurrence of an event explicit. Note that to make contact *on average*, we use Yampas *occasionally* function which requires us to carefully select the right Δt for sampling the system as will be shown in results.

Note the use of $iPre :: a \longrightarrow SF\ a\ a$, which delays the input signal by one sample, taking an initial value for the output at time zero. The reason for it is that we need to delay the transition from susceptible to infected by one step due to the semantics of the *switch* combinator: whenever the switching event occurs, the signal function into which is switched will be run at the time of the event occurrence. This means that a susceptible agent could make a transition to recovered within one time-step, which we want to prevent, because the semantics should be that only one state-transition can happen per time-step.

```
susceptibleAgent :: RandomGen g => g -> SIRAgent
susceptibleAgent g
    = switch
      -- delay switching by 1 step to prevent against transition
      -- from Susceptible to Recovered within one time-step
      (susceptible g >>> iPre (Susceptible, NoEvent))
      (const (infectedAgent g))
  where
    susceptible :: RandomGen g
```

[1] The code of all steps can be accessed freely through the following URL: https://github.com/thalerjonathan/phd/tree/master/public/purefunctionalepidemics/code

```
    => g -> SF [SIRState] (SIRState, Event ())
  susceptible g = proc as -> do
    makeContact <- occasionally g (1 / contactRate) () -< ()
    if isEvent makeContact
      then (do
        -- draw random element from the list
        a <- drawRandomElemSF g -< as
        case a of
          Infected -> do
            -- returns True with given probability
            i <- randomBoolSF g infectivity -< ()
            if i
              then returnA -< (Infected, Event ())
              else returnA -< (Susceptible, NoEvent)
          _        -> returnA -< (Susceptible, NoEvent))
      else returnA -< (Susceptible, NoEvent)
```

To deal with randomness in an FRP way, we implemented additional signal functions built on the *noiseR* function provided by Yampa. This is an example for the stream character and statefulness of a signal function as it allows to keep track of the changed random-number generator internally through the use of continuations and closures. Here we provide the implementation of *randomBoolSF*. *drawRandomElemSF* works similar but takes a list as input and returns a randomly chosen element from it:

```
randomBoolSF :: RandomGen g => g -> Double -> SF () Bool
randomBoolSF g p = proc _ -> do
  r <- noiseR ((0, 1) :: (Double, Double)) g -< ()
  returnA -< (r <= p)
```

An infected agent recovers *on average* after δ time units. This is implemented by drawing the duration from an exponential distribution [5] with $\lambda = \frac{1}{\delta}$ and making the transition to the *Recovered* state after this duration. Thus the infected agent behaves as infected until it recovers, on average after the illness duration, after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note that the infected agent ignores the states of the other agents as its behaviour is completely independent of them.

```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g
    = switch
      -- delay switching by 1 step
      (infected >>> iPre (Infected, NoEvent))
      (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

For running the simulation we use Yampas function *embed*:

```
runSimulation :: RandomGen g => g -> Time -> DTime
              -> [SIRState] -> [[SIRState]]
runSimulation g t dt as
    = embed (stepSimulation sfs as) ((), dts)
  where
    steps     = floor (t / dt)
    dts       = replicate steps (dt, Nothing)
    n         = length as
    (rngs, _) = rngSplits g n [] -- unique rngs for each agent
    sfs       = zipWith sirAgent rngs as
```

What we need to implement next is a closed feedback-loop - the heart of every agent-based simulation. Fortunately, [9, 29] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal functions and states of all agents, runs them all in parallel and returns this step's new agent states. Note the use of *notYet*, which is required because in Yampa switching occurs immediately at $t = 0$. If we don't delay the switching at $t = 0$ until the next step, we would enter an infinite switching loop - *notYet* simply delays the first switching until the next time-step.

```
stepSimulation :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
stepSimulation sfs as =
    dpSwitch
      -- feeding the agent states to each SF
      (\_ sfs' -> (map (\sf -> (as, sf)) sfs'))
      -- the signal functions
      sfs
      -- switching event, ignored at t = 0
      (switchingEvt >>> notYet)
      -- recursively switch back into stepSimulation
      stepSimulation
  where
    switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
    switchingEvt = arr (\ (_, newAs) -> Event newAs)
```

Yampa provides the *dpSwitch* combinator for running signal functions in parallel, which has the following type-signature:

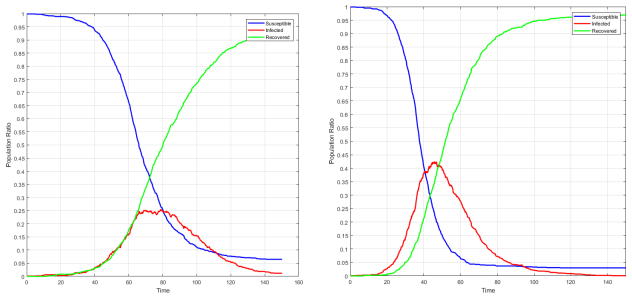
```
dpSwitch :: Functor col
         -- routing function
         => (forall sf. a -> col sf -> col (b, sf))
         -- SF collection
         -> col (SF b c)
         -- SF generating switching event
         -> SF (a, col c) (Event d)
         -- continuation to invoke upon event
         -> (col (SF b c) -> d -> SF a (col c))
         -> SF a (col c)
```

Its first argument is the pairing-function, which pairs up the input to the signal functions - it has to preserve the structure of the signal function collection. The second argument is the collection of signal functions to run. The third argument is a signal function generating the switching event. The last argument is a function, which generates the continuation after the switching event has occurred. *dpSwitch* returns a new signal function, which runs all the signal functions in parallel and switches into the continuation when the switching event occurs. The d in *dpSwitch* stands for decoupled which guarantees that it delays the switching until the next time-step: the function into which we switch is only applied in the next step, which prevents an infinite loop if we switch into a recursive continuation.

Conceptually, *dpSwitch* allows us to recursively switch back into the *stepSimulation* with the continuations and new states of all the agents after they were run in parallel.

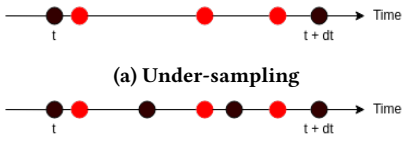
4.1.2 Results. The dynamics generated by this step can be seen in Figure 4.

By following the FRP approach we assume a continuous flow of time, which means that we need to select a *correct* Δt, otherwise we would end up with wrong dynamics. The selection of a correct Δt depends in our case on *occasionally* in the *susceptible* behaviour, which randomly generates an event on average with *contact rate* following the exponential distribution. To arrive at the correct dynamics, this requires us to sample *occasionally*, and thus the whole system, with small enough Δt which matches the frequency of events generated by *contact rate*. If we choose a too large Δt, we loose events, which will result in wrong dynamics as can be seen in Figure 4a. This issue is known as under-sampling and is described in Figure 5.



(a) $\Delta t = 0.1$ **(b)** $\Delta t = 0.01$

Figure 4: FRP simulation of agent-based SIR showing the influence of different Δt. Population size of 1,000 with contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps with respective Δt.



(a) Under-sampling

(b) Super-sampling

Figure 5: A visual explanation of under-sampling and super-sampling. The black dots represent the time-steps of the simulation. The red dots represent virtual events which occur at specific points in continuous time. In the case of under-sampling, 3 events occur in between the two time steps but *occasionally* only captures the first one. By increasing the sampling frequency either through a smaller Δt or super-sampling all 3 events can be captured.

For tackling this issue we have two options. The first one is to use a smaller Δt as can be seen in 4b, which results in the whole system being sampled more often, thus reducing performance. The other option is to implement super-sampling and apply it to *occasionally*, which would allow us to run the whole simulation with $\Delta t = 1.0$ and only sample the *occasionally* function with a much higher frequency.

4.1.3 Discussion. We can conclude that our first step already introduced most of the fundamental concepts of ABS:

- Time - the simulation occurs over virtual time which is modelled explicitly, divided into *fixed* Δt, where at each step all agents are executed.
- Agents - we implement each agent as an individual, with the behaviour depending on its state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.
- Stochasticity - it is an inherently stochastic simulation, which is indicated by the random-number generator and the usage of *occasionally*, *randomBoolSF* and *drawRandomElemSF*.
- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a surprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs *not* in the IO Monad. This guarantees that no external, uncontrollable sources of non-determinism can interfere with the simulation.
- Parallel, lock-step semantics - the simulation implements a *parallel* update-strategy where in each step the agents are run isolated in parallel and don't see the actions of the others until the next step.

Using FRP in the instance of Yampa results in a clear, expressive and robust implementation. State is implicitly encoded, depending on which signal function is active. By using explicit time-semantics with *occasionally* we can achieve extremely fine grained stochastics by sampling the system with small Δt: we are treating it as a truly continuous time-driven agent-based system.

A very severe problem, hard to find with testing but detectable with in-depth validation analysis, is the fact that in the *susceptible* agent the same random-number generator is used in *occasionally*, *drawRandomElemSF* and *randomBoolSF*. This means that all three stochastic functions, which should be independent from each other, are inherently correlated. This is something one wants to prevent under all circumstances in a simulation, as it can invalidate the dynamics on a very subtle level, and indeed we have tested the influence of the correlation in this example and it has an impact. We left this severe bug in for explanatory reasons, as it shows an example where functional programming actually encourages very subtle bugs if one is not careful. A possible but not very elegant solution would be to simply split the initial random-number generator in *sirAgent* three times (using one of the splited generators for the next split) and pass three random-number generators to *susceptible*. A much more elegant solution would be to use the Random Monad which is not possible because Yampa is not monadic.

So far we have an acceptable implementation of an agent-based SIR approach. What we are lacking at the moment is a general treatment of an environment and an elegant solution to the random number correlation. In the next step we make the transition to Monadic Stream Functions as introduced in Dunai [34], which allows FRP within a monadic context and gives us a way for an elegant solution to the random number correlation.

4.2 Generalising to Monadic Stream Functions

A part of the library Dunai is BearRiver, a wrapper which re-implements Yampa on top of Dunai, which should allow us to easily replace Yampa with MSFs. This will enable us to run arbitrary monadic computations in a signal function, solving our problem of correlated random numbers through the use of the Random Monad.

4.2.1 Identity Monad. We start by making the transition to BearRiver by simply replacing Yampas signal function by BearRivers',

which is the same but takes an additional type parameter *m*, indicating the monadic context. If we replace this type-parameter with the Identity Monad, we should be able to keep the code exactly the same, because BearRiver re-implements all necessary functions we are using from Yampa. We simply re-define the agent signal function, introducing the monad stack our SIR implementation runs in:

```
type SIRMonad = Identity
type SIRAgent = SF SIRMonad [SIRState] SIRState
```

4.2.2 Random Monad. Using the Identity Monad does not gain us anything but it is a first step towards a more general solution. Our next step is to replace the Identity Monad by the Random Monad, which will allow us to run the whole simulation within the Random Monad with the full features of FRP, finally solving the problem of correlated random numbers in an elegant way. We start by re-defining the SIRMonad and SIRAgent:

```
type SIRMonad g = Rand g
type SIRAgent g = SF (SIRMonad g) [SIRState] SIRState
```

The question is now how to access this Random Monad functionality within the MSF context. For the function *occasionally*, there exists a monadic pendant *occasionallyM* which requires a MonadRandom type-class. Because we are now running within a MonadRandom instance we simply replace *occasionally* with *occasionallyM*.

```
occasionallyM :: MonadRandom m => Time -> b -> SF m a (Event b)
-- can be used through the use of arrM and lift
randomBoolM :: RandomGen g => Double -> Rand g Bool
-- this can be used directly as a SF with the arrow notation
drawRandomElemSF :: MonadRandom m => SF m [a] a
```

4.2.3 Discussion. Running in the Random Monad solved the problem of correlated random numbers and elegantly guarantees us that we won't have correlated stochastics as discussed in the previous section. In the next step we introduce the concept of an explicit discrete 2D environment.

4.3 Adding an environment

So far we have implicitly assumed a fully connected network amongst agents, where each agent can see and 'knows' every other agent. This is a valid environment and in accordance with the System Dynamics inspired implementation of the SIR model but does not show the real advantage of ABS to situate agents within arbitrary environments. Often, agents are situated within a discrete 2D environment [15] which is simply a finite *NxM* grid with either a Moore or von Neumann neighbourhood (Figure 6). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their interactions directly from the shared read-only environment, which will be passed to the agents as input. This allows agents to read the states of all their neighbours, which tells them if a neighbour is infected or not. To show the benefit over the System Dynamics approach and for purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 6b).

We also implemented this spatial approach in Java using the well known ABS library RePast [30], to have a comparison with a state

(a) von Neumann **(b) Moore**

Figure 6: Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

of the art approach and came to the same results as shown in Figure 7. This supports, that our pure functional approach can produce such results as well and compares positively to the state of the art in the ABS field.

4.3.1 Implementation. We start by defining the discrete 2D environment for which we use an indexed two dimensional array. Each cell stores the agent state of the last time-step, thus we use the *SIRState* as type for our array data. Also, we re-define the agent signal function to take the structured environment *SIREnv* as input instead of the list of all agents as in our previous approach. As output we keep the *SIRState*, which is the state the agent is currently in. Also we run in the Random Monad as introduced before to avoid the random number correlation.

```
type Disc2dCoord = (Int, Int)
type SIREnv      = Array Disc2dCoord SIRState

type SIRAgent g  = SF (Rand g) SIREnv SIRState
```

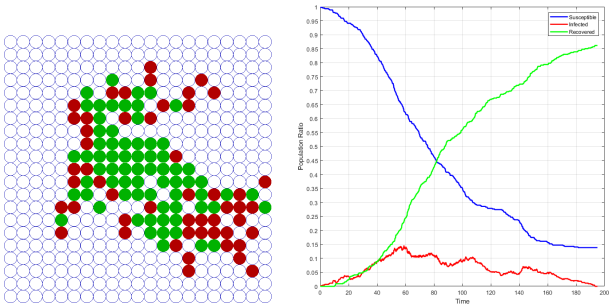
Note that the environment is not returned as output because the agents do not directly manipulate the environment but only read from it. Again, this enforces the semantics of the *parallel* update-strategy through the types where the agents can only see the previous state of the environment and see the actions of other agents reflected in the environment only in the next step.

Note that we could have chosen to use a StateT transformer with the *SIREnv* as state, instead of passing it as input, with the agents then able to arbitrarily read/write, but this would have violated the semantics of our model because actions of agents would have become visible within the same time-step.

The implementation of the susceptible, infected and recovered agents are almost the same with only the neighbour querying now slightly different.

Stepping the simulation needs a new approach because in each step we need to collect the agent outputs and update the environment for the next next step. For this we implemented a separate MSF, which receives the coordinates for every agent to be able to update the state in the environment after the agent was run. Note that we need use *mapM* to run the agents because we are running now in the context of the Random Monad. This has the consequence that the agents are in fact run sequentially one after the other but because they cannot see the other agents actions nor observe changes in the shared read-only environment, it is *conceptually* a *parallel* update-strategy where agents run in lock-step, isolated from each other at conceptually the same time.

```
simulationStep :: RandomGen g => [(SIRAgent g, Disc2dCoord)]
               -> SIREnv -> SF (Rand g) () SIREnv
simulationStep sfsCoords env = MSF (\_ -> do
  let (sfs, coords) = unzip sfsCoords
```



(a) Environment at $t = 50$ **(b) Dynamics over time**

Figure 7: Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 6b), a single infected agent at the center and same SIR parameters as in Figure 2. Simulation run until $t = 200$ with fixed $\Delta t = 0.01$. Last infected agent recovers around $t = 194$. The susceptible agents are rendered as blue hollow circles for better contrast.

```
  -- run agents sequentially but with shared, read-only environment
  ret <- mapM (`unMSF` env) sfs
  -- construct new environment from all agent outputs for next step
  let (as, sfs') = unzip ret
      env' = foldr (\ (a, coord) envAcc -> updateCell coord a envAcc)
                env (zip as coords)

      sfsCoords' = zip sfs' coords
      cont       = simulationStep sfsCoords' env'
  return (env', cont))

updateCell :: Disc2dCoord -> SIRState -> SIREnv -> SIREnv
```

4.3.2 Results. We implemented rendering of the environments using the gloss library which allows us to cycle arbitrarily through the steps and inspect the spreading of the disease over time visually as seen in Figure 7.

Note that the dynamics of the spatial SIR simulation, which are seen in Figure 7b look quite different from the reference dynamics of Figure 2. This is due to a much more restricted neighbourhood which results in far fewer infected agents at a time and a lower number of recovered agents at the end of the epidemic, meaning that fewer agents got infected overall.

4.3.3 Discussion. By introducing a structured environment with a Moore neighbourhood, we showed the ABS ability to place the heterogeneous agents in a generic environment, which is the fundamental advantage of an agent-based approach over other simulation methodologies and allows us to simulate much more realistic scenarios.

Note, that an environment is not restricted to be a discrete 2D grid and can be anything from a continuous N-dimensional space to a complex network - one only needs to change the type of the environment and agent input and provide corresponding neighbourhood querying functions.

4.4 Additional Steps

ABS involves a few more advanced concepts, which we don't fully explore in this paper due to lack of space. Instead we give a short overview and discuss them without presenting code or going into technical details.

4.4.1 Synchronous Agent Interactions. Synchronous agent interactions are necessary when an arbitrary number of interactions between two agents need to happen instantaneously within the same time-step. The use-case for this are price negotiations between multiple agents where each pair of agents needs to come to an agreement in the same time-step [15]. In object-oriented programming, the concept of synchronous communication between agents is implemented directly with method calls. We have implemented synchronous interactions in an additional step. We solved it pure functionally by running the signal functions of the transacting agent pair as often as their protocol requires but with $\Delta t = 0$, which indicates the instantaneous character of these interactions.

4.4.2 Event-Driven Approach. Our approach is inherently time-driven where the system is sampled with fixed Δt. The other fundamental way to implement an ABS in general, is to follow an event-driven approach [28], which is based on the theory of Discrete Event Simulation [48]. In such an approach the system is not sampled in fixed Δt but advanced as events occur, where the system stays constant in between. Depending on the model, in an event-driven approach it may be more natural to express the requirements of the model.

In an additional step we have implemented a rudimentary event-driven approach, which allows the scheduling of events. Using the flexibility of MSFs we added a State transformer to the monad stack, which allows queuing of events into a priority queue. The simulation is advanced by processing the next event at the top of the queue, which means running the MSF of the agent which receives the event. The simulation terminates if there are either no more events in the queue or after a given number of events, or if the simulation time has advanced to some limit. Having made the transition to MSFs, implementing this feature was quite straight forward, which shows the power and strength of the generalised approach to FRP using MSFs.

4.4.3 Conflicts in Environment. The semantics of the agent-based SIR model allowed a straight-forward implementation of the parallel update-strategy. This is not easily possible when there could be conflicts in the environment e.g. moving agents where only a single one can occupy a cell. Most models in ABS [15] solve this by implementing a *sequential* update-strategy [42], where agents are run after another but can already observe the changes by agents run before them in the same time-step. To prevent the introduction of artefacts due to a specific ordering, these models shuffle the agents before running them in each step to average the probability for a specific agent to be run at a fixed position.

It is possible to implement a *sequential* update-strategy using the State Monad but functional programming might offer other conflict resolving mechanisms as well because of immutable data and its different nature of side-effects. One approach could be to still run the agents isolated from each other without a State Monad but in case of conflicts, to randomly select a winner and re-run other conflicting agents signal functions until there is no more conflict. As long as the underlying monadic context is robust to re-runs, e.g. the Random Monad, this is no problem. We argue that

such an approach is conceptually and semantically cleaner and easier implemented in functional programming than in traditional object-oriented approaches.

5 RELATED WORK

The amount of research on using pure functional programming with Haskell in the field of ABS has been moderate so far. Most of the papers are related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [11, 24, 41].

The author of [4] investigated in his master thesis Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the ABS simulation package NetLogo, where agents run within the IO Monad and make use of Software Transactional Memory for a limited form of agent-interactions.

A library for DES and SD in Haskell called *Aivika 3* is described in the technical report [40]. It is not pure, as it uses the IO Monad under the hood and comes only with very basic features for event-driven ABS, which allows to specify simple state-based agents with timed transitions.

Using functional programming for DES was discussed in [24] where the authors explicitly mention the paradigm of FRP to be very suitable to DES.

A domain-specific language for developing functional reactive agent-based simulations was presented in [45]. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Yampa code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

Object-oriented programming and simulation have a long history together as the former one emereged out of Simula 67 [10] which was created for simulation purposes. Simula 67 already supported Discrete Event Simulation and was highly influential for today's object-oriented languages. Although the language was important and influential, in our research we look into different approaches, orthogonal to the existing object-oriented concepts.

Lustre is a formally defined, declarative and synchronous dataflow programming language for programming reactive systems [17]. While it has solved some issues related to implementing ABS in Haskell it still lacks a few important features necessary for ABS. We don't see any way of implementing an environment in Lustre as we do in our approach in Section 4.3. Also the language seems not to come with stochastic functions, which are but the very building blocks of ABS. Finally, Lustre does only support static networks, which is clearly a drawback for ABS in general where agents can be created and terminated dynamically during simulation.

There exists some research [12, 38, 44] of using the functional programming language Erlang [3] to implement ABS. The language is inspired by the actor model [1] and was created in 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The actor model can be seen as quite influential to the development of the concept of agents in ABS, which borrowed it from Multi Agent Systems [47]. It emphasises message-passing concurrency with share-nothing semantics, which maps nicely to functional programming concepts. The mentioned papers investigate how the actor model can be used to close the conceptual gap between agent-specifications, which focus on message-passing and their implementation. Further they also showed that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well. Despite the natural mapping of ABS concepts to such an actor language, it leads to simulations, which despite same initial starting conditions, might result in different dynamics each time due to concurrency.

6 CONCLUSIONS

Our FRP based approach is different from traditional approaches in the ABS community. First it builds on the already quite powerful FRP paradigm. Second, due to our continuous time approach, it forces one to think properly of time-semantics of the model and how small Δt should be. Third it requires one to think about agent interactions in a new way instead of being just method-calls.

Because no part of the simulation runs in the IO Monad and we do not use *unsafePerformIO* we can rule out a serious class of bugs caused by implicit data-dependencies and side-effects, which can occur in traditional imperative implementations.

Also we can statically guarantee the reproducibility of the simulation, which means that repeated runs with the same initial conditions are guaranteed to result in the same dynamics. Although we allow side-effects within agents, we restrict them to only the Random Monad in a controlled, deterministic way and never use the IO Monad, which guarantees the absence of non-deterministic side effects within the agents and other parts of the simulation.

Determinism is also ensured by fixing the Δt and not making it dependent on the performance of e.g. a rendering-loop or other system-dependent sources of non-determinism as described by [35]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [33, 35].

Also we showed how to implement the *parallel* update-strategy [42] in a way that the correct semantics are enforced and guaranteed already at compile time through the types. This is not possible in traditional imperative implementations and poses another unique benefit over the use of functional programming in ABS.

Issues

Currently, the performance of the system does not come close to imperative implementations. We compared the performance of our pure functional approach as presented in Section 4.3 to an implementation in Java using the ABS library RePast [30]. We ran the simulation until $t = 100$ on a 51x51 (2,601 agents) with $\Delta t = 0.1$ (unknown in RePast) and averaged 8 runs. The performance results make the lack of speed of our approach quite clear: the pure functional approach needs around 72.5 seconds whereas the Java RePast version just 10.8 seconds on our machine to arrive at $t = 100$. It must be mentioned, that RePast does implement an event-driven approach to ABS, which can be much more performant [28] than a time-driven one as ours, so the comparison is not completely valid. Still, we have already started investigating speeding up performance through the use of Software Transactional Memory [18, 19], which is quite straight forward when using MSFs. It shows very good

results but we have to leave the investigation and optimization of the performance aspect of our approach for further research as it is beyond the scope of this paper.

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile time, e.g. we can still have infinite loops and run-time errors. This was for example investigated in [37] where the authors use dependent types to avoid some run-time errors in FRP. We suggest that one could go further and develop a domain specific type system for FRP that makes the FRP based ABS more predictable and that would support further mathematical analysis of its properties. Furthermore, moving to dependent types would pose a unique benefit over the traditional object-oriented approach and should allow us to express and guarantee even more properties at compile time. We leave this for further research.

In our pure functional approach, agent identity is not as clear as in traditional object-oriented programming, where there is a quite clear concept of object-identity through the encapsulation of data and methods. Signal functions don't offer this strong identity and one needs to build additional identity mechanisms on top e.g. when sending messages to specific agents.

We can conclude that the main difficulty of a pure functional approach evolves around the communication and interaction between agents, which is a direct consequence of the issue with agent identity. Agent interaction is straight-forward in object-oriented programming, where it is achieved using method-calls mutating the internal state of the agent, but that comes at the cost of a new class of bugs due to implicit data flow. In pure functional programming these data flows are explicit but our current approach of feeding back the states of all agents as inputs is not very general. We have added further mechanisms of agent interaction which we had to omit due to lack of space.

7 FURTHER RESEARCH

We see this paper as an intermediary and necessary step towards dependent types for which we first needed to understand the potential and limitations of a non-dependently typed pure functional approach in Haskell. Dependent types are extremely promising in functional programming as they allow us to express stronger guarantees about the correctness of programs and go as far as allowing to formulate programs and types as constructive proofs, which must be total by definition [2, 27, 43].

So far no research using dependent types in agent-based simulation exists at all. In our next paper we want to explore this for the first time and ask more specifically how we can add dependent types to our pure functional approach, which conceptual implications this has for ABS and what we gain from doing so. We plan on using Idris [6] as the language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq.

We hypothesize that dependent types could help ruling out even more classes of bugs at compile time and encode invariants and model specifications on the type level, which implies that we don't need to test them using e.g. property-testing with QuickCheck. This would allow the ABS community to reason about a model directly in code. We think that a promising approach is to follow the work of [7, 8, 16] in which the authors utilize GADTs to implement an indexed monad, which allows to implementation correct-by-construction software.

- In the SIR implementation one could make wrong state-transitions e.g. when an infected agent should recover, nothing prevents one from making the transition back to susceptible.
  Using dependent types it should be possible to encode invariants and state-machines on the type level, which can prevent such invalid transitions already at compile time. This would be a huge benefit for ABS because many agent-based models define their agents in terms of state-machines.
- An infected agent recovers after a given time - the transition of infected to recovered is a timed transition. Nothing prevents us from *never* doing the transition at all.
  With dependent types we should be able to encode the passing of time in the types and guarantee on a type level that an infected agent has to recover after a finite number of time steps.
- In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent IDs to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation.
  Dependent types could be used to implement agent IDs as a proof that an agent with the given id exists *at the current time-step*. This also implies that such a proof cannot be used in the future, which is prevented by the type system as it is not safe to assume that the agent will still exist in the next step.
- In our implementation, we terminate the SIR model always after a fixed number of time-steps. We can informally reason that restricting the simulation to a fixed number of time-steps is not necessary because the SIR model *has to* reach a steady state after a finite number of steps. This means that at that point the dynamics won't change any more, thus one can safely terminate the simulation. Informally speaking, the reason for that is that eventually the system will run out of infected agents, which are the drivers of the dynamic. We know that all infected agents will recover after a finite number of time-steps *and* that there is only a finite source for infected agents which is monotonously decreasing.
  Using dependent types it might be possible to encode this in the types, resulting in a total simulation, creating a correspondence between the equilibrium of a simulation and the totality of its implementation. Of course this is only possible for models in which we know about their equilibria a priori or in which we can reason somehow that an equilibrium exists.

ACKNOWLEDGMENTS

The authors would like to thank I. Perez, H. Nilsson, J. Greensmith, M. Baerenz, H. Vollbrecht, S. Venkatesan, J. Hey and the anonymous

referees from IFL 2018 for constructive feedback, comments and valuable discussions.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Thorsten Altenkirch, Nils Anders Danielsson, Andres Loeh, and Nicolas Oury. 2010. Pi Sigma: Dependent Types Without the Sugar. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 40–55. https://doi.org/10.1007/978-3-642-12251-4_5

[3] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910

[4] Nikolaos Bezirgiannis. 2013. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism.* Ph.D. Dissertation. Utrecht University - Dept. of Information and Computing Sciences.

[5] Andrei Borshchev and Alexei Filippov. 2004. From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools. Oxford.

[6] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[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. 2016. *State Machines All The Way Down - An Architecture for Dependently Typed Applications.* Technical Report. https://www.idris-lang.org/drafts/sms.pdf

[9] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/871895.871897

[10] Ole-johan Dahl. 2002. The birth of object orientation: the simula languages. In *Software Pioneers: Contributions to Software Engineering, Programming, Software Engineering and Operating Systems Series.* Springer, 79–90.

[11] Tanja De Jong. 2014. *Suitability of Haskell for Multi-Agent Systems.* Technical Report. University of Twente.

[12] Antonella Di Stefano and Corrado Santoro. 2005. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '05)*. IEEE Computer Society, Washington, DC, USA, 679–685. https://doi.org/10.1109/IAT.2005.141

[13] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.

[14] Joshua M. Epstein. 2012. *Generative Social Science: Studies in Agent-Based Computational Modeling.* Princeton University Press. Google-Books-ID: 6jPiuMbKKJ4C.

[15] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA.

[16] Simon Fowler and Edwin Brady. 2014. Dependent Types for Safe and Secure Web Programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL '13)*. ACM, New York, NY, USA, 49:49–49:60. https://doi.org/10.1145/2620678.2620683

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320. https://doi.org/10.1109/5.97300

[18] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[19] Tim Harris and Simon Peyton Jones. 2006. Transactional memory with data invariants. https://www.microsoft.com/en-us/research/publication/transactional-memory-data-invariants/

[20] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, Johan Jeuring and Simon L. Peyton Jones (Eds.). Number 2638 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[22] John Hughes. 2000. Generalising Monads to Arrows. *Sci. Comput. Program.* 37, 1-3 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[23] John Hughes. 2005. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming (AFP'04)*. Springer-Verlag, Berlin, Heidelberg, 73–129. https://doi.org/10.1007/11546382_2

[24] Peter Jankovic and Ondrej Such. 2007. *Functional Programming and Discrete Simulation.* Technical Report.

[25] W. O. Kermack and A. G. McKendrick. 1927. A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 115, 772 (Aug. 1927), 700–721. https://doi.org/10.1098/rspa.1927.0118

[26] C. M. Macal. 2016. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation* 10, 2 (May 2016), 144–156. https://doi.org/10.1057/jos.2016.7

[27] James McKinna. 2006. Why Dependent Types Matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1111037.1111038

[28] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

[29] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

[30] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* 1, 1 (March 2013), 3. https://doi.org/10.1186/2194-3206-1-3

[31] Michael J. North and Charles M. Macal. 2007. *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation.* Oxford University Press, USA. Google-Books-ID: gRATDAAAQBAJ.

[32] Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/507635.507664

[33] Ivan Perez. 2017. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/3122955.3122957

[34] Ivan Perez, Manuel Baerenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

[35] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 2:1–2:27. https://doi.org/10.1145/3110246

[36] Donald E. Porter. 1962. Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18. *Science* 135, 3502 (Feb. 1962), 426–427. https://doi.org/10.1126/science.135.3502.426-a

[37] Neil Sculthorpe and Henrik Nilsson. 2009. Safe Functional Reactive Programming Through Dependent Types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/1596550.1596558

[38] Gene I. Sher. 2013. *Agent-Based Modeling Using Erlang Eliminating The Conceptual Gap Between The Programming Language & ABM.*

[39] Peer-Olaf Siebers and Uwe Aickelin. 2008. Introduction to Multi-Agent Simulation. *arXiv:0803.3905 [cs]* (March 2008). http://arxiv.org/abs/0803.3905 arXiv: 0803.3905.

[40] David Sorokin. 2015. *Aivika 3: Creating a Simulation Library based on Functional Programming.*

[41] Martin Sulzmann and Edmund Lam. 2007. *Specifying and Controlling Agents in Haskell.* Technical Report.

[42] Jonathan Thaler and Peer-Olaf Siebers. 2017. The Art Of Iterating: Update-Strategies in Agent-Based Simulation. Dublin.

[43] Simon Thompson. 1991. *Type Theory and Functional Programming.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[44] Carlos Varela, Carlos Abalde, Laura Castro, and Jose Gulías. 2004. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang (ERLANG '04)*. ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/1022471.1022481

[45] Ivan Vendrov, Christopher Dutchyn, and Nathaniel D. Osgood. 2014. Frabjous A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, William G. Kennedy, Nitin Agarwal, and Shanchieh Jay Yang (Eds.). Number 8393 in Lecture Notes in Computer Science. Springer International Publishing, 385–392. https://doi.org/10.1007/978-3-319-05579-4_47

[46] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 242–252. https://doi.org/10.1145/349299.349331

[47] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

[48] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press. Google-Books-ID: REzmYOQmHuQC.

Delta Debugging Type Errors with a Blackbox Compiler

Joanna Sharrad
University of Kent
Canterbury, UK
jks31@kent.ac.uk

Olaf Chitil
University of Kent
Canterbury, UK
oc@kent.ac.uk

Meng Wang
University of Bristol
Bristol, UK
meng.wang@bristol.ac.uk

ABSTRACT

Debugging type errors is a necessary process that programmers, both novices and experts alike, face when using statically typed functional programming languages. All compilers often report the location of a type error inaccurately. This problem has been a subject of research for over thirty years. We present a new method for locating type errors: We apply the Isolating Delta Debugging algorithm coupled with a blackbox compiler. We evaluate our implementation for Haskell by comparing it with the output of the Glasgow Haskell Compiler; overall we obtain positive results in favour of our method of type error debugging.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Program analysis**;

KEYWORDS

Type Error, Error diagnosis, Blackbox, Delta Debugging, Haskell

ACM Reference Format:
Joanna Sharrad, Olaf Chitil, and Meng Wang. 2018. Delta Debugging Type Errors with a Blackbox Compiler. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/3310232.3310243

1 INTRODUCTION

Compilers for Haskell, OCaml and many other statically typed functional programming languages produce type error messages that can be lengthy, confusing and misleading, causing the programmer hours of frustration during debugging. One role of such a type error message is to tell the programmer the location of a type error within the ill-typed program. Although there has been over thirty years of research [8, 22] on how to improve the way we locate type conflicts and present them to the programmer, type error messages can be misleading. We can trace the cause of inaccurate type error location to an advanced feature of functional languages: type inference. A typical Haskell or OCaml program contains only little type information: definitions of data types, some type signatures for top-level functions and possibly a few more type annotations.

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$$15.00
https://doi.org/10.1145/3310232.3310243

Type inference works by generating constraints for the type of every expression in the program and solving these constraints. An ill-typed program is just a program with type constraints that have no solution. Because the type checker cannot know which program parts and thus constraints are correct, that is, agree with the programmer's intentions, it may start solving incorrect constraints and therefore assume wrong types early on. Eventually, the type checker may note a type conflict when considering a constraint that is actually correct.

1.1 Variations of an Ill-Typed Programs

Consider the following Haskell program from Stuckey et al. [17]:

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3                 | otherwise = x : y : ys
```

The program defines a function that shall insert an element into an ordered list, but the program is ill-typed. Stuckey et al. state that the first line is incorrect and should instead look like below:

```
1 insert x [] = [x]
```

The Glasgow Haskell Compiler (GHC) version 8.2.2 wrongly gives the location of the type error as (part of) line two.

```
2 insert x (y:ys) | x > y    = y : insert x ys
```

Let us see how GHC comes up with this wrong location. GHC derives type constraints and immediately solves them as far as possible. It roughly traverses our example program line by line, starting with line 1. The type constraints for line 1 are solvable and yield the information that `insert` is of type $\alpha \rightarrow [\beta] \rightarrow \alpha$. Subsequently in line 2 the expression `x > y` yields the type constraint that `x` and `y` must have the same type, so together with the constraints for the function arguments `x` and `(y:ys)`, GHC concludes that `insert` must be of type $\alpha \rightarrow [\alpha] \rightarrow \alpha$. Finally, the occurrence of `insert x ys` as subexpression of `y : insert x ys` means that the result type of insert must be the same list type as the type of its second argument. So `insert x ys` has both type $[\alpha]$ and type α, a contradiction reported as type error.

Our program contains no type annotations or signature, meaning we have to infer all types. Surely adding a type signature will ensure that GHC returns the desired type error location? Indeed for

```
1 insert :: Ord a => a -> [a] -> [a]
2 insert x [] = x
3 insert x (y:ys) | x > y     = y : insert x ys
4                 | otherwise = x : y : ys
```

GHC identifies the type error location correctly:

```
2 insert x [] = x
```

However, a recent study showed that type signatures are often wrong, causing 30% of all type errors [23]! GHC trusts that a given type signature is correct and hence for

```
1 insert :: Ord a => a -> [a] -> a
2 insert x [] = x
3 insert x (y:ys) | x > y     = y : insert x ys
4                 | otherwise = x : y : ys
```

GHC wrongly locates the cause in line 2 again:

```
2 insert x (y:ys) | x > y   = y : insert x ys
```

In summary we see that the order in which type constraints are solved determines the reported type error location. There is no fixed order to always obtain the right type error location and requiring type annotations in the program does not help.

As a consequence researchers developed type error slicing [7, 16], which determines a minimal unsatisfiable type constraint set and reports all program parts associated with these constraints as type error slice. However, practical experience showed that these type error slices are often quite big [7] and thus they do not provide the programmer with sufficient information for correcting the type error. Our aim is to determine a smaller type error location, a single line in the program. [1]

1.2 Our Method

Our method is based on the way programmers systematically debug errors without additional tools. The programmer removes part of the program or adds previously removed parts back in. They check for each such variant of the program whether the error still exists or has gone. By doing this systematically, the programmer can determine a small part of the program as the cause of the error.

This general method was termed *Delta Debugging* by Zeller [24]. Specifically, we apply the *Isolating Delta Debugging algorithm*, which determines two variants of the original program that capture a minimal difference between a correct and an erroneous variant of the program. Eventually our method produces the following result:

Result of our type error location method

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3                 | otherwise = x : y : ys
```

This program listing with different highlighting shows that the type error location is in line 1 and that line 1 and 2 together cause the type error; that is, even without line 3 this program is ill-typed.

The Isolating Delta Debugging algorithm has two prerequisites: an input that can be modified repeatedly and a means of inquiring whether these modifications were successful. We fulfil the first prerequisite by employing the raw source code of the programmer's ill-typed program. We work directly on the program text rather than the abstract syntax tree. We make modifications that generate new variants of the program ready for testing to see whether they remain ill-typed. To examine whether they are indeed ill-typed or not, we employ the compiler as a black box. We do not use any location information included in any type error message of the compiler. This black box satisfies the second prerequisite of the Isolating Delta Debugging algorithm.

Once implemented in our tool Gramarye that works on Haskell programs and uses the Glasgow Haskell Compiler as blackbox, we can apply our method to any ill-typed program, no matter how many type errors it contains, to locate one type error. Once our approach has the correct location, the programmer can fix it and reuse the tool to find further type errors.

We evaluated Gramarye against the Glasgow Haskell Compiler using thirty programs containing single type errors and 870 programs generated to include two type errors.

Our paper makes the following contributions:

- We describe how to apply the Isolating Delta Debugging algorithm to type errors (Section 2).
- We use the compiler as a true black box; it can easily be replaced by a different compiler (Section 3.2).
- We implement the method in a tool called Gramarye that directly manipulates Haskell source code (Section 3.3).
- We evaluate our method against the Glasgow Haskell Compiler (Section 4).

Our evaluation shows an improvement in reporting type errors for many programs and demonstrates that our approach has promise in the field of type error debugging.

2 AN ILLUSTRATION OF OUR METHOD

Figure 1 gives an overview of the Gramarye framework. It indicates the steps taken to locate type errors in an ill-typed program.

We start with a single ill-typed Haskell program. This program must contain a type error; otherwise we reject it. Here we work with the original ill-typed program of the Introduction.

From this program, we obtain two programs that the *Isolating Delta Debugging* algorithm will work with. One is the ill-typed program, from which the algorithm removes lines that are irrelevant for the type error. The algorithm aims to minimise this program. The other program is the empty program, which is definitely well-typed. The algorithm moves lines from the ill-typed program to the well-typed program; the algorithm aims to maximise the well-typed program. So we start with:

Step 1: well-typed program

```
1
2
3
```

Step 1: ill-typed program

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3                 | otherwise = x : y : ys
```

[1] There is no fair comparison of size. Our method determines a line as error cause, no matter how long the line. All existing type error slicing algorithms produce slices built from small subexpressions; usually, however, these subexpressions are distributed over many, not necessarily adjacent, lines.

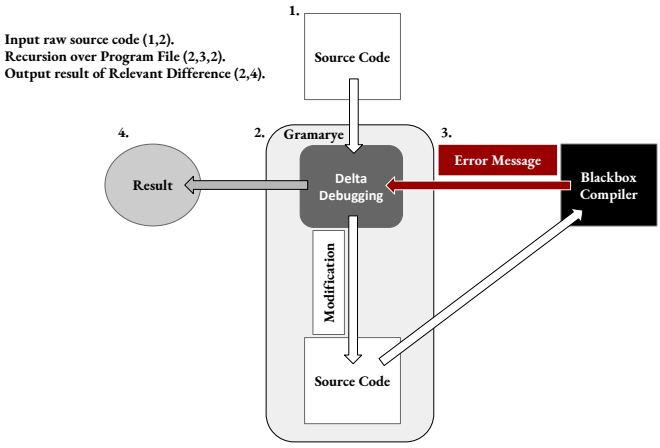


Figure 1: The Gramarye Framework

Now we move a line from the ill-typed program to the well-typed program. We pick line 3, obtaining two new program variants:

Step 1: modified well-typed program

```
1
2
3                | otherwise = x : y : ys
```

Step 1: modified ill-typed program

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3
```

We then send these two programs to the black box compiler for type checking:

- Step 1: modified well-typed program: unresolved.
- Step 1: modified ill-typed program: ill-typed.

The modified well-typed program is not a syntactically valid Haskell program; the compiler yields a parse error. So our black box compiler may yield one of three possible results:

(1) unresolved; compiler yields a non-type error
(2) ill-typed; compiler yields a type error
(3) well-typed; compilation successful

We cannot use an unresolved program for locating a type error, but each of the other two possible results are useful. Our modified ill-typed program is smaller than our original ill-typed program. We now know that the modified variant is ill-typed too, so we can replace our ill-typed program for the next step:

Step 2: well-typed program

```
1
2
3
```

Step 2: ill-typed program

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3
```

The algorithm now repeats: Again we move a single line from the ill-typed program to the well-typed program. Let us pick line 2:

Step 2: modified well-typed program

```
1
2 insert x (y:ys) | x > y     = y : insert x ys
3
```

Step 2: modified ill-typed program

```
1 insert x [] = x
2
3
```

Again we type check these two programs:

- Step 2: modified well-typed program: well-typed.
- Step 2: modified ill-typed program: well-typed.

Because both variants are well-typed and larger than the previous well-typed program, we can use either of them as new well-typed program. We pick the modified well-typed program and thus obtain;

Step 3: well-typed program

```
1
2 insert x (y:ys) | x > y     = y : insert x ys
3
```

Step 3: ill-typed program

```
1 insert x [] = x
2 insert x (y:ys) | x > y     = y : insert x ys
3
```

The well-typed and ill-typed programs differ by only a single line, and hence our algorithm terminates.

The final result is that the difference between well-typed and ill-typed program, here line 1, is the location of the type error. Because the ill-typed program contains only lines 1 and 2 of the original program, we also know that only lines 1 and 2 are needed to make the program ill-typed. Thus we obtain the output shown in the Introduction. If we want to add a compiler type error message for further explanations, we can pick the one we received for the minimised ill-typed program. The error message may be clearer than for the original, larger program.

The isolating delta debugging method is non-deterministic. Often different choices lead to the same final result, but not always. Zeller argues that this non-determinism does not matter and that one result provides insightful debugging information to the programmer [25]. Hence his algorithm is deterministic, making arbitrary choices. Our implementation follows his algorithm and for our example makes the choices described here.

The algorithm is based on an ordering of programs, where a program is just a sequence of strings. A program P_1 is less or equal a program P_2 if they have the same number of lines and for every line, the line content is either the same for both programs, or the line is empty in P_1. All programs that we consider are between the well-typed and ill-typed programs that we start with. The final well-typed and ill-typed programs have minimal distance, that is, they either differ by just one line or programs between them are unresolved; that is, they are not syntactically valid programs.

In this example, in each step, we moved only a single line from the ill-typed to the well-typed program. For programs with hundreds of lines, this simple approach would be expensive in time due to the number of programs needing consideration. Hence we use the full Isolating Delta Debugging algorithm which starts with moving either the first or second half of the program from the ill-typed to the well-typed program. If both modified programs are unresolved, then we increase the granularity of modifications from moving half the program to moving a quarter of the program. In general, every time both modified programs are unresolved, we half the size of our modifications. This increase of granularity can continue until only a single line is modified.

Zeller analysed the complexity of the Isolating Delta Debugging algorithm. In our case complexity is the number of calls to the black box compiler in relation to the number of lines of the original program. In the worst case, if most calls yield unresolved, the number of calls is quadratic. In the best case, when no call yields unresolved, the number of calls is logarithmic [25].

3 IMPLEMENTATION

As illustrated in Figure 1, our Gramarye tool has four components;

- Delta Debugging.
- Blackbox Compiler.
- Source Code Modification.
- Result Processing.

We shall next describe each of the components in greater detail.

3.1 Delta Debugging

Zeller [6, 24–26] defined *Delta Debugging*, a debugging method that systematically applies the scientific approach of *Hypothesis-Test-Result*. When programmers debug, they first use the error message to conjecture a possible cause (hypothesis), then modify the program and recompile (test), and lastly use the outcome of the recompilation (result) to either repeat with a new, improved hypothesis, or terminate with the hypothesis having been proved.

Zeller does not only consider locating a cause in a defective program, but alternatively locating a cause in an input to a program that causes a failure at runtime or locating a cause in the runtime state of a program execution. He abstracts program/input/state by talking about configurations and differences between configurations. In our method a configuration is a program and a difference is a removal of some lines (replacement by empty lines). Our programs are input for the type checker, but the defect(s) that we look for are not in the type checker, but the program.

Essential for delta debugging is the existence of a testing function [24] that places a configuration into one of the following three categories:

(1) Unresolved (?).
(2) Fail (×).
(3) Pass (√).

For our method the testing function is the type checker of the compiler. We use domain specific terminology for the three categories:

(1) Unresolved (any non-type error).
(2) Ill-typed.
(3) Well-typed.

Zeller presents two delta debugging methods. He refers to them as Simplifying and Isolating [25].

3.1.1 Simplifying Delta Debugging. The algorithm determines a smaller variant of the given faulty (ill-typed) program. The result is minimal in that removing any further line makes the program pass (well-typed). The algorithm works by removing parts of a failing program until it no longer fails. The last failing variant of the program is the result of the algorithm. The minimality allows us to surmise that all parts of the program left must contribute to the error. The Simplifying Delta Debugging algorithm has the same disadvantage as program slicing algorithms for type errors [7, 16]: it often reports a rather large program. The second Delta Debugging algorithm, *Isolating*, aims to reduce the size of the reported program slice further.

3.1.2 Isolating Delta Debugging. Isolating Delta Debugging employs the Simplifying algorithm to generate a minimal faulty (ill-typed) program. At the same time the algorithm also produces a maximal passing (well-typed) program. The maximal program is created by taking a passing program (usually starting with the empty program) and adding lines from the faulty program until the program is faulty. The difference between the maximal passing and the minimal faulty program is then considered as cause of the fault. We chose the Isolating Delta Debugging algorithm, because this difference is substantially smaller than a minimal faulty program. Furthermore, Zeller states that in practice the Isolation algorithm is much more efficient than the Simplification algorithm [25].

3.1.3 Granularity. The Isolating Delta Debugging algorithm consists of two parts: granularity and moving of program parts. In our initial illustration of our method we moved individual lines; however, within the algorithm a granularity parameter determines how many lines of code we move between our ill-typed and well-typed programs: the number of lines is difference in the number of lines of the two programs divided by the granularity. A granularity of 2 means that the algorithm resembles a binary chop algorithm, it repeatedly divides the ill-typed program in half. The Isolating Delta Debugging algorithm starts with granularity set to 2, but depending on the results of the testing function, granularity can grow and shrink.

To understand granularity we apply the algorithm to an eight-line program that has a type error on line 8. Only when the testing function returns 'unresolved', the granularity may increase. Hence we assume that lines 1 to 3 and lines 4 and 5 of our program belong together: any program that contains only some lines of these two sets yields 'unresolved'.

At step 1 our well-typed, respectively ill-typed, program have the following lines:

```
{}  {1,2,3,4,5,6,7,8}
```

Because granularity is 2 and the two programs differ by 8 lines, we move the first 8/2 = 4 lines from the ill-typed program to the well-typed one and then test both modified programs:

```
{1,2,3,4}?  {5,6,7,8}?
```

Both programs are unresolved. We cannot move any other 4 lines (we only ever move adjacent lines). Hence granularity is doubled from 2 to 4. So now we move 8/4 = 2 lines from the step 1 ill-typed program to the well-typed program. We first try

```
{1,2}?  {3,4,5,6,7,8}?
```

Both are unresolved, so we try the next two lines:

```
{3,4}?  {1,2,5,6,7,8}?
```

Again both are unresolved, so we continue with another two lines:

```
{5,6}?  {1,2,3,4,7,8}?
```

Still unresolved, so we try:

```
{7,8}×  {1,2,3,4,5,6}√
```

Finally our test function gives a different results. We reset granularity to 2. We select the well-typed program for Step 2; thus we have now the following well-typed and ill-typed program:

```
{1,2,3,4,5,6}  {1,2,3,4,5,6,7,8}
```

The two programs differ by 2 lines. We move 2/2 = 1 line from the ill-typed program to the well-typed program:

```
{1,2,3,4,5,6,7}√  {1,2,3,4,5,6,8}×
```

The first program is well-typed, the second ill-typed. Granularity stays at 2. We select the ill-typed program for Step 3:

```
{1,2,3,4,5,6}  {1,2,3,4,5,6,8}
```

Because the two programs differ only by one line, our algorithm stops. We have identified the single line difference, line 8, as the cause of the type error.

3.1.4 Choices. We noted already in Section 2 that in principle delta debugging is non-deterministic, but we follow Zeller's deterministic implementation [25]. In the preceding section on granularity we stated how the algorithm changes granularity and in which order the algorithm moves lines between the well-typed and ill-typed program. Finally Algorithm 1 shows how the test results for the modified well-typed and ill-typed program decide the choice of the next well-typed and ill-typed program.

ALGORITHM 1: Choices in Isolating Delta Debugging

```
testModProgWell = test(modProgWell)
testModProgIll = test(modProgIll)
if testModProgIll == IllTyped && granularity == 2 then
  | progIll = modProgIll
else if testModProgIll == WellTyped then
  | progWell = modProgIll
else if testModProgWell == IllTyped then
  | progIll = modProgWell
else if testModProgIll == IllTyped then
  | progIll = modProgIll
else if testModProgWell == WellTyped then
  | progWell = modProgWell
- - else: both modified programs are unresolved
try another program modification or terminate
```

3.2 A Blackbox Compiler

We use a compiler as a blackbox, an entity of which we only know the input and the output. Anything that happens within the blackbox remains a mystery to us. Compilers naturally lend themselves to this usage, taking an input (source code), and returning an output: a successfully compiled program or an error. The compiler we chose to use as a blackbox is the Glasgow Haskell Compiler (GHC), which is widely used by the Haskell community. As we can exploit GHC to gather type checking information without the need to alter the compiler itself, we can keep our tool separate. Not modifying the compiler has many benefits; changes made by the compiler developers will not affect the way our method works, users of our tool can avoid downloading a specialist compiler, or having the hassle of patching an existing one. Avoiding modification of the compiler also means our method is not restricted to the Haskell language, giving scope to expand to other functional languages.

We use our blackbox compiler as a type checker. In each iteration of the Isolating Delta Debugging algorithm, we determine the status of our modified ill-typed and well-typed programs as described in Section 2. When using the blackbox compiler, our tool receives the same output a programmer would when they are using GHC. Though the result of compiling with GHC gives a message that includes many details, we are only interested in whether our programs are well-typed, using this information to categorise as

we discuss in Section 3.1. Depending on the categories returned, we modify the source code of our programs in different ways, and again send them to the blackbox compiler for further type checking.

3.3 Source Code Manipulation

When programmers manually debug, they edit their source code directly, looking at where the error is suggested to occur and making changes in the surrounding area. We are also directly manipulating the source code, modifying our programs using the line numbers determined by the Isolating Delta Debugging algorithm. One significant bonus to the strategy of directly changing the source code is that it keeps our approach very simple. As we do not work on the Abstract Syntax Tree (AST) we do not need to parse our source code with each modification, allowing us to avoid making changes to an existing compiler or creating our own parser. Not editing the AST also means we can stay true to the programmer's original program, keeping personal preferences in layout intact by using empty lines as placeholders. In future work this will allow us to provide error messages that refer to the original program.

3.4 Processing the Results

The idea is that if one program is well-typed and the other ill-typed, then the source of the type error lies within the difference of the two; the relevant difference [25]. After the Isolating Delta Debugging algorithm has completed, two programs are left. If a line number does not make an appearance in both of these programs, then we report it as a relevant difference. Reporting whole lines also means that we can easily evaluate how successful we are in locating type errors.

4 EVALUATION

In Section 2 we have shown how we can successfully locate the correct line number of a type error. However, though positive for the example program we have used throughout, a more thorough evaluation was needed to determine the strength of our method in type error locating.

We chose to evaluate our method, implemented in our debugger Gramayre, against a benchmark of programs specially engineered to contain type errors. The programs collated by Chen and Erwig [3] were used to evaluate their Counter-Factual approach to type error debugging. In all, there are 121 programs in the CE benchmark, but not all had what Chen and Erwig called the 'oracle', the knowledge of where the type error lay. Though a program could have many correct solutions to remove a type error, we needed to know the correct location of where the type error occurred to evaluate accurately; so we removed all programs that did not specify the exact cause. To make our evaluation more compact, programs that were ill-typed in similar ways were also removed, reducing our set of test programs to thirty. However, we also wanted to see whether our method could report multiple type errors. To do so we took each pair of different programs from our 30 test programs, and joined the two programs together. Thus we generated a further 870 programs for the evaluation.

Our evaluation answers the following questions;

(1) We apply our method to Haskell programs that each contain a single type error. Do we see improvement in locating the errors compared to the Glasgow Haskell Compiler? (Section 4.1)
(2) We apply our method to Haskell programs that each contain two type errors. Do we see improvement in locating these errors compared to the Glasgow Haskell Compiler? (Section 4.2)
(3) Does our method return a smaller set of type error locations compared to the Glasgow Haskell Compiler? (Section 4.3)

We chose to compare our approach against GHC 8.2.2. We are using GHC as a blackbox compiler within our own tool, but as we use it solely as a type checker we do not use the line numbers that it reports, and thus these line numbers have no interference with our evaluation. GHC and our tool take the CE benchmarks, and type check each one; this results in a set of line numbers suggested as cause of the type error. To judge the success of locating the type error in the tests we have chosen to use the same criterion as Wand [22]. Wand states that even if we get multiple locations returned, the method is classed as a success if the exact location of the type error is within these. As both our tool and GHC can report multiple line numbers for one type error; we use Wand's criterion to allow us to take into consideration all line numbers returned, and not just the first.

4.1 Singular Type Error Evaluation

(1) We apply our method to Haskell programs that each contain a single type error. Do we see improvement in locating the errors compared to the Glasgow Haskell Compiler?

Each program of our first set contains one single type error; if the line number reported matches the 'oracle' response, then our result is accurate. The graph in Figure 2 shows for all 30 ill-typed programs whether Gramarye and GHC correctly discover the position of the type error. The results are positive. Out of the 30 ill-typed programs we accurately locate 23 (77%) of the type errors, compared to 15(50%) for GHC.

In some cases, multiple line numbers were returned. The primary cause of multiple line numbers are large expressions, especially those consisting of several key words, such as If-Then-Else or Let-In expressions. For example, in

Listing 1: Expression over two lines

```
1 doRow (y:ys) r = (if y < r && y > (r-dy) then '*'
2                           else ' ') : doRow r ys
```

our tool identifies both lines as the cause of the type error. In this example GHC wrongly suggests the first line as causing the error. These large expressions are one area for future work.

The evaluation also drew attention to two individual programs that needed more investigating. Initially our debugger failed to discover a type error in program 24 yet GHC did successfully. On inspection program 24 contained a mistake in the original benchmark programs. The program contained two type errors, and not the singular error we were expecting. Removal of one of the type errors returned the results we would expect, with both our debugger and GHC successfully locating the errors. Due to this anomalies we decided to check all programs for multiple type errors. Program 9 also contained two errors. Once fixed our debugger was no

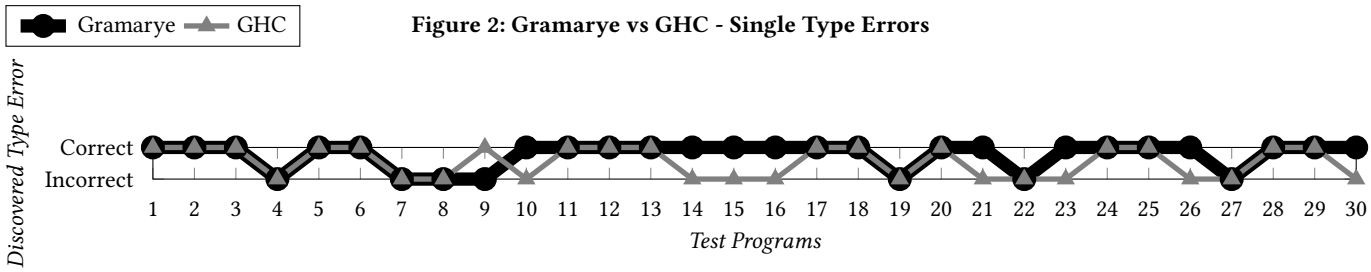


Figure 2: Gramarye vs GHC - Single Type Errors

longer successful in finding the type error. This is due to Program 9 containing an unnecessary call to a variable bound to foldl.

Listing 2: Program 9

```
1 foldleft = foldl
2 intList = [12, 3]
3 zero = 0.0
4 addReciprocals total i = total + (1.0 / i)
5 totalOfReciprocals = foldleft zero addReciprocals intList
```

Our debugger returns line 1 as faulty, yet the type error is on line 5 where the variables 'zero' and 'addReciprocals' should be swapped. If we instead use foldl directly rather than via a variable, our debugger then finds the correct broken line number.

In singular discovery our method has a 27 percentage point success rate over GHC when locating type errors in Haskell source.

4.2 Multiple Type Errors Evaluation

(2) We apply our method to Haskell programs that each contain two type errors. Do we see improvement in locating these errors compared to the Glasgow Haskell Compiler?

To evaluate the locating of multiple errors we merged our singular programs. This gave us programs that each had two self-contained type errors. Self-contained can be described as having two separate functions that do not interact with each other, however both functions containing a single type error. In Listing 3, the first function has an error on line 2 and the second function on line 6, but neither type error affects the other;

Listing 3: Multiple Type Error Example

```
1 addList ls s = if s `elem` ls then ls else s : ls
2 v5 = addList "a" ["b"]
3
4 sumLists = sum2 . map sum2
5 sum2 [] = []
6 sum2 (x:xs) = x + sum2 xs
```

Listing 3 is just one of the programs we generated that contains multiple type errors. We created these by merging the CE benchmark programs. Each set of programs includes the earlier source code with the addition of another CE program attached to the bottom. In all, we generated 870 new ill-typed programs to test.

The success criteria for reporting an accurate discovery of the position of a type error in an ill-typed program that contains multiple errors is similar to what we used for singular errors. The only difference being, that though we have two errors per program we only need one error to be reported to for a success.

Table 1, shows one set of results from a merged file. The first column lists the program number that we are using as the base and the second column indexes the number of the program we merged to the end of the source code. Under the Gramayre and GHC columns, we use ticks and crosses to denote if either correctly reports a type errors location, under this, we total the number of correct matches as a percentage.

With this particular combination of CE benchmark programs, we can see that Gramarye finds 42 percentage points more than GHC when locating type error positions. However, this is not always the case. Table 2, provides the total results for all of our programs as an average. The average was generated by combining the results of each of the groups of programs. Column one lists the base program, and the last two columns show the percentage of how accurate our tool and GHC were at locating type errors.

In total, we can see that Gramarye finds 7 percentage points fewer type errors in our multi-error programs than GHC. The Isolating Delta Debugging algorithm restricts Gramarye to always locate just one type error, the first it comes across. Once it has found this error, the algorithm assumes the job is complete and does not check any further. However, GHC reports many type error messages giving GHC an advantage over our debugger. The more error messages reported the more chance GHC has to report a correct line. We can this effect in the results. Currently, we would expect the programmer to fix one type error at a time, and so will repeatedly use the tool after each implemented fix. Reporting many errors message is only an advantage in this evaluation, not when debugging type errors as a whole.

4.3 Precise Type Error Evaluation

(3) Does our method return a smaller set of type error locations compared to the Glasgow Haskell Compiler?

Though our criteria for success allowed us to check multiple returned line numbers for the correct type error position, reporting many lines to the programmer is not ideal. As we aimed to return just a singular line number as the cause of the type error, an additional evaluation criteria allowed us to pinpoint how specific our tool is compared to GHC. All of the programs we tested had a single type error on a distinct line; our new rule specified that if either

Table 1: Testing a program with two type errors.

Program	Merged	Gramarye	GHC
15	1	✓	✓
15	2	✓	✓
15	3	✓	✓
15	4	✓	×
15	5	✓	✓
15	6	✓	✓
15	7	✓	×
15	8	×	×
15	9	✓	×
15	10	✓	×
15	11	✓	✓
15	12	✓	✓
15	13	✓	✓
15	14	✓	×
15	16	✓	×
15	17	✓	✓
15	18	✓	✓
15	19	✓	×
15	20	✓	✓
15	21	✓	×
15	22	×	×
15	23	✓	×
15	24	✓	✓
15	25	✓	×
15	26	✓	×
15	27	×	×
15	28	✓	✓
15	29	✓	✓
15	30	✓	×
Total		90%	48%

Table 2: Overall testing of programs with two type errors.

Program	Gramarye	GHC
1	69%	100%
2	62%	100%
3	72%	97%
4	72%	52%
5	66%	100%
6	72%	100%
7	62%	48%
8	66%	52%
9	72%	55%
10	62%	52%
11	62%	100%
12	66%	100%
13	62%	100%
14	76%	52%
15	90%	48%
16	62%	52%
17	69%	100%
18	69%	100%
19	79%	21%
20	66%	100%
21	79%	45%
22	38%	45%
23	66%	52%
24	59%	100%
25	62%	100%
26	69%	55%
27	62%	52%
28	69%	100%
29	62%	100%
30	62%	52%
Average	67%	74%

Gramarye or GHC returned a single accurate location, then they were classed as having a "precise success".

Table 3 shows all the programs that had a single type error; a tick denotes if either Gramarye or GHC accurately report a single line number as being the cause of the type error. A report of multiple lines means a cross is displayed, even if a report of a correctly located type error was within them.

Our method had a positive outcome when locating a single line as the cause of the fault. Gramarye reported accurately 16 times (53%), and, GHC does slightly worse at 12 times(40%).

When evaluating programs that included multiple self-contained type errors, we had a slightly different criteria, judging "precise success" under the following rules;

- A single line number containing the location of error one.

```
addList ls s = if s `elem` ls then ls else s : ls
v5 = addList "a" ["b"]

sumLists = sum2 . map sum2
sum2 [] = []
sum2 (x:xs) = x + sum2 xs
```

- A single line number containing the location of error two.

```
addList ls s = if s `elem` ls then ls else s : ls
v5 = addList "a" ["b"]

sumLists = sum2 . map sum2
sum2 [] = []
sum2 (x:xs) = x + sum2 xs
```

- Two line numbers containing the location of both error one and two.

```
addList ls s = if s `elem` ls then ls else s : ls
v5 = addList "a" ["b"]

sumLists = sum2 . map sum2
sum2 [] = []
sum2 (x:xs) = x + sum2 xs
```

All other results, even those that include the correct location, are recorded as failing the "precise success" criterion of discovering type errors. Table 4 presents the test programs that contained two type

Table 3: "precise success" on single type errors.

Program	Gramarye	GHC
1	✓	✓
2	×	✓
3	×	×
4	×	×
5	×	✓
6	×	✓
7	×	×
8	×	×
9	✓	×
10	✓	×
11	×	✓
12	✓	✓
13	✓	✓
14	×	×
15	×	×
16	✓	×
17	✓	×
18	✓	✓
19	×	×
20	✓	✓
21	✓	×
22	×	×
23	✓	×
24	✓	×
25	✓	✓
26	✓	×
27	×	×
28	✓	✓
29	×	✓
30	✓	×
Total	53%	40%

Table 4: "precise success" on programs with two type errors.

Program	Gramarye	GHC
1	48%	38%
2	45%	38%
3	52%	7%
4	48%	0%
5	41%	41%
6	34%	38%
7	41%	0%
8	41%	0%
9	48%	0%
10	48%	0%
11	45%	41%
12	45%	48%
13	41%	38%
14	31%	0%
15	14%	0%
16	41%	0%
17	48%	0%
18	45%	38%
19	10%	0%
20	41%	34%
21	69%	0%
22	21%	0%
23	38%	0%
24	41%	0%
25	45%	34%
26	45%	0%
27	41%	3%
28	45%	41%
29	34%	34%
30	45%	0%
Average	41%	16%

errors. The name of the original program along with the percentage of type error locations deemed to be a "precise success" are shown.

Analysing Table 4 we can see that our method is again successful in reporting the correct type error location using just one line number with 41% accuracy compared to GHC with 16%. GHC tends to report as many line numbers it feels are associated with the type error, very much like slicing. However, our method works on returning the smallest number of lines meaning we achieve a higher rate of receiving only one location at a time.

4.4 Efficiency Evaluation

Our debugger can successfully locate type errors in Haskell source code, however it also needs to be efficient. Efficiency is an important aspect for any programmer wanting to use our tool, so we evaluate it against the following questions:

(1) How many calls to the compiler are needed?
(2) How long does Gramarye take to find the type error?

The evaluation ran on a computer containing an AMD Phenom X4 965, 32GB RAM and a Samsung 850 Solid State Drive, whilst running Ubuntu Linux 16.04 LTS. Table 5 shows the program we are evaluating, how many lines of code each contains, the "clock-time" that the programmer will experience when using the debugger, the number of calls our debugger makes to GHC, and the "clock-time" time for GHC to return the result. On average our debugger took 3.359 seconds to provide a location, which meant calling GHC as a blackbox 11 times.

For programs with single type errors we can see that the majority of the "clock-time" is caused by calling GHC to type check. Reducing the number of calls to the blackbox compiler would increase the efficiency of the debugger. As we are working on a line by line basis we known that there is a risk of producing many unresolved programs that all need to be type-checked. In Table 5 we can also see which categories the programs we type-checked were placed into. In total the programs were categorised as 97 well-typed, 85 ill-typed, and 81 containing unresolved errors, such as a parse failure. The 81 calls to the type checker that were the result of unresolved programs are clearly a source of efficiency drain on the debugger, and a starting point of investigation for future work in this area.

In Table 6 we have condensed the results into averages for each group of programs, denoting the groups by the program number

Table 5: efficiency on programs with one type error.

Program	LoC	Clock-Time(s)	GHC Calls	GHC Clock-Time(s)	Pass	Fail	Unresolved
1	5	2.3	6	0.3	3	2	1
2	8	4.7	14	0.3	2	1	6
3	8	3.0	8	0.3	3	4	1
4	8	2.7	8	0.2	4	2	2
5	5	2.7	8	0.2	2	2	2
6	8	3.2	10	0.3	3	2	3
7	6	2.6	8	0.3	5	3	0
8	5	2.2	6	0.2	3	2	1
9	8	2.9	8	0.3	5	3	0
10	10	5.6	16	0.3	5	4	4
11	5	2.7	8	0.3	2	2	2
12	6	2.5	6	0.2	2	3	1
13	6	2.2	6	0.3	3	3	0
14	8	4.3	18	0.2	2	4	8
15	10	4.3	20	0.2	2	4	8
16	8	4.3	14	0.2	4	3	4
17	5	2.4	6	0.3	4	2	0
18	7	2.1	6	0.2	3	2	1
19	20	9.4	44	0.3	3	6	11
20	8	2.6	8	0.3	4	2	2
21	7	2.8	8	0.3	4	3	1
22	12	2.8	8	0.3	3	3	2
23	8	4.0	14	0.2	4	3	4
24	6	2.7	8	0.2	3	4	1
25	7	4.1	14	0.3	3	2	5
26	7	6.8	24	0.3	4	4	9
27	5	2.2	6	0.3	4	2	0
28	6	2.0	6	0.3	3	3	0
29	5	2.5	8	0.2	2	2	2
30	5	2.2	6	0.3	3	3	0
Average	7	3.4	11	0.3	97	85	81

that was used to generate the programs. When evaluating the programs with multiple type errors we see a similar outcome to the single type errors, with GHC calls tightly associated with the debugger "clock-time". The worst result we received debugging a program with multiple errors took 28.672 seconds and called GHC 110 times. However on average the debugger took 4 seconds and 14 calls to return a type error location.

4.5 Summary

Overall, our evaluation has proven positive towards our method of type error debugging. From the testing, our strength lies in the reporting of singular type errors, be that one per program or the reporting of one instance of type error amongst many. Our results compared to GHC when testing more than one type error in a program prove to be less positive. This is due to the debugger pinpointing a singular type error each time it is run. However, we believe giving an accurate location over a broad suggestion is preferential.

5 RELATED WORK

Type error debugging has taken many forms over the past thirty years; we will not be able to cover all of them. Some core categories within type error debugging include: Slicing [7, 14, 18], Interaction [4, 5, 15, 16, 21], Type Inference Modification [1, 11], and working with Constraints [13, 27]. These solutions are complicated to implement, relying on programmers to patch their compiler or use a bespoke one. However, others do not provide an implementation to use at all, and in the cases where there is an implementation, they are no longer maintained to work with the latest compiler [9].

Delta Debugging is one solution that allows for separation from the compiler, and is the name for two algorithms, one that simplifies and another that isolates[6, 24–26]. There is only one demonstration of the application of the Simplifying Delta Debugging algorithm to types errors, namely an implementation in the Liquid Haskell type checker [19].

Prior works that mention using the idea of a black box include: using the compiler's type inferencer as a black box to construct a type tree to use to debug the program [20], and having an SMT solver as a blackbox to return the satisfiable set of constraints to

Table 6: efficiency on programs with two type errors.

Program	LoC	Clock-Time(s)	GHC Calls	GHC Clock-Time(s)	Pass	Fail	Unresolved
1	11	3.4	11	0.3	3	4	2
2	14	5.3	18	0.3	4	5	6
3	14	3.7	12	0.4	3	5	3
4	14	4.0	14	0.3	3	4	4
5	11	3.5	12	0.3	3	4	3
6	14	4.1	15	0.3	3	4	4
7	12	3.3	11	0.3	3	4	2
8	11	3.1	11	0.3	3	4	2
9	14	4.0	14	0.3	3	4	4
10	16	4.4	16	0.3	3	4	5
11	11	3.2	11	0.3	3	4	2
12	11	3.4	11	0.3	3	4	2
13	12	3.3	11	0.3	3	4	2
14	12	3.9	13	0.5	2	4	4
15	15	4.9	17	0.5	26	4	6
16	14	5.2	19	0.3	4	4	6
17	11	3.4	11	0.3	3	4	2
18	13	3.1	11	0.3	3	4	2
19	26	9.4	34	0.4	3	6	14
20	14	3.6	12	0.3	3	4	3
21	13	3.4	11	0.3	4	4	2
22	18	4.3	15	0.3	4	5	4
23	14	5.2	18	0.3	4	4	6
24	12	3.1	10	0.3	3	4	2
25	13	3.3	11	0.3	3	4	3
26	13	3.2	11	0.3	3	4	3
27	11	3.3	11	0.3	3	4	2
28	12	3.3	11	0.3	3	4	2
29	11	3.7	12	0.3	3	4	3
30	11	3.4	11	0.3	3	4	2
Average	14	3.4	13	0.4	3	4	4

show type errored expressions[12], and SEMINAL [9, 10]. SEMINAL, along with previous solutions of using a blackbox compiler, actually makes modifications to an existing compiler. This requires the programmer to apply a patch; however, that patch is no longer maintained for the latest Ocaml compiler. SEMINAL works by modifying the Abstract Syntax Tree (AST), adding and removing expressions. It returns a location of the type error along with a suggested fix.

Inspired by SEMINAL is an approach that talks about altering source code with a constraint-free tool, however though the author refers to source code modification, the implementation works directly on the AST [14]. Another tool TypeHope also discusses changing the source code of a program to stay true to how a programmer debugs. However, again, the implementation edits the AST [2]. At this point, as far as the authors know, modifying source code directly is a new approach in the type error debugging field.

6 CONCLUSION AND FUTURE WORK

Our method combines the Isolating Delta Debugging algorithm, a black box compiler and direct source code modification to locate type errors. Our tool Gramarye implements the method for Haskell using the Glasgow Haskell compiler as a black box. From our evaluation we have gathered positive results that support our method for type error debugging. For single type errors our tool gives a 27 percentage points improvement over GHC. However, for two separate type errors in a single program GHC was 9 percentage points more successful. When returning only a precise line number for type errors, our method proved positive with 53% for locating singular type errors, and 41% when applied to a program that contained two type errors. A significant practical advantage of our method is that our tool Gramarye has only a small GHC-specific component and thus can easily be modified for other programming languages and compilers.

In the future we will be looking at where Gramarye did well and what its points of failure were. We will then use the outcome of the investigation to improve our algorithm for type error debugging. We will study closer the non-determinism of our method: can we sometimes determine whether one choice is better than another? After we have improved our method to determine the correct line number, we intend to increase the granularity of the tool further to eventually modify programs by single characters instead of lines, thus identifying subexpressions that cause type errors. On the

theoretical side, there is clearly a close link between our method and methods described in the literature that perform type error slicing based on minimal unsolvable constraint sets. We want to formalise that link.

Additional improvements to the tool outside of the algorithm would also be useful. An improved GUI, though not necessary for seeing if our approach is beneficial, does open up the options of not only combining with other methodologies that rely on interaction but also testing with real-life participants. We also would like to conduct empirical research of our solution in combination with evaluating against collected student programs to affirm our strategy.

REFERENCES

[1] Karen L Bernstein and Eugene W Stark. 1995. *Debugging type errors (full version).* Technical Report. State University of New York at Stony Brook, Stony Brook, NY 11794-4400 USA. https://pdfs.semanticscholar.org/814c/164c88ba7dd22e7e501cdd1a951586a3117b.pdf

[2] Bernd Braßel. 2004. Typehope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*. https://www.informatik.uni-kiel.de/~mh/wlp2004/final_papers/paper13.ps

[3] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 583–594. https://doi.org/10.1145/2535838.2535863

[4] Sheng Chen and Martin Erwig. 2014. Guided Type Debugging. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. 35–51. https://doi.org/10.1007/978-3-319-07151-0_3

[5] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*. 193–204. https://doi.org/10.1145/507635.507659

[6] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. 342–351. https://doi.org/10.1145/1062455.1062522

[7] Christian Haack and Joe B. Wells. 2004. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.* 50, 1-3 (2004), 189–224. https://doi.org/10.1016/j.scico.2004.01.004

[8] Gregory F. Johnson and Janet A. Walz. 1986. A Maximum-Flow Approach to Anomaly Isolation in Unification-Based Incremental Type Inference. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 44–57. https://doi.org/10.1145/512644.512649

[9] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 425–434. https://doi.org/10.1145/1250734.1250783

[10] Benjamin S. Lerner, Dan Grossman, and Craig Chambers. 2006. Seminal: searching for ML type-error messages. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. 63–73. https://doi.org/10.1145/1159876.1159887

[11] Bruce J McAdam. 1999. On the unification of substitutions in type inference. *Lecture notes in computer science* 1595 (1999), 137–152. https://link.springer.com/chapter/10.1007/3-540-48515-5_9

[12] Zvonimir Pavlinovic. 2014. General Type Error Diagnostics Using MaxSMT. (2014). https://pdfs.semanticscholar.org/1c14/7bc9f51cc950596dbc3e7cc5121202d160da.pdf

[13] Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electr. Notes Theor. Comput. Sci.* 312 (2015), 197–213. https://doi.org/10.1016/j.entcs.2015.04.012

[14] Thomas Schilling. 2011. Constraint-Free Type Error Slicing. In *Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers*. 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

[15] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 228–242. https://doi.org/10.1145/2951913.2951915

[16] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. 72–83. https://doi.org/10.1145/871895.871903

[17] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving type error diagnosis. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. 80–91. https://doi.org/10.1145/1017472.1017486

[18] Frank Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.* 10, 1 (2001), 5–55. https://doi.org/10.1145/366378.366379

[19] A Tondwalkar. 2016. *Finding and Fixing Bugs in Liquid Haskell.* Master's thesis. University of Virginia. https://pdfs.semanticscholar.org/79b4/22959847253c40aff25c228205372d9ebc60.pdf

[20] Kanae Tsushima and Kenichi Asai. 2012. An Embedded Type Debugger. In *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*. 190–206. https://doi.org/10.1007/978-3-642-41582-1_12

[21] Kanae Tsushima and Olaf Chitil. 2014. Enumerating Counter-Factual Type Error Messages with an Existing Type Checker. In *16th Workshop on Programming and Programming Languages, PPL2014*. http://kar.kent.ac.uk/49007/

[22] Mitchell Wand. 1986. Finding the Source of Type Errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 38–43. https://doi.org/10.1145/512644.512648

[23] Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did?. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[24] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*. 253–267. https://doi.org/10.1007/3-540-48166-4_16

[25] Andreas Zeller. 2009. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition.* Academic Press. http://store.elsevier.com/product.jsp?isbn=9780123745156&pagename=search

[26] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[27] Danfeng Zhang, Andrew C Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. *Diagnosing Haskell type errors.* Technical Report. Technical Report http://hdl.handle. net/1813/39907, Cornell University. https://pdfs.semanticscholar.org/d32f/81a5c1706e225e2255b72c1e4b41f799e8f1.pdf

Received May 2018

HiPErJiT: A Profile-Driven Just-in-Time Compiler for Erlang

Konstantinos Kallas
University of Pennsylvania
Philadelphia, USA
kallas@seas.upenn.edu

Konstantinos Sagonas
Uppsala University
Uppsala, Sweden
kostis@it.uu.se

ABSTRACT

We introduce HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on HiPE, the High Performance Erlang compiler. HiPErJiT uses runtime profiling to decide which modules to compile to native code and which of their functions to inline and type-specialize. HiPErJiT is integrated with the runtime system of Erlang/OTP and preserves aspects of Erlang's compilation which are crucial for its applications: most notably, tail-call optimization and hot code loading at the module level. We present HiPErJiT's architecture, describe the optimizations that it performs, and compare its performance with BEAM, HiPE, and Pyrlang. HiPErJiT offers performance which is about two times faster than BEAM and almost as fast as HiPE, despite the profiling and compilation overhead that it has to pay compared to an ahead-of-time native code compiler. But there also exist programs for which HiPErJiT's profile-driven optimizations allow it to surpass HiPE's performance.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Functional languages*; *Concurrent programming languages*;

KEYWORDS

Just-in-Time compilation, profile-driven optimization, HiPE, Erlang

ACM Reference Format:
Konstantinos Kallas and Konstantinos Sagonas. 2018. HiPErJiT: A Profile-Driven Just-in-Time Compiler for Erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3310232.3310234

1 INTRODUCTION

Erlang is a concurrent functional programming language with features that support the development of scalable concurrent and distributed applications, and systems with requirements for high availability and responsiveness. Its main implementation, the Erlang/OTP system, comes with a byte code compiler that produces portable and reasonably efficient code for its virtual machine, called BEAM. For applications with requirements for better performance, an ahead-of-time native code compiler called HiPE (High Performance Erlang) can be selected. In fact, byte code and native code can happily coexist in the Erlang/OTP runtime system.

Despite this flexibility, the selection of the modules of an application to compile to native code is currently manual. Perhaps it would be better if the system itself could decide on which parts to compile to native code in a just-in-time fashion. Moreover, it would be best if this process was guided by profiling information gathered during runtime, and was intelligent enough to allow for the continuous run-time optimization of the code of an application.

This paper makes a first big step in that direction. We have developed HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on HiPE. HiPErJiT employs the tracing support of the Erlang runtime system to profile the code of bytecode-compiled modules during execution and choose whether to compile these modules to native code, currently employing all optimizations that HiPE also performs by default. For the chosen modules, it additionally decides which of their functions to inline and/or specialize based on runtime type information. We envision that the list of additional optimizations to perform based on profiling information will be extended in the future, especially if HiPErJiT grows to become a JiT compiler which performs lifelong feedback-directed optimization of programs. Currently, JiT compilation is triggered only once for each loaded instance of a module, and the profiling of their functions is stopped at that point.

The main part of this paper presents the architecture of HiPErJiT and the rationale behind some of our design decisions (Section 3), the profile-driven optimizations that HiPErJiT performs (Section 4), and the performance it achieves (Section 5). Before all that, in the next section, we review the current landscape of Erlang compilers. Related work is scattered throughout.

2 ERLANG AND ITS COMPILERS

In this section, we present a brief account of the various compilers for Erlang. Our main aim is to set the landscape for our work and present a comprehensive high-level overview of the various compilers that have been developed for the language, rather than a detailed technical one. For the latter, we refer the reader to the sites of these compilers and to publications about them.

2.1 JAM and BEAM

Technically, JAM and BEAM are abstract (virtual) machines for Erlang, not compilers. However, they both have lent their name to compilers that generate byte code for these machines. JAM is the older one, but since 1998, when the Erlang/OTP system became open source, the system has been exclusively based on BEAM.

The BEAM compiler is a fast, module-at-a-time compiler that produces relatively compact byte code, which is then loaded into the

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310234

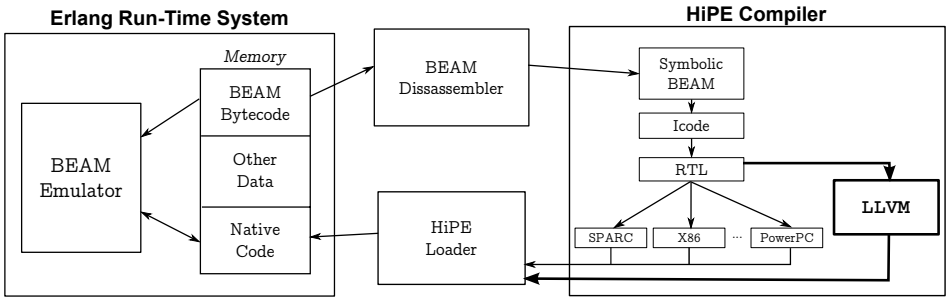


Figure 1: Architecture of some Erlang/OTP components: ERTS, HiPE and ErLLVM (from [24]).

Erlang Run-Time System and expanded to indirectly threaded code for the VM interpreter (called "emulator" in the Erlang community). In the process, the BEAM loader specializes and/or merges byte code instructions. By now, the BEAM compiler comes with a well-engineered VM interpreter offering good performance for many Erlang applications. As a result, some other languages (e.g., Elixir) have chosen to use BEAM as their target. In recent years, the term "BEAM ecosystem" has been used to describe these languages, and also to signify that BEAM is important not only for Erlang.

2.2 HiPE and ErLLVM

The High-Performance Erlang (HiPE) compiler [13, 23] is an ahead-of-time native code compiler for Erlang. It has backends for SPARC, x86 [21], x86_64 [20], PowerPC, PowerPC 64, and ARM, has been part of the Erlang/OTP distribution since 2002, and is mature by now. HiPE aims to improve the performance of Erlang applications by allowing users to compile their time-critical modules to native code, and offer flexible integration between interpreted and native code. Since 2014, the ErLLVM compiler [24], which generates native code for Erlang using the LLVM compiler infrastructure [15] (versions 3.5 or higher), has also been integrated into HiPE as one of its backends, selectable by the `to_llvm` compiler option. In general, ErLLVM offers similar performance to the native HiPE backends [24].

Figure 1 shows how the HiPE compiler fits into Erlang/OTP. The compilation process typically starts from BEAM byte code. HiPE uses three intermediate representations, namely Symbolic BEAM, Icode, and RTL, and then generates target-specific assembly either directly or outsources its generation to ErLLVM.

Icode is a register-based intermediate language for Erlang. It supports an infinite number of registers which are used to store arguments and temporaries. All values in Icode are proper Erlang terms. The call stack is implicit and preserves registers. Finally, as Icode is a high-level intermediate language, bookkeeping operations (such as heap overflow checks and context switching) are implicit.

RTL is a generic three-address register transfer language. Call stack management and the saving and restoring of registers before and after calls are made explicit in RTL. Heap overflow tests are also made explicit and are propagated backwards in order to get merged. Registers in RTL are separated in tagged and untagged, where the untagged registers hold raw integers, floating-point numbers, and addresses.

RTL code is then translated to machine-specific assembly code (or to LLVM IR), virtual registers are assigned to real registers, symbolic references to atoms and functions are replaced by their real values in the running system, and finally, the native code is loaded into the Erlang Run-Time System (ERTS) by the HiPE loader.

ERTS allows seamless interaction between interpreted and native code. However, there are differences in the way interpreted and native code execution is managed. Therefore, a transition from native to interpreted code (or vice versa) triggers a *mode switch*. Mode switches occur in function calls, returns, and exception throws between natively-compiled and interpreted functions. HiPE was designed with the goal of no runtime overhead as long as the execution mode stays the same, so mode switches are handled by instrumenting calls with special mode switch instructions and by adding extra call frames that cause a mode switch after each function return. Frequent mode switching can negatively affect execution performance. Therefore, it is recommended that the most frequently executed functions of an application are all executed in the same mode [23].

2.3 BEAMJIT and Pyrlang

Unsurprisingly, JiT compilation has been investigated in the context of Erlang several times in the past, both distant and more recent. For example, both Jerico [12], the first native code compiler for Erlang (based on JAM) circa 1996, and early versions of HiPE contained some support for JiT compilation. However, this support never became robust to the point that it could be used in production.

More recently, two attempts to develop tracing JiTs for Erlang have been made. The first of them, BEAMJIT [6], is a tracing just-in-time compiling runtime for Erlang. BEAMJIT uses a tracing strategy for deciding which code sequences to compile to native code and the LLVM toolkit for optimization and native code emission. It extends the base BEAM implementation with support for profiling, tracing, native code compilation, and support for switching between these three (profiling, tracing, and native) execution modes. Performance-wise, back in 2014, BEAMJIT reportedly managed to reduce the execution time of some small Erlang benchmarks by 25–40% compared to BEAM, but there were also many other benchmarks where it performed worse than BEAM [6]. Moreover, the same paper reported that "HiPE provides such a large performance improvement compared to plain BEAM that a comparison to BEAMJIT would be uninteresting" [6, Sect. 5]. At the time of this writing (May 2018), BEAMJIT is not yet a complete implementation of Erlang; for example, it does not yet support floats. Although work is ongoing in extending BEAMJIT, its overall performance, which

has improved, does not yet surpass that of HiPE.[1] Since BEAMJIT is not available, not even as a binary, we cannot compare against it.

The second attempt, Pyrlang [11], is an alternative virtual machine for the BEAM byte code which uses RPython's meta-tracing JiT compiler [5] as a backend in order to improve the sequential performance of Erlang programs. Meta-tracing JiT compilers are tracing JiT compilers that trace and optimize an interpreter instead of the program itself. The Pyrlang paper [11] reports that Pyrlang achieves average performance which is 38.3% faster than BEAM and 25.2% slower than HiPE, on a suite of sequential benchmarks. Currently, Pyrlang is a research prototype and not yet a complete implementation of Erlang, not even for the sequential part of the language. On the other hand, unlike BEAMJIT, Pyrlang is available, and we will directly compare against it in Section 5.

2.4 Challenges of Compiling Erlang

The Erlang programming language comes with some special characteristics which make its efficient compilation quite challenging. On the top of the list is *hot code loading*: the requirement to be able to replace modules, on an individual basis, while the system is running and without imposing a long stop to its operation. The second characteristic, which is closely related to hot code loading, is that the language makes a semantic distinction between module-local calls, which have the form `f(...)`, and so called *remote calls* which are module-qualified, i.e., have the form `m:f(...)`, and need to look up the most recently loaded version of module `m`, even from a call within `m` itself.

These two characteristics, combined with the fact that Erlang is primarily a concurrent language in which a large number of processes may be executing code from different modules at the same time, effectively impose that compilation happens in a module-at-a-time fashion, without opportunities for cross-module optimizations. The alternative, i.e., performing cross-module optimizations, implies that the runtime system must be able to quickly undo optimizations in the code of some module that rely on information from other modules, whenever those other modules change. Since such undoings can cascade arbitrarily deep, this alternative is not very attractive engineering-wise.

The runtime system of Erlang/OTP supports hot code loading in a particular, arguably quite ad hoc, way. It allows for *up to two* versions of each module (m_{old} and $m_{current}$) to be simultaneously loaded, and redirects all remote calls to $m_{current}$, the most recent of the two. Whenever m_{new}, a new version of module m, is about to be loaded, all processes that still execute code of m_{old} are abruptly terminated, $m_{current}$ becomes the new m_{old}, and m_{new} becomes $m_{current}$. This quite elaborate mechanism is implemented by the code loader with support from ERTS, which has control over all Erlang processes.

Unlike BEAMJIT and Pyrlang, we decided, at least for the time being, to leave the Erlang Run-Time System unchanged as far as hot code loading is concerned. This also means that, like HiPE, the unit of JiT compilation of HiPErJiT is the entire module.

[1] Lukas Larsson (member of Erlang/OTP team), private communication, May 2018.

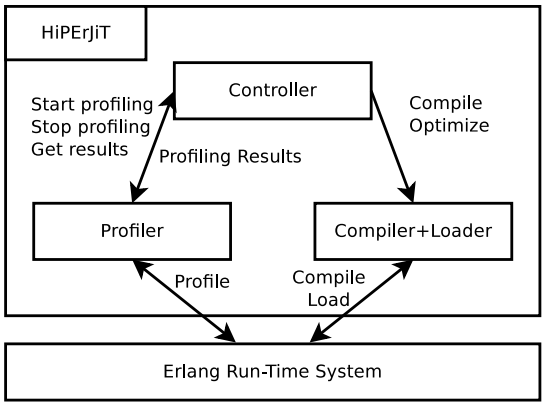


Figure 2: High-level architecture of HiPErJiT.

3 HIPERJIT

The functionality of HiPErJiT can be briefly described as follows. It profiles executed modules, maintaining runtime data such as execution time and call graphs. It then decides which modules to compile and optimize based on the collected data. Finally, it compiles and loads the JiT-compiled modules in the runtime system. Each of these tasks is handled by a separate component.

Controller The central controlling unit which decides which modules should be profiled and which should be optimized based on runtime data.

Profiler An intermediate layer between the controller and the low-level Erlang profilers. It gathers profiling information, organizes it, and propagates it to the controller for further processing.

Compiler+Loader An intermediate layer between the controller and HiPE. It compiles the modules chosen by the controller and then loads them into the runtime system.

The architecture of the HiPErJiT compiler can be seen in Fig. 2.

3.1 Controller

The controller, as stated above, is the fundamental component of HiPErJiT. It chooses the modules to profile, and uses the profiling data to decide which modules to compile and optimize. It is essential that no user input is needed to drive decision making. Our design extends a lot of concepts from the work on Jalapeño JVM [1].

Traditionally, many JiT compilers use a lightweight call frequency method to drive compilation [3]. This method maintains a counter for each function and increments it every time the function is called. When the counter surpasses a specified threshold, JiT compilation is triggered. This approach, while having very low overhead, does not give precise information about the program execution.

Instead, HiPErJiT makes its decision based on a simple cost-benefit analysis. Compilation for a module is triggered when its predicted future execution time when compiled, combined with its compilation time, would be less than its future execution time when interpreted:

$$FutureExecTime_c + CompTime < FutureExecTime_i$$

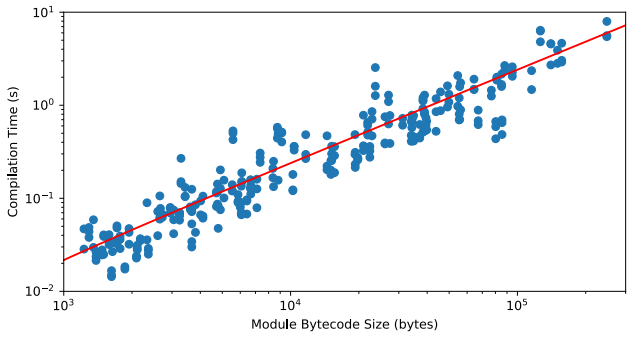


Figure 3: Estimating compilation time as a function of module byte code size. Observe that both axes are log-scale.

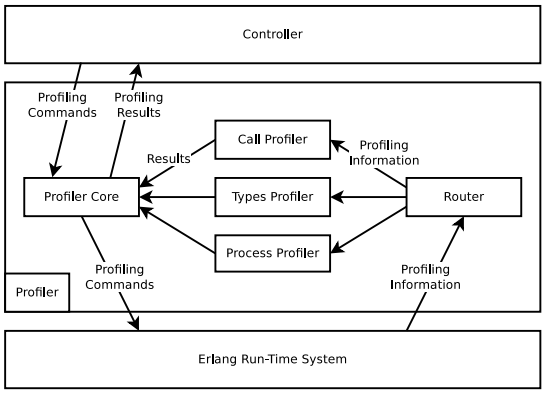


Figure 4: Profiler architecture and its components.

Of course, it is not possible to predict accurately the future execution time of a module or the time needed to compile it, so some estimations need to be made. First of all, we need to estimate future execution time. The results of a study about the lifetime of UNIX processes [8] show that the total execution time of UNIX processes follows a Pareto (heavy-tailed) distribution. The mean remaining waiting time of this distribution is analogous to the amount of time that has passed already. Motivated by those results, and assuming that the analogy between a module and a UNIX process holds, we consider future execution time of a module to be equal to its execution time until now $FutureExecTime = ExecTime$. In addition, we consider that compiled code has a constant relative speedup to the interpreted code, thus $ExecTime_c * Speedup_c = ExecTime_i$. Finally, we consider that the compilation time for a module depends linearly on its size, so $CompTime = C * Size$. Based on the above assumptions, the condition to check is:

$$\frac{ExecTime_i}{Speedup_c} + C * Size < ExecTime_i$$

If this condition holds, the module is "worth" compiling.

We conducted two experiments in order to find suitable values for the condition parameters $Speedup_c$ and C. In the first one, we executed all the benchmark programs, that were used for evaluation, before and after compiling their modules to native code. The average speedup we measured was 2 and we used it as an estimate for the $Speedup_c$ parameter. In the second experiment, we compiled all the modules of the above programs, measured their compilation time, and fitted a line to the set of given points using the Least Squares method (cf. Fig. 3).

The line has a slope of $2.5e^{-5}$ so that is also the estimated compilation cost (in seconds) per byte of byte code. It is worth mentioning that, in reality, the compilation time does not depend only on module size, but on many other factors (e.g., branching, exported functions, etc). However, we consider this first estimate to be adequate for our goal.

Finally, HiPErJiT uses a feedback-directed scheme to improve the precision of the compilation decision condition when possible. HiPErJiT measures the compilation time of each module it compiles to native code and stores it in a persistent key-value storage, where the key is a pair of the module name and the MD5 hash value of its byte code. If the same version of that module (one that has the same name and MD5 hash) is considered for compilation at a subsequent time, HiPErJiT will use the stored compilation time measurement in place of $CompTime$.

3.2 Profiler

The profiler is responsible for efficiently profiling executed code using the ERTS profiling infrastructure. Its architecture, which can be seen in Fig. 4, consists of:

- The profiler core, which receives the profiling commands from the controller and transfers them to ERTS. It also receives the profiling results from the individual profilers and transfers them back to the controller.
- The router, which receives all profiling messages from ERTS and routes them to the correct individual profiler.
- The individual profilers, which handle profiling messages, aggregate them, and transfer the results to the profiler core. Each individual profiler handles a different subset of the execution data, namely function calls, execution time, argument types, and process lifetime.

We designed the profiler in a way that facilitates the addition and removal of individual profilers.

3.2.1 Profiling Execution Time. In order to profile the execution time that is being spent in each function, the profiler receives a time-stamped message from ERTS for each function call, function return, and process scheduling action. Messages have the form $\langle Action, Timestamp, Process, Function \rangle$, where $Action$ can be any of the following values: $call$, $return\_to$, $sched\_in$, $sched\_out$.

For each sequence of trace messages that refer to the same process P, i.e., a sequence of form $[(sched\_in, T_1, P, F_1), (A_2, T_2, P, F_2), \ldots, (A_{n-1}, T_{n-1}, P, F_{n-1}), (sched\_out, T_n, P, F_n)]$, we can compute the time spent at each function by finding the difference of the timestamps in each pair of adjacent messages, so $[(T_2 - T_1, F_1), (T_3 - T_2, F_2), \ldots, (T_n - T_{n-1}, F_{n-1})]$. The sum of all the differences for each function gives the total execution time spent in each function. An advantage of this method is that it allows us to also track the number of calls between each pair of functions, which is later used for profile-driven call inlining (Section 4.2).

The execution time profiler is able to handle the stream of messages sent during the execution of sequential programs. However,

in concurrent programs, the rate at which messages are sent to the profiler can increase uncontrollably, thus flooding its mailbox, leading to high memory consumption.

In order to tackle this problem, we implemented a probing mechanism that checks whether the number of unhandled messages in the mailbox of the execution time profiler exceeds a threshold (currently 1 million messages), in essence checking whether the arrival rate of messages is higher than their consumption rate. If the mailbox exceeds this size, no more trace messages are sent from ERTS until the profiler handles the remaining messages in the mailbox.

3.2.2 Type Tracing. The profiler is also responsible for type tracing. As we will see, HiPErJiT contains a compiler pass that uses type information for the arguments of functions in order to create type-specialized, and hopefully better performing, versions of functions. This type information is extracted from type specifications in Erlang source code (if they exist) and from runtime type information which is collected and aggregated as described below.

The function call arguments to functions in modules that have been selected for profiling are recorded by HiPErJiT. The ERTS low-level profiling functionality returns the complete argument values, so in principle their types can be determined. However, functions could be called with complicated data structures as arguments, and determining their types precisely requires a complete traversal. As this could lead to a significant performance overhead, the profiler approximates their types by a limited-depth term traversal. In other words, *depth-k type abstraction* is used. Every type T is represented as a tree with leaves that are either singleton types or type constructors with zero arguments, and internal nodes that are type constructors with one or more arguments. The depth of each node is defined as its distance from the root. A depth-k type T_k is a tree where every node with depth $\geq k$ is pruned and over-approximated with the top type (`any()`). For example, in Erlang's type language [18], which supports unions of singleton types, the Erlang term `{foo,{bar,[{a,17},{a,42}]}}` has type `{'foo',{'bar',list({'a',17|42})}}`, where `'a'` denotes the singleton atom type `a`. Its depth-1 and depth-2 type abstractions are `{'foo',{any(),any()}}` and `{'foo',{'bar',list(any())}}`.

The following two properties should hold for depth-k type abstractions:

(1) $\forall k.k \geq 0 \Rightarrow T \sqsubseteq T_k$ where $\sqsubseteq$ is the subtype relation.
(2) $\forall t.t \neq T \Rightarrow \exists k.\forall i.i \geq k \Rightarrow t \not\sqsubseteq T_i$

The first property guarantees correctness while the second allows us to improve the approximation precision by choosing a greater k, thus allowing us to trade performance for precision or vice versa.

Another problem that we faced is that Erlang collections (lists and maps) can contain elements of different types. Traversing them in order to find their complete type could also create undesired overhead. As a result, we decided to optimistically estimate the collection element types. Although Erlang collections can contain elements of many different types, this is rarely the case in practice. In most programs, collections usually contain elements of the same type. This, combined with the fact that HiPErJiT uses the runtime type information to specialize some functions for specific type arguments, gives us the opportunity to be more optimistic while deducing the types of values. Therefore, we decided to approximate the types of Erlang collections by considering only a small subset of their elements.

What is left is a way to generalize and aggregate the type information acquired through the profiling of many function calls. Types in Erlang are internally represented using a subtyping system [18], thus forming a type lattice. Because of that, a supremum operation can be used to aggregate type information that has been acquired through different traces.

3.2.3 Profiling Processes. Significant effort has been made to ensure that the overhead of HiPErJiT does not increase with the number of concurrent processes. Initially, performance was mediocre when executing concurrent applications with a large number ($\gg 100$) of spawned processes because of profiling overhead. While profiling a process, every function call triggers an action that sends a message to the profiler. The problem arises when many processes execute concurrently, where a lot of execution time is needed by the profiler process to handle all the messages being sent to it. In addition, the memory consumption of the profiler skyrocketed as messages arrived with a higher rate than they were consumed.

To avoid such problems, HiPErJiT employs *genealogy based statistical profiling*, or genealogy profiling in short. The idea is based on the following observation. Massively concurrent applications usually consist of a lot of sibling processes that have identical functionality. Because of that, it is reasonable to sample a part of the hundreds (or even thousands) of processes and only profile them to get information about the system as a whole.

This sample should not be taken at random, but it should rather follow the principle described above. We maintain a process tree by monitoring the lifetime of all processes. The nodes of the process tree are of the following type:

```
-type ptnode() :: {pid(), mfa(), [ptnode()]}.
```

The `pid()` is the process identifier, the `mfa()` is the initial function of the process, and the `[ptnode()]` is a list of its children processes, which is empty if it is a leaf. The essence of the process tree is that sibling subtrees which share the same structure and execute the same initial function usually have the same functionality. Thus we could only profile a subset of those subtrees and get relatively accurate results. However, finding subtrees that share the same structure can become computationally demanding. Instead, we decided to just find leaf processes that were spawned using the same MFA and group them. So, instead of profiling all processes of each group, we just profile a randomly sampled subset. The sampling strategy that we used as a starting point is very simple but still gives satisfactory results. When the processes of a group are more than a specified threshold (currently 50) we sample only a percentage (currently 10%) of them. The profiling results for these subsets are then scaled accordingly (i.e., multiplied by 10) to better represent the complete results.

3.3 Compiler+Loader

This is the component that is responsible for the compilation and loading of modules. It receives the collected profiling data (i.e., type information and function call data) from the controller, formats them, and calls an extended version of the HiPE compiler to compile the module and load it. The HiPE compiler is extended with two

additional optimizations that are driven by the collected profiling data; these optimizations are described in Section 4.

There are several challenges that HiPErJiT has to deal with, when loading an Erlang module, as also mentioned in Section 2.4. First of all, Erlang supports hot code loading, so a user can reload the code of a module while the system is running. If this happens, HiPErJiT automatically triggers the process of re-profiling this module.

Currently ERTS allows only up to two versions of the same module to be simultaneously loaded. This introduces a rather obscure but still possible race condition between HiPErJiT and the user. If the user tries to load a new version of a module after HiPErJiT has started compiling the current one, but before it has completed loading it, HiPErJiT could load JiT-compiled code for an older version of the module after the user has loaded the new one. Furthermore, if HiPErJiT tries to load a JiT-compiled version of a module m when there are processes executing m_{old} code, this could lead to processes being killed. Thus, HiPErJiT loads a module m as follows. First, it acquires a module lock, not allowing any other process to reload this specific module during that period. Then, it calls HiPE to compile the target module to native code. After compilation is complete, HiPErJiT repeatedly checks whether any process executes m_{old} code. When no process executes m_{old} code, the JiT-compiled version is loaded. Finally, HiPErJiT releases the lock so that new versions of module m can be loaded again. This way, HiPErJiT avoids that loading JiT-compiled code leads to processes being killed.

Of course, the above scheme does not offer any progress guarantees, as it is possible for a process to execute m_{old} code for an indefinite amount of time. In that case, the user would be unable to reload the module (e.g., after fixing a bug). In order to avoid this, HiPErJiT stops trying to load a JiT-compiled module after a user-defined timeout (the default value is 10 seconds), thus releasing the module lock.

Let us end this section with a brief note about decompilation. Most JiT compilers support decompilation, which is usually triggered when a piece of JiT-compiled code is not executed frequently enough anymore. The main benefit of doing this is that native code takes up more space than byte code, so decompilation often reduces the memory footprint of the system. However, since code size is not a big concern in today's machines, and since decompilation can increase the number of mode switches (which are known to cause performance overhead) HiPErJiT currently does not support decompilation.

4 PROFILE-DRIVEN OPTIMIZATIONS

In this section, we describe two optimizations that HiPErJiT performs, based on the collected profiling information, in addition to calling HiPE: type specialization and inlining.

4.1 Type Specialization

In dynamically typed languages, there is typically very little type information available during compilation. Types of values are determined at runtime and because of that, compilers for these languages generate code that handles all possible value types by adding type tests to ensure that operations are performed on terms of correct type. Also all values are tagged, which means that their runtime representation contains information about their type. The combination of these two features leads to compilers that generate less efficient code than those for statically typed languages.

This problem has been tackled with static type analysis [16, 23], runtime type feedback [10], or a combination of both [14]. Runtime type feedback is essentially a mechanism that gathers type information from calls during runtime, and uses this information to create specialized versions of the functions for the most commonly used types. On the other hand, static type analysis tries to deduce type information from the code in a conservative way, in order to simplify it (e.g., eliminate some unnecessary type tests). In HiPErJiT, we employ a combination of these two methods.

4.1.1 Optimistic Type Compilation. After profiling the argument types of function calls as described in Section 3.2.2, the collected type information is used. However, since this information is approximate or may not hold for all subsequent calls, the optimistic type compilation pass adds type tests that check whether the arguments have the appropriate types. Its goal is to create specialized (optimistic) function versions, whose arguments are of known types.

Its implementation is straightforward. For each function `f` where the collected type information is non-trivial:

- The function is duplicated into an optimized `f$opt` and a "standard" `f$std` version of the function.
- A header function that contains all the type tests is created. This function performs all necessary type tests for each argument, to ensure that they satisfy any assumptions upon which type specialization may be based. If all tests pass, the header calls `f$opt`, otherwise it calls `f$std`.
- The header function is inlined in every local function call of the specified function `f`. This ensures that the type tests happen on the caller's side, thus improving the benefit from the type analysis pass that happens later on.

4.1.2 Type Analysis. Optimistic type compilation on its own does not offer any performance benefit. It simply duplicates the code and forces execution of possibly redundant type tests. Its benefits arise from its combination with type analysis and propagation.

Type analysis is an optimization pass performed by HiPE on Icode that infers type information for each program point and then simplifies the code based on this information. It removes checks and type tests that are unnecessary based on the inferred type information. It also removes some boxing and unboxing operations from floating-point computations. A brief description of the type analysis algorithm [18] is as follows:

(1) Construct the call graph for all the functions in a module and sort it topologically based on the dependencies between its strongly connected components (SCCs).
(2) Analyze the SCCs in a bottom-up fashion using a constraint-based type inference to find the most general success typings [19] under the current constraints.
(3) Analyze the SCCs in a top-down order using a data-flow analysis to propagate type information from the call sites to module-local functions.
(4) Add new constraints for the types, based on the propagated information from the previous step.
(5) If a fix-point has not been reached, go back to step 2.

Initial constraints are mostly generated using the type information of Erlang Built-In Functions (BIFs) and functions from the standard library which are known to the analysis. Guards and pattern matches are also used to generate type constraints.

After type analysis, code optimizations are performed. First, all redundant type tests or other checks are completely removed. Then some instructions, such as floating-point arithmetic, are simplified based on the available type information. Finally, control flow is simplified and dead code is removed.

It is important to note that type analysis and propagation is restricted to the module boundary. No assumptions are made about the arguments of the exported functions, as those functions can be called from modules which are not yet present in the system or available for analysis. Thus, modules that export only few functions benefit more from this analysis as more constraints are generated for their local functions and used for type specializations.

4.1.3 An Example. We present a simple example that illustrates type specialization. Listing 1 contains a function that computes the power of two values, where the base of the exponentiation is a number (an integer or a float) and the exponent is an integer.

```
-spec power(number(), integer(), number()) -> number().
power(_V1, 0, V3) -> V3;
power(V1, V2, V3) -> power(V1, V2-1, V1*V3).
```

Listing 1: The source code of a simple power function.

Without the optimistic type compilation, HiPE generates the Icode shown in Listing 2.

```
power/3(v1, v2, v3) ->
12:
    v5 := v3
    v4 := v2
    goto 1
1:
    _ := redtest()    (primop)
    if is_{integer,0}(v4) then goto 3 else goto 10
3:
    return(v5)
10:
    v8 := '-'(v4, 1)  (primop)
    v9 := '*'(v1, v5) (primop)
    v5 := v9
    v4 := v8
    goto 1
```

Listing 2: Generated Icode for the power function.

If we consider that this function is mostly called with a floating point number as the first argument, then HiPE with optimistic type compilation generates the Icode in Listing 3. The Icode for `power$std` is not included as it is the same as the Icode for `power` without optimistic type compilation.

As it can be seen, optimistic type compilation has used type propagation and found that both `v1` and `v3` are always floats. The code was then modified so that they are untagged unsafely in every iteration and multiplied using a floating point instruction, whereas without optimistic type compilation they are multiplied using the standard general multiplication function, which checks the types of both arguments and untags (and unboxes) them before performing the multiplication.

```
power/3(v1, v2, v3) ->
16:
    _ := redtest()    (primop)
    if is_float(v1) then goto 3 else goto 14
3:
    if is_integer(v2) then goto 5 else goto 14
5:
    if is_float(v3) then goto 7 else goto 14
7:
    power$opt/3(v1, v2, v3)
14:
    power$std/3(v1, v2, v3)

power$opt/3(v1, v2, v3) ->
24:
    v5 := v3
    v4 := v2
    goto 1
1:
    _ := redtest() (primop)
    if is_{integer,0}(v4) then goto 3 else goto 20
3:
    return(v5)
20:
    v8 := '-'(v4, 1)  (primop)
    _ := gc_test<3>() (primop) -> goto 28, #fail 28
28:
    fv10 := unsafe_untag_float(v5) (primop)
    fv11 := unsafe_untag_float(v1) (primop)
    _ := fclearerror()             (primop)
    fv12 := fp_mul(fv11, fv10)     (primop)
    _ := fcheckerror()             (primop)
    v5 := unsafe_tag_float(fv12)   (primop)
    v4 := v8
    goto 1
```

Listing 3: Generated Icode for the power function with optimistic type compilation.

4.2 Inlining

Inlining is the process of replacing a function call with the body of the called function. This improves performance in two ways. First, it mitigates the function call overhead. Second, and most important, it enables more optimizations to be performed later, as most optimizations usually do not cross function boundaries. That is the reason why inlining is usually performed in early phases of compilation so that later phases become more effective.

However, aggressive inlining has several potential drawbacks, both in compilation time, as well as in code size increase (which in turn can also lead to higher execution times because of caching effects). However, code size is less of a concern in today's machines, except of course in application domains where memory is not abundant (e.g., in IoT or embedded systems).

In order to get a good performance benefit from inlining, the compiler must achieve a fine balance between inlining every function call and not inlining anything at all. Therefore, the most important issue when inlining is choosing which function calls to inline. There has been a lot of work on how to make this decision with compile-time [22, 25, 26, 28] as well as run-time information in the context of JiT compilers [4, 7, 9, 27]. HiPErJiT makes its inlining decisions based on run-time information, but also keeps its algorithm fairly lightweight so as to not to impose a big overhead.

4.2.1 Inlining Decision. Our mechanism borrows two ideas from previous work, the use of call frequency [4] and call graphs [27] to

guide the inlining decision. Recall that HiPErJiT compiles whole modules at a time, thus inlining decisions are made based on information from all the functions in a module. Due to hot code loading, inlining across modules is not performed.

Finding the most profitable, performance-wise, function calls to inline and also the most efficient order in which to inline them is a heavy computational task and thus we decided to use heuristics to greedily decide which function calls to inline and when. The call frequency data that are used is the number of calls between each pair of function, as also described in Section 3.2.

The main idea behind our decision mechanism is the assumption that call sites which are visited the most are the best candidates for inlining. Because of that our mechanism greedily chooses to inline F_b into F_a when:

$$\forall i, j. NumberOfCalls_{a,b} \geq NumberOfCalls_{i,j}$$

where $NumberOfCalls_{i,j}$ is the number of calls from F_i to F_j.

Of course, inlining has to be restrained, so that it does not happen for every function call in the program. We achieve this by limiting the maximum code size of each module as seen below:

$$MaxCodeSize = min\left(\frac{SmallCodeSize}{InitialCodesize} + 1.1, 2.1\right) * InitialCodeSize$$

We measured code size as the number of instructions in the Icode of a module and that *SmallCodeSize* is the size of the smallest module in the benchmarks that were used for evaluation. To better clarify, a module m that has $InitialCodeSize_m = 2 * SmallCodeSize$, will be allowed to grow up to $MaxCodeSize_m = 1.6 * InitialCodeSize_m$. Note that the exact configuration of the maximum code size formula have been chosen after evaluating its performance against some alternatives. However, we have not extensively tuned it, and it could certainly be improved.

A detailed description of the decision algorithm follows. Inlining decisions are made iteratively until there are no more calls to inline or until the module has reached the maximum code size. The iteration acts on the following data structures.

- A priority queue that contains pairs of functions (F_i, F_j) and the number of calls $NoC_{i,j}$ from F_i to F_j for each such pair. This priority queue supports two basic operations:
  - Delete-Maximum: which deletes and returns the pair with the maximum number of calls.
  - Update: which updates a pair with a new number of calls.
- A map of the total calls to each function, which is initially constructed for each F_x by adding the number of calls for all pairs on the priority queue $TC_x = \sum_i NoC_{i,x}$.
- A map of the code size of each function, which is used to compute whether the module has become larger than the specified limit.

The main loop works as follows:

(1) Delete-Maximum pair of functions (F_x, F_y) from the priority queue.
(2) Check whether inlining the pair (F_x, F_y) makes the module exceed the maximum code size limit. If it does, then the loop continues, otherwise:
   (a) All the local calls to F_y in F_x are inlined.
   (b) The set of already inlined pairs is updated with the pair (F_x, F_y).
   (c) The map of function sizes is updated for function F_x based on its new size after the inline.
   (d) The number of calls of every pair (F_x, F_i) in the priority queue is updated to $NoC_{x,i} + NoC_{x,y} * NoC_{y,i}/TC_y$. In practice this means that every call that was done from F_y will now be also done from F_x.

4.2.2 Implementation. Our inlining implementation is fairly standard except for some details specific to Icode. The steps below describe the inlining process. In case the inlined call is a tail-call, only the first two steps need to be performed.

(1) Variables and labels in the body of the callee are updated to fresh ones so that there is no name clash with the variables and labels in the body of the caller.
(2) The arguments of the function call are moved to the parameters of the callee.
(3) Tail-calls in the body of the callee are transformed to a normal call and a return.
(4) Return instructions in the body of the callee are transformed into a move and a goto instruction that moves the return value to the destination of the call and jumps to the end of the body of the callee.

It is worth mentioning that Erlang uses cooperative scheduling which is implemented by making sure that processes are executed for a number of reductions and then yield. When the reductions of a process are depleted, the process is suspended and another process is scheduled in instead. Reductions are implemented in Icode by the `redtest()` primitive operation (primop) in the beginning of each function body; cf Listing 2. When inlining a function call, the call to `redtest()` in the body of the callee is also removed. This is safe to do, because inlining is bounded anyway.

5 EVALUATION

In this section, we evaluate the performance of HiPErJiT against BEAM, which serves as a baseline for our comparison, and against two systems that also aim to surpass the performance of BEAM. The first of them is the HiPE compiler.[2] The second is the Pyrlang[3] meta-tracing JiT compiler for Erlang, which is a research prototype and not a complete implementation; we report its performance on the subset of benchmarks that it can handle. We were not able to obtain a version of BEAMJIT to include in our comparison, as this system is still (May 2018) not publicly available. However, as mentioned in Section 2.3, BEAMJIT does not achieve performance that is superior to HiPE anyway.

We conducted all experiments on a laptop with an Intel Core i7-4710HQ @ 2.50GHz CPU and 16 GB of RAM running Ubuntu 16.04.

5.1 Profiling Overhead

Our first set of measurements concerns the profiling overhead that HiPErJiT imposes. To obtain a rough worst-case estimate, we used a modified version of HiPErJiT that profiles programs as they run but

[2] For both BEAM and HiPE, we used the 'master' branch of Erlang/OTP 21.0.
[3] We used the latest version of Pyrlang (https://bitbucket.org/hrc706/pyrlang/overview) built using PyPy 5.0.1.

does not compile any module to native code. Note that this scenario is very pessimistic for HiPErJiT, because the common case is that JiT compilation will be triggered for some modules, HiPErJiT will stop profiling them at that point, and these modules will then most likely execute faster, as we will soon see. But even if native code compilation were not to trigger for any module, it would be very easy for HiPErJiT to stop profiling after a certain time period has passed or some other event (e.g., the number of calls that have been profiled has exceeded a threshold) has occurred. In any case, we executed all the benchmark programs, that were used for evaluation, using the modified version of HiPErJiT and our measurements showed that the overhead caused by genealogy profiling is around 10%. More specifically, the overhead we measured ranged from 5% (in most cases) up to 40% for some heavily concurrent programs.

We also separately measured the profiling overhead on all concurrent benchmarks with and without genealogy profiling, to measure the benefit of genealogy over standard profiling. The average overhead that standard profiling imposes on concurrent benchmarks is 19%, while the average overhead of genealogy profiling on such benchmarks is 13%.

5.2 Evaluation on Small Benchmark Programs

The benchmarks we used come from the ErLLVM benchmark suite[4], which has been previously used for the evaluation of HiPE [13], ErLLVM [24], BEAMJIT [6], and Pyrlang [11]. The benchmarks can be split in two sets: (1) a set of small, relatively simple but quite representative Erlang programs, and (2) the set of Erlang programs from the Computer Language Benchmarks Game (CLBG)[5] as they were when the ErLLVM benchmark suite was created.

Comparing the performance of a Just-in-Time with an ahead-of-time compiler on small benchmarks is tricky, as the JiT compiler also pays the overhead of compilation during the program's execution. Moreover, a JiT compiler needs some time to warm up. Because of that, we have executed each benchmark enough times to allow HiPErJiT to compile the relevant application modules to native code. More specifically we run each benchmark $2 * N$ times, where after approximately N times HiPErJiT had compiled all relevant modules to native code. We report three numbers for HiPErJiT: the speedup achieved when considering the total execution time, the speedup achieved when disregarding the first run, and the speedup achieved in the last N runs, when a steady state has been reached.[6] We also use two configurations of HiPE: one with the maximum level of optimization (`o3`), and one where static inlining (using the `{inline_size,25}` compiler directive) has been performed besides `o3`.

The speedup of HiPErJiT, HiPE, and Pyrlang compared to BEAM for the small benchmarks is shown in Figs. 5 and 6. (We have split them into two figures based on scale of the y-axis, the speedup over BEAM.) Note that the speedups that we report are averages of five different executions of each benchmark multi-run. The black vertical lines indicate the standard deviation. The overall average speedup for each configuration is summarized in Table 1.

[4] https://github.com/cstavr/erllvm-bench
[5] http://benchmarksgame.alioth.debian.org/
[6] At the moment, HiPErJiT does not perform any code decompilation or deoptimization and therefore it always reaches a steady state after it has compiled and optimized all relevant modules.

Table 1: Speedup over BEAM for the small benchmarks.

Configuration	Speedup
HiPE	2.04
HiPE + Static inlining	2.09
Pyrlang	1.14
HiPErJiT	1.63
HiPErJiT w/o 1st run	1.98
HiPErJiT last 50% runs	2.31

Table 2: Speedup over BEAM for the CLBG programs.

Configuration	Speedup
HiPE	2.10
HiPE + Static inlining	2.10
HiPErJiT	1.77
HiPErJiT w/o 1st run	2.03
HiPErJiT last 50% runs	2.15

Overall, the performance of HiPErJiT is almost two times better than BEAM and Pyrlang, but slightly worse than HiPE. However, there also exist five benchmarks (nrev, qsort, fib, smith and tak) where HiPErJiT, in its steady state (the last 50% of runs), surpasses HiPE's performance. This strongly indicates that the profile-driven optimizations that HiPErJiT performs lead to more efficient code, compared to that of an ahead-of-time native code compiler performing static inlining.

Out of those five benchmarks, the most interesting one is smith. It is an implementation of the Smith-Waterman DNA sequence matching algorithm. The reason why HiPErJiT offers better speedup over HiPE is that profile-driven inlining manages to inline functions `alpha_beta_penalty/2` and `max/2` in `match_entry/5`, which is the most time-critical function of the program, thus allowing further optimizations to improve performance. Static inlining on the other hand does not inline `max/2` in `match_entry/5` even if one chooses very large values for the inliner's thresholds.

On the ring benchmark (Fig. 6), where a message is cycled through a ring of processes, HiPErJiT performs worse than both HiPE and BEAM. To be more precise, all systems perform worse than BEAM on this benchmark. The main reason is that the majority of the execution time is spent on message passing (around 1.5 million messages are sent per second), which is handled by BEAM's runtime system. Finally, there are a lot of process spawns and exits (around 20 thousand per second), which leads to considerable profiling overhead, as HiPErJiT maintains and constantly updates the process tree.

Another interesting benchmark is stable (also Fig. 6), where HiPErJiT performs slightly better than BEAM but visibly worse than HiPE. This is mostly due to the fact that there are a lot of process spawns and exits in this benchmark (around 240 thousand per second), which leads to significant profiling overhead.

The speedup of HiPErJiT and HiPE compared to BEAM for the CLBG benchmarks is shown in Figs. 7 and 8. The overall average speedup for each configuration is shown in Table 2.

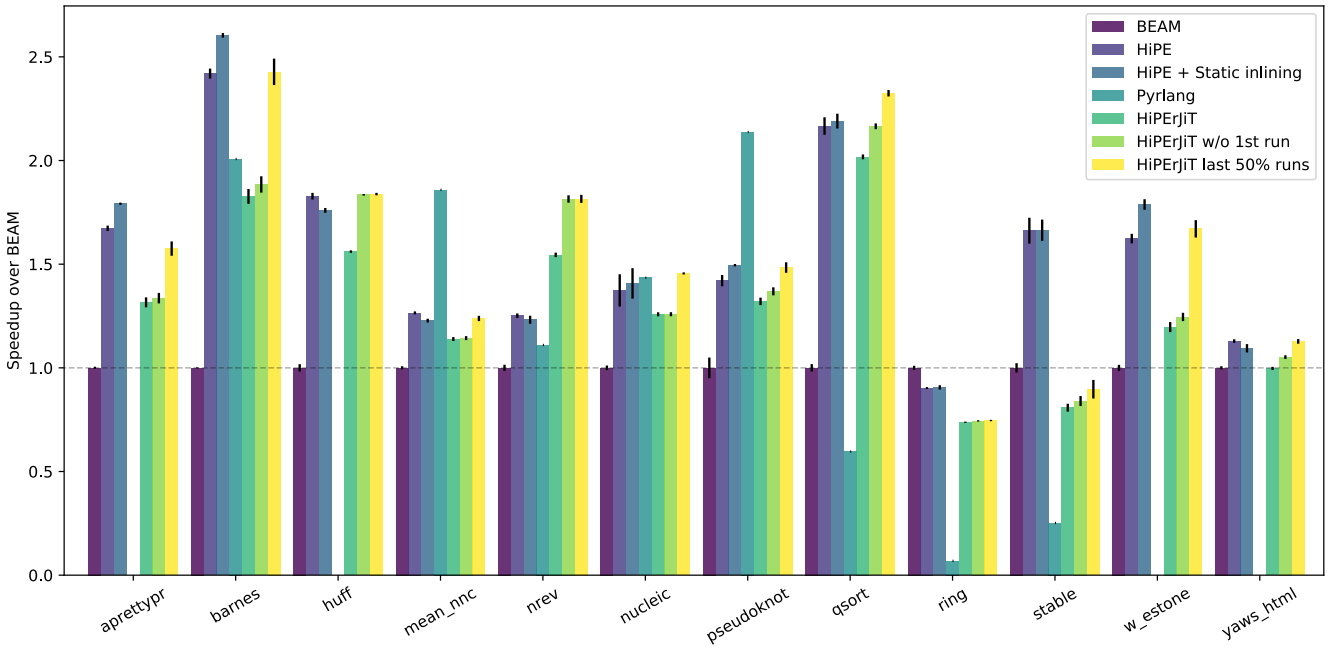


Figure 5: Speedup over BEAM on small benchmarks.

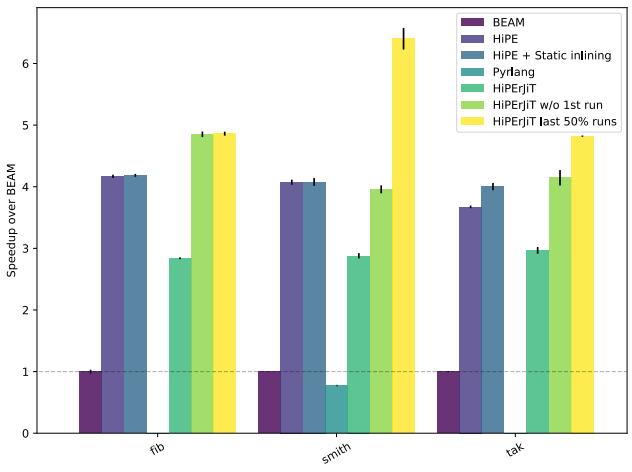


Figure 6: Speedup over BEAM on small benchmarks.

As with the small benchmarks, the performance of HiPErJiT mostly lies between BEAM and HiPE. When excluding the first run, HiPErJiT outperforms both BEAM and HiPE in several benchmarks (except, fannkuchredux, fasta, fibo, and nestedloop). However, HiPErJiT's performance on some benchmarks (revcomp, and threadring) is worse than both BEAM and HiPE because the profiling overhead is higher than the benefit of compilation.

5.3 Evaluation on a Bigger Program

Besides small benchmarks, we also evaluate the performance of HiPErJiT on a program of considerable size and complexity, as results in small or medium-sized benchmarks may not always provide a complete picture for the expected performance of a compiler. The Erlang program we chose, the Dialyzer [17] static analysis tool, is big (about 30, 000 LOC), complex, and highly concurrent [2]. It has also been heavily engineered over the years and comes with hard-coded knowledge of the set of 26 modules it needs to compile to native code upon its start to get maximum performance for most use cases. Using an appropriate option, the user can disable this native code compilation phase, which takes more than a minute on the laptop we use, and in fact this is what we do to get measurements for BEAM and HiPErJiT.

We use Dialyzer in two ways. The first builds a Persistent Lookup Table (PLT) containing cached type information for all modules under erts, compiler, crypto, hipe, kernel, stdlib and syntax_tools. The second analyzes these applications for type errors and other discrepancies. The speedups of HiPErJiT and HiPE compared to BEAM for the two use ways of using Dialyzer are shown in Table 3.

The results show that HiPErJiT achieves performance which is better than BEAM's but worse than HiPE's. There are various reasons for this. First of all, Dialyzer is a complex application where functions from many different modules of Erlang/OTP (50–100) are called with arguments of significant size (e.g., the source code of the applications that are analyzed). This leads to considerable tracing and bookkeeping overhead. Second, some of the called modules contain very large, often compiler-generated, functions and their compilation takes considerable time (the total compilation time is about 70 seconds, which is a significant portion of the total time).

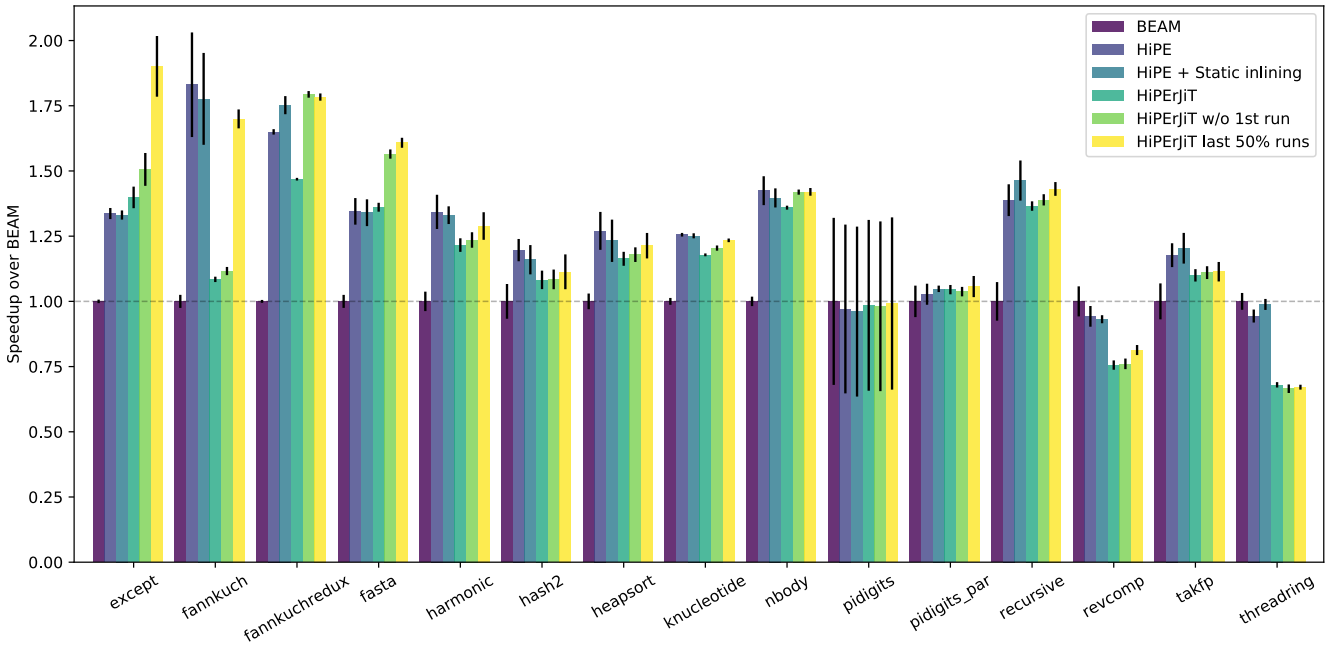


Figure 7: Speedup over BEAM on the CLBG programs.

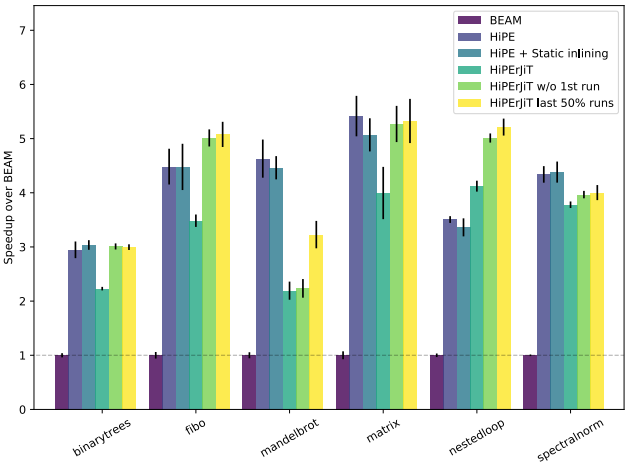


Figure 8: Speedup over BEAM on the CLBG programs.

Table 3: Speedups over BEAM for two Dialyzer use cases.

	Dialyzer Speedup	
Configuration	Building PLT	Analyzing
HiPE	1.78	1.73
HiPE + Static inlining	1.78	1.77
HiPErJiT	1.46	1.42
HiPErJiT w/o 1st run	1.61	1.55
HiPErJiT last 50% runs	1.67	1.60

Finally, HiPErJiT does not compile all modules from the start, which means that a percentage of the time is spent running interpreted code and performing mode switches which are more expensive than same-mode calls. In contrast, HiPE has hard-coded knowledge of "the best" set of modules to natively compile before the analysis starts, and runs native code from the beginning of the analysis.

6 CONCLUDING REMARKS AND FUTURE WORK

We have presented HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on the HiPE native code compiler. It offers performance which is better than BEAM's and comparable to HiPE's, on most benchmarks. Aside from performance, we have been careful to preserve features such as hot code loading that are considered important for Erlang's application domain, and have made design decisions that try to maximize the chances that HiPErJiT remains easily maintainable and in-sync with components of the Erlang/OTP implementation. In particular, besides employing the HiPE native code compiler for most of its optimizations, HiPErJiT uses the same concurrency support that the Erlang Run-Time System provides, and relies upon the tracing infrastructure that it offers. Thus, it can straightforwardly profit from any improvements that may occur in these components.

Despite the fact that the current implementation of HiPErJiT is quite robust and performs reasonably well, profile-driven JiT compilers are primarily engineering artifacts and can never be considered completely "done". Besides making the current optimizations more effective and implementing additional ones, one item which is quite high on our "to do" list is investigating techniques that

reduce the profiling overhead, especially in heavily concurrent applications. For the current state of HiPErJiT, the profiling overhead is bigger than we would like it to be but, on the other hand, it's not really a major concern because once HiPErJiT decides to compile a module to native code the profiling of its functions stops and the overhead drops to zero. But it will become an issue if HiPErJiT becomes a JiT compiler that performs lifelong feedback-directed optimization of programs, which is a direction that we want to pursue.

ACKNOWLEDGMENTS

Most of this work was done at the National Technical University of Athens, Greece, as part of the first author's diploma thesis. The research time of the second author was partly supported by the Swedish Research Council (Vetenskapsrådet). We thank the IFL'2018 reviewers for comments that have improved the presentation of our work.

REFERENCES

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. 2000. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[2] Stavros Aronis and Konstantinos Sagonas. 2013. On Using Erlang for Parallelization — Experience from Parallelizing Dialyzer. In *Trends in Functional Programming, 13th International Symposium, TFP 2012, Revised Selected Papers (LNCS)*, Vol. 7829. Springer, 295–310. https://doi.org/10.1007/978-3-642-40447-4_19

[3] John Aycock. 2003. A Brief History of Just-in-time. *ACM Comput. Surv.* 35, 2 (2003), 97–113. https://doi.org/10.1145/857076.857077

[4] Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[6] Frej Drejhammar and Lars Rasmusson. 2014. BEAMJIT: A Just-in-time Compiling Runtime for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang (Erlang '14)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2633448.2633450

[7] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J Eggers. 1999. An Evaluation of Staged Run-time Optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 293–304. https://doi.org/10.1145/301618.301683

[8] Mor Harchol-Balter and Allen B Downey. 1997. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Trans. Comput. Syst.* 15, 3 (1997), 253–285. https://doi.org/10.1145/263326.263344

[9] Kim Hazelwood and David Grove. 2003. Adaptive Online Context-sensitive Inlining. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 253–264. http://dl.acm.org/citation.cfm?id=776261.776289

[10] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[11] Ruochen Huang, Hidehiko Masuhara, and Tomoyuki Aotani. 2016. Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler. In *Post-Proceeding of the 17th Symposium on Trends in Functional Programming*. https://tfp2016.org/papers/TFP{_}2016{_}paper{_}16.pdf

[12] Erik Johansson and Christer Jonsson. 1996. *Native Code Compilation for Erlang*. Technical Report. Computing Science Department, Uppsala University.

[13] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. 2000. A High Performance Erlang System. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '00)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/351268.351273

[14] Madhukar N Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. 2013. Improved Type Specialization for Dynamic Scripting Languages. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2508168.2508177

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75—-. http://dl.acm.org/citation.cfm?id=977395.977673

[16] Tobias Lindahl and Konstantinos Sagonas. 2003. Unboxed Compilation of Floating Point Arithmetic in a Dynamically Typed Language Environment. In *Proceedings of the 14th International Conference on Implementation of Functional Languages (IFL'02)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–149. http://dl.acm.org/citation.cfm?id=1756972.1756981

[17] Tobias Lindahl and Konstantinos Sagonas. 2004. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, Wei-Ngan Chin (Ed.). Springer-Verlag, Berlin, Heidelberg, 91–106. https://doi.org/10.1007/978-3-540-30477-7_7

[18] Tobias Lindahl and Konstantinos Sagonas. 2005. TypEr: A Type Annotator of Erlang Code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang (Erlang '05)*. ACM, New York, NY, USA, 17–25. https://doi.org/10.1145/1088361.1088366

[19] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical Type Inference Based on Success Typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1140335.1140356

[20] Daniel Luna, Mikael Pettersson, and Konstantinos Sagonas. 2004. HiPE on AMD64. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 38–47. https://doi.org/10.1145/1022471.1022478

[21] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. 2002. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Proceedings (LNCS)*, Vol. 2441. Springer, Berlin, Heidelberg, 228–244. https://doi.org/10.1007/3-540-45788-7_14

[22] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (2002), 393–434. https://doi.org/10.1017/S0956796802004331

[23] Konstantinos Sagonas, Mikael Pettersson, Richard Carlsson, Per Gustafsson, and Tobias Lindahl. 2003. All You Wanted to Know About the HiPE Compiler: (but Might Have Been Afraid to Ask). In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang (Erlang '03)*. ACM, New York, NY, USA, 36–42. https://doi.org/10.1145/940880.940886

[24] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. 2012. ErLLVM: an LLVM Backend for Erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*. ACM, New York, NY, USA, 21–32. https://doi.org/10.1145/2364489.2364494

[25] Andre Santos. 1995. *Compilation by transformation for non-strict functional languages*. Ph.D. Dissertation. University of Glasgow, Scotland. https://www.microsoft.com/en-us/research/publication/compilation-transformation-non-strict-functional-languages/

[26] Manuel Serrano. 1997. Inline expansion: When and how?. In *Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP '97 Including a Special Track on Declarative Programming Languages in Education Southampton, UK, September 3–5, 1997 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 143–157. https://doi.org/10.1007/BFb0033842

[27] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2002. An Empirical Study of Method In-lining for a Java Just-in-Time Compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. USENIX Association, Berkeley, CA, USA, 91–104. http://dl.acm.org/citation.cfm?id=648042.744889

[28] Oscar Waddell and R Kent Dybvig. 1997. Fast and effective procedure inlining. In *Static Analysis*, Pascal Van Hentenryck (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–52.

Spine-local Type Inference

Christopher Jenkins
University of Iowa
christopher-jenkins@uiowa.edu

Aaron Stump
University of Iowa
aaron-stump@uiowa.edu

ABSTRACT

We present *spine-local* type inference, a method of partially inferring omitted type annotations for System F based on local type inference. *Local type inference* relies on bidirectional rules to propagate type information into and out of adjacent nodes of the AST and restricts type-argument inference to a single node. Spine-local inference relaxes this restriction, allowing it to occur only within an *application spine*, and improves upon it by using *contextual type-argument inference*. As our goal is to explore the design space of local type inference, we show that, relative to other variants, spine-local type inference better supports desirable features such as first-class curried applications and partial type applications, and it has the ability to infer types for some terms not otherwise possible. Our approach enjoys usual properties of a bidirectional system of having a specification for our inference algorithm and predictable requirements for typing annotations, and in particular maintains some advantages of local type inference such as a relatively simple implementation and a tendency to produce good-quality error messages when type inference fails.

CCS CONCEPTS

• **Software and its engineering → Language features**;

KEYWORDS

bidirectional typechecking, polymorphism, type errors

ACM Reference Format:
Christopher Jenkins and Aaron Stump. 2018. Spine-local Type Inference. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310233

1 INTRODUCTION

Local type inference[16] is a simple yet effective partial method for inferring types for programs. In contrast to complete methods (e.g. the Damas-Milner type system[3]) which can type programs without any annotations by restricting the type language, *partial* methods require explicit annotations in some cases and, in exchange, are suitable for languages with rich type features such as impredicativity and subtyping[13, 16], dependent types[23], and higher-rank types[14], where complete type inference may be undecidable.

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310233

Local type inference is also contrasted with *global* inference methods (usually based on unification) which are able to infer more missing annotations by solving typing constraints generated from the entire program. Though more powerful, global inference methods can also be more difficult for programmers to use when type inference fails, as they can generate type errors whose root cause is distant from the location the error is reported[9]. For example, for the expression $\lambda\, x : \mathbb{B}.\, x$ an error message like

`error: expected` $\mathbb{N} \to \mathbb{N}$ `, got` $\mathbb{B} \to \mathbb{B}$

requires users to work backwards to understand why the typechecker tried to unify these two types, which can be non-trivial.

Local type inference address this issue by only propagating typing information between adjacent nodes of the abstract syntax tree (AST), allowing programmers to reason *locally* about type errors. It achieves this by using two main techniques: *bidirectional type inference rules* and *local type-argument inference*.

The first of these techniques, bidirectional type inference, is not unique to local type inference (see [5, 14, 21]), and uses two main judgment forms, often called *synthesis* and *checking* mode. When a term t synthesizes type T, we view this typing information as coming up and out of t and as available for use in typing nearby terms; when t checks against type T (called in this paper the *contextual* type), this information is being pushed down and in to t and is provided by nearby terms. The second of these techniques, local type-argument inference, finds the missing types arguments in polymorphic function applications by using only the type information available at an application node of the AST. For a simple example, consider the expression `id` z where `id` has type $\forall\, X.\, X \to X$ and z has type $\mathbb{N}$. Here we can perform *synthetic* type-argument inference by synthesizing the type of z and comparing this to the type of `id` to infer we should instantiate type variable X with $\mathbb{N}$.

Local type inference has a number of desirable properties. Using just these two techniques it can in practice infer a good deal of type annotations, and those it cannot are predictable and coincide with programmers' expectations that they serve as useful and machine-checked documentation[7, 16]. Without further instrumentation, local type inference already tends to report errors close to where further annotations are required; recently, it has been used as the basis for developing autonomous type-driven debugging and error explanations[17]. The type inference algorithms of [13, 16] admit a specification for their behavior, helping programmers understand the system without requiring knowledge of its implementation. Add to this its relative simplicity and robustness when extended to richer type systems and it seems unsurprising that it is a popular choice for type inference in programming languages.

Unfortunately, local type inference can fail in unituitive ways even allowing for their restricted scope of inferece. Consider trying to check that the expression `pair` $(\lambda\, x.\, x)\; z$ has type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$, assuming `pair` has type $\forall\, X.\, \forall\, Y.\, X \to Y \to \langle X \times Y\rangle$). The inference systems presented in [13, 16] will fail here because $\lambda\, x.\, x$ does not

synthesize a type. The techniques proposed in the literature for dealing with such cases include classifying and avoiding such "hard-to-synthesize" terms[7] and utilizing the partial type information provided by polymorphic functions[13]; the former was dismissed as unsatisfactory by the same authors that introduced it and the latter is of no help in this situation, since the type of `pair` tells us nothing about the expected type of $\lambda\, x.\, x$. What we need in this case is *contextual* type-argument inference, utilizing the information available from the contextual type of the result of the application to know argument $\lambda\, x.\, x$ is expected to have type $\mathbb{N} \to \mathbb{N}$.

Additionally, languages using local type inference usually use fully-uncurried applications in order to maximize the notion of "locality" for type-argument inference, improving its effectiveness. The programmer can still use curried applications if desired, but "they are second-class in this respect."[16]. It is also usual for type arguments to be given in an "all or nothing" fashion in such languages, meaning that even if only *one* cannot be inferred, all must be provided. We believe that currying and partial type applications are useful idioms for functional programming and wish to support them as first-class language features.

1.1 Contributions

In this paper, we explore the design space of local type inference in the setting of System F[6] by developing *spine-local* type inference, an approach that both expands the locality of type-argument inference to an *application spine* and augments its effectiveness by using the *contextual* type of the spine. In doing so, we

- show that we better support first-class currying and partial type applications, and can infer types for some "hard-to-synthesize" terms that other local systems would not type;
- provide a specification for contextual type-argument inference with respect to which we show our algorithm is sound and complete
- give a weak completeness theorem indicating where the programmer can expect type inference to succeed and where it needs additional annotations when it fails.

Spine-local type inference is implemented in Cedille[18], a functional programming language based on the pure type-theory CDLE that contains System F as a sub-language and aims to be a simple language in which one can derive inductive datatypes through impredicative lambda-encodings rather than baking them into its core theory. Therefore, type-inference systems for Cedille need not address these features explicitly, as focusing on the more fundamental task of inferring type annotations improves their usability. Though the setting for this paper is only a sub-language of Cedille, experience tells us that spine-local type inference already makes using Cedille's rich type features much more convenient.

The rest of this paper is organized as follows: in Section 2 we cover the syntax and some useful terminology for our setting; in Section 3 we present the type inference rules constituting a specification for contextual type-argument inference, consider its annotation requirements, and illustrate its use, limitations, and the type errors it presents to users; in Section 4 we show the prototype-matching algorithm implementing contextual type-argument inference; and in Section 5 we discuss how this work compares to other approaches to type inference.

2 INTERNAL AND EXTERNAL LANGUAGE

Type inference can be viewed as a relation between an *internal* language of terms, where all needed typing information is present, and an *external* language, in which programmers work directly and where some of this information can be omitted for their convenience. Under this view, type inference for the external language not only associates a term with some type but also with some *elaborated* term in the internal language in which all missing type information has been restored. In this section, we present the syntax for our internal and external languages as well as introduce some terminology that will be used throughout the rest of this paper.

2.1 Syntax

We take as our internal language explicitly typed System F (see [6]); we review its syntax below:

$$\begin{array}{rc} \textbf{Types} & S, T, U, V ::= X, Y, Z \mid S \to T \mid \forall X.\, T \\ \textbf{Contexts} & \Gamma ::= \cdot \mid \Gamma, X \mid \Gamma, x{:}T \\ \textbf{Internal Terms} & e, p ::= x \mid \lambda\, x{:}T.\, e \mid \Lambda X.\, e \mid e\ e' \mid e[T] \end{array}$$

Types consist of type variables, arrow types, and type quantification, and typing contexts consist of the empty context, type variables (also called the context's *declared* type variables), and term variables associated with their types. The internal language of terms consists of variables, λ-abstractions with annotations on bound variables, Λ-abstractions for polymorphic terms, and term and type applications. Our notational convention in this paper is that term meta-variable e indicates an elaborated term for which all type arguments are known, and p indicates a *partially* elaborated term where some elaborated type arguments are type meta-variables (discussed in Section 3).

The external language contains the same terms as the internal language as well as bare λ-abstractions – that is, λ-abstractions missing an annotation on their bound variable:

$$\textbf{External Terms} \quad t, t' ::= x \mid \lambda\, x{:}T.\, t \mid \lambda\, x.\, t \mid \Lambda X.\, t \mid t\ t' \mid t[T]$$

Types and contexts are the same as for the internal language.

2.2 Terminology

In both the internal and external languages, we say that the *applicand* of a term or type application is the term in the function position. A *head* is either a variable or abstraction (term or type), and an *application spine*[2] (or just *spine*) is a view of an application as consisting of a head (called the *spine head*) followed by a sequence of (term and type) arguments. The *maximal application* of a sub-expression is the spine in which it occurs as an applicand, or just the sub-expression itself if it does not. For example, spine $x[S]\ y\ z$ is the maximal application of itself and its applicand sub-expressions x, $x[S]$, and $x[S]\ y$, with x as head of the spine. Predicate $App(t)$ indicates term t is some term or type application (in either language) and we define it formally as $(\exists\, t_1, t_2.\ t = t_1\ t_2) \vee (\exists\, t', S.\ t = t'[S])$.

Turning to definitions for types and contexts, function $DTV(\Gamma)$ calculates the set of *declared type variables* of context Γ and is

defined recursively by the following set of equations:

$$\begin{aligned} DTV(\cdot) &= \varnothing \\ DTV(\Gamma, X) &= DTV(\Gamma) \cup \{X\} \\ DTV(\Gamma, x{:}T) &= DTV(\Gamma) \end{aligned}$$

Predicate $WF(\Gamma, T)$ indicates that type T is *well-formed* under Γ – that is, all free type variables of T occur as declared type variables in Γ (formally $FV(T) \subseteq DTV(\Gamma)$).

3 TYPE INFERENCE SPECIFICATION

The typing rules for our internal language are standard for explicitly typed System F and are omitted (see Ch. 23 of [15] for a thorough discussion of these rules). We write $\Gamma \vdash e : T$ to indicate that under context Γ internal term e has type T. For type inference in the external language, Figure 1 shows judgment $\vdash_\delta$ which consists mostly of standard (except for *AppSyn* and *AppChk*) bidirectional inference rules with elaboration to the internal language, and Figure 2 shows the specification for contextual type-argument inference. Judgment $\vdash^{\mathrm{P}}$ in Figure 2b handles traversing the spine and judgment $\vdash^{\cdot}$ in Figure 2c types its term applications and performs type-argument inference (both synthetic and contextual). Figure 2a gives a "shim" judgment $\vdash^{\mathrm{I}}$ which bridges the bidirectional rules with the specification for rhetorical purposes (discussed below). While these rules are not fully syntax-directed, they *are* subject-directed, meaning that for each judgment the shape of the term we are typing (i.e. the *subject* of typing) uniquely determines which rule applies.

Bidirectional Rules. We now consider more closely each judgment form and its rules starting with $\vdash_\delta$, the point of entry for type inference. The two modes for type inference, checking and synthesizing, are indicated resp. by $\vdash_\Downarrow$ (suggesting pushing a type down and into a term) and $\vdash_\Uparrow$ (suggesting pulling a type up and out of a term). Following the notational convention of Peyton Jones et al.[14] we abbreviate two inference rules that differ only in their direction to one by writing $\vdash_\delta$, where δ is a parameter ranging over $\{\Uparrow, \Downarrow\}$. We read judgment $\Gamma \vdash_\Uparrow t : T \rightsquigarrow e$ as: "under context Γ, term t synthesizes type T and elaborates to e," and a similar reading for checking mode applies for $\vdash_\Downarrow$. When the direction does not matter, we will simply say that we can *infer* t has type T.

Rule *Var* is standard. Rule *Abs* says we can infer missing type annotation T on a λ-abstraction when we have a contextual arrow type $T \to S$. Rules *AAbs* and *TAbs* say that Λ- and annotated λ-abstractions can have their types either checked or synthesized. *TApp* says that a type application $t[S]$ has its type inferred in either mode when the applicand t synthesizes a quantified type. The reason for this asymmetry between the modes of the conclusion and the premise is that even when in checking mode, it is not clear how to work backwards from type $[S/X]T$ to $\forall X.\, T$.

AppSyn and *AppChk* are invoked on maximal applications and are the first non-standard rules. To understand how these rules work, we must 1) explain the "shim" judgment $\vdash^{\mathrm{I}}$ serving as the interface for contextual type-argument inference and 2) define meta-language function *MV*. Read $\Gamma; T_? \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (p, \sigma)$ as "under context Γ and with (optional) contextual type $T_?$, we partially infer application $t\ t'$ has type T with elaboration p and solution σ," where $T_?$ indicates either a type T (as in *AppChk*) or no contextual type information ? (as in *AppSyn*) and σ is a substitution mapping a subset of the meta-variables (i.e. the originally omitted type arguments) in p to contextually-inferred type arguments.

In rule *AppSyn*, ? is an artifice provided to $\vdash^{\mathrm{I}}$ solely to indicate no contextual type is available and is not propagated throughout the other rules. We constrain σ to be the identity substitution (notation σ_{id}) and insist the elaborated term has no unsolved meta-variables, matching our intuition that all type arguments must be inferred synthetically. In rule *AppChk*, we provide the contextual type to $\vdash^{\mathrm{I}}$ and check (implicitly) that it equals $\sigma\ T$ and (explicitly) that all remaining meta-variables in p are solved by σ, then elaborate $\sigma\ p$ (the replacement of each meta-variable in p with its mapping in σ). Shared by both is the second premise of the (anonymous) rule introducing $\vdash^{\mathrm{I}}$ that σ solves precisely the meta-variables of the partially inferred type T for application $t\ t'$.

Meta-variables. What do we mean by the "meta-variables" of partial elaborations and types? When t is a term application with some type arguments omitted in its spine, then (under context Γ) its partial elaboration p from type-argument inference fills in each missing type argument with either a type (well-formed under Γ) or with a *meta-variable* (a type variable not declared in Γ) depending on whether it was inferred through synthesizing the types of the term arguments of t. For example, if $t = \mathtt{pair}\ (\lambda x.\, x)\ z$ and we wanted to check that it has type $T = \langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$ under a typing context Γ associating `pair` with type $\forall X.\, \forall Y.\, X \to Y \to \langle X \times Y\rangle$ and z with type $\mathbb{N}$, then we could derive

$$\Gamma; T \vdash^{\mathrm{I}} t : \langle X \times \mathbb{N}\rangle \rightsquigarrow (\mathtt{pair}[X][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z, [\mathbb{N} \to \mathbb{N}/X])$$

(assuming some base type $\mathbb{N}$, some family of base types $\langle S \times T\rangle$ for all types S and T, and assuming X is not declared in Γ.) Looking at the partial elaboration of t, we would see that type argument X was inferred from its contextual type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$ and that Y was inferred from the synthesized types of the arguments to `pair` (in this case z).

Meta-variables never occur in a judgment formed by $\vdash_\delta$, only in the judgments of Figure 2. In particular, these rules enforce that meta-variables in a partial elaboration p can occur *only* as type arguments in its spine, not within its head or term arguments. This restriction guarantees *spine-local* type-argument inference and helps to narrow the programmer's focus when debugging type errors. Furthermore, meta-variables correspond to omitted type arguments *injectively*, simplifying the kind of reasoning needed for debugging type errors. We make this precise by defining meta-language function $MV(\Gamma, -)$ which yields the set of meta-variables occurring in its second argument with respect to the context Γ. *MV* is overloaded to take both types and elaborated terms for its second argument: for types we define $MV(\Gamma, T) = FV(T) - DTV(\Gamma)$, the set of free variables in T less the declared type variables of Γ; for terms, $MV(\Gamma, p)$ is defined recursively by the following equations:

$$\begin{aligned} MV(\Gamma, p) &= \varnothing && \text{when } \neg App(p) \\ MV(\Gamma, p[X]) &= MV(\Gamma, p) \cup \{X\} && \text{when } X \notin DTV(\Gamma) \\ MV(\Gamma, p[S]) &= MV(\Gamma, p) && \text{when } WF(\Gamma, S) \\ MV(\Gamma, p\ e) &= MV(\Gamma, p) \end{aligned}$$

Using our running example where the subject t is $\mathtt{pair}\ (\lambda x.\, x)\ z$ we can now show how the meta-variable checks are used in rules *AppSyn* and *AppChk*. We have for our partially elaborated term that

$$\boxed{\Gamma \vdash_\delta t : T \rightsquigarrow e} \qquad \frac{}{\Gamma \vdash_\delta x : \Gamma(x) \rightsquigarrow x}\ Var \qquad \frac{\Gamma, x{:}T \vdash_\Downarrow t : S \rightsquigarrow e}{\Gamma \vdash_\Downarrow \lambda x.\, t : T \to S \rightsquigarrow \lambda x{:}T.\, e}\ Abs$$

$$\frac{\Gamma, x{:}T \vdash_\delta t : S \rightsquigarrow e}{\Gamma \vdash_\delta \lambda x{:}T.\, t : T \to S \rightsquigarrow \lambda x{:}T.\, e}\ AAbs \qquad \frac{\Gamma, X \vdash_\delta t : T \rightsquigarrow e}{\Gamma \vdash_\delta \Lambda X.\, t : \forall X.T \rightsquigarrow \Lambda X.\, e}\ TAbs \qquad \frac{\Gamma \vdash_\Uparrow t : \forall X.T \rightsquigarrow e}{\Gamma \vdash_\delta t[S] : [S/X]T \rightsquigarrow e[S]}\ TApp$$

$$\frac{\Gamma; ? \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (e, \sigma_{id}) \quad MV(\Gamma, e) = \varnothing}{\Gamma \vdash_\Uparrow t\ t' : T \rightsquigarrow e}\ AppSyn \qquad \frac{\Gamma; \sigma\ T \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (p, \sigma) \quad MV(\Gamma, p) = dom(\sigma)}{\Gamma \vdash_\Downarrow t\ t' : \sigma\ T \rightsquigarrow \sigma\ p}\ AppChk$$

Figure 1: Bidirectional inference rules with elaboration

(a) Shim (specification)

$$\frac{\Gamma \vdash^{\mathrm{P}} t\ t' : T \rightsquigarrow (p, \sigma) \quad MV(\Gamma, T) = dom(\sigma) \quad T_? \in \{?, \sigma\ T\}}{\Gamma; T_? \vdash^{\mathrm{I}} t\ t' : T \rightsquigarrow (p, \sigma)}$$

(b) $\boxed{\Gamma \vdash^{\mathbf{P}} t : T \rightsquigarrow (p, \sigma)}$

$$\frac{\neg App(t) \quad \Gamma \vdash_\Uparrow t : T \rightsquigarrow e}{\Gamma \vdash^{\mathrm{P}} t : T \rightsquigarrow (e, \sigma_{id})}\ PHead \qquad \frac{\Gamma \vdash^{\mathrm{P}} t : \forall X.T \rightsquigarrow (p, \sigma)}{\Gamma \vdash^{\mathrm{P}} t[S] : [S/X]T \rightsquigarrow (p[S], \sigma)}\ PTApp \qquad \frac{\Gamma \vdash^{\mathrm{P}} t : T \rightsquigarrow (p, \sigma) \quad \Gamma \vdash^{\cdot} (p{:}T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')}{\Gamma \vdash^{\mathrm{P}} t\ t' : T' \rightsquigarrow (p', \sigma')}\ PApp$$

(c) $\boxed{\Gamma \vdash^{\cdot} (p{:}T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')}$

$$\frac{\sigma'' \in \{\sigma, [S/X] \circ \sigma\} \quad WF(\Gamma, S) \quad \Gamma \vdash^{\cdot} (p[X]{:}T, \sigma'') \cdot t' : T' \rightsquigarrow (p', \sigma')}{\Gamma \vdash^{\cdot} (p{:}\forall X.\, T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')}\ PForall \qquad \frac{MV(\Gamma, \sigma\ S) = \varnothing \quad \Gamma \vdash_\Downarrow t' : \sigma\ S \rightsquigarrow e'}{\Gamma \vdash^{\cdot} (p{:}S \to T, \sigma) \cdot t' : T \rightsquigarrow (p\ e', \sigma)}\ PChk$$

$$\frac{MV(\Gamma, \sigma\ S) = \overline{Y} \neq \varnothing \quad \Gamma \vdash_\Uparrow t' : \overline{[U/Y]}\ \sigma\ S \rightsquigarrow e'}{\Gamma \vdash^{\cdot} (p{:}S \to T, \sigma) \cdot t' : \overline{[U/Y]}\ T \rightsquigarrow ((\overline{[U/Y]}\ p)\ e', \sigma)}\ PSyn$$

Figure 2: Specification for contextual type-argument inference

$MV(\Gamma, \texttt{pair}[X][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z) = \{X\}$ and also for our type that $MV(\Gamma, \langle X \times \mathbb{N} \rangle) = \{X\}$. If we have a derivation of the judgment above formed by $\vdash^{\mathrm{I}}$ we can then derive with rule *AppChk*

$$\Gamma \vdash_\Downarrow t : \langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle \rightsquigarrow \texttt{pair}[\mathbb{N} \to \mathbb{N}][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z)$$

because substitution $[\mathbb{N} \to \mathbb{N}/X]$ solves the remaining meta-variable X in the elaborated term and type, and when utilized on the partially inferred type $\langle X \times \mathbb{N} \rangle$ yields the contextual type for the term. However, we would not be able to derive with rule *AppSyn*

$$\Gamma \vdash_\Uparrow t : \langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle \rightsquigarrow \texttt{pair}[\mathbb{N} \to \mathbb{N}][\mathbb{N}]\ (\lambda x{:}\mathbb{N}.\, x)\ z)$$

since we do not have σ_{id} as our solution and we have meta-variable X remaining in our partial elaboration and type. Together, the checks in *AppSyn* and *AppChk* ensure that meta-variables are never passed up and out of a maximal application during type inference.

Specification Rules. Judgment $\vdash^{\mathrm{I}}$ serves as an interface to spine-local type-argument inference. In Figure 2a it is defined in terms of the specification for contextual type-argument inference given by judgments $\vdash^{\mathrm{P}}$ and $\vdash^{\cdot}$; we call it a "shim" judgment because in Figure 4a we give for it an alternative definition using the algorithmic rules in which the condition $MV(\Gamma, T) = dom(\sigma)$ is not needed. Its purpose, then, is to cleanly delineate what we consider specification and implementation for our inference system.

Though the details of maintaining spine-locality and performing synthetic type-argument inference permeate the inference rules for $\vdash^{\mathrm{P}}$ and $\vdash^{\cdot}$, these rules form a specification in that they fully abstract away the details of contextual type-argument inference, describing how solutions are used but omitting how they are generated. Spine-locality in particular contributes to our specification's size – what would be one or two rules in a fully-uncurried language with all-or-nothing type argument applications is decomposed in our system into multiple inference rules to support currying and partial type applications.

Judgment $\vdash^{\mathrm{P}}$ contains three rules and serves to dig through a spine until it reaches its head, then work back up the spine typing its term and type applications. The reading for it is the same as for $\vdash^{\mathrm{I}}$, less the optional contextual type. Rule *PHead* types the spine head t by deferring to $\vdash_\Uparrow$; our partial solution is σ_{id} since no meta-variables are present in a judgment formed by $\vdash_\Uparrow$. *PTApp* is similar to *TApp* except it additionally propagates solution σ. Rule *PApp* is used for term applications: first it partially synthesizes a type T for the applicand and then it uses judgment $\vdash^{\cdot}$ to ensure that a function of type T can be applied to the argument t'

Judgment $\vdash^{\cdot}$ performs synthetic and contextual type-argument inference and ensures that term applications with omitted type arguments are well-typed. We read $\Gamma \vdash^{\cdot} (p{:}T, \sigma) \cdot t' : T' \rightsquigarrow (p', \sigma')$

as "under context Γ, elaborated applicand p of partial type T together with solution σ can be applied to term t'; the application has type T' and elaborates p' with solution σ'."

Contextual type-argument inference happens in rule *PForall*, which says that when the applicand has type $\forall X.\, T$ we can choose to guess any well-formed S for our contextual type argument by picking $\sigma'' = [S/X] \circ \sigma$ (indicating σ'' contains all the mappings present in σ and an additional mapping S for X), or choose to attempt to synthesize it later from an argument by picking $\sigma'' = \sigma$. *The details of which S to guess, or whether we should guess at all, are not present in this specificational rule.* In both cases, we elaborate the applicand to $p[X]$ of type T and check that it can be applied to t' – we do this even when we guess S for X to maintain the invariant that all partial elaborations p and solutions σ we generate satisfy $dom(\sigma) \subseteq MV(\Gamma, p)$, needed when checking in the (specificational) rule for $\vdash^{\mathrm{I}}$ that these guessed solutions are ultimately justified by the contextual type (if any) of our maximal application.

We illustrate the use of *PForall* with an example: if the subject of (i.e. input to) judgment $\vdash^{\cdot}$ is

$$(\mathtt{pair} : \forall X.\, \forall Y.\, X \to Y \to \langle X \times Y \rangle, \sigma_{id}) \cdot (\lambda\, x.\, x)$$

then after two uses of rule *PForall* where we guess $\mathbb{N} \to \mathbb{N}$ for X and decline to guess for Y the subject would be:

$$(\mathtt{pair}[X][Y] : X \to Y \to \langle X \times Y \rangle, [\mathbb{N} \to \mathbb{N}/X]) \cdot (\lambda\, x.\, x)$$

After working through omitted type arguments, $\vdash^{\cdot}$ requires that we eventually reveal some arrow type $S \to T$ to type a term application. When it does we have two cases, handled resp. by *PChk* and *PSyn*: either the domain type S of applicand p together with solution σ provide enough information to fully know the expected type for argument t' (i.e. $MV(\Gamma, \sigma\ p) = \varnothing$), or else they do not and we have some non-empty set of unsolved meta-variables $\overline{Y}$ in S corresponding to type arguments we must synthesize. Having full knowledge, in *PChk* we check t' has type $\sigma\ S$; otherwise, in *PSyn* we try to solve meta-variables $\overline{Y}$ by synthesizing a type for t' and checking it is instantiation $\overline{[U/Y]}$ (vectorized notation for the simultaneous substitution of types $\overline{U}$ for $\overline{Y}$) of $\sigma\ S$. Once done, we conclude with result type $\overline{[U/Y]}\ T$ and elaboration $(\overline{[U/Y]}\ p)\ e$ for the application, as the meta-variables $\overline{Y}$ of p corresponding to omitted type arguments have now been fully solved by type-argument synthesis. Together, *PChk* and *PSyn* prevent meta-variables from being passed down to term argument t', as we require that it either check against or synthesize a well-formed type.

We illustrate the use of rule *PSyn* with and example: suppose that under context Γ the subject of judgment $\vdash^{\cdot}$ is

$$(\mathtt{pair}[X][Y]\ (\lambda\, x : \mathbb{N}.\, x) : Y \to \langle X \times Y \rangle, [\mathbb{N} \to \mathbb{N}/X]) \cdot z$$

and furthermore that $\Gamma \vdash_{\Uparrow} z : \mathbb{N}$. Then we have instantiation $[\mathbb{N}/Y]$ from synthetic type-argument inference and use it to produce for the application the result type $[\mathbb{N}/Y]\ \langle X \times Y \rangle = \langle X \times \mathbb{N} \rangle$ and the elaboration $\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x : \mathbb{N}.\, x)\ z$. Note that synthesized type arguments are used *eagerly*, meaning that the typing information synthesized from earlier arguments can in some cases be used to infer the types of later arguments in *checking* mode (see Section 3.2). This is reminiscent of *greedy* type-argument inference for type systems with subtyping[1, 4], which is known to cause unintuitive type inference failures due to sub-optimal type arguments (i.e. less general wrt to the subtyping relation) being inferred. As System F lacks subtyping, this problem does not affect our type inference system and we can happily utilize synthesized type arguments eagerly (see Section 5 for more discussion of this).

3.1 Soundness, Weak Completeness, and Annotation Requirements

The inference rules in Figure 2 for our external language are *sound* with respect to the typing rules for our internal language (i.e. explicitly typed System F), meaning that elaborations of typeable external terms are typeable at the same type[1]:

Theorem 1. *(Soundness of $\vdash_{\delta}$):*
If $\Gamma \vdash_{\delta} t : T \rightsquigarrow e$ *then* $\Gamma \vdash e : T$.

Our inference rules also enjoy a trivial form of completeness that serves as a sanity-check with respect to the internal language: since any fully annotated internal language term e is in the external language, we expect that e should be typeable using the typing rules for external terms:

Theorem 2. *(Trivial Completeness of $\vdash_{\delta}$):*
If $\Gamma \vdash e : T$ *then* $\Gamma \vdash_{\delta} e : T \rightsquigarrow e$

A more interesting form of completeness comes from asking *which* external terms can be typed – after all, this is precisely what a programmer needs to know when trying to debug type inference failures! Since our external language contains terms without any annotations and our type language is impredicative System F, we know from [22] that type inference is in general undecidable. Therefore, to state a completeness theorem for type inference we must first place some restrictions on the set of external terms that can be the subject of typing.

We start by defining what it means for t to be a *partial erasure* of internal term e. The grammar given in Section 2 for the external language does not fully express where we hope our inference rules will restore missing type information. Specifically, the rules in Figures 1 and 2 will try to infer annotations on bare λ-abstractions and only try to infer missing type arguments that occur in the applicand of a term application. For example, given (well-typed) internal term $x[S_1][S_2]\ y[T]$ and external term $x\ y$, our inference rules will try to infer the missing type arguments S_1 and S_2 but *will not* try to infer the missing T.

A more artificial restriction on partial erasures is that the sequence of type arguments occurring between two terms in an application can only be erased in a right-to-left fashion. For example, given internal term $x[S_1][S_2]\ y[T_1][T_2]\ z$, the external term $x\ y[T_1]\ z$ is a valid erasure (S_1 and S_2 are erased between x and y, and between y and z rightmost T_2 is erased), but term $x[S_2]\ y[T_2]\ z$ is not. This restriction helps preserve soundness of the external type inference rules by ensuring that every explicit type argument preserved in an erasure of an internal term e instantiates the same type variable it did in e; it is artificial because we could instead have introduced notation for "explicitly erased" type arguments in the external language, such as $x[\_][S_2]\ y$, to indicate the first type argument has been erased, or allow type arguments to be given *by*

[1] A complete list of proofs for non-trivial theorems can be found in the proof appendix at http://homepage.cs.uiowa.edu/~cwjnkins/Papers/JS18_Spine-local/proof-appendix.pdf

name such as $x[?X_2 = S_2]\ y$, but chose not to do so to simplify the presentation of our inference rules and language.

The above restrictions for partial erasure are made precise by the functions $\lfloor \_ \rfloor$ and $\lfloor \_ \rfloor_a$ which map an internal term e to sets of partial erasures $\lfloor e \rfloor$. They are defined mutually recursively below:

$$\begin{aligned}
\lfloor \lambda\, x{:}T.\, e \rfloor &= \{\lambda\, x{:}T.\, t \mid t \in \lfloor e \rfloor\} \cup \{\lambda\, x.\, t \mid t \in \lfloor e \rfloor\} \\
\lfloor \Lambda\, X.\, e \rfloor &= \{\Lambda\, X.\, t \mid t \in \lfloor e \rfloor\} \\
\lfloor e\ e' \rfloor &= \{t\ t' \mid t \in \lfloor e \rfloor_a \wedge t' \in \lfloor e' \rfloor\} \\
\lfloor e[S] \rfloor &= \{t[S] \mid t \in \lfloor e \rfloor\} \\
\\
\lfloor e[S] \rfloor_a &= \{t \mid t \in \lfloor e \rfloor_a\} \cup \{t[S] \mid t \in \lfloor e \rfloor\} \\
\lfloor e \rfloor_a &= \lfloor e \rfloor \text{ otherwise}
\end{aligned}$$

We are now ready to state a weak completeness theorem for typing terms in the external language which over-approximates the annotations required for type inference to succeed (we write $\forall\, \overline{X}.\, T$ to mean some number of type quantifications over type T)

Theorem 3. *(Weak completeness of $\vdash_{\Uparrow}$):*
Let e be a term of the internal language and t be an external language term such that $t \in \lfloor e \rfloor$. If $\Gamma \vdash e : T$ then $\Gamma \vdash_{\Uparrow} t : T \rightsquigarrow e$ when the following conditions hold for each sub-expression e' of e, corresponding sub-expression t' of t, and corresponding sub-derivation $\Gamma' \vdash e' : T'$ of $\Gamma \vdash e : T$:

1. *If $e' = \lambda\, x : S.\, e''$ for some S and e'', then $t' = \lambda\, x : S.\, t''$ for some t'*
2. *If e' is a maximal term application in e and if $\Gamma' \vdash^P t' : T'' \rightsquigarrow (p, \sigma_{id})$ for some T'' and p, then $MV(\Gamma, p) = \varnothing$.*
3. *If e' is a term application and $t' = t_1\ t_2$ for some t_1 and t_2, and if $\Gamma' \vdash^P t_1 : T'' \rightsquigarrow (p, \sigma_{id})$ for some T'' and p, then $T'' = \forall\, \overline{X}.\, S_1 \to S_2$ for some S_1 and S_2.*
4. *If e' is a type application and $t' = t''[S]$ for some t'' and S, and $\Gamma' \vdash^P t'' : T'' \rightsquigarrow (p, \sigma_{id})$ for some T'' and p, then $T'' = \forall\, X.\, S'$ for some S'.*

Theorem 3 only considers synthetic type-argument inference, and in practice condition (1) is too conservative thanks to contextual type-argument inference. Though a little heavyweight, our weak completeness theorem can be translated into a reasonable guide for where type annotations are required when type synthesis fails. Conditions (3) and (4) suggest that when the applicand of a term or type application already partially synthesizes some type, the programmer should give enough annotations to at least reveal it has the appropriate shape (resp. a type arrow or quantification) for the application. (2) indicates that type variables that do not occur somewhere corresponding to a term argument of an application should be instantiated explicitly, as there is no way for synthetic type-argument inference to do so. For example, in the expression $f\ z$ if f has type $\forall\, X.\, \forall\, Y.\, Y \to X$ there is no way to instantiate X from synthesizing argument z. Finally, condition (1) we suggest as the programmer's last resort: if the above advice does not help it is because some λ-abstractions need annotations.

Note that in conditions (2), (3), and (4) we are not circularly assuming type synthesis for sub-expressions of partial erasure t succeeds in order to show that it succeeds for t, only that *if* a certain sub-expression can be typed *then* we can make some assumptions about the shape of its type or elaboration. Conditions (3) and (4) in particular are a direct consequence of a design choice we made for our algorithm to maintain injectivity of meta-variables to omitted type arguments. As an alternative, we could instead refine meta-variables when we know something about the shape of their instantiation. For example, if we encountered a term application whose applicand has a meta-variable type X, we know it must have some arrow type and could refine X to $X_1 \to X_2$, where X_1 and X_2 are fresh meta-variables. However, doing so means type errors may now require non-trivial reasoning from users to determine why some meta-variables were introduced in the first place.

Still, we find it somewhat inelegant that our characterization of annotation requirements for type inference is not fully independent of the inference system itself. For programmers using these guidelines, this implies that there must be some way to interactively query the type-checker for different sub-expressions of a program during debugging. Fortunately, many programming languages offer just such a feature in the form of a REPL, meaning that in practice this is not too onerous a requirement to make.

Theorem 3 only states when an external term will synthesize its type, but what about when a term can be *checked* against a type? It is clear from the typing rules in Figure 1 that some terms that fail to synthesize a type may still be successfully checked against a type. Besides typing bare λ-abstractions (which can only have their type checked), checking mode can also reduce the annotation burden implied by condition (2) of Theorem 3: consider again the example $f\ z$ where f has type $\forall\, X.\, \forall\, Y.\, Y \to X$. If instead of attempting type synthesis we were to check that it has some type T then we would not need to provide an explicit type argument to instantiate X.

From these observations and our next result, we have that checking mode of our type inference system can infer the types of strictly more terms than can synthesizing mode – whenever a term synthesizes a type, it can be checked against the same type.

Theorem 4. *(Checking extends synthesizing):*
If $\Gamma \vdash_{\Uparrow} t : T \rightsquigarrow e$ then $\Gamma \vdash_{\Downarrow} t : T \rightsquigarrow e$

3.2 Examples

Successful Type Inference. We conclude this section with some example programs for which the type inference system in Figures 1 and 2 will and will not be able to type. We start with the motivating example from the introduction of checking that the expression $\texttt{pair}\ (\lambda\, x.\, x)\ z$ has type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$, which is not possible in other variants of local type inference. For convenience, we assume the existence of a base type $\mathbb{N}$ and a family of base types $\langle S \times T \rangle$ for all types S and T. These assumptions are admissible as we could define these types using lambda encodings. A full derivation for typing this program is given in Figure 3, including the following abbreviations:

$$\begin{aligned}
I_\times\ X\ Y &= X \to Y \to \langle X \times Y \rangle \\
\Gamma &= \texttt{pair}{:}\forall\, X.\, \forall\, Y.\, I_\times\ X\ Y,\ z{:}\mathbb{N} \\
\sigma &= [\mathbb{N} \to \mathbb{N}/X] \\
p &= \texttt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z
\end{aligned}$$

To type this application $\texttt{pair}\ (\lambda\, x.\, x)\ z$ we first dig through the spine, reach the head pair, and synthesize type $\forall\, X.\, \forall\, Y.\, \mathtt{I}_\times\ X\ Y$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\Uparrow} \mathtt{pair} : \forall X.\, \forall Y.\, I_\times\ X\ Y \rightsquigarrow \mathtt{pair}}\ Var}{\Gamma \vdash^{\mathrm{P}} \mathtt{pair} : \forall X.\, \forall Y.\, I_\times\ X\ Y \rightsquigarrow (\mathtt{pair}, \sigma_{id})}\ PHead \quad \mathcal{D}_1}{\Gamma \vdash^{\mathrm{P}} \mathtt{pair}\ (\lambda\, x.\, x)\ : Y \to \langle X \times \mathbb{N} \rangle \rightsquigarrow (\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PApp \quad \mathcal{D}_2}{\Gamma \vdash^{\mathrm{P}} \mathtt{pair}\ (\lambda\, x.\, x)\ z : \langle X \times \mathbb{N} \rangle \rightsquigarrow (p, \sigma)}\ PApp \quad MV(\Gamma, X \times \mathbb{N}) = dom(\sigma)}{\Gamma \vdash^{\mathrm{I}} \mathtt{pair}\ (\lambda\, x.\, x)\ z : \langle X \times \mathbb{N} \rangle \rightsquigarrow (p, \sigma)} \quad MV(\Gamma, p) = dom(\sigma)}{\Gamma \vdash_{\Downarrow} \mathtt{pair}\ (\lambda\, x.\, x)\ z : \langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle \rightsquigarrow \mathtt{pair}[\mathbb{N} \to \mathbb{N}][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z}\ AppChk$$

$$\mathcal{D}_1 = \dfrac{\dfrac{\dfrac{MV(\Gamma, \sigma\ X) = \varnothing \quad \dfrac{\dfrac{}{\Gamma, x{:}\mathbb{N} \vdash_{\Downarrow} x : \mathbb{N} \rightsquigarrow x}\ Var}{\Gamma \vdash_{\Downarrow} \lambda\, x.\, x : \mathbb{N} \to \mathbb{N} \rightsquigarrow \lambda\, x{:}\mathbb{N}.\, x}\ Abs}{\Gamma \vdash^{\cdot} (\mathtt{pair}[X][Y]{:}I_\times\ X\ Y, \sigma) \cdot (\lambda\, x.\, x) : Y \to \langle X \times Y \rangle \rightsquigarrow (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PChk}{\Gamma \vdash^{\cdot} (\mathtt{pair}[X]{:}\forall Y.\, I_\times\ X\ Y, \sigma) \cdot (\lambda\, x.\, x) : Y \to \langle X \times Y \rangle \rightsquigarrow (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PForall}{\Gamma \vdash^{\cdot} (\mathtt{pair}{:}\forall X.\, \forall Y.\, I_\times\ X\ Y, \sigma_{id}) \cdot (\lambda\, x.\, x) : Y \to \langle X \times Y \rangle \rightsquigarrow (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x), \sigma)}\ PForall$$

$$\mathcal{D}_2 = \dfrac{MV(\Gamma, Y) = \{Y\} \quad \dfrac{}{\Gamma \vdash_{\Uparrow} z : \mathbb{N} \rightsquigarrow z}\ Var}{\Gamma \vdash^{\cdot} (\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x){:}Y \to \langle X \times Y \rangle, \sigma) \cdot z : \langle X \times \mathbb{N} \rangle \rightsquigarrow (\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z, \sigma)}\ PSyn$$

$$\text{where} \quad \begin{array}{rclcrcl} I_\times\ X\ Y & = & X \to Y \to \langle X \times Y \rangle & \quad & \Gamma & = & \mathtt{pair}{:}\forall X.\, \forall Y.\, I_\times\ X\ Y,\ z{:}\mathbb{N} \\ \sigma & = & [\mathbb{N} \to \mathbb{N}/X] & & p & = & \mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z \end{array}$$

Figure 3: Example typing derivation with the specification rules

No meta-variables are generated by judgment $\vdash_{\Uparrow}$ and thus there can be no meta-variable solutions, so we generate solution σ_{id}.

Next we type the first application, `pair` $(\lambda\, x.\, x)$, shown in sub-derivation $\mathcal{D}_1$. In the first invocation of rule *PForall* we guess solution σ for X, and in the second invocation we decline to guess an instantiation for Y (in this example we could have also guessed $\mathbb{N}$ for Y as this information is also available from the contextual type, but choose not to in order to demonstrate the use of all three rules of $\vdash^{\cdot}$). Then using rule *PChk* we check argument $\lambda\, x.\, x$ against $\sigma\ X = \mathbb{N} \to \mathbb{N}$. *This is the point at which the local type inference systems of [13, 16] will fail*: as a bare λ-abstraction this argument will not synthesize a type, and the expected type X as provided by the applicand `pair` alone does not tell us what the missing type annotation should be. However, by using the information provided by the contextual type of the entire application we know it must have type $\mathbb{N} \to \mathbb{N}$. The resulting partial type of the application is $Y \to \langle X \times Y \rangle$, and we propagate solution σ to the rest of the derivation. Note that we elaborate the argument $\lambda\, x.\, x$ of this application to $\lambda\, x{:}\mathbb{N}.\, x$ – we never pass down meta-variables to term arguments, keeping type-argument inference local to the spine.

In sub-derivation $\mathcal{D}_2$ we type (`pair` $(\lambda\, x.\, x)$) z (parentheses added) where our applicand has partial type $Y \to \langle X \times Y \rangle$. We find that we have unsolved meta-variable Y as the expected type for z, so we use rule *PSyn* and synthesize the type $\mathbb{N}$ for z. Using solution $[\mathbb{N}/Y]$, we produce $\langle X \times \mathbb{N} \rangle$ for the resulting type of the application and elaborate the application to $\mathtt{pair}[X][\mathbb{N}]\ (\lambda\, x{:}\mathbb{N}.\, x)\ z$, wherein type argument Y is replaced by $\mathbb{N}$ in the original elaborated applicand $\mathtt{pair}[X][Y]\ (\lambda\, x{:}\mathbb{N}.\, x)$.

Finally, in rule *AppChk* we confirm that the meta-variables remaining in our partial type synthesis of the application is precisely those for which we knew the solutions contextually. For this example, the only remaining meta-variable in both the partially synthesized type and elaboration is X, which is also the only mapping in σ, so type inference succeeds. We use σ to replace all occurrences of X with $\mathbb{N} \to \mathbb{N}$ in the type and elaboration and conclude that term `pair` $(\lambda\, x.\, x)\ z$ can be checked against type $\langle (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \rangle$.

The next example shows how eager use of *synthetic* type-argument inference can type some terms not possible in other variants of local type inference. Consider checking that the expression `rapp` $x\ \lambda\, y.\, y$ has type $\mathbb{N}$, where `rapp` has type $\forall X.\, \forall Y.\, X \to (X \to Y) \to Y$ and x has type $\mathbb{N}$. From the contextual type we know that Y should be instantiated to $\mathbb{N}$, and when we reach application `rapp` x, we learn that X should be instantiated to $\mathbb{N}$ from the synthesized type of x. Together, this gives us enough information to know that argument $\lambda\, y.\, y$ should have type $\mathbb{N} \to \mathbb{N}$. Such eager instantiation is neither novel, nor is it necessarily desirable when extended to richer type languages or more powerful methods of inference (see Section 5), but in our setting it is a useful technique that we can use to infer types for expressions like the one above.

Type Inference Failures. To see where type inference can fail, we again use `pair` $(\lambda\, x.\, x)\ z$ but now ask that it *synthesize* its type. Rule *AppSyn* insists that we make no guesses for meta-variables (as there is no contextual type for the application that they could have come from), so we would need to synthesize a type for argument $\lambda\, x.\, x$ – but our rules do not permit this! In this case the user can expect an error message like the following:

```
expected type: ?X
        error: We are not in checking mode, so bound
               variable x must be annotated
```

where ?X indicates an unsolved meta-variable corresponding to type variable X in the type of `pair`. The situation above corresponds to condition (1) of Theorem 3: in general, if there is not enough information from the type of an applicand and the contextual type of the application spine in which it occurs to fully know the expected types of arguments that are λ-abstractions, then such arguments require explicit type annotations.

We next look at an example corresponding to condition (2) of Theorem 3, namely that the type variables of a polymorphic function that do not correspond to term arguments in an application should be instantiated explicitly. Here we will assume a family of base types $S + T$ for every type S and T, a variable `right` of type $\forall X.\, \forall Y.\, Y \to (X + Y)$, and a variable z of type $\mathbb{N}$. In trying to synthesize a type for the application `right` z the user can expect an error message like:

```
synthesized type: (?X + ℕ)
           error: Unsolved meta-variable ?X
```

indicating that type variable X requires an explicit type argument be provided. Fortunately for the programmer, and unlike the local type inference systems of [13, 16], our system supports partial explicit type application, meaning that X can be instantiated without also explicitly (and redundantly) instantiating Y. On the other hand, these systems are in the setting of System $F_{\leq}$ and can succeed in typing `right` z *without* additional type arguments, as they will instantiate X to the minimal type (with respect to their subtyping relation) `Bot`. Partial type application, then, is more useful for our setting of System F where picking an instantiation in this situation would be arbitrary.

A similar error can occur when synthesizing types for partial (term) applications. For example, the expression `pair` $\lambda\, x : \mathbb{N}.\, x$ would raise the message

```
synthesized type: ?Y → ⟨ℕ → ℕ, ?Y⟩
           error: Unsolved meta-variable ?Y
```

As discussed in Section 3.1, our system is artificially sensitive to the order of quantifiers and would require instantiating type variable X to instantiate Y. However, another option is available to users in this case that is *not* available to other forms of local type inference: if this partial application were *checked* against a type like $\mathbb{N} \to (\mathbb{N} \to \mathbb{N}, \mathbb{N})$, then contextual type-argument inference can infer the instantiation for Y and thus the expression would be well-typed.

A more subtle point of failure for our algorithm corresponds to conditions (3) and (4) of Theorem 3. Even when the head and all arguments of an application spine can synthesize their types, the programmer may still be required to provide some additional type arguments. Consider the expression `bot z`, where `bot`$:\forall X.\, X$ and `z`$:\mathbb{N}$. Even with some contextual type for this expression, type inference still fails because the rules in Figure 2c require that the type of the applicand of a term application reveals some arrow, which $\forall X.\, X$ does not. The programmer would be met with the following error message:

```
applicand type: ?X
         error: The type of an applicand in a term
                application must reveal an arrow
```

prompting the user to provide an explicit type argument for X. To make expression `bot` z typeable, the programmer could write `bot`$[\mathbb{N} \to \mathbb{N}]\ z$, or even `bot`$[\forall Y.\, Y \to Y]\ z$ – our inference rules are able to solve meta-variables introduced by explicit (and even synthetically-inferred) type arguments, as long as there is enough information to reveal a quantifier or arrow in the type of a term or type applicand.

For our last type error example, we consider the situation where the programmer has written an ill-typed program. Local type inference enjoys the property that type errors can be understood *locally*, without any "spooky action" from a distant part of the program. In particular, with local type inference we would like to avoid error messages like the following:

```
synthesized type: 𝔹 → 𝔹
   expected type: ?X := ℕ → ℕ
           error: type mismatch
```

From this error message alone the programmer has no indication of why the expected type is $\mathbb{N} \to \mathbb{N}$! In our type system we have expanded the distance information travels by allowing it to flow from the contextual type of an application to its arguments – so can programmers expect now to see such error messages?. As an example, the error message above could have been generated when checking the expression `pair` $(\lambda\, x : \mathbb{B}.\, x)\ z$ against type $\langle(\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\rangle$, specifically when inferring the type of the first argument. Fortunately, our notion of locality is still quite small and we can easily demystify the type error:

```
synthesized type: 𝔹 → 𝔹
   expected type: ?X := ℕ → ℕ
contextual match: ⟨?X × ?Y⟩ := ⟨(ℕ → ℕ) × ℕ⟩
```

where `contextual match` tells the programmer to compare the partially synthesized and contextual return types of the application spine to determine why X was instantiated to $\mathbb{N} \to \mathbb{N}$. A similar field, `synthetic match`, could tell the programmer that the type of an earlier argument informs the expected type of current one.

4 ALGORITHMIC INFERENCE RULES

The type inference system presented in Section 3 do not constitute an algorithm. Though the rules forming judgment $\vdash^{\cdot}$ indicate *where* and *how* we use contextually-inferred type arguments, they do not specify *what* their instantiations are or even *whether* this information is available to use, and it is not obvious how to work backwards from the second premise in Figure 2a to develop an algorithm.

Figure 4 shows the algorithmic rules implementing contextual type-argument inference. The full algorithm for spine-local type inference, then, consists of the rules in Figure 1 with the shim judgment $\vdash^{\mathrm{I}}$ as defined in Figure 4a. At the heart of the implementation is our prototype matching algorithm; to understand the details of how we implement contextual type-argument inference, we must first discuss this algorithm and the two new syntactic categories it introduces, prototypes and decorated types.

4.1 Prototype Matching

Figure 4d lists the rules for the prototype matching algorithm. We read the judgment $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)$ as: "solving for meta-variables $\overline{X}$, we match type T to prototype P and generate solution σ and decorated type W," and we maintain the invariant that $dom(\sigma) \subseteq \overline{X}$. Meta-variables can only occur in T, thus these are

(a) Shim (algorithm)

$$\frac{\Gamma; T_? \Vdash^? t\ t' : T \rightsquigarrow (p, \sigma) \quad T_? \in \{?, \sigma\ T\}}{\Gamma; T_? \vdash^! t\ t' : T \rightsquigarrow (p, \sigma)}$$

(b) $\boxed{\Gamma; P \Vdash^? t : W \rightsquigarrow (p, \sigma)}$

$$\frac{\neg App(t) \quad \Gamma \Vdash_{\Uparrow} t : T \rightsquigarrow e \quad \varnothing \Vdash^{:=} T := ? \to P \Rightarrow (\sigma_{id}, W)}{\Gamma; ? \to P \Vdash^? t : W \rightsquigarrow (e, \sigma_{id})}\ ?Head \qquad \frac{\Gamma; ? \to P \Vdash^? t : \forall X{=}R.\ W \rightsquigarrow (p, \sigma) \quad R \in \{X, S\}}{\Gamma; ? \to P \Vdash^? t[S] : [S/X]W \rightsquigarrow (p[S], \sigma)}\ ?TApp$$

$$\frac{\Gamma; ? \to P \Vdash^? t : W \rightsquigarrow (p, \sigma) \quad \Gamma \Vdash^{\cdot} (p{:}W, \sigma) \cdot t' : W' \rightsquigarrow (p', \sigma')}{\Gamma; P \Vdash^? t\ t' : W' \rightsquigarrow (p', \sigma')}\ ?App$$

(c) $\boxed{\Gamma \Vdash^{\cdot} (p{:}W, \sigma) \cdot t' : W \rightsquigarrow (p', \sigma')}$

$$\frac{\sigma'' = \text{ if } R{=}X \text{ then } \sigma \text{ else } [R/X]{\circ}\sigma \quad \Gamma \Vdash^{\cdot} (p[X]{:}W, \sigma'') \cdot t' : W' \rightsquigarrow (p', \sigma')}{\Gamma \Vdash^{\cdot} (p{:}\forall X{=}R.\ W, \sigma) \cdot t' : W' \rightsquigarrow (p', \sigma')}\ ?Forall \qquad \frac{MV(\Gamma, \sigma\ S) = \varnothing \quad \Gamma \Vdash_{\Downarrow} t' : S \rightsquigarrow e}{\Gamma \Vdash^{\cdot} (p{:}S \to W, \sigma) \cdot t' : W \rightsquigarrow (p\ e', \sigma)}\ ?Chk$$

$$\frac{MV(\Gamma, \sigma\ S) = \overline{Y} \neq \varnothing \quad \Gamma \Vdash_{\Uparrow} t : \overline{[U/Y]}\ \sigma\ S \rightsquigarrow e}{\Gamma \Vdash^{\cdot} (p{:}S \to W) \cdot t' : \overline{[U/Y]}\ W \rightsquigarrow ((\overline{[U/Y]}\ p)\ e', \sigma)}\ ?Syn$$

(d) $\boxed{\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)}$

$$\frac{\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)}{\overline{X} \Vdash^{:=} S \to T := ? \to P \Rightarrow (\sigma, S \to W)}\ MArr \qquad \frac{\overline{X}, X \Vdash^{:=} T := ? \to P \Rightarrow (\sigma, W)}{\overline{X} \Vdash^{:=} \forall X.\ T := ? \to P \Rightarrow (\sigma - X, \forall X{=}\sigma(X).\ W)}\ MForall$$

$$\frac{\overline{Y} = FV(T) \cap \overline{X} \quad FV(\overline{U}) \cap (BTV(S) \cup \overline{X}) = \varnothing \quad \overline{[U/Y]}\ T = S}{\overline{X} \Vdash^{:=} T := S \Rightarrow (\overline{[U/Y]}, T)}\ MType \qquad \frac{}{\overline{X} \Vdash^{:=} T := ? \Rightarrow (\sigma_{id}, T)}\ M? \qquad \frac{X \in \overline{X}}{\overline{X} \Vdash^{:=} X := ? \to P \Rightarrow (\sigma_{id}, (X, ? \to P))}\ MCurr$$

Figure 4: Algorithm for contextual type argument inference

matching (not unification) rules. The grammar for prototypes and decorated types is given below:

$$\begin{array}{rl} \textbf{Prototypes} & P ::= ? \mid T \mid ? \to P \\ \textbf{Decorated Types} & W ::= T \mid S \to W \mid \forall X{=}X.\ W \mid \forall X{=}S.\ W \\ & \quad \mid (X, ? \to P) \end{array}$$

Prototypes carry the contextual type of the maximal application of a spine. In the base case they are either the uninformative ? (as in *AppSyn*), indicating no contextual type, or they are informative of type T (as in *AppChk*). In this way, prototypes generalize the syntactic category $T_?$ we introduced earlier for optional contextual types. We use the last prototype former $? \to$ as we work our way down an application spine to track the expected arity of its head. For example, if we wished to check that the expression `id suc x` has type $\mathbb{N}$, then when we reached the head `id` using the rules in Figure 4b we would generate for it prototype $? \to ? \to \mathbb{N}$

Decorated types consist of types (also called *plain-decorated* types), an arrow with a regular type as the domain (as prototypes only carry the result type of a maximal application, not of the types of arguments), quantified types whose bound variable X may be decorated with the type to which we expect to instantiate it, and "stuck" decorations. On quantifiers, decoration $X = X$ indicates that P did not inform us of an instantiation for X – we sometimes abbreviate the two cases as $\forall X{=}R.\ W$, where $R \in \{X, S\}$ and $S \neq X$. Finally, we define operation $\lfloor W \rfloor$ erasing decorations from W and producing a type in the expected way, $\lfloor \forall X{=}R.\ W \rfloor = \forall X.\ \lfloor W \rfloor$ and $\lfloor (X, ? \to P) \rfloor = X$

To explain the role of stuck decorations, consider again `id suc x`. Assuming `id` has type $\forall X.\ X \to X$, matching this with prototype $? \to ? \to \mathbb{N}$ generates decorated type $\forall X = X.\ X \to (X, ? \to \mathbb{N})$, meaning that we only know that X will be instantiated to some type that matches $? \to \mathbb{N}$. Stuck decorations occur when the expected arity of a spine head (as tracked by a given prototype) is greater than the arity of the head's synthesized type and are the mechanism by which we propagate a contextual type to a head that is "over-applied" – a not-uncommon occurrence in languages with curried applications!

Turning to the prototype matching algorithm in Figure 4d, rule *MArr* says that we match an arrow type and prototype when we can match their codomains. Rule *MType* says that when the prototype is some type S we must find an instantiation $\overline{[U/Y]}$ (where $\overline{Y} \subseteq \overline{X}$) such that $\overline{[U/Y]}\ T = S$, and rule *M?* says that any type matches with ? with no solutions generated (thus we call ? "uninformative"). In rule *MForall* we match a quantified type with a prototype by

adding bound variable X to our meta-variables and matching the body T to the same prototype; the substitution in the conclusion, $\sigma - X$, is the solution generated from this match less its mapping for X, which is placed in the decoration $X = \sigma(X)$. For example, matching $\forall X.\, \forall Y.\, X \to Y \to X$ with prototype $? \to ? \to \mathbb{N}$ generates decorated type $\forall X{=}\mathbb{N}.\, \forall Y{=}Y.\, X \to Y \to X$. Finally, rule *MCurr* applies when there is incomplete information (in the form of $? \to P$) on how to instantiate a meta-variable; we generate a stuck decoration with identity solution σ_{id}.

We conclude by showing that our prototype matching rules are *functional*; when $\overline{X}$, T, and P are considered as inputs then there is at most one output pair (σ, W):

Theorem 5. *(Function-ness of $\Vdash^{:=}$):*
Given $\overline{X}$, T, and P, if $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)$
and $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma', W')$, then $\sigma = \sigma'$ and $W = W'$

It should also be clear that these rules are *syntax-directed* (i.e. that the rules are non-overlapping and that the inputs to premises and the outputs to the conclusions are uniquely determined), meaning they can be straightforwardly translated to an algorithm.

4.2 Decorated Type Inference

We now discuss the rules in Figures 4b and 4c which implement contextual type-argument inference (as specified by Figures 2b and 2c) by using the prototype matching algorithm. We begin by giving a reading for judgments $\Vdash^{?}$ – read $\Gamma; P \Vdash^{?} t : W \rightsquigarrow (p, \sigma)$ as: "under context Γ and with prototype P, t synthesizes decorated type W and elaborates p with solution σ," where σ again represents the contextually-inferred type arguments.

In rule *AppSyn* we required that the solution generated by $\vdash^{\mathrm{I}}$ in its premise is σ_{id}; in *AppChk* we (implicitly) required that the contextual type is equal to $\sigma\ T$; and now with the algorithmic definition for $\vdash^{\mathrm{I}}$ we appear to be requiring in both that the decorated type generated by $\Vdash^{?}$ is a plain-decorated type T. With the algorithmic rules, these are not requirements but *guarantees* that the specification makes of the algorithm:

Lemma 1. *Let $arr_P(P)$ be the number of prototype arrows prefixing P and $arr_W(W)$ be the number of decorated arrows prefixing W. If $\Gamma; P \Vdash^{?} t : W \rightsquigarrow (p, \sigma)$ then $arr_W(W) \leq arr_P(P)$*

Theorem 6. *(Soundness of $\Vdash^{?}$ wrt $\Vdash^{:=}$): If $\Gamma; P \Vdash^{?} t : W \rightsquigarrow (p, \sigma)$ then $MV(\Gamma, p) \Vdash^{:=} \lfloor W \rfloor := P \Rightarrow (\sigma, W)$*

Assuming prototype inference succeeds, when we specialize P in Theorem 6 to $?$ we have immediately by rule $M?$ that $\sigma = \sigma_{id}$; when we specialize it to some contextual type T' for an application, then by the premise of *MType* we have $\sigma\ T = T'$. Theorem 1 and 6 together tell us that we generate plain-decorated types in both cases, as in particular we cannot have leading (decorated) arrows or stuck decorations with prototypes $?$ or T'.

Next we discuss the rules forming judgment $\Vdash^{?}$ in Figure 4b, constituting the algorithmic version of the rules in Figure 2b. In rule *?Head*, after synthesizing a type T for the application head we match this type against expected prototype $? \to P$ (we are guaranteed the prototype has this shape since a maximal term application begins all derivations of $\Vdash^{?}$). No meta-variables occur in T initially – as we perform prototype matching these will be generated by rule *MForall* from quantified type variables in T and their solutions will be left as quantifier decorations in the resulting decorated type W. We are justified in requiring that matching T to $? \to P$ generates empty solution σ_{id} since we have in general that the meta-variables solved by our prototype matching judgment are a subset of the meta-variables it was asked to solve:

Lemma 2. *If $\overline{X} \Vdash^{:=} T := P \Rightarrow (\sigma, W)$ then $dom(\sigma) \subseteq \overline{X}$*

In *?TApp*, we can infer the type of a type application $t[S]$ when t synthesizes a decorated type $\forall X{=}R.\, W$ and R is either an uninformative decoration X or is precisely S (that is, the programmer provided explicitly the type argument the algorithm contextually inferred). We synthesize $[S/X]W$ for the type application, where we extend type substitution to decorated types by the following recursive partial function:

$$\sigma\ S \to W = (\sigma\ S) \to (\sigma\ W)$$
$$\sigma\ \forall X{=}R.\, W = \forall X{=}R.\, \sigma\ W$$
$$\sigma\ (X, ? \to P) = W \qquad \text{if } \varnothing \Vdash^{:=} \sigma(X) :=? \to P \Rightarrow (\sigma_{id}, W)$$

This definition is straightforward except for the case dealing with stuck decorations. Here, σ (representing type arguments given explicitly or inferred synthetically) may provide information on how to instantiate X and this must match our current (though incomplete) information $? \to P$ about this instantiation. For example, if we have decorated type $W = X \to (X, ? \to \mathbb{N})$, then $[\mathbb{N} \to \mathbb{N}/X]\ W$ would require we match $\mathbb{N} \to \mathbb{N}$ with $? \to \mathbb{N}$ and matching would generate (plain) decorated type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$

The definition of substitution on decorated types is partial since prototype matching may fail (consider if we used substitution $[\mathbb{N}/X]$ in the above example instead). When a decorated type substitution $\sigma\ W$ appears in the conclusion of our algorithmic rules, such as in *?TApp* or *?Syn*, we are implicitly assuming an additional premise that the result is defined.

The last rule for judgment $\Vdash^{?}$ is *?App*, and like *PApp* it benefits from a reading for judgment $\Vdash^{\cdot}$ occurring in its premise. We read $\Gamma \Vdash^{\cdot} (p : W, \sigma) \cdot t' : W' \rightsquigarrow (p', W')$ as: "under Γ, elaborated applicand p of decorated type W together with solution σ can be applied to t'; the application has decorated type W' and elaborates p' with solution σ'." Thus, *?App* says that to synthesize a decorated type for a term application $t\ t'$ we synthesize the decorated type of the applicand t and ensure that the resulting elaboration p, along with its decorated type and solution, can be applied to t'.

We now turn to the rules for the last judgment $\Vdash^{\cdot}$ of our algorithm. Rule *?Forall* clarifies the non-deterministic guessing done by the specificational rule *PForall*: the contextually-inferred type arguments we build during contextual type-argument inference are just the accumulation of quantified type decorations. The solution σ'' we provide to the second premise of *?Forall* contains mapping $[R/X]$ if R is an informative decoration, and as we did in rule *PForall* we provide elaborated term $p[X]$ to track the contextually-inferred type arguments separately from those synthetically inferred.

Rule *?Chk* works similarly to *PChk*: when the only meta-variables in the domain S of our decorated type are solved by σ, we can check that argument t' has type $\sigma\ S$. In rule *?Syn* we have some meta-variables $\overline{Y}$ in S not solved by σ – we synthesize a type for the

argument, ensure that it is some instantiation $\overline{[U/Y]}$ of $\sigma\ S$, and use this instantiation on the meta-variables in p as well as the decorated codomain type W, potentially unlocking some stuck decoration to reveal more arrows or decorated type quantifications.

We conclude this section by noting that the specificational and algorithmic type inference system are equivalent, in the sense that they type precisely the same set of terms:

Theorem 7. *(Soundness of $\Vdash_\delta$ wrt $\vdash_\delta$):*
If $\Gamma \Vdash_\delta t : T \rightsquigarrow e$ *then* $\Gamma \vdash_\delta t : T \rightsquigarrow e$

Theorem 8. *(Completeness of $\Vdash_\delta$ wrt $\vdash_\delta$):*
If $\Gamma \vdash_\delta t : T \rightsquigarrow e$ *then* $\Gamma \Vdash_\delta t : T \rightsquigarrow e$

(where $\Vdash_\delta$ indicates $\vdash^{\mathrm{I}}$ is defined as in Figure 4a), and that the algorithmic rules are decidable:

Theorem 9. *(Decidability of Type-checking)*

- *For any context* Γ *and term* t*, it is decidable whether* $\Gamma \Vdash_{\Uparrow} t : T \rightsquigarrow e$ *for some* T *and* e
- *For any context* Γ*, term* t*, and type* T*, it is decidable whether* $\Gamma \Vdash_{\Downarrow} t : T \rightsquigarrow e$ *for some* e

Taken together, Theorems 7 and 8 justify our claim that the rules of Figure 2 constitute a specification for contextual type-argument inference – it is not necessary that the programmer know the notably more complex details of prototype matching or type decoration to understand how some type arguments are inferred contextually. Indeed, the judgment $\vdash^{\cdot}$ provides more flexibility in reasoning about type inference than does $\Vdash^{\cdot}$, as in rule *PForall* we may freely decline to guess a contextual type argument even when this would be justified and instead try to learn it synthetically. In contrast, algorithmic rule ?*Forall* requires that we use any informative quantifier decoration. We use this flexibility when giving guidelines for the required annotations in Section 3.1 for typing external terms, as the required conditions for typeability in Theorem 3 would be further complicated if we could not restrict ourselves to using only synthetic type-argument inference.

5 DISCUSSION & RELATED WORK

5.1 Local Type Inference and System $\mathrm{F}_{\leq}$

Local Type Inference. Our work is most influenced by the seminal paper by Pierce and Turner[16] that lays out the approach of local type inference systems, including bidirectional typing rules, local type-argument inference, and the design-space restriction that polymorphic function applications be fully-uncurried to maximize the benefit of these techniques. In their system, either all term arguments to polymorphic functions must be synthesized or else all type arguments must be given – no compromise is available when only a *few* type arguments suffice to type an application, be they provided explicitly or inferred contextually. Our primary motivation in this work was addressing these issues – improving support for first-class currying and partial type application, and using the contextual type of an application for type-argument inference – while maintaining some of the desirable properties of local type inference and staying in the spirit of their approach.

Colored Local Type Inference. Odersky, Zenger, and Zenger[13] extend the type system of Pierce and Turner by allowing *partial* type information to be propagated when inferring types for term arguments. Their insight was to internalize the two modes of bidirectional type inference to the very structure of types, allowing different parts to be synthetic or contextual. In contrast, we use an "all or nothing" approach to type propagation, requring a term argument to fully synthesize its type when we have incomplete information. On the other hand, their system uses only the typing information provided by the application head, whereas we combine this with the contextual type of an application, allowing us to type some expressions their system cannot.

The syntax for prototypes in our algorithm was directly inspired by the prototypes used in the algorithmic inference rules for [13]. Our use of prototypes complements theirs; ours propagates the partial type information provided by contextual type of an application spine to its head, whereas theirs propagates the partial type information provided by an application head to its arguments. In future work, we hope to combine these two notions of prototype to propagate *partially* the type information coming from the application's contextual type *and* head to its arguments.

Subtyping. Local type inference is usually studied in the setting of System $\mathrm{F}_{\leq}$ which combines impredicative parametric polymorphism and subtyping. The reason for this is two-fold: first, a partial type inference technique is needed as complete type inference for $\mathrm{F}_{\leq}$ is undecidable[19]; second, global type inference systems fail to infer principal types in $\mathrm{F}_{\leq}$ [10, 12], whereas local type inference is able to promise that it infers the "locally best"[16] type arguments. The setting for our algorithm is System F, so the reader may ask whether our developments can be extended gracefully to handle subtyping. We believe the answer is yes, though with some modification on how synthetic type arguments are used.

In rule *PSyn* in Figure 2c, meta-variables $\overline{Y}$ are instantiated to types $\overline{U}$ immediately. In the presence of subtyping this would make our rules *greedy*[1, 4] and we would not be able to guarantee synthetic type-argument inference produced locally best types, possibly causing type inference to fail later in the application spine. To illustrate this, consider the expression $\texttt{rapp}\ x\ \texttt{neg}$, assuming $\texttt{rapp} : \forall X.\forall Y.X \to (X \to Y) \to Y$, $x : \mathbb{N}$, $\texttt{neg} : \mathbb{Z} \to \mathbb{Z}$, and some subtyping relation $\leq$ where $\mathbb{N} \leq \mathbb{Z}$. Greed causes us to instantiate X with $\mathbb{N}$, but in order to type the expression we would need to instantiate it to $\mathbb{Z}$ instead!

To correct this, we could instead collect these constraints and solve them only when the function is fully applied to its arguments (i.e., when we reach a stuck decoration). This mirrors the requirement in [16] that constraints are solved at fully uncurried applications, maintaining currying but losing a syntactically-obvious location for synthetic type-argument inference.

We would also need to justify our use of contextual type-argument inference for checking the types of term arguments. Happily, this does not appear to be an intractable problem like greed: unlike in synthesis mode, checking mode for applications in [16] does not require that the synthesized type arguments minimize the result type of the application, so there is greater freedom in choosing the instantiations for contextually-inferred type arguments. Hosoya and Pierce note in [7] that the optimal instantiations for these type arguments are ones that "maximize the expected type corresponding to the [argument]," as the type that the programmer meant for

the argument (if type correct) will be a subtype of this. Though the informal approach they proposed (and later dismissed) for inferring the types of hard-to-synthesize terms differs from ours in the use of a "slightly ad-hoc" analysis of arguments, it anticipated contextual type-argument inference and suggests the way forward for using it in the presence of subtyping.

5.2 Bidirectional Type Inference and System F

Predicative Polymorphism. Bidirectional type inference sees use outside of purely local systems. Dunfield and Krishnaswami[5] introduced a simple and elegant type inference system for predicative System F using a dedicated application judgment that instantiates type arguments at term applications. Their application judgment was the direct inspiration for our own, though there are some significant differences between the two. First, our rules distinguish between checking the argument of an application with a fully known expected type and synthesizing its argument when incomplete information is available to keep meta-variables *spine-local*, whereas in their approach meta-variables and typing constraints are passed downwards to check term arguments. Our system also contains the additional judgment form $\vdash^{\mathrm{P}}$ that theirs does not, again to contain meta-variables within an application spine.

Approaches to type inference for System F (impredicative and predicative alike) often make use of some form of subsumption rule to decrease the required type annotations in terms. A popular basis for such rules is the "more polymorphic than" subtyping relation introduced by Odersky and Läufer in [11] which stratifies polymorphic and monomorphic types and is able to perform deeply nested monomorphic type instantiation. This line of work also includes [5] above as well as work by Peyton Jones et. al. [14], both of which are able to infer arbitrary-rank types in the setting of predicative System F. In contrast our type inference algorithm supports more powerful impredicative polymorphism at the cost of significant increase in required type annotations.

Impredicative Polymorphism. The "more-polymorphic-than" subtyping relation for impredicative System F is undecidable[19], so type inference systems wishing to use a subsumption rule in this setting must make some compromises. With $\mathbf{ML}^{\mathbf{F}}$ [8] Le Botlan and Rémy develop a type language with bounded type quantification and an inference system using type *instantiation* (a covariant restriction of subtyping). *Boxy type inference*[21] by Vytiniotis et al. uses an idea similar to [13] of propagating partial type information (though with a very different implementation) to allow inference for polymorphic types only in checking mode; its later development in **FPH**[20] both simplifies the specification for type inference and extends boxy types to synthesis mode to allow "boxy monotypes" to be inferred for polymorphic functions. These inference systems add additional constructs (resp. bounded quantifications and boxy types) to System F types in their specification, whereas we reserve our new constructs (decorated types and prototypes) for the algorithmic rules only. Our use of first-order matching when typing applications of and arguments to polymorphic functions can be viewed as a crude form of subtyping via shallow type instantiation – it is significantly easier for programmers to understand but at the same time significantly less powerful than the subtyping used in the type inference systems above.

ACKNOWLEDGMENTS

We thank Larry Diehl, Anthony Cantor, and Ernesto Copello for their feedback on earlier versions of this paper which helped us clarify some points of terminology and improve the readability of the more technical sections of the paper. We gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

REFERENCES

[1] Luca Cardelli. 1997. An implementation of F<:. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.6158
[2] Iliano Cervesato and Frank Pfenning. 2003. A Linear Spine Calculus. *J. Log. Comput.* 13 (2003), 639–688.
[3] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. ACM, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176
[4] Joshua Dunfield. 2009. Greedy Bidirectional Polymorphism. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML (ML '09)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1596627.1596631
[5] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. *SIGPLAN Not.* 48, 9 (Sept. 2013), 429–442. https://doi.org/10.1145/2544174.2500582
[6] Jean-Yves Girard. 1986. The system F of variable types, fifteen years later. *Theoretical Computer Science* 45 (1986), 159 – 192. https://doi.org/10.1016/0304-3975(86)90044-7
[7] Haruo Hosoya and Benjamin Pierce. 1999. How Good is Local Type Inference? (07 1999).
[8] Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F. *SIGPLAN Not.* 38, 9 (Aug. 2003), 27–38. https://doi.org/10.1145/944746.944709
[9] Bruce McAdam. 2002. Trends in Functional Programming. Intellect Books, Exeter, UK, UK, Chapter How to Repair Type Errors Automatically, 87–98. http://dl.acm.org/citation.cfm?id=644403.644412
[10] Martin Odersky. 2002. Inferred Type Instantiation for GJ. Note sent to the types mailing list.
[11] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729
[12] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theor. Pract. Object Syst.* 5, 1 (Jan. 1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4
[13] Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Inference. *SIGPLAN Not.* 36, 3 (Jan. 2001), 41–53. https://doi.org/10.1145/373243.360207
[14] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2005. Practical type inference for arbitrary-rank types. 17 (January 2005), 1–82. https://www.microsoft.com/en-us/research/publication/practical-type-inference-for-arbitrary-rank-types/
[15] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
[16] Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100
[17] Hubert Plociniczak. 2016. *Decrypting Local Type Inference.* Ph.D. Dissertation. École polytechnique fédérale de Lausanne. http://dx.doi.org/10.5075/epfl-thesis-6741
[18] Aaron Stump. 2017. The calculus of dependent lambda eliminations. *J. Funct. Program.* 27 (2017), e14. https://doi.org/10.1017/S0956796817000053
[19] J. Tiuryn and P. Urzyczyn. 1996. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 74–85. https://doi.org/10.1109/LICS.1996.561306
[20] Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2008. FPH: First-class polymorphism for Haskell, In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. https://www.microsoft.com/en-us/research/publication/fph-first-class-polymorphism-for-haskell/
[21] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy Types: Inference for Higher-rank Types and Impredicativity. *SIGPLAN Not.* 41, 9 (Sept. 2006), 251–262. https://doi.org/10.1145/1160074.1159838
[22] J. B. Wells. 1998. Typability and Type Checking in System F Are Equivalent and Undecidable. *ANNALS OF PURE AND APPLIED LOGIC* 98 (1998), 111–156.
[23] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

Verifiably Lazy

Verified Compilation of Call-by-Need

George Stelle
Los Alamos National Laboratory
Los Alamos, New Mexico, United States
University of New Mexico
Albuquerque, New Mexico, United States
stelleg@lanl.gov

Darko Stefanovic
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico, United States
darko@cs.unm.edu

ABSTRACT

Call-by-need semantics underlies the widely used programming language Haskell. Unfortunately, unlike call-by-value counterparts, there are no verified compilers for call-by-need. In this paper we present the first verified compiler for call-by-need semantics. We use recent work on a simple call-by-need abstract machine as a way of reducing the formalization burden. We discuss some of the difficulties in verifying call-by-need, and show how we overcome them.

ACM Reference Format:
George Stelle and Darko Stefanovic. 2019. Verifiably Lazy: Verified Compilation of Call-by-Need. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Non-strict languages, such as Haskell, rely heavily on call-by-need semantics to ensure efficient execution. Without the memoization of results provided by call-by-need, Haskell would be prohibitively inefficient, often exponentially slower than its call-by-value counterparts. Thanks to careful restriction of side effects, reasoning about correctness in Haskell is easier than most mainstream languages. It is for this reason that we would like to have a compiler that gives formal guarantees about preservation of call-by-need semantics. We wish to ensure that any reasoning we do about our non-strict functional programs is preserved through compilation.

Unfortunately, one of the challenges for formalization of non-strict compilers is that the semantics of call-by-need abstract machines tend to be complex, incorporating complex optimizations into the semantics, requiring preprocessing of terms, and closures of variable sizes [6, 13]. Recently we developed a particularly simple abstract machine for call-by-need, the $\mathscr{CE}$ machine [16]. In addition to being exceedingly simple to implement and reason about, the machine shows performance comparable to state-of-the-art.

Verified compilers provide powerful guarantees about the code they generate and its relation to the corresponding source code [4, 8, 10]. In particular, for higher order functional languages, they ensure that the non-trivial task of compiling lambda calculus and its extensions to machine code is implemented correctly, preserving source semantics. The amortized return on investment for verified compilers is high: any reasoning about any program which is compiled with a verified compiler is provably preserved.

Existing verified compilers have focused on call-by-value semantics [4, 8, 10]. This semantics has the property of being historically easier to implement than call-by-need, and therefore likely easier to reason about formally. In this paper, we build on recent work developing a simple method for implementing call-by-need semantics, which has enabled us to implement, and reason formally about the correctness of, call-by-need. We use the Coq proof assistant [2] to implement and prove the correctness of our compiler. We start with a source language of λ calculus with de Bruijn indices:

$$t ::= t\ t \mid x \mid \lambda\ t$$
$$x \in \mathbb{N}$$

Our source semantics is the big-step operational semantics of the $\mathscr{CE}$ machine, which uses shared environments to share results between instances of a bound variable. To strengthen the result, and relate it to a better-known semantics, we also show that the call-by-name $\mathscr{CE}$ machine implements Curien's call-by-name calculus of closures.

It may surprise the reader to see that we do not start with a better known call-by-need semantics; we address this concern in Section 8. We hope that the proof of compiler correctness, along with the proof that our call-by-name version of the semantics implements Curien's call-by-name semantics, convinces the reader that we have indeed implemented a call-by-need semantics, despite not using a better known definition of call-by-need.

For our target, we define a simple instruction machine, described in Section 5. This simple target allows us to describe the compiler and proofs concisely for the paper, while still allowing flexibility in eventually verifying a compiler down to machine code for some set of real hardware, e.g., x86, ARM, or Power.

Our main results is a proof that whenever the source semantics evaluates to a value, the compiled code evaluates to the same value. While there are stronger definitions of what qualifies as a verified compiler, we argue that this is sufficient in Section 8. This main result, along with the proof that the call-by-name version of our semantics implements Curien's calculus of closures, are the primary contributions of this paper. We are unaware of any existing verified non-strict compilers, much less a verified compiler of call-by-need.

IFL'18, September 2018, Lowell, MA, USA

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

The paper is structured as follows. In Section 2 we give the necessary background. In Section 3 we describe the source syntax and semantics (the big-step $\mathscr{CE}$ semantics) in detail. We also use this section to define a call-by-name version of the semantics, and show that it implements Curien's calculus of closures [5]. In Section 4 we describe the small-step $\mathscr{CE}$ semantics and its relation to the big-step semantics. In Section 5 we describe the instruction machine syntax and semantics. In Section 6 we describe the compilation from machine terms to assembly language. In Section 7 we describe how the evaluation of compiled programs is related to the small-step $\mathscr{CE}$ semantics. We compose this proof with the proof that the small-step semantics implement the big-step semantics to show that the instruction machine implements the big-step semantics. In Section 8 we discuss threats to validity, future work, and related work. We conclude in Section 9. The Coq source code with all the definitions and proofs described in this document is available at https://github.com/stelleg/cem_coq.

2 BACKGROUND

Programming languages fall roughly into two camps: those with *strict* and those with *non-strict* semantics. A strict language is one in which arguments at a call-site are always evaluated, while a non-strict language only evaluates arguments when they are needed. One can further break non-strict into two categories: call-by-name and call-by-need. Call-by-name is essentially evaluation by substitution: an argument term or closure is substituted for every instance of a corresponding variable. This has the downside that it can result in exponential slowdown due to repeated work: every variable dereference must re-evaluate the corresponding argument. Call-by-need is an evaluation strategy devised to address this shortcoming. By sharing the result of argument evaluation between instances of a variable, one avoids duplicated work. Unsurprisingly, call-by-need is the default semantics implemented by compilers for non-strict languages like Haskell [13].

Also perhaps unsurprisingly, call-by-need implementations tend to be more complicated than their strict counterparts. For example, even attempts at simple call-by-need abstract machines such as the Three Instruction Machine [6] require lambda lifting and shared indirections, both of which make formal reasoning more difficult. Our $\mathscr{CE}$ machine avoids these complications by using shared environments to share evaluation results between instances of a variable. We showed in previous work that in addition to being simpler to implement and reason about, performance of this approach may be able compete with the state of the art [16].

With recent improvements in higher order logics, machine verification of algorithms has become a valuable tool in software development. Instead of relying heavily on tests to check the correctness of programs, verification can prove that algorithms implement their specification for *all* inputs. Implementing both the specification and the proof in a machine-checked logic removes the vast majority of bugs found in hand-written proofs, ensuring far higher confidence in correctness than other standard methods. Other approaches, such as fuzz testing, have confirmed that verified programs remove effectively all bugs [18].

This approach applies particularly well to compilers. Often, the specification for a compiler is complete: source level semantics for some languages are exceedingly straightforward to specify, and target architectures have lengthy specifications that are amenable to mechanization. In addition, writing tests for compilers that cover all cases is even more hopeless than most domains, due to the size and complexity of the domain and codomain. The amortized return on investment is also high: all reasoning about programs compiled with a verified compiler is provably preserved.

Due to the complexities discussed above involved in implementing lazy languages, existing work has focused on compiling strict languages [4, 8, 10]. Here we use the simple $\mathscr{CE}$ machine as a base for a verified compiler of a lazy language, using the Coq proof assistant.

As with many areas of research, the devil is in the details. What exactly does it mean to claim a compiler is verified? Essentially, a verified compiler of a functional language is one that preserves computation of values. That is, we have an implication: *if the source semantics computes a value, then the compiled code computes an equivalent value* [4]. The important thing to note is that the implication is only in one direction. If the source semantics never terminates, this class of correctness theorem says nothing about the behavior of the compiled code. This has consequences for Turing-complete source languages. If we are unsure if a source program terminates, and wish to run it to check experimentally if it does, if we run the compiled code and it returns a value, we cannot be certain that it corresponds to a value computed in the source semantics.

While in theory one could solve this by proving the implication the other direction, that is, *if the compiled code computes a value then the source semantics computes an equivalent value*, in practice this is prohibitively difficult. Effectively, the induction rules for the abstract machine make constructing such a proof monumentally tricky.

One approach for getting around this issue is to try and capture the divergent behavior by defining a diverging semantics explicitly [12]. Then we can safely say that *if the source semantics diverges according to our diverging semantics, then the compiled code also diverges*.

For this paper, we choose to take the approach of [4] and define verification as the first implication above, focusing on the case in which the source semantics evaluates to a value. This is still a strong result: any source program that has meaning compiles to an executable with equivalent meaning. In addition, if we ever choose to extend the language with a type system that ensures termination, or some notion of progress, then we can use this proof in combination with our verification proof to prove the other direction.

3 $\mathscr{CE}$ BIG-STEP SEMANTICS

In this section we define our big-step source semantics. A big-step semantics has the advantage of powerful, easy-to-use induction properties. This eases reasoning about many program properties. We shall also define a small-step semantics and prove that it implements the big-step semantics, but by showing that our implementation preserves the big-step semantics, we prove preservation of any inductive reasoning on the structure of evaluation tree.

As discussed in Section 1, our source syntax is the lambda calculus with de Bruijn indices. De Bruijn indices count the number of

intermediate lambdas between the occurrence of the variable and its binding lambda.

$$t ::= t\ t \mid x \mid \lambda\ t$$
$$x \in \mathbb{N}$$

The essence of the $\mathscr{CE}$ semantics (Figure 1) is that we implement a shared environment, and use its structure to share results of computations. This makes possible a simple abstract machine that operates on the lambda calculus directly, which is uncommon among call-by-need abstract machines [6, 7, 9, 13]. This simplifies formalization, as we do not need to prove that intermediate transformations, e.g. lambda lifting, are semantics-preserving. Another advantage to the $\mathscr{CE}$ machine is that it has constant-sized closures, obviating the need to reason about re-allocating the results of computation and adding indirections due to closure size changes from thunk to value [13]. We operate on closures, which combine terms with pointers into the shared environment, which is implemented as a heap. Every heap location contains a cell, which consists of a closure and a pointer to the next environment location, which we will refer to as the environment continuation. Variable dereferences index into this shared environment structure, and if/when a dereferenced location evaluates to a value, the original closure (potentially a thunk or closure not evaluated to WHNF) will be replaced with that value. The binding of a new variable extends the shared environment structure with a new cell. This occurs during application, which evaluates the left hand side to an abstraction, then extends the environment with the argument term closed under the environment pointer of the application. The App rule ensures that two variables bound to the same argument closure will point to the same location in the shared environment. Because they point to the same location by construction of the shared environment, we can update that location with the value computed at the first variable dereference, and then each subsequent dereference will point to this value. The variable rule applies the update by indexing into the shared environment structure and replacing the closure at that location with the resulting value. It is worth noting that while the closures in the heap cells are mutable, the shared environment structure is never mutated. This property is crucial when reasoning about variable dereferences. The $\mu\ (l, i)$ function looks up a variable index in the shared environment structure by following environment continuation pointers, returning the location and cell pointed to by the final step. See the Coq source for a formal treatment. Note that we require that fresh heap locations are greater than zero. This is required for reasoning about compilation to the instruction machine, which we will return to in Section 5. While here we constrain fresh heap locations to not be fresh with respect to the entire heap domain, for a real implementation, this is far too strong a constraint, as it doesn't allow any sort of heap re-use. We return to this issue in Section 8, and discuss how this could be relaxed to either allow reasoning about garbage collection or direct heap reuse.

The fact that our natural semantics is defined on the lambda calculus with de Bruijn indices differs from most existing definitions of call-by-need, such as Ariola's call-by-need [1] or Launchbury's lazy semantics [9]. These semantics are defined on the lambda calculus with named variables. While it should be possible to relate our semantics to these [1], the comparison is certainly made more difficult by this disparity. A more fruitful relation to semantics operating on the lambda calculus with named variables would likely be relating Curien's calculus of closures to call-by-name semantics implemented with substitution. We return to this discussion in Section 8.

Syntax

$$t ::= i \mid \lambda t \mid t\ t \quad \text{(Term)}$$
$$i \in \mathbb{N} \quad \text{(Variable)}$$
$$c ::= t\ [l] \quad \text{(Closure)}$$
$$v ::= \lambda t\ [l] \quad \text{(Value)}$$
$$\mu ::= \varepsilon \mid \mu\ [l \mapsto \rho] \quad \text{(Heap)}$$
$$\rho ::= \bullet \mid c \cdot l \quad \text{(Environment)}$$
$$l, f \in \mathbb{N} \quad \text{(Location)}$$
$$s ::= (c, \mu) \quad \text{(Configuration)}$$

Semantics

$$\frac{\mu\ (l, i) = l' \mapsto c \cdot l'' \quad (c, \mu) \Downarrow (v, \mu')}{(i\ [l], \mu) \Downarrow (v, \mu'\ [l' \mapsto v \cdot l''])} \quad \text{(Id)}$$

$$\frac{(t\ [l], \mu) \Downarrow (\lambda t_2\ [l'], \mu') \quad f \notin \mathrm{dom}\ (\mu') \quad (t_2\ [f], \mu'\ [f \mapsto t_3\ [l] \cdot l']) \Downarrow (v, \mu'')}{(t\ t_3\ [l], \mu) \Downarrow (v, \mu'')} \quad \text{(App)}$$

$$\frac{}{(\lambda t\ [l], \mu) \Downarrow (\lambda t\ [l], \mu)} \quad \text{(Abs)}$$

Figure 1: Big step $\mathscr{CE}$ syntax and semantics (call-by-need)

Finally, given a term t, we define the initial configuration as $(t\ [0], \varepsilon)$. As discussed, the choice of the null pointer for the environment pointer is not completely arbitrary, but chosen across our semantics uniformly to represent failed environment lookup.

3.1 Call-By-Name

In this section we define a call-by-name variant of our big-step semantics and prove that it is an implementation of Curien's call-by-name calculus of closures [5].

[1] Both of these well known existing semantics have known problems that arise during formalization, as discussed in Section 8.

See Figure 3 for the definition of our call-by-name semantics. Note that the only change from our call-by-need semantics is that we do not update the heap location with the result of the dereferenced computation. This is the essence of the difference between call-by-name and call-by-need.

A well known existing call-by-name semantics is Curien's calculus of closures [5]. Refer to Figure 2 for a formalization of this semantics. This semantics defines closures as a term, environment pair, where an environment is a list of closures. Abstractions are in weak head normal form, variables index into the environment, and applications evaluate the left hand side to a value, then extend the environment of the value with the closure of the argument.

We define a heterogeneous equivalence relation between our shared environment and Curien's environment. Effectively, this relation is the proposition that the shared environment structure is a linked list implementation of the environment list in Curien's semantics. This is defined inductively, and we require that every closure reachable in the environment is also equivalent. We say two closures are equivalent if their terms are identical and their environments are equivalent.

Given these definitions, we can prove that our call-by-name semantics implement Curien's call by name semantics:

THEOREM 3.1. *If a closure c in Curien's call-by-name semantics is equivalent to a configuration c', and c steps to v, then there exists a v' that our call-by-name semantics steps to from c' that is equivalent to v.*

PROOF OUTLINE. The proof proceeds by induction on Curien's step relation. The abstraction rule is a trivial base case. The variable lookup rule uses a helper lemma that proves by induction on the variable that if the two environments are equivalent and the variable indexes to a closure, then the μ function will look up an equivalent closure. The application rule uses a helper lemma which proves that a fresh allocation will keep any equivalent environments equivalent, and that the new environment defined by the fresh allocation will be equivalent to the extended environment of Curien's semantics.

By proving that Curien's semantics is implemented by the call-by-name variant of our semantics, we provide further evidence that our call-by-need is a meaningful semantics. While eventually we would like to prove that the call-by-need semantics implements an optimization of the call-by-name, we leave that for future work.

One important note is that nowhere do we require that a term being evaluated is closed under its environment. Indeed, it's possible that a term with free variables can be evaluated by both semantics to a value as long as a free variable is never dereferenced. This theme will recur through the rest of the paper, so it is worth keeping in mind.

4 $\mathscr{CE}$ SMALL-STEP SEMANTICS

In this section we discuss the small-step semantics of the $\mathscr{CE}$ machine, and show that it implements the big-step semantics of Section 3. This is a fairly straightforward transformation implemented by adding a stack. The source language is the same, and we simply add a stack to our configuration (and call it a state). The stack elements are either argument closures or update markers. Update markers are pushed onto the stack when a variable dereferences that location in the heap.

Syntax

$$t ::= i \mid \lambda t \mid t\ t \quad \text{(Term)}$$
$$i \in \mathbb{N} \quad \text{(Variable)}$$
$$c ::= t\ [\rho] \quad \text{(Closure)}$$
$$v ::= \lambda t\ [\rho] \quad \text{(Value)}$$
$$\rho ::= \bullet \mid c \cdot \rho \quad \text{(Environment)}$$

Semantics

$$\frac{t_1\ [\rho] \Downarrow \lambda t_2\ [\rho'] \quad t_2\ [t_3\ [\rho] \cdot \rho'] \Downarrow v}{t_1 t_3\ [\rho] \Downarrow v} \quad \text{(LEval)}$$

$$\frac{c_i \Downarrow v}{i\ [c_0 \cdot c_1 \cdot ... c_i \cdot \rho] \Downarrow v} \quad \text{(LVar)}$$

Figure 2: Curien's call-by-name calculus of closures

Syntax

$$t ::= i \mid \lambda t \mid t\ t \quad \text{(Term)}$$
$$i \in \mathbb{N} \quad \text{(Variable)}$$
$$c ::= t\ [l] \quad \text{(Closure)}$$
$$v ::= \lambda t\ [l] \quad \text{(Value)}$$
$$\mu ::= \varepsilon \mid \mu\ [l \mapsto \rho] \quad \text{(Heap)}$$
$$\rho ::= \bullet \mid c \cdot l \quad \text{(Environment)}$$
$$l, f \in \mathbb{N} \quad \text{(Location)}$$
$$s ::= (c, \mu) \quad \text{(Configuration)}$$

Semantics

$$\frac{\mu\ (l, i) = l' \mapsto c \cdot l'' \quad (c, \mu) \Downarrow (v, \mu')}{(i\ [l], \mu) \Downarrow (v, \mu')} \quad \text{(Id)}$$

$$\frac{(t\ [l], \mu) \Downarrow (\lambda t_2\ [l'], \mu') \quad f \notin \mathrm{dom}\,(\mu') \quad (t_2\ [f], \mu'\ [f \mapsto t_3\ [l] \cdot l']) \Downarrow (v, \mu'')}{(t\ t_3\ [l], \mu) \Downarrow (v, \mu'')} \quad \text{(App)}$$

$$\frac{}{(\lambda t\ [l], \mu) \Downarrow (\lambda t\ [l], \mu)} \quad \text{(Abs)}$$

Figure 3: Big step call-by-name $\mathscr{CE}$ syntax and semantics

When they are popped by an abstraction, the closure at that location is replaced by said abstraction, so that later dereferences by the same variable in the same scope dereference the value, and do not repeat the computation. Argument closures are pushed onto the stack by applications, with the same environment pointer duplicated in the current closure and the argument closure. Argument closures are popped off the stack by abstractions, which allocate a fresh memory location, write the argument closure to it, write the environment continuation as the current environment pointer, then enter the body of the abstraction with the fresh environment pointer. This is the

Syntax

$$
\begin{aligned}
s &::= \langle c, \sigma, \mu \rangle && \text{(State)}\\
t &::= i \mid \lambda t \mid t\ t && \text{(Term)}\\
i &\in \mathbb{N} && \text{(Variable)}\\
c &::= t\ [l] && \text{(Closure)}\\
v &::= \lambda t\ [l] && \text{(Value)}\\
\mu &::= \varepsilon \mid \mu\ [l \mapsto \rho] && \text{(Heap)}\\
\rho &::= \bullet \mid c \cdot l && \text{(Environment)}\\
\sigma &::= \square \mid \sigma\ c \mid \sigma\ u && \text{(Stack)}\\
l, u, f &\in \mathbb{N} && \text{(Location)}
\end{aligned}
$$

Semantics

$$
\begin{aligned}
\langle v, \sigma\ u, \mu \rangle &\to \langle v, \sigma, \mu\ (u \mapsto v \cdot l) \rangle \text{ where } c \cdot l = \mu\ (u) && \text{(Upd)}\\
\langle \lambda t\ [l], \sigma\ c, \mu \rangle &\to \langle t\ [f], \sigma, \mu\ [f \mapsto c \cdot l] \rangle f \notin \mathrm{dom}\ (\mu) && \text{(Lam)}\\
\langle t\ t'\ [l], \sigma, \mu \rangle &\to \langle t\ [l], \sigma\ t'\ [l], \mu \rangle && \text{(App)}\\
\langle i\ [l], \sigma, \mu \rangle &\to \langle c, \sigma\ l'', \mu \rangle \text{ where } l'' \mapsto c \cdot l' = \mu\ (l, i) && \text{(Var1)}
\end{aligned}
$$

Figure 4: Syntax and semantics of the $\mathscr{CE}$ machine

mechanism used for extending the shared environment structure. The semantics is defined formally in Figure 4.

Note that the presentation given here differs slightly from our previous presentation [16], which inlined the lookup into the machine steps. This is to simplify formalization and relation to the big-step semantics, but does not change the semantics of the machine. As a trade-off, it does make the relation to the instruction machine in the later sections slightly more involved, but it is generally a superficial change.

4.1 Relation to Big Step

Here we prove that the small-step semantics implements the big-step semantics of Section 3. This requires first a notion of reflexive transitive closure, which we define in the standard way. We also make use of the fact that the reflexive transitive closure can be defined equivalently to extend from the left or right.

LEMMA 4.1. *If the big-step semantics evaluates from one configuration to another, then the reflexive transitive closure of the small-step semantics evaluates from the same starting configuration with any stack to the same value configuration with that same stack.*

PROOF OUTLINE. The proof proceeds by induction on the big-step relation. We define our induction hypothesis so that it holds for all stacks, which gives us the desired case of the empty stack as a simple specialization. The rule for abstractions is the trivial base case. Var rule applies as the first step, and the induction hypothesis applies to the stack with the update marker on it. To ensure that the Upd rule applies we use the fact that the big-step semantics only evaluates to abstraction configurations, and the fact that the reflexive transitive closure can be rewritten with steps on the right. For the Application rule, we take advantage of the fact that we can append two evaluations together, as well as extend a reflexive transitive closure from the left or the right. As with the Var rule we use the fact that the induction rule is defined for all stacks to ensure we evaluate the left hand side to a value with the argument on the top of the stack. Finally, we extend the environment with the argument closure, and evaluate the result to a value by the second induction hypothesis.

Adding a stack in this fashion is a standard approach to converting between big step and small-step semantics. Still, we appreciate that this approach applies here in a straightforward way.

5 INSTRUCTION MACHINE

Here we describe in full the instruction machine syntax and semantics. We choose a simple stack machine with a Harvard architecture (with separate instruction and heap memory). We use natural numbers for pointers, though it shouldn't be too difficult to replace these with standard-sized machine words, e.g., 64 bits, making the stack and malloc operations partial. Our stack is represented as a list of pointers, though again it should be a relatively straightforward exercise to represent the stack in contiguous memory. With the fixed machine word size, we would need to make push operations partial to represent stacks this way. We define our machine to have only four registers: an instruction pointer, an environment pointer, and two scratch registers. Our instruction set is minimal, consisting only of a conditional jump instruction, pop and push instructions, a move instruction, and a new instruction for allocating new memory. Note that for our program memory, we have pointers to basic blocks, but for simplicity of proofs we choose to not increment the instruction pointer within a basic block. Instead, the instruction pointer is constant within a basic block, only changing between basic blocks. In fact, we represent the program as a list of basic blocks, with pointers indexing into the list. This has the advantage of letting us easily reason about sublists and their relation to terms. As with other design decisions, this also should be fairly unproblematic for formalization to a more realistic hardware design. The full syntax of the machine is given in Figure 5. Note that curly brackets $\{\}$ denote optionality, while stars $*$ denote zero or more elements, represented as a list. Note that we'll use some common list terminology, such as bracket notation for indexing, i.e., $l\ [i]$ accesses the i'th element in l (we don't worry about the partiality in this presentation of this operation; see the Coq implementation for a full treatment). We also use $+\!\!+$ for list concatenation, and :: for consing an element onto the head of a list.

We separate read (ro) and write (wo) operands. Write operands can be registers or memory (defined by a register and a constant offset). Read operands can be any write operand or a constant. For reading, we have a read relation, which takes a read operand and a state and is inhabited when the third argument can be read from that read operand in that state. Similarly, a write relation is inhabited when writing the second argument into the first in a state defined by the third argument results in the state defined by the fourth argument.

The machine semantics should be fairly unsurprising. A State, consists of a register file, program memory, a stack, and a heap. The push instruction takes a read operand and pushes it onto the stack. The pop instruction pops the top of the stack into a write operand. The mov instruction moves a machine word from a read operand to a write operand. The jump instruction is parameterized by an optional pair, which, if present, reads the first element of the pair from a read operand, checks if it is zero, and if so sets the IP to the second

$$
\begin{aligned}
n, l, w &\in \mathbb{N} && \text{(Machine Word)}\\
r &:= ip \mid ep \mid r1 \mid r2 && \text{(Registers)}\\
wo &:= r \mid r\%n && \text{(Write Operands)}\\
ro &:= wo \mid n && \text{(Read Operands)}\\
i &:= push\ ro \mid pop\ wo \mid new\ n\ wo \mid mov\ ro\ wo && \text{(Instructions)}\\
bb &:= i : bb \mid jump\ \{ro, l\}\ ro && \text{(Basic Block)}\\
p &:= bb* && \text{(Program)}\\
s &:= w* && \text{(Stack)}\\
h &:= (l, w)* && \text{(Heap)}\\
S &:= \langle rf, p, s, h \rangle && \text{(State)}
\end{aligned}
$$

Figure 5: Instruction Machine Syntax

element of the pair, which is a constant pointer. If the condition is not zero, then it sets the IP to the instruction pointer contained in the second jump argument. If we pass nothing as the first argument, then it becomes an unconditional set of the IP to the value read from the second argument. Note that the second argument is a read operand, so it can either be a constant or read from a register or memory. This means it can be effectively either a direct or indirect jump, both of which are used in the compilation of lambda terms. The new instruction allocates a contiguous block of new memory and writes the resulting pointer to the fresh memory into a write operand. We take the approach of not choosing a particular allocation strategy. Instead, we follow existing approaches and parameterize our proof on the existence of such functionality [4]. For simplicity, we assume that the allocation function returns completely fresh memory, though it should be possible to modify this assumption to be less restrictive, i.e., let it re-use heap locations that are no longer live. The complete semantics of the machine is given in Figure 6. Note that we separate instruction steps and basic block steps. Recall that a basic block is a sequence of instructions that ends with a jump. The Step BB relation will execute the instructions in the basic block in order, then set the IP in accordance with the jump semantics. The Step relation dereferences a basic block at the current IP, and if executing the basic block results in a new state, then the machine executes to that state.

$$\frac{read\ ro\ \langle rf, ps, h \rangle\ v \quad bb, \langle rf, p, v :: s, h \rangle \rightarrow_{bb} S}{push\ ro : bb, \langle rf, p, s, h \rangle \rightarrow_{bb} S} \quad \text{(Push)}$$

$$\frac{write\ wo\ w \langle rf, p, s, h \rangle S' \quad bb, S' \rightarrow_{bb} S}{pop\ wo : bb, \langle rf, p, w :: s, h \rangle \rightarrow_{bb} S} \quad \text{(Pop)}$$

$$\frac{\forall i < n, f + i \notin \mathrm{dom}\,(h) \quad write\ wo\ f \langle rf, p, s, zeroes\ n\ f +\!\!+ h \rangle S' \quad bb, S' \rightarrow_{bb} S}{new\ n\ wo : bb, \langle rf, p, s, h \rangle \rightarrow_{bb} S} \quad \text{(New)}$$

$$\frac{read\ ro\ s\ v \quad write\ wo\ v\ S\ S' \quad bb, S' \rightarrow_{bb} S''}{mov\ ro\ wo : bb, S \rightarrow_{bb} S''} \quad \text{(Mov)}$$

$$\frac{read\ ro\ S\ 0 \quad write\ ip\ k\ S\ S'}{jump\ (ro, k)\ j, S \rightarrow_{bb} S'} \quad \text{(Jump 0)}$$

$$\frac{l > 0 \quad read\ ro\ S\ l \quad read\ j\ S\ k \quad write\ ip\ k\ S\ S'}{jump\ (ro, k')\ j, S \rightarrow_{bb} S'} \quad \text{(Jump S)}$$

$$\frac{read\ ro\ S\ l \quad write\ ip\ l\ S\ S'}{jump\ ro : S \rightarrow_{bb} S'} \quad \text{(Jump)}$$

$$\frac{read\ ip\ \langle rf, p, s, h \rangle\ k \quad p\,[k] = bb \quad bb, \langle rf, p, s, h \rangle \rightarrow_{bb} S'}{\langle rf, p, s, h \rangle \rightarrow S'} \quad \text{(Enter)}$$

Figure 6: Instruction Machine Semantics

6 COMPILER

In this section we describe the compiler, which compiles lambda terms with de Bruijn indices to programs. The compiler proceeds by recursion on lambda terms, keeping a current index into the program to ensure correct linking without a separate pass. For variables, when we get to zero we push the current environment pointer and a null instruction pointer to denote the update marker to the location of the closure being entered. Then we *mov* the closure at that location into *r1* and *ep*, and *jump* to *r1*, recalling that the *jump* sets the *ip*. For nonzero variables, we replicate traversing the environment pointer i times before loading the closure. For applications, we calculate the program location of the argument basic block, and push that and the current environment pointer onto the stack, effectively pushing an argument closure on top of the stack. We then *jump* to the left hand side of the application, as is standard for push-enter evaluation. For abstractions, we use a conditional jump depending on whether the top of the stack is a null pointer (and therefore an update marker) or a valid instruction pointer (and therefore an argument). If it is an update marker, we update the heap location defined by the update marker with the current value instruction pointer and the current environment pointer. We must point to the first of the three abstractions basic blocks, as this value could later update another heap location as well. In the case that the top of the stack was a valid instruction pointer, we allocate a new chunk of 3 word of memory, and *mov* the argument closure into it, with the current environment pointer as the environment continuation. We then set our current environment pointer to this fresh location. This is the process by which we extend our shared environment structure in the instruction machine. Finally, we perform an unconditional jump to the next basic block, which is the first basic block of the compiled body of the lambda. As this is an unconditional jump to the next basic block, for real machine code this jump can be omitted.

$$
\begin{aligned}
var\ 0 :=\ & push\ ep : \\
& push\ 0 : \\
& mov\ \left(ep\%0\right)\ r1 : \\
& mov\ \left(ep\%1\right)\ ep : \\
& jump\ r1 \\
var\ \left(i+1\right) :=\ & mov\ \left(ep\%2\right)\ ep : \\
& var\ i \\
compile\ i\ k :=\ & \left[var\ i\right] \\
compile\ \left(m\ n\right)\ k :=\ & let\ ms = compile\ m\ \left(k+1\right)\ in \\
& let\ nk = 1 + k + length\ ms\ in \\
& push\ ep : \\
& push\ nk : \\
& jump\ \left(k+1\right) :: \\
& ms +\!\!+ compile\ n\ nk \\
compile\ \left(\lambda b\right)\ k :=\ & pop\ r1 : \\
& jump\ \left(r1, k+1\right)\ \left(k+2\right) :: \\
& pop\ r1 : \\
& mov\ k\ r1\%0 : \\
& mov\ ep\ r1\%1 : \\
& jump\ k :: \\
& new\ 3\ r2 : \\
& mov\ r1\ \left(r2\%0\right) : \\
& pop\ \left(r2\%1\right) : \\
& mov\ ep\ \left(r2\%2\right) : \\
& mov\ r2\ ep : \\
& jump\ \left(k+3\right) :: \\
& compile\ b\ \left(k+3\right)
\end{aligned}
$$

Figure 7: Compiler Definition

Being able to define the full compiler this simply is crucial to this verification project. Other, more sophisticated implementations of call-by-need, such as the STG machine, are much harder to implement and reason about. It is worth noting that despite this simplicity, initial tests suggest that performance is not as horrible as one might suspect, and is often competitive with state of the art [16].

As with the relation discussed in Section 3, note that even when compiling, we do not require that a term is closed to compile it. Indeed, we will happily generate code that if entered, will attempt to dereference the null pointer, leaving the machine stuck. Because we are only concerned with proving that we implement the source semantics in the case that it evaluates to a value, this is not a problem. If we wanted to strengthen our proof further, we would try to show that if the source semantics gets stuck trying to dereference a free variable, the implementation would get stuck in the same way, both failing to dereference a null pointer.

7 COMPILER CORRECTNESS

In this section we define a relation between the state of the small-step semantics and the state of the instruction machine semantics, and show that the instruction machine implements the small-step semantics under that relation.

In general, we implement closures as instruction pointer, environment pointer pairs. For the instruction pointers, we relate them to terms via the compile function defined in Section 6. Essentially, we require that the instruction pointer points to a list of basic blocks that the related term compiles to. For the current closure, we relate the instruction pointer register in the instruction machine to the current term in the small-step source semantics. The environment pointers of each machine are more similar. Given a relation between the heaps of the two machines, we define the relation between two environment pointers as existing in the relation of the heaps, or both being the null pointers. While it should be possible to avoid this special case, during the proof it became apparent that not having the special case made the proof significantly harder. This forces us to add the constraint to all machines that pointers are non-null, which for real hardware shouldn't be an issue.

We use null pointers in two crucial ways. One is to explicitly define the root of the shared environment structure in both the source semantics and the machine semantics. The other use is for instruction pointers. To differentiate between update markers and pointers to basic blocks, we use a null pointer to refer to an update marker, and a non-null pointer as an instruction pointer for an argument closure. Note that in fact, while the null pointers in heaps required us to only allocate non-null fresh locations in the heaps of our semantics, using null pointers to denote update markers requires no change to our program generation, due to the fact that an argument term of an application cannot occur at position 0 in the program.

The relation between the heaps of the small-step source semantics and the instruction machine is the trickiest part of the state relation. Note that for each location in the source semantics heap, we have a cell with a closure and environment continuation pointer. Naturally, the instruction machine represents these as three pointers: two for the closure (the instruction pointer and environment pointer) and one for the environment continuation. The easiest approach turned out to be to use the structure of the heap constructs to define a one-to-three mapping between this single cell and the three machine words. The structure used for each of the heaps is a list of pointer, value bindings. We use the ordering of these bindings in the list to define a one binding to three binding mapping between the source heap and the machine heap. We define a membership relation that defines when an element is in our heap relation, proceeding recursively on the inductive relation structure. This allows us to define a notion of which pairs of each type of closure are in the heap, along with their respective locations. Due to the ordering in which they are allocated in the heap during evaluation, each pair of memory allocations corresponds to an equivalent cell. We use this property as a heap equivalence property that is preserved through evaluation: every binding pair in the heap relation property described above defines equivalent closures and environment continuations. For the relation between our stacks, we define a similar notion. For update markers, we require that every update marker points to related environments (they are two pointers that exist in the heap relation). For argument

closures, we require that the closures are equivalent (the instruction pointer and environment pointer are equivalent to their respective counterparts in the small-step semantics).

In summary, we require that the current closure in the small-step semantics is equivalent to the closure represented by the instruction pointer, environment pointer pair, and that the stacks and the heaps are equivalent. The actual Coq implementation of this relation is too involved to relate directly here. We encourage the reader to read the linked Coq source to fully appreciate it.

Given this relation between heaps, we can state our primary lemma.

LEMMA 7.1. *Given that an instruction machine state i is related to a small-step semantics state s, and that small-step semantics state steps to a new state s', the instruction machine will step in zero or more steps to a related state i'.*

PROOF OUTLINE. Our proof proceeds by case analysis on the step rules for the small-step semantics. We'll focus on the second half of the proof, that i' is related to s'. The proofs that i evaluates to i' follow fairly directly from the compiler definition given in Section 6. For the Var rule, because we need to proceed by induction, we have to define a separate lemma and proceed by induction on a basic block while forgetting the program, as the induction hypothesis is invalid in the presence of the program. We then use the lemma to show that evaluation of a compiled variable implements the evaluation of the variable in the small-step semantics. In particular, we use the null environment as a base case for our induction, as we know the only way lookup could fail is if both environment pointers are null, but that cannot be the case due to the fact that we know that the small-step semantics must have successfully looked up its environment pointer in the heap. Therefore the only option is for both environment pointers to exist in the heap relation, which when combined with the heap equivalence relation in the outer proof gives us the necessary property that the environment continuations are equivalent. Finally, because the last locations reached must have been in the heap relation, we know they are equivalent environment pointers, and therefore the stack relation is preserved when we push the update marker onto the heap. For the App rule, we use the definition of our compiler to prove that the argument term and argument instruction pointer are equivalent and that the left hand side term and instruction pointer are also equivalent. They share an environment pointer which is equivalent by the fact that the application closures are related. This proves that the stack relation is preserved as well as the current closure, while the heap is unchanged. For the Lam rule, we allocate a fresh variable and because of our stack relation we can be sure that the closures that we allocate are equivalent, as well as the environment continuations, as they are taken from the previous current continuation. Because of how we define it, the new allocations are equivalent under our heap relation, and preserve heap equivalence. Finally, the Upd rule trivially preserves the stack and current closure relations, and for proving that the relation is preserved for the heap, we proceed by induction on the heap relation. In addition, we must prove a supporting lemma that all environment relations are preserved by the update.

We now have a proof that the small-step semantics implements the big-step semantics, and a proof that the instruction machine implements the small-step semantics. We can now combine these to get our correct compiler theorem.

THEOREM 7.2. *If a term t placed into the initial configuration for the big-step semantics evaluates to a value configuration v, then the instruction machine starting in the initial state with* `compile 0 t` *as its program will evaluate to a related state v'.*

PROOF OUTLINE. We first require that the relation defined between the small-step semantics state and the instruction machine state holds for the initial configurations. This follows fairly directly from the definition of the initial conditions and the compile function. Second, we have by definition of reflexive transitive closure that Lemma 7.1 implies that if the reflexive transitive closure of the small-step relation evaluates in zero or more steps from a state c to a state v, then a related state of the instruction machine c' will evaluate to a state v' which is related to v. We use these two facts, along with the proof that the small-step implements the big-step for any stack, specialized on the empty stack, to prove our theorem.

It is worth recalling exactly what the relation implies about the two value states. Namely, in addition to the value closures being equivalent, their heaps and environments are equivalent, so that every reachable closure in the environment is equivalent between the two.

8 DISCUSSION

Here we reflect on what we have accomplished, including threats to validity, future work, related work, and general discussion of the results.

One thing we'd like to communicate is the difficulty we had in writing comprehensible proofs. The reader is discouraged from attempting to understand the proofs in any way by reading the Coq tactic source code. While we attempted to keep our definitions and lemmas as clean and comprehensible as possible, we found it extremely difficult to do the same with tactics. Partially this may be a failure on our part to become more familiar with the tactic language of Coq, but we suspect that the imperative nature of tactic proofs prevents composability of tactic meta-programs.

Another lesson was the importance of good induction principles. For example, in Section 8.1, we will discuss the issue of only proving the implication of correctness in one direction. This is effectively a product of the power of the inductive properties of high level semantics, which makes them so much easier to reason about. Indeed, this lesson resonates with the purpose of the paper, which is that we'd like to reason about high level semantics, because they are so much easier to reason about due to their pleasant inductive properties, and have that reasoning preserved through compilation.

8.1 Threats to Validity

There are a few potential threats to validity that we address in this section. The first is the one mentioned in Section 2, that we only show that our compiler is correct in the case of termination of the source semantics. In other words, if the source semantics doesn't terminate, we can say nothing about how the compiled code behaves. This means that we could have a compiled program that terminates when the source semantics does not terminate.

One argument in defense of our verification is that *we generally only care about preservation of semantics for preserving reasoning*

about our programs. In other words, if we have a program that we can't reason about, and therefore may not terminate, we care less about having a proof that semantics are preserved. Of course, this is a claim about most uses of program analysis. There are possible analyses that could say things along the lines of *if* the source program terminates, then we can conclude x. We claim these cases are rare, and therefore the provided proof of correctness can still be applied to most use cases.

Another potential threat to validity is the use of a high level instruction machine language. While we claim that its high level and simplicity should make it possible to show that a set of real ISAs implement this instruction machine, we haven't formally verified this step. We believe that this would make for valuable future work, and hope that the reader agrees that nothing in the design of our high level instruction machine would prevent such work.

As a dual to the issue of a high level instruction machine language, some readers may take issue with calling lambda calculus with de Bruijn indices an "input language". Indeed, we do not advocate writing programs in such a language. Still, the conversion between lambda calculus with named variables and lambda calculus with de Bruijn indices is a well understood topic, and we believe it would distract from the presentation of the verified compiler provided here. Indeed, as noted below, semantics using named variables and substitution can be hard to get right [3, 11], so we stand by our decision to use a semantics based on lambda calculus with de Bruijn indices. One potential approach for future work would be to prove that a call-by-name semantics using substitution is equivalent to Curien's calculus of closures, which when combined with a proof that our call-by-need implements our call-by-name, would prove the compiler implements the semantics of a standard lambda calculus with named variables.

A third threat to the validity of this work is the question of whether we have really proved that we have implemented *call-by-need*. The question naturally arises of what exactly it means to prove an implementation of call-by-need is correct. There are certainly well-established semantics [1, 9], so one option would be to directly prove that the $\mathscr{CE}$ semantics implements one of those existing semantics. Unfortunately, recent work has shown that both of these have small issues that arise when formalized that require fixes. Indeed, we did stray down this path a good ways and discovered one of these issues which has been previously described in the literature [11]. This raises the question of whether or not semantics that aren't obviously correct are a good base for what it means to be a call-by-need semantics. Instead, we have chosen to relate our call-by-name semantics formally to a semantics that is obviously correct, Curien's calculus of closures. Along with the tiny modification required for memoization of results, we hope that we have convinced the reader that it is *extremely likely* that the memoization of results is correct. Of course, further evidence such as examples of correct evaluation would go further to convince the reader, and for that we encourage readers to play with a toy implementation at https://github.com/stelleg/cem_pearl. Finally, a more convincing result would be a proof that the call-by-need semantics implement the call-by-name semantics.

Yet another threat to validity is our approach (or lack of approach) to heap-reuse. For simplicity, we have assumed that our fresh locations are fresh with respect to all existing bindings in the heap. Of course, this is unsatisfactory when compared to real implementations. It would be preferable to have our freshness constraint relaxed to only be fresh with respect to live bindings on the heap. We believe that this modification should be possible, at the cost of increased complexity in the proofs.

8.2 Future Work

In addition to some of the future work discussed as ways of addressing issues in Section 8.1, there are some additional features that we think make for exciting areas of future work.

One such area is reasoning about preservation of operational properties such as time and space requirements. This would enable reasoning about time and space properties at the source level and ensuring that these are preserved through compilation. In addition, there is the possibility of verified optimizations, where one can prove that some optimizations are both *correct*, in that they provably preserve semantics, and *true* optimizations, in that they only improve performance with respect to some performance model. By defining a baseline compiler and proving that it preserved operational properties such as time and space usage, one would have a good platform for which to apply this class of optimizations, resulting in a full compiler that verifiably preserves bounds on time and space consumption. As with correctness, reasoning about operational properties is often likely to be easier in the context of the easy-to-reason-about high level semantics, and having that reasoning provably preserved would be extremely valuable.

Another exciting area of future work is powerful proofs of type preservation through compilation. While there has been existing work on type-preserving compilers, fully verified compilers like this one provide such a strong property that type-safety should fall out directly.

One useful feature of Coq is the ability to extract Coq programs out to other implementations, e.g. Haskell. This raises the possibility of extracting the verified compiler out to a Haskell implementation that could be incorporated into GHC, providing a path towards a verified Haskell compiler.

8.3 Related Work

Chlipala implements a compiler from a STLC to a simple instruction machine in [4]. In many ways it is more sophisticated than our work: it converts to CPS, performs closure conversion, and proves a similar compiler correctness theorem to the one we've proved here. The primary difference is that we've defined a call-by-need compiler, which forces us to reason about updating thunks in the heap, a challenge not faced by call-by-value implementations.

Breitner formalizes Launchbury's natural semantics and proves an optimization is sound with respect to the semantics [3, 9]. By relating his formalization with ours, these projects could be combined to prove a more sophisticated lazy compiler correct: one with non-trivial optimizations applied.

CakeML [8] is a verified compiler for a large subset of the Standard ML language formalized in HOL4 [14]. Like Chlipala's work, this is a call-by-value language, though they prove correctness down to an x86 machine model, and are working with a much larger real-world source language. They also make divergence arguments along

the lines of [12], strengthening their correctness theorem in the presence of nontermination. It's also worth noting that like [10], they are also formalizing a front end to the compiler.

As part of the DeepSpec project, Weirich et al. have been working on formalizing Haskell's core semantics [15, 17]. We believe there is opportunity to use this effort in combination with the DeepSpec project to implement and verify a full-featured Haskell compiler.

9 CONCLUSION

We have presented the first verified compiler of a non-strict lambda calculus. In addition to proving that our call-by-need semantics is preserved through compilation, we have proved that Curien's calculus of closures is implemented by our call-by-name semantics. We argue that this provides compelling evidence that our compiler is a true verified compiler of call-by-need.

We hope that this work can serve as a foundation for future work on real-world verified compilers. While it is clearly a toy compiler, we have reason to believe that performance is acceptable and can be further improved [16]. In combination with efforts to formalize semantics of real-world languages like Haskell, we hope that this work can help us move towards fully verified non-strict programs.

REFERENCES

[1] Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246. ACM, 1995.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[3] J. Breitner. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation*. PhD thesis, Karlsruher Instituts für Technologie, 2017.

[4] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 42, pages 54–65. ACM, 2007.

[5] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, 1991.

[6] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architecture*, pages 34–45. Springer, 1987.

[7] T. Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN Notices*, 1984.

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535841. URL http://doi.acm.org/10.1145/2535838.2535841.

[9] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM, 1993.

[10] X. Leroy. The CompCert C verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[11] K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.

[12] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *Programming Languages and Systems*, pages 589–615, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49498-1.

[13] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of functional programming*, 2(2):127–202, 1992.

[14] K. Slind and M. Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[15] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 14–27, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5586-5. doi: 10.1145/3167092. URL http://doi.acm.org/10.1145/3167092.

[16] G. Stelle, D. Stefanovic, S. L. Olivier, and S. Forrest. Cactus environment machine: Shared environment call-by-need. In *Proceedings of the 17th ACM symposium on Trends in Functional Programming*, page to appear. ACM, 2017.

[17] S. Weirich, A. Voizard, P. H. A. de Amorim, and R. A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, Aug. 2017. ISSN 2475-1421. doi: 10.1145/3110275. URL http://doi.acm.org/10.1145/3110275.

[18] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993532. URL http://doi.acm.org/10.1145/1993498.1993532.

ESVERIFY: Verifying Dynamically-Typed Higher-Order Functional Programs by SMT Solving

Christopher Schuster
University of California, Santa Cruz
cschuste@ucsc.edu

Sohum Banerjea
University of California, Santa Cruz
sobanerj@ucsc.edu

Cormac Flanagan
University of California, Santa Cruz
cormac@ucsc.edu

ABSTRACT

Program verifiers statically check correctness properties of programs with annotations such as assertions and pre- and postconditions. Recent advances in SMT solving make it applicable to a wide range of domains, including program verification. In this paper, we describe ESVERIFY, a program verifier for JavaScript based on SMT solving, supporting functional correctness properties comparable to languages with refinement and dependent function types. ESVERIFY supports both higher-order functions and dynamically-typed idioms, enabling verification of programs that static type systems usually do not support. To verify these programs, we represent functions as universal quantifiers in the SMT logic and function calls as instantiations of these quantifiers. To ensure that the verification process is decidable and predictable, we describe a bounded quantifier instantiation algorithm that prevents matching loops and avoids ad-hoc instantiation heuristics. We also present a formalism and soundness proof of this verification system in the Lean theorem prover and a prototype implementation.

CCS CONCEPTS

• **Theory of computation → Pre- and post-conditions**; *Logic and verification*; • **Software and its engineering → Language types**; Functional languages;

KEYWORDS

program verification, functional programming, SMT solving

ACM Reference Format:
Christopher Schuster, Sohum Banerjea, and Cormac Flanagan. 2018. ESVERIFY: Verifying Dynamically-Typed Higher-Order Functional Programs by SMT Solving. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310240

1 INTRODUCTION

The goal of program verification is to statically check programs for properties such as robustness, security and functional correctness across all possible inputs. For example, a program verifier might statically verify that the result of a sorting routine is sorted and is a permutation of the input.

In this paper, we present ESVERIFY, a program verification system for JavaScript. JavaScript, a dynamically-typed scripting language, was chosen as target because its broad user base suggests many beneficial use cases for static analysis, and because its availability in browsers enables accessible online demos without local installation.

JavaScript programs often include idioms and patterns that do not adhere to standard typing rules. For instance, the latest edition of the JavaScript/ECMAScript standard [ECMA-262 2017] introduces *promises* such that a promise can be composed with other promises and with arbitrary objects, as long as these objects have a `"then"` method. Since ESVERIFY does not rely on static types, it can easily accommodate these idioms.

JavaScript programs also often use higher-order functions. In order to support verification of these functions, ESVERIFY introduces a new syntax to constrain function values in terms of their pre- and postconditions. Similarly, other JavaScript values such as numbers, strings, arrays and classes can be used in assertions, including invariants on array contents and class instances.

The implementation of ESVERIFY, including its source code[1] and a live demo[2] are available online. In summary, if a JavaScript program uses features unsupported by ESVERIFY, it will be rejected early; otherwise, verification conditions are generated based on annotations, and each verification condition is transformed according to a quantifier instantiation algorithm and then checked by an SMT solver.

In addition to describing the design and implementation or ESVERIFY, we formally define a JavaScript-inspired, statically verified but dynamically typed language, called λ^S. Functions in λ^S are annotated with pre- and postconditions, rendered as logical propositions. These propositions can include operators, refer to variables in scope, denote function results with uninterpreted function calls, and constrain the pre- and postconditions of function values. The verification rules for λ^S involve checking verification conditions for validity. This checking is performed by an SMT solver augmented with decidable theories for linear integer arithmetic, equality, data types and uninterpreted functions. The key difficulty is that verification conditions can include quantifiers, as function definitions in the source program correspond to universally quantified formulas in verification conditions. Unfortunately, SMT solvers may not always perform the right instantiations, and therefore quantifiers imperil the decidability of the verification process [Ge and de Moura 2009; Reynolds et al. 2013]. We ensure that the verification process remains decidable and predictable, by proposing

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310240

[1] Implementation Source Code: https://github.com/levjj/esverify/
[2] Online live demo of ESVERIFY: https://esverify.org/try

a bounded quantifier instantiation algorithm such that function calls in the source program act as hints ("triggers") that instantiate these quantifiers. The algorithm only performs a bounded number of trigger-based instantiations and thereby avoids brittle instantiation heuristics and matching loops. Using this decision procedure for verification conditions, we show that verification of λ^S is sound, i.e. verifiable λ^S programs do not get stuck. The proof is formalized in the Lean Theorem Prover and available online[3].

To evaluate the expressiveness of this approach, we also include a brief comparison with static refinement types [Vazou et al. 2014]. While a formalization and proof is beyond the scope of this paper, refined base type and dependent function types can be translated to assertions such that the resulting program is verifiable if the original program is well-typed. This suggests that esverify is at least as expressive as a language with refinement types.

To summarize, the main contributions of this paper are

(1) an approach for verifying dynamically-typed, higher-order JavaScript programs,
(2) a bounded quantifier instantiation algorithm that enables trigger-based instantiations without heuristics or matching loops,
(3) a prototype implementation called esverify, and
(4) a formalization of the verification rules and a proof of soundness in the Lean theorem prover.

The structure of the rest of the paper is as follows: Section 2 illustrates common use cases and relevant features of esverify, Section 3 outlines the verification process and the design of the implementation, Section 4 formally defines quantifier instantiation, as well as the syntax, semantics, verification rules and a soundness theorem for a core language λ^S, Section 5 compares the program verification approach to refinement type systems, Section 6 discusses related work, and finally Section 7 concludes the paper.

2 VERIFYING JAVASCRIPT PROGRAMS

Our program verifier, esverify, targets a subset of ECMAScript/-JavaScript. By supporting a dynamically-typed scripting language, esverify is unlike existing verifiers for statically-typed programming languages. We do not aim to support complex and advanced JavaScript features such as prototypical inheritance and metaprogramming, leaving these extensions for future work. Instead, the goal is to support both functional as well as object-oriented programming paradigms with an emphasis on functional JavaScript programs with higher-order functions.

2.1 Annotating JavaScript with Assertions

esverify extends JavaScript with source code annotations such as functions pre- and postconditions, loop invariants and statically-checked assertions. These are written as *pseudo function calls* with standard Javascript syntax. While some program verification systems specify these in comments such as ESC/Java [Flanagan et al. 2002], this approach enables a better integration with existing tooling support such as refactoring tools and syntax highlighters.

[3]Formal definitions and proofs in Lean: https://github.com/levjj/esverify-theory/

```
1  function max(a, b) {
2    requires(typeof(a) === 'number');
3    requires(typeof(b) === 'number');
4    ensures(res => res >= a);
5    ensures(res => res >= b); // does not hold
6    if (a >= b) {
7      return a;
8    } else {
9      return a; // bug
10   }
11 }
```

Listing 1: A JavaScript function `max` annotated with pre- and postconditions.

```
1  function sumTo (n) {
2    requires(Number.isInteger(n) && n >= 0);
3    ensures(res => res === (n + 1) * n / 2);
4    let i = 0;
5    let s = 0;
6    while (i < n) {
7      invariant(Number.isInteger(i) && i <= n);
8      invariant(Number.isInteger(s));
9      invariant(s === (i + 1) * i / 2);
10     i++;
11     s = s + i;
12   }
13   return s;
14 }
```

Listing 2: A JavaScript function that shows $\sum_{i=0}^{n} i = \frac{(n+1)\cdot n}{2}$. Loop invariants are not inferred and need to be specified explicitly for all mutable variables in scope.

The assertion language is a subset of JavaScript. It does not support all of JavaScript's semantics. In particular, it is restricted to pure expressions that do not contain function definitions.

2.2 `max`: A Simple Example

Listing 1 shows an example of an annotated JavaScript program. The calls to `requires` and `ensures` in lines 2–5 are only used for verification purposes and excluded from evaluation. Instead of introducing custom type annotations, the standard JavaScript `typeof` operator is used to constrain the possible values passed as function arguments. Due to a bug in line 9, the `max` function does not return the maximum of the arguments if `b` is greater than `a`, violating the postcondition in line 5.

2.3 Explicit Loop Invariants

For programs without loops or recursion, static analysis can check various correctness properties precisely. However, the potential behavior of programs with loops or recursion cannot be determined statically. esverify "overapproximated" the behavior of the program, i.e. correct programs may be rejected if the program lacks a sufficiently strong loop invariant or pre- or postcondition, but verified programs are guaranteed to not violate an assertion regardless of the number of iterations or recursive function calls.

```
1  function inc (x) {
2    requires(Number.isInteger(x));
3    ensures(y => Number.isInteger(y) && y > x);
4    // implicit: ensures(y => y === x + 1);
5    return x + 1;
6  }
7  function twice (f, n) {
8    requires(spec(f, (x)   => Number.isInteger(x),
9                     (x,y) => Number.isInteger(y) &&
10                              y > x));
11   requires(Number.isInteger(n));
12   ensures(res => res >= n + 2);
13   return f(f(n));
14 }
15 const n = 3;
16 const m = twice(inc, n); // 'inc' satisfies spec
17 assert(m > 4);           // statically verified
```

Listing 3: The higher-order function `twice` restricts its function argument `f` with a maximum precondition and a minimum postcondition. The function `inc` has its body as implicit postcondition and therefore satisfies this `spec`.

Listing 2 shows a JavaScript function that computes the sum of the first `n` natural numbers with a **`while`** loop. The loop requires annotated invariants for mutable variables including their types and bounds[4]. Without these loop invariants, the state of `i` and `s` would be unknown in line 13 except for the fact that `i < n` is false. However, when combined with the loop invariants, the equality $i == n$ can be inferred after the loop and thereby the postcondition in line 3 can be verified. ESVERIFY internally uses standard SMT theorems for integer arithmetic to establish that the invariants are maintained for each iteration of the loop.

There is extensive prior work on automatically inferring loop invariants [Furia and Meyer 2010]. Recent research suggest that automatic inference can also be extended to program invariants [Ernst et al. 2001] and specifications [Henkel and Diwan 2003]. However, this topic is orthogonal to the program verification approach presented in this paper.

2.4 Higher-order Functions

In order to support function values as arguments and results, ESVERIFY introduces a **`spec`** construct in pre-, postconditions and assertions. Listing 3 illustrates this syntax in lines 8–10 of the higher-order `twice` function. The argument `f` needs to be a function that satisfies the given constraints, and therefore the call `twice(inc,n)` in line 17 requires ESVERIFY to compare the pre- and postconditions of `inc` with pre- and postconditions in lines 8–10. It is important to note that ESVERIFY implicitly strengthens the stated postcondition of `inc` by inlining its function body `x + 1`. Recursive functions are only inlined by one level, so these need to be explicitly annotated with adequate pre- and postconditions for verification purposes, similarly to loop invariants.

[4]Here, `Number.isInteger(i)` ensures that `i` is an actual integer, while **`typeof`**`(i) === 'number'` is also true for floating point numbers.

```
1  function f (a) {
2    requires(a instanceof Array);
3    requires(a.every(e => e > 3));
4    requires(a.length >= 2);
5    assert(a[0] > 2); // holds
6    assert(a[1] > 4); // fails as a[1] might be 4
7    assert(a[2] > 1); // a may have only 2 elements
8  }
```

Listing 4: ESVERIFY includes basic support for immutable arrays. The elements of an array can be described with `every`.

2.5 Arrays and Objects

In addition to floating point numbers and integers, ESVERIFY also supports other standard JavaScript values such as boolean values, strings, functions, arrays and objects. However, ESVERIFY restricts how objects and arrays can be used. Specifically, mutation of arrays and objects is not currently supported and objects have to be either immutable *dictionaries* that map string keys to values or instances of user-defined classes with a fixed set of fields without inheritance.

The elements of an array can be described with a quantified proposition, corresponding to the standard array method **`every`**. This is illustrated in Listing 4.

Despite these restrictions, it is possible to express complex recursive data structures. For example, Listing 5 shows a user-defined linked list class that is parameterized by a predicate. Here, the `each` field is actually a function that returns **`true`** for each element in the linked list. To simplify reasoning, the pseudo call **`pure`**`()` in the postcondition ensures the absence of side effects. Mapping over the elements of the list with a function `f` requires that `f`' can be invoked with elements that satisfy **`this`**`.each` and that return values of `f` satisfy the new predicate `newEach`. This demonstrates how generic data structures can be used to verify correctness in a similar way to parameterized types. It is important to note that function calls in an assertion context are uninterpreted, so the call `newEach(y)` in line 19 only refers to the function return value but does not actually invoke the function.

2.6 Dynamic Programming Idioms

JavaScript programs often include functions that have polymorphic calling conventions. A common example is the jQuery library which provides a function "`$`" whose behavior varies greatly depending on the arguments: given a function argument, the function is scheduled for deferred execution, while other argument types select and return portions of the current webpage.

Even standard JavaScript objects use dynamic programming idioms to provide a more convenient programming interface. For example, the latest edition of the ECMAScript standard [ECMA-262 2017] includes *Promises* [Liskov and Shrira 1988] and specifies a polymorphic `Promise.resolve()` function. This function behaves differently depending on whether it is called with a promise, an arbitrary non-promise object with a method called `"then"`, or a non-promise object without such a method. ESVERIFY can accurately express these kinds of specifications in pre- and postconditions as

```
class List {
  constructor (head, tail, each) {
    this.head = head; this.tail = tail; this.each = each;
  }
  invariant () {
    // this.each is a predicate that is true for this element and the rest of the list
    return spec(this.each, x => true, (x, y) => pure() && typeof(y) === 'boolean') &&
          (true && this.each)(this.head) && // same as 'this.each(this.head)' but without binding 'this'
          (this.tail === null || (this.tail instanceof List && this.each === this.tail.each));
  }
}
function map (f, lst, newEach) {
  // newEach needs to be a predicate
  // (a pure function without precondition that returns a boolean)
  requires(spec(newEach, x => true, (x, y) => pure() && typeof(y) === 'boolean'));
  // the current predicate 'this.each' must satisfy the precondition of 'f'
  // and the return value of 'f' needs to satisfy the new predicate 'newEach'
  requires(lst === null || spec(f, x => (true && lst.each)(x), (x, y) => pure() && newEach(y)));
  requires(lst === null || lst instanceof List);
  ensures(res => res === null || (res instanceof List && res.each === newEach));
  ensures(pure()); // necessary as recursive calls could otherwise invalidate the class invariant
  return lst === null ? null : new List(f(lst.head), map(f, lst.tail, newEach), newEach);
}
```

Listing 5: Custom linked list class with a field `each` which is a function that is true for all elements. Mapping over the list results in a new list whose elements satisfy a new predicate analogous to a map function in a parametrized type system.

```
class Promise {
  constructor (value) { this.value = value; }
}
function resolve (fulfill) {
  // "fulfill" is promise, then-able or
  // a value without a "then" property
  requires(fulfill instanceof Promise ||
           spec(fulfill.then, () => true,
                              () => true) ||
           !('then' in fulfill));
  ensures(res => res instanceof Promise);
  if (fulfill instanceof Promise) {
    return fulfill;
  } else if ('then' in fulfill) {
    return new Promise(fulfill.then());
  } else {
    return new Promise(fulfill);
  }
}
```

Listing 6: The standard `Promise.resolve()` function in JavaScript has complex polymorphic behavior. This simplified mock definition illustrates how ESVERIFY enables such dynamic programming idioms.

shown in Listing 6, while standard type systems need to resort to code changes, such as sum types and injections.

2.7 Complex Programs: MergeSort

We also demonstrate non-trivial programs such as MergeSort and verify their functional correctness[5]. The implementation is purely functional and uses a linked list data type that is defined as a class. Interestingly, about 48 out of a total 99 lines are verification annotations, including invariants, pre- and postconditions and the predicate function `isSorted`. `isSorted` is primarily used in specifications, but the implementations of `merge` and `sort` also include calls to it. These calls are used as *triggers*, hints to the underlying SMT solver that do not contribute to the result. In other verified languages such as Dafny [Leino 2013], `isSorted` would correspond to a "ghost function", but ESVERIFY does not currently differentiate between verification-only and regular implementation functions.

2.8 JavaScript as Theorem Prover

A simple induction proof over natural numbers can be written as a while loop as previously shown in Listing 2. This idea can be generalized by using the `spec` construct to reify propositions.

In particular, the postcondition of a function need not only describe its return value; it can also state a proposition such that a value that satisfies the function specification acts as proof of this proposition – analogous to the Curry-Howard isomorphism. Such a "function" can then be supplied as argument to higher-order functions to build up longer proofs. For an example, Listing 7 includes a proof written in JavaScript showing that any locally increasing integer-ranged function is globally increasing. This example was previously used to illustrate refinement reflection in LiquidHaskell [Vazou et al. 2018].

[5]The source code of a MergeSort algorithm in ESVERIFY is available at https://esverify.org/try#msort.

```
function proof_f_mono (f, proof_f_inc, n, m) {
  // f is a function from non-negative int to int
  requires(spec(f,
      (x) => Number.isInteger(x) && x >= 0,
      (x, y) => Number.isInteger(y) && pure()));
  // proof_f_inc states that f is increasing
  requires(spec(proof_f_inc,
      x => Number.isInteger(x) && x >= 0,
      x => f(x) <= f(x + 1) && pure()));
  requires(Number.isInteger(n) && n >= 0);
  requires(Number.isInteger(m) && m >= 0);
  requires(n < m);
  // show that f is increasing for arbitrary n,m
  ensures(f(n) <= f(m));
  ensures(pure()); // no side effects
  proof_f_inc(n); // instantiate proof for n
  if (n + 1 < m) {
    // invoke induction hypothesis (I.H.)
    proof_f_mono(f, proof_f_inc, n + 1, m);
  }
}
function fib (n) {
  requires(Number.isInteger(n) && n >= 0);
  ensures(res => Number.isInteger(res));
  ensures(pure());
  if (n <= 1) {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
// A proof that fib is increasing
function proof_fib_inc (n) {
  requires(Number.isInteger(n) && n >= 0);
  ensures(fib(n) <= fib(n + 1));
  ensures(pure());
  fib(n); // unfolds fib at n
  fib(n + 1);
  if (n > 0) {
    fib(n - 1);
    proof_fib_inc(n - 1); // I.H.
  }
  if (n > 1) {
    fib(n - 2);
    proof_fib_inc(n - 2); // I.H.
  }
}
function proof_fib_mono (n, m) {
  requires(Number.isInteger(n) && n >= 0);
  requires(Number.isInteger(m) && m >= 0);
  requires(n < m);
  ensures(fib(n) <= fib(m));
  ensures(pure());
  proof_f_mono(fib, proof_fib_inc, n, m);
}
```

Listing 7: A proof about monotonous integer functions in JavaScript and an instantiation for `fib`. This example was previously used to illustrate refinement reflection in the statically-typed LiquidHaskell system [Vazou et al. 2018].

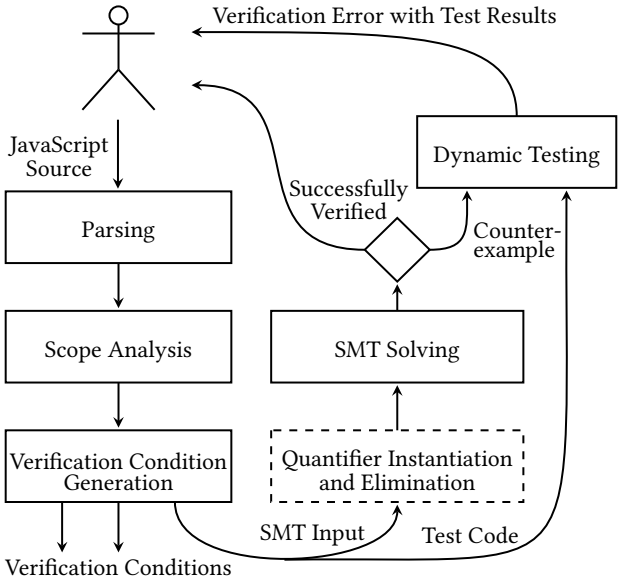


Figure 1: The basic verification workflow: ESVERIFY generates and statically checks verification conditions by SMT solving.

3 IMPLEMENTATION

The ESVERIFY prototype implementation[6] is available online. Because the implementation itself is written in TypeScript, a dialect of JavaScript, it can be used in a browser. Indeed, there is a browser-based editor with ESVERIFY checking[7]. Alternative integrations such as extensions for Vim and Emacs also exist.

The basic verification process and overall design of ESVERIFY is depicted in Figure 1.

The first step of the process involves **parsing** the source code and restricting the input language to a subset of JavaScript supported by ESVERIFY. Some of these restrictions may be lifted in future versions of ESVERIFY, such as support for regular expressions or functions with a variable number of arguments. However, other JavaScript features would involve immense complexity for accurate verification due to their dynamic character and their interactions with the rest of the program, such as metaprogramming with `eval` or `new Function()`. Additionally, ESVERIFY does not support features that have been deprecated in newer versions of strict mode JavaScript such as `arguments.callee`, `this` outside of functions or the `with` statement. The parser also differentiates between expressions and assertions. For example, `spec` can only used in assertions while function definitions can only appear in the actual program implementation.

During the second step, **scope analysis** determines variable scopes and rejects programs with scoping errors and references to unsupported global objects. In addition to user-provided definitions, it includes a whitelist of globals supported by ESVERIFY, such as `Array`, `Math` and `console`. The analysis also takes mutability into account. For example, mutable variables cannot be referenced in

[6] Implementation Source Code: https://github.com/levjj/esverify/
[7] Online live demo of ESVERIFY: https://esverify.org/try

$$\begin{array}{c} \forall a \forall b. \\ \texttt{typeof}(a) = \text{“}number\text{”} \\ \wedge\ \texttt{typeof}(b) = \text{“}number\text{”} \\ \wedge\ (a > b) \implies result = a \\ \wedge\ \neg(a > b) \implies result = a \\ \implies result \geq a \end{array}$$

Figure 2: A simplified verification condition for the postcondition of the `max` function in line 4 of Listing 1.

$$\begin{array}{ll} (\forall a, b. \texttt{max}(a, b) \geq a) & (\forall a, b. \texttt{max}(a, b) \geq a) \\ \implies \texttt{max}(3, 5) > 0 & \wedge\ \texttt{max}(3, 5) \geq 3 \\ & \implies \texttt{max}(3, 5) > 0 \end{array}$$

Figure 3: The proposition on the left has a universal quantifier. On the right, this quantifier is instantiated with concrete values of a and b, yielding an augmented proposition that can be verified with simple arithmetic.

class invariants, and the `old(x)` syntax in a postcondition requires `x` to be a mutable variable.

The main **verification** step is implemented as a traversal of the source program that generates verification conditions and maintains a *verification context*. Most notably, the verification context includes a logical proposition that acts as precondition and a set of variables with unknown values. Generated verification conditions combine this context with an assertion, such as a function postcondition. Figure 2 illustrates this process for a simple example. The verification condition checks whether the preconditions and the translated function body imply the postcondition. Section 4.4 describes the verification rules in more detail.

The verification condition is then transformed with a **quantifier instantiation** procedure. As illustrated by Figure 3, quantified propositions in verification conditions need to be instantiated with concrete values in order to determine satisfiability of the formula. Quantifiers are instantiated based on matching triggers and remaining quantifiers are then erased from the proposition[8]. The resulting quantifier-free proposition can be checked by SMT solving, ensuring that the verification process remains predicable. However, this approach to quantifier instantiation requires the programmer to provide explicit triggers as function calls. Alternatively, the trigger-based quantifier instantiation can be skipped and the proposition passed directly to the SMT solver, which internally performs instantiations based on heuristics.

The final step of the verification process involves checking the verification condition with an **SMT solver** such as z3 [de Moura and Bjørner 2008] or CVC4 [Barrett and Berezin 2004; Barrett et al. 2011]. If the solver cannot find a solution for the negated verification condition, i.e. if the solver cannot refute the proposition, verification succeeded. Otherwise, the returned model includes an assignment of free variables that acts as a counterexample.

Given a counterexample and a synthesized unit test with holes, the verification condition can be **dynamically tested**. This involves dynamic checking of assertions and serves two purposes. On the one hand, the test might not be able to reproduce an error or assertion violation. This indicates that the static analysis did not accurately model the actual program behavior due to a loop invariant or assertion not being sufficiently strong. In this case, the programmer can use the variable assignment in the counterexample to better understand the shortcomings of the analysis and improve those annotations. On the other hand, the test might lead to an error or assertion violation. In this case, the programmer is presented with both a verification error message as well as a concrete witness that assists in the debugging process similar to existing test generators [Tillmann and de Halleux 2008].

[8]The formal definition of the quantifier instantiation algorithm is given in Section 4.2.

$$\begin{array}{lll} \phi \in \text{Propositions} & ::= & \tau \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid pre_1(\otimes, \tau) \mid \\ & & pre_2(\oplus, \tau, \tau) \mid pre(\tau, \tau) \mid post(\tau, \tau) \mid \forall x.\phi \\ \tau \in \text{Terms} & ::= & v \mid x \mid \otimes\, \tau \mid \tau \oplus \tau \mid \tau(\tau) \\ \otimes \in \text{UnaryOperators} & ::= & \neg \mid isInt \mid isBool \mid isFunc \\ \oplus \in \text{BinaryOperators} & ::= & + \mid - \mid \times \mid / \mid \wedge \mid \vee \mid = \mid < \\ v \in \text{Values} & ::= & \text{true} \mid \text{false} \mid n \mid \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma \rangle \\ \sigma \in \text{Environments} & ::= & \emptyset \mid \sigma[x \mapsto v] \\ n \in \mathbb{N} & & f, x, y, z \in \text{Variables} \end{array}$$

Figure 4: Syntax of logical propositions used in the verifier.

4 FORMALISM

In order to reason about ESVERIFY, this section introduces a formal development of λ^S, a JavaScript-inspired, statically verified but dynamically typed language, and shows that its verification is sound.

The verification rules of λ^S use verification conditions whose validity is checked with a custom decision procedure, so this section first formally defines logical propositions and axiomatizes their validity and then describes the decision procedure including quantifier instantiation. Finally, the syntax and semantics of λ^S are defined and its verification rules are shown to be sound.

The definitions, axioms and theorems in this section are also formalized in the Lean theorem prover and available online[9].

4.1 Logical Foundation

Figure 4 formally defines the syntax of propositions, terms, values and environments. Propositions ϕ can use terms, connectives $\neg$, $\wedge$ and $\vee$, symbols $pre_1, pre_2, pre, post$ and universal quantifiers. Here, terms τ are either values, variables, unary or binary operations or uninterpreted function calls. Finally, values v include boolean and integer constants as well as *closures* which are opaque values that will be explained in Section 4.3.

Instead of defining validity of propositions in terms of an algorithm such as SMT solving, this formal development uses an axiomatization of the validity judgement $\vdash \phi$.

Standard axioms for logical connectives and quantifiers are omitted here for brevity but can be found Lean proof. Most noteworthy, axioms about unary and binary operators that are specified in terms of a partial function δ, e.g. $\delta(+, 2, 3) = 5$.

Axiom 1. Iff $\delta(\otimes, v_x) = v$ then $\vdash v = \otimes v_x$. $\boxed{\vdash \phi}$

[9]Formal definitions and proofs in Lean: https://github.com/levjj/esverify-theory/

Axiom 2. Iff $\delta(\oplus, v_x, v_y) = v$ then $\vdash v = v_x \oplus v_y$.

The propositions $pre_1(\otimes, v_x)$ and $pre_2(\otimes, v_x, v_y)$ can be used to reason about the domain of operators.

Axiom 3. If $\vdash pre_1(\otimes, v_x)$ then $(\otimes, v_x) \in dom(\delta)$.

Axiom 4. If $\vdash pre_2(\oplus, v_x, v_y)$ then $(\oplus, v_x, v_y) \in dom(\delta)$.

Similarly, the constructs $pre(f, x)$ and $post(f, x)$ in propositions denote the pre- and postcondition of a function f when applied to a given argument x. However, in contrast to pre_1 and pre_2, the logical foundations do not contain axioms for pre and $post$. The interpretation of pre and $post$ is instead determined by their use in the generated verification condition.

A valid proposition is not necessarily closed. In fact, free variables occurring in a proposition are assumed to be implicitly universally quantified.

Axiom 5. If x is free in ϕ and $\vdash \phi$ then $\vdash \forall x.\ \phi$.

It is important to note that the validity judgement may not be decidable for all propositions due to the use of quantifiers, so in addition to this (undecidable) validity judgement, we also introduce a notion of satisfiability by an SMT solver.

Definition 1 (Satisfiability)**.** $Sat(\phi)$ denotes that the SMT solver found a model that satisfies ϕ.

$$\boxed{Sat(\phi)}$$

Theorem 1. If ϕ is quantifier-free, then $Sat(\phi)$ terminates and $Sat(\phi)$ iff $\sigma \models \phi$ for some model σ.

Proof. SMT solving is not decidable for arbitrary propositions but the QF-UFLIA fragment of quantifier-free formulas with equality, linear integer arithmetic and uninterpreted function is known to be decidable [Christ et al. 2012; Nelson and Oppen 1979]. □

4.2 Quantifier Instantiation Algorithm and Decision Procedure

As described above, verification of λ^S involves checking the validity of verification conditions that include quantifiers. Quantifier instantiation in SMT solvers is an active research topic [Ge and de Moura 2009; Reynolds et al. 2013] and often requires heuristics or explicit matching triggers. However, heuristics can cause unpredictable results and trigger-based instantiation might lead to infinite matching loops. This section describes a bounded quantifier instantiation algorithm that avoids matching loops and brittle heuristics, thus enabling a predictable decision procedure for verification conditions.

$$\begin{aligned} P \in \text{VerificationConditions} ::= \\ \tau \mid \neg P \mid P \wedge P \mid P \vee P \mid pre_1(\otimes, \tau) \mid pre_2(\oplus, \tau, \tau) \mid \\ pre(\tau, \tau) \mid post(\tau, \tau) \mid call(\tau) \mid \forall x.\{call(x)\} \Rightarrow P \mid \exists x.P \end{aligned}$$

The syntax of verification conditions P used in verification rules is similar to the syntax for propositions but universal quantifiers in verification conditions (VCs) have explicit matching patterns to indicate that instantiation requires a *trigger*. Accordingly, the construct $call(x)$ is introduced to act as an instantiation trigger that does not otherwise affect validity of propositions, i.e. $call(x)$ can always assumed to be true. Intuitively, $call(x)$ represents a function call or an asserted function specification while $\forall x.\{call(x)\} \Rightarrow P$ corresponds to a function definition or an assumed function specification. For the trigger $call(x)$, x denotes the argument of the call; the callee is omitted as triggers are matched irrespective of their callees by the instantiation algorithm.

The complete decision procedure for VCs including quantifier instantiation is shown in Figure 5.

To make the definition more concise, we first define contexts $P^+[\circ]$ and $P^-[\circ]$ for a VC with positive and negative polarity regarding negation. Using this definition, the set of call triggers in negative positions can be defined as the set of triggers for which there exists a context with negative polarity.

The procedure $lift^+$ matches universal quantifiers in positive and existential quantifiers in negative positions. In both cases, an equivalent VC without the quantifier can be obtained by renaming the quantified variable to a fresh variable that is implicitly universally quantified. It is important to note that the matching pattern $call(y)$ of a universal quantifier now becomes a part of an implication thereby available to instantiate further quantifiers. The lifting is repeated until no more such quantifiers can be found.

The procedure $instantiateOnce^-$ performs one round of trigger-based instantiation such that each universal quantifier with negative polarity is instantiated with all available triggers in negative position. All such instantiations are conjoined with the original quantifier.

Both lifting and instantiation are repeated for multiple iterations by the recursive $instantiate^-$ procedure. As a final step, $erase^-$ removes all remaining triggers and quantifiers in negative positions.

The overall decision procedure $\langle P \rangle$ performs n rounds of instantiations where n is the maximum level of quantifier nesting. The original VC P is considered valid if SMT solving cannot refute the resulting proposition.

VCs P can syntactically include both existential and universal quantifiers in both positive and negative positions. However, we can show that VCs generated by the verifier have existential quantifiers only in negative positions.

Theorem 2 (Decision Procedure Termination)**.** If P does not contain existential quantifiers in negative positions, the decision procedure $\langle P \rangle$ terminates.

Proof. The $lift^+$ function eliminates a quantifier during each recursive call and therefore terminates when there are no more matching quantifiers in the formula. $instantiateOnce^-$ and $erase^-$ are non-recursive and trivially terminate. Since the maximum level of nesting is finite, $\langle P \rangle$ performs only a finite number of instantiations. With existential quantifiers only in negative positions, the erased and lifted result is quantifier-free, so according to Lemma 1, the final SMT solving step also terminates. □

It is now possible to compare the axiomatized validity judgement for propositions $\vdash \phi$ with the decision procedure for verification conditions $\langle P \rangle$ by translating the VC P to a proposition ϕ without triggers or matching patterns.

Definition 2 (Proposition Translation)**.** $prop(P)$ denotes a proposition such that triggers and matching patterns in P are removed and existential quantifiers $\exists x.\ P$ translated to $\neg\forall x.\ \neg prop(P)$.

$$P^+[\circ] ::= \quad \circ \mid \neg P^-[\circ] \mid P^+[\circ] \wedge P \mid P \wedge P^+[\circ] \mid P^+[\circ] \vee P \mid P \vee P^+[\circ] \qquad calls^+(P) \stackrel{\text{def}}{=} \{\, call(\tau) \mid P = P^+[call(\tau)] \,\}$$

$$P^-[\circ] ::= \quad \neg P^+[\circ] \mid P^-[\circ] \wedge P \mid P \wedge P^-[\circ] \mid P^-[\circ] \vee P \mid P \vee P^-[\circ] \qquad calls^-(P) \stackrel{\text{def}}{=} \{\, call(\tau) \mid P = P^-[call(\tau)] \,\}$$

$$\begin{array}{lll} lift^+(P) \stackrel{\text{def}}{=} & \text{match } P \text{ with} & \\ & P^+[\forall x.\{call(x)\} \Rightarrow P'] \rightarrow lift^+\left(P^+[call(y) \Rightarrow P'[x \mapsto y]]\right) & (y \text{ fresh}) \\ & P^-[\exists x.P'] \rightarrow lift^+\left(P^-[P'[x \mapsto y]]\right) & (y \text{ fresh}) \\ & \text{otherwise} \rightarrow P & \end{array}$$

$$instantiateOnce^-(P) \stackrel{\text{def}}{=} P\left[P^-\left[(\forall x.\{call(x)\} \Rightarrow P')\right] \mapsto P^-\left[(\forall x.\{call(x)\} \Rightarrow P') \wedge \bigwedge_{call(\tau) \in calls^-(P)} P'[x \mapsto \tau]\right]\right]$$

$$erase^-(P) \stackrel{\text{def}}{=} P\left[P^-\left[(\forall x.\{call(x)\} \Rightarrow P'')\right] \mapsto P^-\left[\text{true}\right],\ P^+\left[call(\tau)\right] \mapsto P^+\left[\text{true}\right],\ P^-\left[call(\tau)\right] \mapsto P^-\left[\text{true}\right]\right]$$

$$instantiate^-(P, n) \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } erase^-(lift^+(P)) \text{ else } instantiate^-(instantiateOnce^-\left(lift^+(P)\right), n-1)$$

$$\langle P \rangle \stackrel{\text{def}}{=} \text{let } n = \text{maximum level of quantifier nesting of } P \text{ in } \neg Sat\left(\neg instantiate^-(P, n)\right) \qquad \boxed{\langle P \rangle}$$

Figure 5: The decision procedure lifts, instantiates and finally eliminates quantifiers. The number of iterations is bounded by the maximum level of quantifier nesting.

$$\begin{array}{rl} e \in \text{Expressions} & ::= \\ & \text{let } x = \text{true in } e \mid \text{let } x = \text{false in } e \mid \text{let } x = n \text{ in } e \mid \\ & \text{let } f(x) \text{ req } R \text{ ens } S = e \text{ in } e \mid \text{let } y = \otimes x \text{ in } e \mid \\ & \text{let } z = x \oplus y \text{ in } e \mid \text{let } y = f(x) \text{ in } e \mid \text{if } (x)\ e \text{ else } e \mid \text{return } x \\ R, S \in \text{Specs} & ::= \tau \mid \neg R \mid R \wedge R \mid R \vee R \mid \text{spec } \tau(x) \text{ req } R \text{ ens } S \\ \kappa \in \text{Stacks} & ::= (\sigma, e) \mid \kappa \cdot (\sigma, \text{let } y = f(x) \text{ in } e) \end{array}$$

Figure 6: Syntax of λ^S programs. Function definitions have pre- and postconditions written as simple logical propositions with the *spec* syntax for higher-order functions.

Theorem 3 (Quantifier Instantiation Soundness)**.** If P has no existential quantifiers in negative positions, then $\langle P \rangle$ implies $\vdash prop(P)$.

Proof. By Axiom 5, $list^+$ preserves equisatisfiability. Note that any conjuncts inserted by $instantiateOnce^-$ could also be obtained via classical (not trigger-based) instantiation. Furthermore, since $erase^-$ only removes quantifiers in negative positions and (inconsequential) triggers, the resulting propositions are implied by the original non-erased VC. Finally, with existential quantifiers only in negative positions, the erased and lifted result is quantifier-free. Therefore, Axiom 1 can be used to show that a valid VC according to the decision procedure is also valid without trigger-based instantiation[10]. □

4.3 Syntax and Operational Semantics of λ^S

Figure 6 defines the syntax of λ^S. Programs are assumed to be in A-normal form [Flanagan et al. 1993] and the dynamic semantics uses environments and stack configurations. This formalism avoids substitution in expressions and assertions in order to simplify subsequent proofs. Here, a function definition let $f(x)$ req R ens S = e_1 in e_2 is annotated with a precondition R and a postcondition S. These specifications can include terms τ such as constants, program variables, and uninterpreted function application $\tau(\tau)$ as well as logical connectives, and a special syntax "spec $\tau(x)$ req R ens S" that describes the pre- and postcondition of a function value.

The operational semantics of λ^S is specified by a small-step evaluation relation over stack configurations κ, as shown in Figure 7. Most noteworthily, the callee function name is added to the environment at each call to enable recursion, and function pre- and postconditions are not checked or enforced during evaluation.

The evaluation of a stack configuration terminates either by getting stuck or by reaching a successful completion configuration.

Definition 3 (Evaluation Finished)**.** A stack κ has terminated successfully, abbreviated with $terminated(\kappa)$, if there exists σ and x such that $\kappa = (\sigma, \text{return } x)$ and $x \in \sigma$.

4.4 Program Verification

The verification rules of λ^S are inductively defined in terms of a verification judgement $P \vdash e : Q$ as shown in Figure 8. Given a known precondition P and an expression e, a verification rule checks potential verification conditions and generates a postcondition Q. This postcondition contains a hole $\bullet$ for the evaluation result of e. Since λ^S is purely functional, P still holds after evaluating e, so we call Q the *marginal postcondition* and $P \wedge Q[\bullet]$ the *strongest postcondition*.

As an example, a unary operation such as let $y = \otimes x$ in e is verified with the rule VC-UNOP. It requires x to be a variable in scope, i.e. a variable that is free in the precondition P. To avoid name clashes, the result y should not be free. Additionally, the VC $\langle P \Rightarrow pre(\otimes, x) \rangle$ needs to be valid for all assignments of free variables (such as x). This check ensures that the value of x is in the domain of the operator $\otimes$. The rules VC-BINOP, VC-IF, etc. follow analogously.

For function applications $f(x)$, an additional $call(x)$ trigger is assumed to instantiate quantified formulas that correspond to the function definition or specification of the callee.

[10] A complete proof is available at: https://github.com/levjj/esverify-theory/

$$\boxed{\kappa \hookrightarrow \kappa}$$

$$\begin{array}{rll}
(\sigma, \text{let } x = v \text{ in } e) \hookrightarrow (\sigma[x \mapsto v], e) & \text{where } v \in \{\text{true}, \text{false}, n\} & \text{[E-VAL]} \\
(\sigma, \text{let } f(x) \text{ req } R \text{ ens } S = e_1 \text{ in } e_2) \hookrightarrow (\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e_1\}, \sigma\rangle], e_2) & & \text{[E-CLOSURE]} \\
(\sigma, \text{let } y = \otimes x \text{ in } e) \hookrightarrow (\sigma[y \mapsto v], e) & \text{where } v = \delta(\otimes, \sigma(x)) & \text{[E-UNOP]} \\
(\sigma, \text{let } z = x \oplus y \text{ in } e) \hookrightarrow (\sigma[z \mapsto v], e) & \text{where } v = \delta(\oplus, \sigma(x), \sigma(y)) & \text{[E-BINOP]} \\
(\sigma, \text{let } z = f(y) \text{ in } e) \hookrightarrow (\sigma_f[g \mapsto \sigma(f), x \mapsto \sigma(y)], e_f) \cdot (\sigma, \text{let } z = f(y) \text{ in } e) & & \text{[E-CALL]} \\
& \text{where } \sigma(f) = \langle g(x) \text{ req } R \text{ ens } S\ \{e_f\}, \sigma_f\rangle & \\
(\sigma, \text{return } z) \cdot (\sigma_2, \text{let } y = f(x) \text{ in } e_2) \hookrightarrow (\sigma_2[y \mapsto \sigma(z)], e_2) & & \text{[E-RETURN]} \\
(\sigma, \text{if } (x)\ e_1 \text{ else } e_2) \hookrightarrow (\sigma, e_1) & \text{if } \sigma(x) = \text{true} & \text{[E-IF-TRUE]} \\
(\sigma, \text{if } (x)\ e_1 \text{ else } e_2) \hookrightarrow (\sigma, e_2) & \text{if } \sigma(x) = \text{false} & \text{[E-IF-FALSE]} \\
\kappa \cdot (\sigma, \text{let } y = f(x) \text{ in } e) \hookrightarrow \kappa' \cdot (\sigma, \text{let } y = f(x) \text{ in } e) & \text{if } \kappa \hookrightarrow \kappa' & \text{[E-CONTEXT]}
\end{array}$$

Figure 7: Operational semantics

$$\boxed{P \vdash e : Q}$$

$$\begin{array}{rcl}
Q[\bullet] \in \text{PropositionContexts} & ::= & P \mid \eta[\bullet] \mid \neg Q[\bullet] \mid Q[\bullet] \wedge Q[\bullet] \mid Q[\bullet] \vee Q[\bullet] \mid pre_1(\otimes, \eta[\bullet]) \mid pre_2(\oplus, \eta[\bullet], \eta[\bullet]) \mid \\
& & pre(\eta[\bullet], \eta[\bullet]) \mid post(\eta[\bullet], \eta[\bullet]) \mid call(\eta[\bullet]) \mid \forall x.\{call(x)\} \Rightarrow Q[\bullet] \mid \exists x.\ Q[\bullet] \\
\eta[\bullet] \in \text{TermContexts} & ::= & \bullet \mid \tau \mid\ \otimes \eta[\bullet] \mid \eta[\bullet] \oplus \eta[\bullet] \mid \eta[\bullet](\eta[\bullet])
\end{array}$$

$$\frac{x \notin FV(P) \quad v \in \{\text{true}, \text{false}, n\} \quad P \wedge x = \text{v} \vdash e : Q}{P \vdash \text{let } x = v \text{ in } e : \exists x.\ x = v \wedge Q}\ \text{VC-VAL} \qquad \frac{x \in FV(P) \quad y \notin FV(P) \quad \langle P \Rightarrow pre(\otimes, x)\rangle \quad P \wedge y = \otimes x \vdash e : Q}{P \vdash \text{let } y = \otimes x \text{ in } e : \exists y.\ y = \otimes x \wedge Q}\ \text{VC-UNOP}$$

$$\frac{x \in FV(P) \quad y \in FV(P) \quad z \notin FV(P) \quad \langle P \Rightarrow pre(\oplus, x, y)\rangle \quad P \wedge z = x \oplus y \vdash e : Q}{P \vdash \text{let } z = x \oplus y \text{ in } e : \exists z.\ z = x \oplus y \wedge Q}\ \text{VC-BINOP}$$

$$\frac{\begin{array}{c} f \notin FV(P) \quad x \notin FV(P) \quad f \neq x \quad x \in FV(R) \quad FV(R) \subseteq FV(P) \cup \{f, x\} \quad FV(S) \subseteq FV(P) \cup \{f, x\} \\ P \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e_1 : Q_1 \quad \langle P \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \wedge Q_1[f(x)] \Rightarrow S\rangle \\ P \wedge \text{spec } f(x) \text{ req } R \text{ ens } (Q_1[f(x)] \wedge S) \vdash e_2 : Q_2 \end{array}}{P \vdash \text{let } f(x) \text{ req } R \text{ ens } S = e_1 \text{ in } e_2 : \exists f.\ \text{spec } f(x) \text{ req } R \text{ ens } (Q_1[f(x)] \wedge S) \wedge Q_2}\ \text{VC-FUNC}$$

$$\frac{f \in FV(P) \quad x \in FV(P) \quad y \notin FV(P) \quad \langle P \wedge call(x) \Rightarrow isFunc(f) \wedge pre(f, x)\rangle \quad P \wedge call(x) \wedge post(x) \wedge y = f(x) \vdash e : Q}{P \vdash \text{let } y = f(x) \text{ in } e : \exists y.\ call(x) \wedge post(f, x) \wedge y = f(x) \wedge Q}\ \text{VC-APP}$$

$$\frac{x \in FV(P) \quad \langle P \Rightarrow isBool(f)\rangle \quad P \wedge x \vdash e_1 : Q_1 \quad P \wedge \neg x \vdash e_2 : Q_2}{P \vdash \text{if } (x)\ e_1 \text{ else } e_2 : (x \Rightarrow Q_1) \wedge (\neg x \Rightarrow Q_2)}\ \text{VC-ITE} \qquad \frac{x \in FV(P)}{P \vdash \text{return } x : x = \bullet}\ \text{VC-RETURN}$$

Figure 8: The judgement $P \vdash e : Q$ verifies the expression e given a known context P.

The most complex rule concerns the verification of function definitions, such as let $f(x)$ req R ens $S = e_1$ in e_2. Here, the annotated precondition R, the specification of f and the marginal postcondition $Q_1[f(x)]$ together have to imply the annotated postcondition S. Any recursive calls of f appearing in its function body will instantiate its (non-recursive) specification, while subsequent calls of f in e_2 will use a postcondition that is strengthened by the generated marginal postcondition. This corresponds to expanding or inlining the function definition by one level at each non-recursive callsite.

The special syntax spec $\tau(x)$ req R ens S, as used in verification rules, user-provided pre- and postconditions, is a notation that desugars to a universal quantifier when appearing in a verification condition.

Notation 1 (Function Specifications). spec $\tau(x)$ req R ens $S \equiv$ $isFunc(\tau) \wedge \forall x.\{call(x)\} \Rightarrow ((R \Rightarrow pre(\tau, x)) \wedge (post(\tau, x) \Rightarrow S))$

That is, if a function call instantiates this quantifier, the precondition R of the *spec* satisfies the precondition of f and the postcondition S of the *spec* is implied by the postcondition of f. For a concrete function call, this means that R needs to be asserted by the calling context and S can be assumed at the callsite.

4.5 Soundness

Based on the decision procedure and the verification rules described in the previous sections, it is possible to show that verified programs evaluate to completion without getting stuck. While annotated assertions are not directly enforced by the operational semantics, the preconditions of operators have to be satisfied and can

be arbitrarily complex. Therefore, this soundness property also ensures that annotated assertions, such as postconditions, hold during evaluation for concrete values of free variables.

First, it is important to note that quantifiers in generated VCs only appear in certain positions.

Lemma 1. If P is a proposition with existential quantifiers only in positive positions, then each VC used in the derivation tree of $P \vdash e : Q$ has existential quantifiers only in negative positions.

Proof. All VCs in the verification rules shown in Figures 8 are implications of the form $\langle P'' \Rightarrow Q'' \rangle$. In each of these implications, there are no existential quantifiers in Q'', as user-supplied postconditions S have no existential quantifiers. Additionally, all propositions P'' on the left-hand side have existential quantifiers only in positive positions, since existential quantifiers in marginal postconditions are always in positive positions. □

From Lemma 1 and Theorem 2, it follows that verification always terminates, ensuring a predictable verification process.

As mentioned in section 4.1, the axiomatization of logical propositions does not include evaluation and treats terms $\tau(\tau)$ as uninterpreted symbols rather than function calls. However, for a proof of verification soundness it is necessary to establish equalities about function application for a given closure and argument value.

Axiom 6. If $(\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x], e) \longrightarrow^* (\sigma', y)$ and $\sigma'(y) = v$ then $\vdash \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle(v_x) = v$.

Similarly, axioms about $pre(f, x)$ and $post(f, x)$ can be added for concrete values of f and x.

Axiom 7. Iff $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x] \models R$ then $\vdash pre(\langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, v_x)$.

Axiom 8. If $\vdash \sigma : Q_1$ and $Q_1 \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e : Q_2[\bullet]$ and $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x] \models Q_2[f(x)] \wedge S$ then $\vdash post(\langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, v_x)$.

Axiom 9. If $\vdash \sigma : Q_1$ and $Q_1 \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e : Q_2[\bullet]$ and $\vdash post(\langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, v_x)$ then $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x] \models Q_2[f(x)] \wedge S$

Based on these axioms, definitions and verification rules, it can now be shown that verifiable expressions evaluate to completion without getting stuck, i.e. all reachable configurations either terminate normally or can be further evaluated.

Theorem 4 (Verification Safety)**.** If $true \vdash e : Q$ and $(\emptyset, e) \hookrightarrow^* \kappa$ then $terminated(\kappa)$ or $\kappa \hookrightarrow \kappa'$ for some κ'.

Proof. Due to the complex quantifier instantiation of the decision procedure, we first show verification safety for a similar but undecidable verification judgement without quantifier instantiation. With theorem 3, soundness of this verification judgement also implies soundness with trigger-based quantifier instantiation. The verification safety proof for this alternate judgement uses a standard progress/preservation proof strategy, where the notion of verifiability is extended to stack configurations. Here, a given runtime stack is considered verifiable if, at each stack frame, the expression is verifiable with the translation of σ as precondition. A complete proof is available at: https://github.com/levjj/esverify-theory/. □

$$
\begin{array}{lll}
c \in \text{ClassNames} & fd \in \text{FieldNames} & this \in \text{Variables} \\
v \in \text{Values} & ::= \ldots \mid C(\bar{v}) & \\
e \in \text{Expressions} & ::= \ldots \mid \text{let } y = \text{new } C(\bar{x}) \text{ in } e \mid \text{let } y = x.fd \text{ in } e & \\
\tau \in \text{Terms} & ::= \ldots \mid \tau.fd \mid C(\bar{\tau}) & \\
P \in \text{VCs} & ::= \ldots \mid fd \text{ in } \tau \mid \tau \text{ instanceof } C \mid & \\
 & \quad access(\tau) \mid \forall x.\{access(x)\} \Rightarrow P & \\
D \in \text{ClassDefs} & ::= \text{class } C(\overline{fd}) \text{ inv } S &
\end{array}
$$

$$
\frac{x \in FV(P) \quad y \notin FV(P) \quad \langle P \Rightarrow fd \text{ in } x\rangle \quad P \wedge y = x.fd \vdash e : Q}{P \vdash \text{let } y = x.fd \text{ in } e : \exists y.\ y = x.fd \wedge Q}
$$

$$
\frac{\begin{array}{c}\overline{x \in FV(P)} \quad y \notin FV(P) \quad \text{class } C(\overline{fd}) \text{ inv } S \in \overline{D} \\ \langle P \wedge this = C(\bar{x}) \wedge this \text{ instanceof } C \Rightarrow S\rangle \\ P \wedge y = C(\bar{x}) \wedge y \text{ instanceof } C \vdash e : Q\end{array}}{P \vdash \text{let } y = \text{new } C(\bar{x}) \text{ in } e : \exists y.\ y = C(\bar{x}) \wedge y \text{ instanceof } C \wedge Q}
$$

Figure 9: Extending the verification rules of λ^S with simple immutable classes with class invariants.

4.6 Extensions

The core language λ^S includes higher-order functions but does not address other language features supported by esverify, such as imperative programs and complex recursive data types.

Extending λ^S for imperative programs would entail syntax, semantics and verification rules for allocating, mutating and referencing values stored in the heap. Most noteworthy, loops and recursion invalidate previous facts about heap contents and therefore require precise invariants. This issue can be addressed with segmentation logic and dynamic frames [Smans et al. 2009].

Additionally, λ^S can be extended to support "classes" as shown in Figure 9. These classes are immutable and more akin to recursive data types as they do not support inheritance. Each class definition consists of an ordered sequence of fields and an invariant S that is specified in terms of a free variable *this*. The class invariant can be used to express complex recursive data structures such as the parameterized linked list shown in Section 2.5.

The class invariant has to be instantiated for concrete instances of the class, so a trigger $access(x)$ is inserted into verification conditions at each field access, similarly to $call(x)$ trigger for function calls. However, unlike function definitions, class definitions $\overline{D}$ are global. Therefore, we augment verification conditions such that for each class $C(\overline{fd})$ inv $S \in \overline{D}$ the following quantifier is assumed:

$$\forall x.\{access(x)\} \Rightarrow \left(x \text{ instanceof } C \Rightarrow \left(\overline{x \text{ has } fd} \wedge S[this \mapsto x]\right)\right)$$

This quantifier is instantiated by an $access(x)$ trigger and the instantiated formula includes both the class invariant as well as a description of its fields.

5 COMPARISON WITH REFINEMENT TYPES

Despite being dynamically typed, the verification rules shown in Figure 8 resemble static typing rules. In this section, we provide a brief comparison of this program verification approach with static type checking.

$$
\begin{array}{rcl}
t \in \text{TypedExpressions} & ::= & \ldots \mid \text{let } f(x:T):T = t \text{ in } t \\
T \in \text{Types} & ::= & \{\, x : B \mid R \,\} \mid x:T \to T \\
B \in \text{BaseTypes} & ::= & \text{Bool} \mid \text{Int} \\
\Gamma \in \text{TypeEnvironments} & ::= & \emptyset \mid \Gamma, x:T
\end{array}
$$

$\boxed{\Gamma \vdash t : T}$

$$
\frac{\begin{array}{c} x, f \notin dom(\Gamma) \quad FV(T_x) \subseteq dom(\Gamma) \quad FV(T) \subseteq dom(\Gamma) \cup \{x\} \\ \Gamma, x:T_x, f:(x:T_x \to T) \vdash t_1 : T_1 \qquad \Gamma, x:T_x \vdash T_1 <: T \\ \Gamma, x:T_x, f:(x:T_x \to T) \vdash t_2 : T_2 \end{array}}{\Gamma \vdash \text{let } f(x:T):T = t_1 \text{ in } t_2 : T_2} \text{ T-FN}
$$

$\boxed{\Gamma \vdash T <: T}$

$$
\frac{B_1 = B_2 \qquad \langle [\![\Gamma]\!] \wedge R \Rightarrow S \rangle \qquad x \notin dom(\Gamma)}{\Gamma \vdash \{\, x : B_1 \mid R \,\} <: \{\, x : B_2 \mid S \,\}} \text{ ST-REF}
$$

$$
\frac{\Gamma \vdash T'_x <: T_x \qquad \Gamma, x:T'_x \vdash T <: T'}{\Gamma \vdash (x:T_x \to T) <: (x:T'_x \to T')} \text{ ST-FUN}
$$

$\boxed{[\![\Gamma]\!]}$

$$
\begin{array}{rcl}
[\![\emptyset]\!] & \stackrel{\text{def}}{=} & \text{true} \\
[\![\Gamma, x:T]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!] \wedge [\![x:T]\!]
\end{array}
$$

$\boxed{[\![\tau : T]\!]}$

$$
\begin{array}{rcl}
[\![\tau : \{x : \text{Bool} \mid R\}]\!] & \stackrel{\text{def}}{=} & isBool(\tau) \ \wedge \ R[x \mapsto \tau] \\
[\![\tau : \{x : \text{Int} \mid R\}]\!] & \stackrel{\text{def}}{=} & isInt(\tau) \ \wedge \ R[x \mapsto \tau] \\
[\![\tau : (x:T_x \to T)]\!] & \stackrel{\text{def}}{=} & \text{spec } \tau(x) \text{ req } [\![x:T_x]\!] \text{ ens } [\![\tau(x) : T]\!]
\end{array}
$$

Figure 10: Selected typing and subtyping rules of a statically typed language λ^T. Functions are annotated with types where refinements R are analogous to specifications in λ^S.

An comprehensive formalization of refinement and dependent type systems and a formal proof that examines their expressiveness is beyond the scope of this paper. However, by describing a translation of types to assertions and investigating concrete examples, we enable a comparison of ESVERIFY with systems such as LiquidHaskell [Vazou et al. 2014] and conjecture that it is at least as expressive.

First, we assume a language λ^T similar to λ^S but with type annotations instead of pre- and postconditions. Figure 10 shows an excerpt of such a language. Here, a type is either a dependent function type or a refined base type where refinements R are consistent with specifications in λ^S.

Given such a language, the typing rule for function definitions (T-FN) checks the function body t_1 and compares its type T_1 with the annotated return type T. This return type might refer to the function argument in order to support dependent types. However, other free variables in refinements can break hygiene, so T-FN restricts free variables in user-provided types T_x and T accordingly.

The subtyping relation is also shown in Figure 10. Most importantly, subtyping of refined base types requires checking an implication between the refinements and it requires translating the type environment Γ to a logical formula $[\![\Gamma]\!]$, where function types translate to the function specifications with the *spec* syntax.

Intuitively, the logical implication used for refinements also extends to translated function types, so if $\Gamma \vdash T <: T'$ then for all terms τ, $[\![\Gamma]\!] \wedge [\![\tau : T]\!]$ implies $[\![\tau : T']\!]$.

As an example, the following λ^T expression is well-typed as the return type is a subtype of the argument type:

$$
\begin{array}{l}
\text{let } f\,(g : \ (x : \{x : \text{Int} \mid x > 3\} \to \{y : \text{Int} \mid y > 8\})) : \\
\qquad\qquad (x : \{x : \text{Int} \mid x > 4\} \to \{y : \text{Int} \mid y > 7\}) \ = \ g \text{ in } \ldots
\end{array}
$$

Translated into ESVERIFY, we obtain a program with **spec** in pre- and postcondition:

```
function f (g) {
  requires(  spec(g, x => x > 3, (x,y) => y > 8));
  ensures(r=>spec(r, x => x > 4, (x,y) => y > 7));
  return g;
}
```

This program is verifiable with the quantifier instantiation algorithm described in section 4.2. The second **spec** is translated to a universal quantifier in positive position that will be lifted, introducing a free global variable x. This also exposes a $call(x)$ trigger in negative position that now instantiates the quantifier in the antecedent. The resulting proposition can now be checked without further instantiations by comparing the argument and return propositions of both functions for all possible values of x.

This suggests that the translation of λ^T to λ^S programs preserves verifiability, i.e. well-typed λ^T programs translate to verifiable λ^S programs.

Conjecture 1 (Translated well-typed expressions are verifiable)**.** If $[\![t]\!]$ is the translation of a λ^T expression t to λ^S, then $\Gamma \vdash t : T$ implies $[\![\Gamma]\!] \vdash [\![t]\!] : Q$ for some Q.

A formal proof of this conjecture needs to take quantifier instantiation and the quantifier nesting bound into account. This introduces immense complexity for the proof and goes beyond the scope of this paper but might be addressed in future work.

Coincidentally, a sound translation of types to annotations also enables seamless interweaving of statically-typed λ^T expressions with dynamically-typed λ^S programs in a sound way. This might be a step towards a full spectrum type system that bridges the gap between verification and type checking.

6 RELATED WORK

There have been decades of prior work on software verification. In particular, static verification of general purpose programming languages based on pre- and postconditions has previously been explored in verifiers such as ESC/Java [Flanagan et al. 2002; Leino 2001], JaVerT [Fragoso Santos et al. 2018], Dafny [Leino 2010, 2013, 2017] and LiquidHaskell [Vazou et al. 2017, 2014, 2018].

ESC/Java [Flanagan et al. 2002] proposed the idea of using undecidable but SMT-solvable logic to provide more powerful static checking than traditional type systems. Their proposed extended static checking gave up on soundness to do so and instead focused on the utility of tools to find bugs.

JaVert [Fragoso Santos et al. 2018] is a more recent program verifier for JavaScript. It supports object-oriented programs but, in contrast to ESVERIFY, does not support higher-order functions. Other related work on static analysis of JavaScript include Loop-Sensitive Analysis [Park and Ryu 2015], the TAJS Type analyzer for JavaScript [Andreasen and Møller 2014], and type systems such as TypeScript, Flow and Dependent JavaScript [Chugh et al. 2012].

esverify follows as different approach as it relies on manually annotated assertions that are generally more expressive than types.

Dafny [Leino 2010] seeks to provide a full verification language with support for both functional and imperative programs. Dafny offers comprehensive support for verified programming, such as ghost functions and parameters, termination checking, quantifiers in user-supplied annotations, and reasoning about the heap. However, in contrast to esverify and LiquidHaskell, Dafny requires function calls in an assertion context to satisfy the precondition instead of treating these as uninterpreted calls. Therefore, Dafny does not support higher-order proof functions such as those shown in Section 2.8. Additionally, quantifier instantiation in Dafny is often implicit and based on heuristics, which often results in a brittle and unpredictable verification process.

In trying to find a compromise, with predictable checking but also a larger scope than traditional type systems, LiquidHaskell is most closely related to esverify. In fact, the refinement type system discussed in Section 5 loosely resembles its formalization by Vazou et. al. [Vazou et al. 2014]. More recently, LiquidHaskell introduced *refinement reflection* [Vazou et al. 2018], which enables external proofs in a similar way as the **spec** construct in esverify, and *proof by logical evaluation* which is a close cousin to the quantifier instantiation algorithm in Section 4.2 but is not based on triggering matching patterns. In contrast to LiquidHaskell, esverify is not based on static type checking and thus also supports dynamically-typed programming idioms such as dynamic type checks instead of injections.

Finally, trigger-based quantifier instantiation, as used by the decision procedure described in Section 4.2, has been studied by extensive prior work [Dross et al. 2016; Ge and de Moura 2009; Leino and Pit-Claudel 2016; Reynolds et al. 2013]. The instantiation in esverify is specifically bounded in order to prevent matching loops, but further research could provide this kind of instantiation as a built-in feature of off-the-shelf SMT solvers.

7 CONCLUSION

This paper introduced esverify, a program verifier for dynamically-typed JavaScript programs. esverify supports both dynamic programming idioms as well as higher-order functional programs, and thus has an expressiveness comparable to and potentially greater than common refinement type systems. Internally, the verifier relies on a bounded quantifier instantiation algorithm and SMT solving, yielding concrete counterexamples for verification errors. We showed that this approach to program verification is sound by formalizing the quantifier instantiation algorithm and the verification rules in the Lean Theorem prover. While esverify enables verification of non-trivial programs such as MergeSort, it lacks termination checking and support for object-oriented programming. However, it would be possible to combine it with an external termination checker for total correctness [Sereni and Jones 2005], and to extend it with reasoning about the heap, such as regions or dynamic frames [Smans et al. 2009]. Finally, while the approach presented in this paper is purely static, future work might use runtime checks similar to hybrid and gradual type checking [Ahmed et al. 2011; Knowles and Flanagan 2010; Siek and Taha 2006] to enable sound execution of programs that are not fully verified.

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *POPL '11*.

Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *OOPSLA '14*.

Clark Barrett and Sergey Berezin. 2004. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *CAV'04*.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV'11*.

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *SPIN'12*.

Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent Types for JavaScript. In *OOPSLA '12*.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08*.

Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. 2016. Adding decision procedures to SMT solvers using axioms with triggers. *Journal of Automated Reasoning* (2016).

ECMA-262. 2017. *ECMAScript 2017 Language Specification* (6 / 2017 ed.).

M.D. Ernst, J. Cockrell, William G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on* 27, 2 (Feb 2001), 99–123. https://doi.org/10.1109/32.908957

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *PLDI'02*.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI '93*.

José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *POPL'18* (2018).

Carlo Alberto Furia and Bertrand Meyer. 2010. Inferring Loop Invariants Using Postconditions. In *Fields of Logic and Computation*.

Yeting Ge and Leonardo de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *CAV'09*.

Johannes Henkel and Amer Diwan. 2003. Discovering Algebraic Specifications from Java Classes. In *ECOOP 2003 Object-Oriented Programming*, Luca Cardelli (Ed.). Lecture Notes in Computer Science, Vol. 2743. Springer Berlin Heidelberg, 431–456. https://doi.org/10.1007/978-3-540-45070-2_19

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *TOPLAS* (2010).

K. Rustan M. Leino. 2001. Extended Static Checking: A Ten-Year Perspective. In *Informatics - 10 Years Back. 10 Years Ahead.*

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10).*

K. Rustan M. Leino. 2013. Developing Verified Programs with Dafny. In *International Conference on Software Engineering, (ICSE'13).*

K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* (2017).

K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *CAV'16*.

B. Liskov and L. Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *PLDI '88*.

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *TOPLAS* (1979).

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *ECOOP'15*.

Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. 2013. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *CADE'13*.

Damien Sereni and Neil D. Jones. 2005. Termination Analysis of Higher-order Functional Programs. In *APLAS'05*.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Workshop on Scheme and Functional Programming*.

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP 2009*.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *TAP'08*.

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq. *International Symposium on Haskell*.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *ICFP'14*.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan Scott, Ryan Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. In *POPL'18*.

MIL, a Monadic Intermediate Language for Implementing Functional Languages

Mark P. Jones
Portland State University
Portland, Oregon, USA
mpj@pdx.edu

Justin Bailey
Portland, Oregon, USA
jgbailey@gmail.com

Theodore R. Cooper
Portland State University
Portland, Oregon, USA
ted.r.cooper@gmail.com

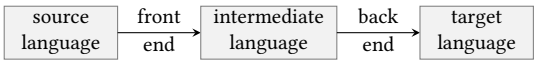
ABSTRACT

This paper describes MIL, a "monadic intermediate language" that is designed for use in optimizing compilers for strict, strongly typed functional languages. By using a notation that exposes the construction and use of closures and algebraic datatype values, for example, the MIL optimizer is able to detect and eliminate many unnecessary uses of these structures prior to code generation. One feature that distinguishes MIL from other intermediate languages in this area is the use of a typed, parameterized notion for basic blocks. This both enables new optimization techniques, such as the ability to create specialized versions of basic blocks, and leads to a new approach for implementing changes in data representation.

ACM Reference Format:
Mark P. Jones, Justin Bailey, and Theodore R. Cooper. 2018. MIL, a Monadic Intermediate Language for Implementing Functional Languages. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310238

1 INTRODUCTION

Modern compilers rely on intermediate languages to simplify the task of translating programs in a rich, high-level source language to equivalent, efficient implementations in a low-level target. For example, a compiler may use two distinct compilation steps, with a *front end* that translates source programs to an intermediate language (IL) and a *back end* that translates from IL to the target:



This splits the compilation process in to smaller, conceptually simpler components, each of which is easier to implement and maintain. Moreover, with a well-defined IL, it is possible for the front and back end components to be developed independently and to be reused in other compilers with the same source or target.

The biggest challenge in realizing these benefits is in identifying a suitable intermediate language. A good IL, for example, should retain some high-level elements, allowing a relatively simple translation from source to IL, and avoiding a premature commitment to low-level details such as data representation and register allocation. At the same time, it should also have some low-level characteristics, exposing implementation details that are hidden in the original source language as opportunities for optimization, but ultimately also enabling a relatively simple translation to the target.

In this paper, we describe MIL, a "monadic intermediate language", inspired by the monadic metalanguage of Moggi [12] and the monad comprehensions of Wadler [16], that has been specifically designed for use in implementations of functional programming languages. MIL bridges the tension between the high-level and low-level perspectives, for example, by including constructs for constructing, defining, and entering closures. These objects, and associated operations, are important in the implementation of higher-order functions, but the details of their use are typically not explicit in higher-level languages. At the same time, MIL does not fix a specific machine-level representation for closures, and does not specify register-level protocols for entering closures or for accessing their fields, delegating these decisions instead to individual back ends. This combination of design choices enables an optimizer for MIL, for example, to detect and eliminate many unnecessary uses of closures before the resulting code is passed to the back end.

MIL was originally developed as a platform for experimenting with the implementation and optimization of functional languages [2]. Since then, it has evolved in significant ways, both in the language itself—which now supports a type system with multi-value returns, for example—and its implementation—including an aggressive whole-program optimizer and novel representation transformation and specialization components. The implementation can be used as a standalone tool, but it can also be integrated in to larger systems. We currently use MIL, for example, as one of several intermediate languages in a compiler for the Habit programming language [15] with the following overall structure:

In the initial stage of this compiler, Habit programs are desugared to LC, a simple functional language that includes lambda and case constructs (LC is short for "LambdaCase" and is named for those two features) but lacks the richer syntax and semantics of Habit. LC programs are then translated in to MIL, processed by the tools described in this paper, and used to generate code for LLVM [9, 10], which is a popular, lower-level IL. There is, for example, no builtin notion of closures in LLVM, so it would be difficult to implement analogs of MIL closure optimizations at this stage of the compiler. However, the LLVM tools do implement many important, lower-level optimizations and can ultimately be used to generate assembly code ("asm" in the diagram) for an executable program.

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310238

1.1 Outline and Contributions

We begin the remaining sections of this paper with a detailed overview of the syntax and features of MIL (Section 2). To evaluate its suitability as an intermediate language, we have built a practical implementation that uses MIL: as the target language for a simple front end (Section 3); as a framework for optimization (Section 5), and as a source language for an LLVM back end (Section 6). Along the way, we describe new global program transformations techniques for implementing changes in data representation (Section 4).

MIL has much in common with numerous other ILs for functional languages, including approaches based on CPS [1, 8], monads [12, 16], and other functional representations [5, 11]; these references necessarily only sample a very small portion of the literature. Our work makes some new contributions by exploring, for example, a parameterized form of basic block, and the new optimizations for "derived blocks" that this enables (Section 5.5), as well as new techniques for representation transformations (Section 4). An additional contribution of our work is the combination of many small design and engineering decisions that, together, result in a coherent and effective IL design, scaling to a practical system that can be used either as a standalone tool or as part of a larger compiler.

2 AN OVERVIEW OF MIL

We begin with an informal introduction to MIL, highlighting fundamental concepts, introducing terminology, and illustrating the concrete syntax for programs. We follow Haskell conventions [14] for basic syntax—from commenting and use of layout, to lexical details, such as the syntax of identifiers. That said, while we expect some familiarity with languages like Haskell, we will not assume deep knowledge of any specific syntax or features.

2.1 Types in MIL

MIL is a strongly typed language and all values that are manipulated in MIL must have a valid type. MIL provides several builtin types, including `Word` (machine words) and `Flag` (single-bit booleans). MIL also supports first class functions with types of the form $[d_1,\ldots,d_n]\twoheadrightarrow[r_1,\ldots,r_m]$ that map a tuple of input values with types d_i to a tuple of results with types r_j. (The prefixes `d` and `r` were chosen as mnemonics for domain and range, respectively.) Curried functions can also be described by using multiple $\twoheadrightarrow$ types. For example, the type `[Word]↠[[Word]↠[Flag]]` could be used for a curried function that compares `Word` values.

MIL programs can also include definitions for parameterized algebraic datatypes, as illustrated by the following familiar examples:

```
data List a   = Nil | Cons a (List a)
data Pair a b = MkPair a b
```

Each definition introduces a new type name (`List` and `Pair` in these examples) and one or more *data constructors* (`Nil`, `Cons`, and `MkPair`). Each data constructor has an *arity*, which is the number of fields it stores. For example, `Nil` has arity 0 and `Cons` has arity 2.

MIL also supports `bitdata` types [4] that specify a bit-level representation for all values of the type. In other respects, however, values of these types can be used in the same way as values of algebraic datatypes. With the following definitions, for example: Every value of type `Bool` is either `False` or `True`, with either option being represented by a single bit; and a `PCI` address can be described by a 16 bit value with subfields of width 8, 5, and 3:

```
bitdata Bool = False [B0] | True [B1]
bitdata PCI  = PCI [bus::Bit 8 | dev::Bit 5 | fun::Bit 3]
```

Bitdata types allow programmers to match the bit-level representation of data structures that are used in low-level systems applications, which is a key domain for Habit. Ultimately, however, `bitdata` values are represented by sequences of `Word` values (Section 4.2). For completeness, we mention that MIL also supports several other builtin types for systems programming, including the `Bit n` types seen here that represent bit vectors of width `n`.

2.2 MIL programs

The core of a MIL program is a sequence of block, closure, and top-level definitions, as suggested by the following grammar:

```
prog ::=  def₁ ... defₙ                      -- program
def  ::=  b[v₁,...,vₙ] = c                   -- block
       |  k{v₁,...,vₙ} [u₁,...,uₘ] = c       -- closure
       |  [v₁,...,vₙ] ← t                    -- top-level
       |  ...                                -- other
```

The last line here uses "..." as a placeholder for other forms of definition, such as those introducing new types (see the previous section). The full details, however, are not central to this paper.

2.3 Blocks

Blocks in MIL are like basic blocks in traditional optimizing compilers except that each block also has a list of zero or more parameters. Intuitively, these can be thought of as naming the "registers" that must be loaded with appropriate values before the block is executed. The syntax of block definitions is reminiscent of Haskell's monadic notation (without the `do` keyword), as in the following schematic example for a block `b` with arguments $u_1,\ldots,u_n$:

```
b[u₁,...,uₙ] = v₁ ← t₁  -- run t₁, capture result in v₁
               ...
               vₙ ← tₙ  -- run tₙ, capture result in vₙ
               e
```

The body of this block is executed by running the statements $v_i \leftarrow t_i$ in sequence, capturing the result of each computation t_i in a variable v_i so that it can be referenced in later steps. The expression `e` on the last line may be either a tail expression (such as a `return` or an unconditional jump to another block) or a `case` or `if` construct that performs a conditional jump.

MIL relies on type checking to ensure basic consistency properties (for example, to detect calls with the wrong number or types of arguments). The code fragment above assumes that each t_i returns exactly one result but it is possible to use computations that return zero or more results at the same time, as determined by the type of t_i. In these cases, the binding for t_i must specify a corresponding list of variables, enclosed in brackets, as in $[w_1,\ldots,w_m] \leftarrow t_i$.

2.4 Tail Expressions

MIL allows several different forms of "tail" expressions (each of which may be used in place of the expressions $t_1,\ldots,t_n$ in the preceding discussion), as summarized in the following grammar.

```
t  ::=  return [a1,...,an]  -- monadic return
    |   b[a1,...,an]        -- basic block call
    |   p((a1,...,an))      -- primitive call
    |   C(a1,...,an)        -- allocate data value
    |   C i a               -- select component
    |   k{a1,...,an}        -- allocate closure
    |   f @ [a1,...,an]     -- enter closure
```

Every tail expression represents a computation that (potentially) returns zero or more results:

- `return [a1,...,an]` returns the values $a_1, \ldots, a_n$ immediately with no further action. The most frequently occurring form produces a single result, `a`, and may be written without brackets as `return a`.
- `b[a1,...,an]` is a call to a block `b` with the given arguments. When a block call occurs at the end of a code sequence, it can potentially be implemented as a tail call (i.e., a jump). This is why we refer to expressions in the grammar for `t` as *tail* expressions or sometimes *generalized tail calls*.
- `p((a1,...,an))` is a call to a primitive `p` with the given arguments. Primitives typically correspond to basic machine instructions (such as `add`, `and`, or `shr`), but can also be used to link to external functions. Primitive calls are written with double parentheses around the argument list so that they are visibly distinguished from block and constructor calls.
- `C(a1,...,an)` allocates a data value with constructor `C` and components $a_1, \ldots, a_n$, where `n` is the arity of `C`.
- `C i a` is a selector that returns the `i`th component of the value in `a`, assuming that it was constructed with `C`. For example, having executed `v ← C(a1,...,an)`, a subsequent `C i v` call (with $0 \le i < n$) will return a_{i+1}. (Note that `i` must be a constant and that the first component has index `0`.)
- `k{a1,...,an}` allocates a closure (i.e., the representation of a function value) with a code pointer specified by `k` and arguments $a_1, \ldots, a_n$. This is only valid in a program that includes a corresponding definition for `k`. Closures are an important feature of MIL and are the topic of further discussion throughout this paper.
- `f @ [a1,...,an]` is a call to an unknown function, `f`, with arguments $a_1, \ldots, a_n$. For this to be valid, `f` must hold a closure for a function that expects exactly `n` arguments. The process of executing an expression of this form is often referred to as *entering* a closure. Mirroring the syntax for `return`, we write `f @ a` in the case where there is exactly one argument. As another special case, an unknown function `f` can be applied to an empty list of arguments, as in `f @ []`; this can be used, for example, to invoke a (monadic) thunk.

2.5 Atoms

In the preceding examples, the symbols $a, a_1, \ldots, a_n$, and `f` represent arbitrary *atoms* that are either numeric constants (integer literals) or variable names. The latter correspond either to *temporaries* (i.e., local variables holding parameter values or intermediate results) or to variables defined at the top level (see Section 2.9):

```
a  ::=  i  -- a constant (i.e., an integer literal)
    |   v  -- a variable name
```

Atoms are the only form of expression that can be used as arguments in tail expressions. If a program requires the evaluation of a more complex expression, then it must be computed in steps using temporaries to capture intermediate results. For example, the following block calculates the sum of the squares of its inputs using the `mul` and `add` primitives for basic arithmetic:

```
sumSqrs[x,y] = u ← mul((x,x))  -- compute x*x in u
               v ← mul((y,y))  -- compute y*y in v
               add((u,v))      -- return the sum
```

2.6 Block Types

Blocks are not first class values in MIL, so they cannot be stored in variables, passed as inputs to a block, or returned as results. Nevertheless, it is still useful to have a notion of types for describing the inputs and results of each block, and we use the notation `[d1,...,dn] >>= [r1,...,rm]` for this where d_i and r_j are the types of the block's inputs and results, respectively. For example, the type of `sumSqrs` can be declared explicitly as follows:[1]

```
sumSqrs :: [Word,Word] >>= [Word]
```

Blocks may have polymorphic types, as in the following examples, the second of which also illustrates a block with multiple results:

```
idBlock    :: forall (a::type). [a] >>= [a]
idBlock[x]  = return x

dupBlock   :: forall (a::type). [a] >>= [a, a]
dupBlock[x] = return [x, x]
```

The explicit `forall` quantifiers here give a precise way to document polymorphic types but they are not required; those details can be calculated automatically instead using kind and type inference.

It is important to distinguish block types, formed with `>>=`, from the first-class function types using ↠ that were introduced in Section 2.1. The notations are similar because both represent a kind of function, but there are also fundamental differences. In particular, blocks are executed by jumping to a known address and not by entering a closure, as is done with a first-class function.

2.7 Code Sequences

The syntax for the code sequences on the right of each block definition is captured in the following grammar:

```
c    ::=  [v1,...,vn] ← t; c     -- monadic bind
      |   assert a C; c          -- data assertion
      |   t                      -- tail call
      |   if v then bc1 else bc2 -- conditional
      |   case v of alts         -- case construct
alts ::=  {alt1;...;altn}        -- alternatives
alt  ::=  C → bc                 -- match against C
      |   _ → bc                 -- default branch
bc   ::=  b[a1,...,an]           -- block call
```

As illustrated previously, any given code sequence begins with zero or more *monadic bind* steps (or *statements*) of the form `vs ← t` (where `vs` is either a single temporary or a bracketed list). Each such step describes a computation that begins by executing the tail

[1] The same block type is also used for the `mul` and `add` primitives.

t, binding any results to the temporaries in vs, and then continuing with the entries in vs now in scope. The grammar also includes an assert construct; its purpose will be illustrated later.

Where space permits, we write the steps in a code sequence using a vertical layout, relying on indentation to reflect its structure. However, we will also use a more compact notation, matching the preceding grammar with multiple statements on a single line:

```
v₁ ← t₁; v₂ ← t₂; ...; vₙ ← tₙ; e
```

However they are written, every code sequence is ultimately terminated by an expression e that is either an unconditional jump (i.e., a tail expression, t); or a conditional jump (i.e., an if or case construct). Note that, like a traditional basic block, there are no labels in the middle of code sequences. As a result, we can only enter a code sequence at the beginning and, once entered, we cannot leave until we reach the end expression e.

2.8 Conditional Jumps

An if construct ends a code sequence with a conditional jump; it requires an atom, v, of type Flag as the test expression and block calls for each of the two branches. Beyond these details, if constructs work in the usual way. For example, the following two blocks can be used to calculate the maximum of two values u and v:

```
b1[u,v] = t ← primGt((u,v))  -- primitive for u>v
          if t then b2[u] else b2[v]
b2[x]   = return x
```

Other conditional jumps can be described with a case v of alts construct, using the value of v—the "discriminant"—to choose between the alternatives in alts. Specifically, a case construct is executed by scanning the list of alternatives and executing the block call bc for the first one whose constructor, C, matches v. The following code sequence corresponds to an if-then-else construct for a variable v of type Bool: the code will execute b1[x,y] if v was built using True(), or b2[z] if v was built using False():

```
case v of { True → b1[x,y]; False → b2[z] }
```

We can also write the previous expression using a default branch (signaled by an underscore) instead of the case for False:

```
case v of { True → b1[x,y]; _ → b2[z] }
```

Once again, we often write examples like this with a vertical layout, eliding the punctuation of semicolons and braces:

```
case v of
  True → b1[x,y]
  _    → b2[z]
```

Each alternative in a case only tests the outermost constructor, C, of the discriminant, v. If the constructor matches but subsequent computation requires access to components of v, then they must be referenced explicitly using C i v selectors, where i is the appropriate field number. The following blocks, for example, can be used to compute the length of a list:

```
length[list] = loop[0,list]
loop[n,list] = case list of
                 Nil  → done[n]
                 Cons → step[n,list]
done[n]      = return n
step[n,list] = assert list Cons
               tail ← Cons 1 list
               m ← add(n,1)
               loop[m,tail]
```

The length block passes the given list to loop, with an initial (accumulating) parameter 0. The loop block uses a case to examine the list. A Nil value represents an empty list, in which case we branch to done and immediately return the current value of n. Otherwise, list must be a Cons, and we jump to step, which uses a Cons 1 list selector to find the tail of the list. After calculating an incremented count in m, step jumps back to examine the rest of the list. The assert here indicates that list is known to be a Cons node; this information ensures that the Cons selector in the next line is valid, and can also be leveraged during optimization.

2.9 Top-level Definitions

MIL programs can use definitions of the form [v₁,...,vₙ] ← t to introduce top-level bindings for variables called v₁,...,vₙ. Variables defined in this way are initialized to the values that are produced by executing the tail expression t. In the common case where a single variable is defined, we omit the brackets on the left of the ← symbol and write v₁ ← t. Note that variables introduced in a definition like this are not the same as *global variables* in an imperative programming language because their values cannot be changed by a subsequent assignment operation.

Top-level definitions like this are typically used to provide names for constants or data structures (including closures, as shown below). For example, the following top-level definitions, each with a data constructor on the right hand side of the ← symbol, will construct a static list data structure with two elements:

```
list1 ← Cons(1, list2)
list2 ← Cons(2, nil)
nil   ← Nil()
```

It is possible to specify types for variables introduced in a top-level definition using declarations of the form v₁, ..., vₙ :: t. For example, we might declare types for the list variables defined previously by writing:

```
list1, list2, nil :: List Word
```

However, there is no requirement to provide types for top-level variables because they can also be inferred in the usual way.

2.10 Closures and Closure Definitions

First-class functions in MIL are represented by *closure* values that are constructed using tail expressions of the form k{a₁,...,aₙ}. Here, a₁,...,aₙ are atoms that will be stored in the closure and accessed when the closure is entered (i.e., applied to an argument) and the code identified by k is executed. For each different k that is used in a given program, there must be a corresponding *closure definition* of the form k{v₁,...,vₙ} [u₁,...,uₘ] = t. Here, the vᵢ are variables representing the values stored in the closure; the uᵢ are variables representing the arguments that are required when the closure is entered; and *t* is a tail expression that may involve any of these variables. Conceptually, each such closure definition corresponds to the code that must be executed when a closure is entered, loading stored values and arguments as necessary in to

registers before branching to the code as described by `t`. Following our usual pattern, in the case where there is just one argument, u_1, the brackets may be omitted and the definition can be written `k{`v_1`,...,`v_n`} `u_1` = t`. The ability to pass multiple (or zero) arguments u_i to a closure, however, is important because it allows us to work with values whose representation may be spread across multiple components; we will return to this in Section 4.2.

As an example, given the definition `k{n} x = add((n,x))`, we can use a closure with code pointer `k` and a stored free variable `n` to represent the function (`λx → n + x`) that will return the value `n+x` whenever it is called with an argument `x`.

The type of any closure can be written in the form:

```
{t1,...,tn} [d1,...,dm] ↠ [r1,...,rp]
```

where the t_i are the types of the stored components, the d_j are the types of the inputs (the domain), and the r_k are the types of the results (the range). This is actually a special case of an *allocator type* `{t1,...,tn} t`, which corresponds to a value of type `t` whose representation stores values of the types listed in the braces. For example, the type of the closure `k` defined above is `{Word} [Word] ↠ [Word]`, while the type of the `Cons` constructor for lists can be written `{a, List a} List a`.

2.11 Example: Implementing map in MIL

In this section, we illustrate how the components of MIL described previously can be used together to provide an implementation for the familiar `map` function. In a standard functional language, we might define this function using the following two equations:

```
map f Nil         = Nil
map f (Cons y ys) = Cons (f y) (map f ys)
```

Using only naive techniques, this function can be implemented by the following MIL code (presented here in two columns) with one top-level, two closure, and three block definitions:

```
map      ← k0{}              b2[f,xs] = assert xs Cons
k0{}  f  = k1{f}                      y  ← Cons 0 xs
k1{f} xs = b0[f,xs]                   ys ← Cons 1 xs
b0[f, xs] = case xs of                z  ← f @ y
              Nil  → b1[]             m  ← map @ f
              Cons → b2[f,xs]         zs ← m @ ys
b1[]     = Nil()                      Cons(z,zs)
```

This implementation starts with a top-level definition for `map`, binding it to a freshly constructed closure `k0{}`. When the latter is entered with an argument `f`, it captures that argument in a new closure structure `k1{f}`. No further work is possible until this second closure is entered with an argument `xs`, which results in a branch to the block `b0`, and a test to determine whether the list value is either a `Nil` or `Cons` node. In the first case, the `map` operation is completed by returning `Nil` in block `b1`. Otherwise, we execute the code in `b2`, extracting the head, `y`, and tail, `ys`, from the argument list. This is followed by three closure entries: the first calculates the value of `f y`, while the second and third calculate `map f ys`, with one closure entry for each argument. The results are then combined using `Cons(z,zs)` to produce the final result for the `map` call.

The MIL definition of `map` reveals concrete implementation details, such as the construction of closures, that are not immediately visible in the original code. For an intermediate language, this is exactly what we need to facilitate optimization, and we will revisit this example in Section 5.4, showing how the code in `b2` can be rewritten to avoid unnecessary construction of closures.

2.12 Notes on Formalization of MIL

Although we do not have space here for many details, we have developed both a formal type system and an abstract machine for MIL. The former has served as a guide in the implementation of the MIL typechecker, which is useful in practice for detecting errors in MIL source programs (and, occasionally, for detecting bugs in our implementation). Perhaps the most interesting detail here is the appearance of a type constructor, M—representing the underlying monad in which MIL computations take place—in rules like the following (for type checking bindings in code sequences):

$$\frac{A \vdash \mathtt{t} : M\,[r_1, \ldots, r_n] \quad A, \mathtt{v}_1 : r_1, \ldots, \mathtt{v}_n : r_n \vdash \mathtt{c} : M\,a}{A \vdash (\mathtt{[v_1,\ldots,v_n]} \leftarrow \mathtt{t;\ c}) : M\,a}$$

An interesting direction for future work is to allow the fixed `M` to be replaced with a type variable, `m`, and to perform a monadic effects analysis by collecting constraints on `m`.

The design of an abstract machine for MIL has also had practical impact, guiding the implementation of a bytecode interpreter that is useful for testing. In the future, the abstract machine may also provide a formal semantics for verifying the program rewrites used in the MIL optimizer (Section 5).

3 COMPILING LC TO MIL

In this section, we discuss the work involved in compiling programs written in LC—a simple, high-level functional language—into MIL. This serves two important practical goals: First, by using MIL as the target language, we demonstrate that it has sufficient features to serve as an intermediate language for a functional language with higher-order functions, pattern matching, and monadic operations. Second, in support of testing, it is easier to write programs in LC and compile them to MIL than to write MIL code directly.[2]

3.1 Translating LC Types to MIL

The task of translating LC types to corresponding MIL types is almost trivial: the two languages have the same set of primitive types and use the same mechanisms for defining new data and bitdata types. The only complication is in dealing with function types in LC, which map a single input to a single result, instead of the tuples of inputs and results that are used in MIL. To bridge this gap, we define the LC function type, written using a conventional infix → symbol, as an algebraic datatype:

```
data d → r = Func ([d] ↠ [r])
```

With this definition, we now have three different notions of function types for MIL: ↠ and → describe first-class function values while >>= is for block types (Section 2.6). The following definitions show how all of these function types can be used together in a MIL implementation of the LC identity function, (`λx → x`):

[2]Our implementation actually accepts combinations of MIL and LC source files as input; this allows users to mix higher-level LC code that is translated automatically to MIL with handwritten MIL code. The latter is useful in practice for implementing libraries of low-level primitives that cannot be expressed directly in LC.

```
k    :: {} [a] ↠ [a]  -- a closure definition for
k{} x = return x      -- the identity function

b  :: [] >>= [a → a]  -- create a closure value and
b[] = c ← k{}         -- package it as a → function
      Func(c)

id :: a → a           -- set top-level name id to the
id ← b[]              -- value produced by b[]
```

A legitimate concern here is that every use of an LC function (→) requires extra steps to wrap or unwrap the `Func` constructor around a MIL function (↠). Fortunately, we will see that it is possible to eliminate these overheads (Section 4.1).

3.2 Translating LC Code to MIL

There is a large body of existing work on compilation of functional languages, much of which can be easily adapted to the task of translating LC source programs in to MIL. As a simple example, to compile a lambda expression like `λx → e` with free variables `fvs`, we just need to: pick a new closure name, `k`; add a definition `k{fvs} x = e'` to the MIL program (where `e'` is compiled code for `e`); and then use `k{fvs}` in place of the original lambda expression. Beyond these general comments, we highlight the following details from our implementation:

- To simplify code generation, we use a lambda lifting transformation [6] to move locally defined recursive definitions to the top-level (possibly with added parameters).
- Our code generator is based on compilation schemes for "compiling with continuations" [8]. As a rough outline (in pseudo-Haskell notation), it can be described as a function:

  ```
  compile :: Expr → (Tail → CM Code) → CM Code
  ```

  The first argument is the LC expression, `exp`, that we want to compile. The second argument is a continuation that takes a MIL tail expression, `t`, corresponding to `exp`, and embeds it in a MIL code sequence for the rest of the program. For instance, if `exp` is the lambda expression `λx → e` in the example at the start of this section, then `t` will be the closure allocator `k{fvs}`. Note that `CM` in the type above represents a "compilation monad", with operations for generating new temporaries, and for adding new block, closure, or top-level definitions to the MIL program as compilation proceeds.
- Continuation based techniques require special care with conditionals like `if c then t else f`). In particular, we need to ensure that the `(Tail → CM Code)` continuation is not applied separately to each of the tail expressions for `t` and `f`, which could lead to duplicated code. Fortunately, it is easy to avoid this in MIL by placing the continuation code in a new block, serving as a "join point" [11], and then having the code in each branch end with a jump to that block.

4 REPRESENTATION TRANSFORMATIONS

Although MIL is more broadly applicable, our current implementation assumes a whole-program compilation model. One benefit of this is that we can consider transformations that require changes across many parts of input programs. In the following subsections, we describe three specific transformations of this kind, all implemented in our toolset, that change the representation of data values in MIL programs. Each of these typically requires modifications in both code and type definitions. For example, if a program uses values of type `T` that can be more efficiently represented as values of type `T'`, then implementing a change of representation will require updates, not only to code that manipulates values of type `T`, but also to any type definitions that mention `T`. Our tools allow these transformations to be applied in any order or combination, running the MIL type checker after each pass as a sanity check (although the original types are not preserved, each transformation is expected to preserve typeability). In addition, because these transformations can create new opportunities for optimization, it is also generally useful to (re)run the MIL optimizer (Section 5) after each pass.

4.1 Eliminating Single Constructor Types

Our first application for representation transformations deals with 'single constructor', non-recursive algebraic datatypes with definitions of the form: `data T` a_1 `...` a_n `= C t'`. Types like this are often used to create wrappers that avoid type confusion errors. For example, by defining `data Time = Millis Word`, we ensure that values of type `Word` are not used accidentally where `Time` values are actually required. And, as discussed in Section 3.1, we also use a type of this form to map between the → and ↠ function types when compiling LC to MIL. These types are useful because they can enforce correct usage in source programs. But for compilation purposes, the extra constructors—like `Millis` and `Func`—add unnecessary runtime overhead, and can block opportunities for optimization. To avoid this, we can rewrite the MIL program to replace every tail of the form `C 0 a` or `C(a)` with `return a` (effectively treating selection and construction as the identity function) and every `case v of C → bc` that 'matches' on `C` with a direct block call `bc`. In addition, the original definition of `T` can be discarded, but every remaining type of the form `T` t_1 `...` t_n must be replaced with the corresponding instance $[t_1/a_1, \ldots, t_n/a_n]$`t'` of `t'`.

4.2 Representation Vectors

Our second application was initially prompted by the use of `bitdata` types in MIL (see Section 2.1) but is also useful with some algebraic datatypes. In the early stages of compilation, we manipulate values of bitdata types like `Bool` or `PCI` with the same pattern matching and constructor mechanisms as algebraic datatypes. At some point, however, the compiler must transition to the bit-level representation that is specified for each of these types. This means, for example, that values of type `Bool` should be represented by values of type `Flag`. Similarly, `PCI` values might be represented using `Word` values, with the associated selectors for `bus`, `dev`, and `fun` replaced by appropriate bit-twiddling logic using shift and mask operations.

To specify representation changes like this, we define a function that maps every MIL type `t` to a suitable *representation vector*: a list of zero or more types that, together, provide a representation for `t`. We use a vector, rather than a single type, to accommodate types that require multi-word representations. A `Bit 64` value, for example, cannot be stored in one 32 bit word, but can be supported by using a representation vector `[Word,Word]` with two `Word` values. Similarly, a zero length vector, `[]`, can be used for `Bit 0` and,

indeed, for any other singleton type, such as the standard unit type, `()`. This reflects the fact that, if a type only has one possible value, then it does not require a runtime representation.

The resulting representation vectors can be used to guide a program transformation that replaces each variable `v` of a type `t` with a list of zero or more variables $v_1, \ldots, v_n$, where `n` is the length of the representation vector for `t`. This was the primary technical motivation for using tuples of values throughout MIL because it allows us to rewrite tails like `f @ a` and top-level definitions like `v ← t`, for example, as `f @ [a1,a2]` or `[v1,v2] ← t` when `a` and `v` are each represented by two words. Of course, additional rewrites are needed for selectors and primitive calls that use values whose representation is changed. A convenient way to manage this is to replace the operations in question with a calls to new blocks that will be inlined and optimized with the surrounding code.

Support for the representation transformation described here is a distinguishing features of our toolset: to our knowledge, no other current system implements a transformation of this kind.

4.3 Specializing Polymorphic Definitions

Our third application, included to satisfy a requirement of the LLVM backend (Section 6), is a transformation that eliminates polymorphic definitions and generates specialized code for each monomorphic instance that is needed in a given program. One problem is that this transformation cannot be used in some programs that rely on polymorphic recursion [13]. Another concern is that specialization has the potential to cause an explosion in code size. In practice, however, specialization has proved to be an effective tool in domain-specific [3] and general-purpose functional language compilers [17], and even in implementations of overloading [7].

Representation transformation is relevant here when we consider the task of generating code for specific monomorphic instances of polymorphic functions. As part of this process, our implementation of specialization also eliminates all uses of parameterized datatypes. A program that uses a value of type `Pair PCI PCI`, for example, might be transformed in to code that uses a value of a new type `data` $\texttt{Pair}_1$ `= MkPair1 PCI PCI`, that is generated by the specializer. This provides an interesting opportunity for using type-specific representations. For example, the $\texttt{Pair}_1$ type described here should hold two 16-bit `PCI` values, so it could easily be represented using a single, 32-bit `Word` with no need for heap allocation. We have already started to explore the possibility of inferring bit-level representations for a wide-range of types like this, and expect to include support for this in a future release of our toolset.

5 OPTIMIZATION

A common goal in the design of an intermediate language is to support optimization: program transformations that preserve program behavior but improve performance, reduce memory usage, etc. For MIL programs, we have identified a collection of rules that describe how some sections of code can be rewritten to obtain better performance. A small but representative set of these rules is presented in Figure 1. The full set has been used to build an optimizer for MIL that works by repeatedly applying rewrites to input programs. Individual rewrites typically have limited impact, but using many rewrites in sequence can yield significant improvements by reducing code size, eliminating unnecessary operations, and replacing expensive operations with more efficient equivalents.

The table in Figure 1 has three columns that provide, for each rule: a short name; a rewrite; and, in several cases, a set of side conditions that must be satisfied in order to use the rule. The rewrites are written in the form `e` $\Longrightarrow$ `e'` indicating that an expression matching `e` should be replaced by the corresponding `e'`. To avoid ambiguity, we use two variants of this notation with $\Longrightarrow_c$ for rewrites on code sequences and $\Longrightarrow_t$ for rewrites on tails. In the rest of this section, we will walk through each of the rewrites in Figure 1 in more detail (Sections 5.1–5.5), describe additional optimizations that are supported by our implementation (Section 5.6), and reflect on the overall effectiveness of our optimizer for MIL (Section 5.7).

5.1 Using the Monad Laws

The first group of rules are by standard laws for monads (as described, for example, by Wadler [16]), but are also recognizable as traditional compiler optimizations. Rule (1), for example, implements a form of *copy* or *constant propagation*, depending on whether the atom `a` is a variable or a constant. The notation `[a/x]c` here represents the substitution of `a` for all free occurrences of `x` in the code sequence `c`; concretely, instead of creating an extra variable, `x`, to hold the value of `a`, we can just use that value directly. Rule (2) corresponds to the right monad law, which can also be seen as eliminating an unnecessary `return` at the end of a code sequence and potentially as creating a new opportunity for a tail call. Finally, Rules (3) and (4) are based on the associativity law for monads, but can also be understood as descriptions of function inlining rules; we use the terms *prefix* and *suffix* to distinguish between cases where the block being inlined appears at the beginning or the end of the code sequence. To better understand the relationship with the associativity law, note that a naive attempt to inline the block `b` in the code sequence `v ← b[x]; c`, using the definition of `b` in the figure, would produce `v ← (`v_0 `←` t_0`;` t_1`); c`. With the previous grammar for MIL code sequences, this is not actually a valid expression. Using associativity, however, it can be flattened/rewritten as the code sequence on the right hand side of the rule. As is always the case, unrestricted use of inlining can lead to an explosion in code size with little or no benefit in performance. To avoid such problems, our implementation uses a typical set of heuristics—limiting inlining to small blocks or to blocks that are only used once, for example—to strike a good balance between the benefits and potential risks of an aggressive inlining strategy.

5.2 Eliminating Unnecessary Code

The second group of rewrites in Figure 1 provide ways to simplify MIL programs by trimming unnecessary code. Rule (5), for example, uses the results of a simple analysis to detect tail expressions `t` that do not return (e.g., because they call a primitive that terminates the program, or enter an infinite loop). In these situations, any code that follows `t` can be deleted without a change in semantics.

Rules (6) and (7) allow us to detect, and then, respectively, to eliminate code that has no effect. Rule (6), sets the result variable name for a statement to underscore to flag situations where the original variable is not used in the following code. A subsequent

Name	Rewrite			Conditions
Using the monad laws, where block b is defined by b[x] = $v_0 \leftarrow t_0$; t_1				
1) Left monad law (constant propagation)	x ← return a; c	$\Longrightarrow_c$	[a/x]c	–
2) Right monad law (tail call introduction)	x ← t; return x	$\Longrightarrow_c$	t	–
3) Prefix inlining	v ← b[x]; c	$\Longrightarrow_c$	$v_0 \leftarrow t_0$; v ← t_1; c	–
4) Suffix inlining	v ← t; b[x]	$\Longrightarrow_c$	v ← t; $v_0 \leftarrow t_0$; t_1	–
Eliminating unnecessary code				
5) Unreachable code elimination	v ← t; c	$\Longrightarrow_c$	t	t does not return
6) Wildcard introduction	v ← t; c	$\Longrightarrow_c$	_ ← t; c	v is not free in c
7) Dead tail elimination	_ ← t; c	$\Longrightarrow_c$	c	t is pure
8) Common subexpression elimination	t	$\Longrightarrow_t$	return v	{v=t}
Using algebraic identities (focusing here on bitwise and and writing M, N, and P for arbitrary integer constants)				
9) Identity laws	and((v,0))	$\Longrightarrow_t$	return 0	–
10) Idempotence	and((v,v))	$\Longrightarrow_t$	return v	–
11) Constant folding	and((M,N))	$\Longrightarrow_t$	return (M&N)	–
12) Commutativity	and((M,v))	$\Longrightarrow_t$	and((v,M))	–
13) Associative folding (u&M)&N = u&(M&N)	and((v,N))	$\Longrightarrow_t$	and((u,M&N))	{v=and((u,M))}
14) Distributive folding (1) (u|M)&N = (u&N)|(M&N)	and((v,N))	$\Longrightarrow_c$	v' ← and((u,N)) or((v',M&N))	{v=or((u,M))}
15) Distributive folding (2) ((u|M)&N)|P = (u&N)|((M&N)|P)	or((w,P))	$\Longrightarrow_c$	v' ← and((u,N)) or((v',(M&N)|P))	{v=or((u,M)), w=and((v,N))}
Known structures, where closure k is defined by k{x} y = t				
16) Known constructor	case v of ...; C → b'[...]; ...	$\Longrightarrow_c$	b'[...]	{c=C(...)}
17) Known closure	f @ y	$\Longrightarrow_t$	t	{f=k{x}}
Derived blocks, where block b is defined by b[x] = $v_0 \leftarrow t_0$; t_1				
18) Known structure	b[v] where b'[y] = x ← C(y); $v_0 \leftarrow t_0$; t_1	$\Longrightarrow_t$	b'[y]	{v=C(y)}
19) Trailing enter	f ← b[x]; f @ a where b'[x,a] = $v_0 \leftarrow t_0$; f ← t_1; f @ a	$\Longrightarrow_c$	b'[x,a]	–
20) Trailing case	v ← b[x]; case v of ... where b'[x,...] = $v_0 \leftarrow t_0$; v ← t_1; case v of ...	$\Longrightarrow_c$	b'[x,...]	–

Figure 1: A representative set of rewrite rules for MIL optimization

use of Rule (7) will then eliminate the statement altogether if the associated tail expression t has no externally visible effects (for example, if t is a call to a pure primitive function like add or mul, or if t is a closure or data allocator). Our presentation of this process using two separate rules reflects the fact that our optimizer actually applies these two rules in separate passes over the abstract syntax.

Rule (8) uses a local dataflow analysis to detect situations where the result of a previous computation can be reused. To implement this, our optimizer calculates a set of "facts", each of which is a statement of the form v=t, as it traverses the statements in each code sequence. Starting with an empty set, the optimizer will add (or "generate") a new fact v=t for every statement v ← t that it encounters with a pure t. At the same time, for each statement v ← t, it will also remove (or "kill") any facts that mention v because the variable that they reference will no longer be in scope. The rightmost column in Figure 1 documents the facts that are required to apply each rewrite. In this case, given v=t, we can avoid recalculating t and just return the value v that it produced previously. (In practice, the return introduced here will often be eliminated later using Rules (1) or (2).)

5.3 Using Algebraic Identities

The next group (Rules (9)–(15)) take advantage of (mostly) well-known algebraic identities to simplify programs using the builtin primitives for arithmetic, logic, and comparison operations. We only show a small subset of the (more than 100) rewrites of this kind that are used in our implementation, all of which involve the bitwise and primitive. In combination with additional rewrites involving bitwise or and shift operations, these rules are very effective in simplifying the bit-twiddling code that is generated when manipulating or constructing Habit-style bitdata types [4, 15].

Rules (9), (10), and (11), for example, each eliminate a use of and in a familiar special case. Rule (12) is not useful as an optimization by itself, but instead is a first step in rewriting expressions in to

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           m  ← map @ f
           zs ← m @ ys
           Cons(z,zs)
```
(a)

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           m  ← k1{f}
           zs ← m @ ys
           Cons(z,zs)
```
(b)

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           m  ← k1{f}
           zs ← b0[f,ys]
           Cons(z,zs)
```
(c)

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           zs ← b0[f,ys]
           Cons(z,zs)
```
(d)

Figure 2: An example illustrating the optimization of known closures (Rule (17))

a canonical form that enables subsequent optimizations. In this case, the rewrite ensures that, for an `and` with one constant and one unknown argument, the constant is always the second argument. This explains, for example, why we do not need a variant of Rule (9) for tails of the form `and((0,v))`, and also why we do not need four distinct variations of the pattern in Rule (13) where an unknown `u` is combined via bitwise `and`s with two constants `M` and `N`. In this case, we rely on facts produced by the dataflow analysis to determine that the value of `v` in `and((v,N))` was calculated using a prior `and((u,M))` call. The rewrite here replaces one `and` with another, so it may not result in an immediate program optimization. However, there are two ways in which this *might* open up opportunities for subsequent rewrites: One possibility is that `M&N` may be zero, in which case we will be able to eliminate the `and` using Rule (9). Another possibility is that, by rewriting the call to `and` in terms of `u`, we may eliminate all references to `v` and can then eliminate the statement defining `v` as dead code using Rules (6) and (7).

In a similar way, Rules (14) and (15) take advantage of standard distributivity laws to implement more complex rewrites. By using these rules in combination, we can rewrite any expression involving bitwise `and`s and `or`s of an unknown `u` with an arbitrary sequence of constants into an expression of the form `(u&M)|N` for some constants `M` and `N`, with exactly one use of each primitive. These rewrites are potentially dangerous because they increase the size of the MIL code, replacing one primitive call on the left of the rewrite with two on the right. In practice, however, these rules often turn the definitions of the variables `v` and `w` that they reference in to dead code that can be eliminated by subsequent rewrites.

5.4 Known Structures

MIL provides `case` and @ constructs that work with arbitrary data values and closures, respectively. But the general operations are not needed in situations where we are working with known structures. Rule (16), for example, eliminates a conditional jump if the constructor, `C`, that was used to build `v` is already known: we can just make a direct jump using the alternative for `C` (or the default branch if there is no such alternative), and delete all other parts of the original `case` construct.

In a similar way, Rule (17) can be used to avoid a general closure entry operation when the specific closure is known. To see how this works in practice, consider the example in Figure 2, with the original code for b_2 in our implementation of `map` (Section 2.11) in Column (a). From the other definitions in this program, we know that `map` is a reference to the closure $k_0\{\}$, and that $k_0\{\}$ `f` $= k_1\{f\}$. By allowing our dataflow analysis to derive the fact `map=`$k_0\{\}$ from its top-level definition, we can use Rule (17) to rewrite the tail defining `m`, as shown in Column (b). After this transformation, the local dataflow analysis will reach the definition of `zs` with a list of facts that includes `m=`$k_1\{f\}$, and so we can apply Rule (17) again to obtain the code in Column (c). This removes the only reference to `m`, and allows subsequent uses of Rules (6) and (7) will eliminate its definition, producing the code definition in Column (d). This example shows that the original implementation of `map` would have allocated a fresh closure, $k_1\{f\}$, for every element of the input list. The transformations we have applied here, however, eliminate this overhead and substitute a more efficient, direct recursive call.

5.5 Derived Blocks

Optimizing compilers often use collections of basic blocks, connected together in control flow graphs, as a representation for programs. Most of the rules that we have described so far are traditionally considered local optimizations because they only consider the code within a single block. While much can be accomplished using local optimizations, it is also useful to take a more global view of the program, and to use optimizations that span multiple blocks. In other words, in addition to the *content* of individual blocks, we would also like to account for the *context* in which they appear.

One interesting way that we have been able to handle this in MIL is by using the code of existing blocks to generate new versions—which we refer to as *derived blocks*—that are specialized for use in a particular context. The final group of rewrites in Figure 1 corresponds to different strategies for generating derived blocks that we have found to be effective in the optimization of MIL programs. Of course, adding new blocks to a program increases program size and does not immediately provide an optimization. But, in practice, the addition of new derived blocks often opens new opportunities for optimization—by bringing the construction and matching of a data value into the same block, for example—and the original source block often becomes dead code that will be removed from the program once any specialized versions have been generated.

Rule (18) illustrates one way of using derived blocks to take advantage of information produced by our local dataflow analysis and to obtain results that typically require a global analysis. The techniques that we describe here can be applied very broadly, but, for this paper, we restrict ourselves to a special case: a call to a block `b` that has just one parameter and a very simple definition.

In this situation, if the argument, `v`, to `b` is known to have been constructed using a tail `C(y)`, then we can replace the call `b[v]` with a call of the form `b'[y]`. Here, `b'` is a new block that begins with a statement that recomputes `v` ← `C(y)` and then proceeds in the same way as the original block `b`. In theory, this could result in an 'optimized' program that actually allocates twice as many `C(y)` objects as necessary; that would obviously not be a good outcome. In practice, however, this transformation often enables subsequent optimizations both in the place where the original `b[v]` call appeared (for example, the statement that initialized `v` may now be dead code) and in the new block, the latter resulting from a new fact, `v=C(y)`, that can be propagated through the code for `b'`. Note that Rule (18) also extends naturally to calls with multiple parameters and to cases where one or more of those parameters is a known closure; in that case the process of generating a new derived block has much the same effect as specializing a higher-order function to a known function argument.

Rules (19) and (20) deal with situations where a block is called and its result is immediately used as a closure or data value, respectively. The code on the left side of these rewrites essentially *forces* the allocation of a closure or data object in `b`, just so that value can be returned and then, most likely, discarded after one use. The right sides deal with this by introducing a tail call in the caller and then turning the use of whatever value is produced in to a 'trailing' action in the new block. As in other examples, this does not produce an immediate optimization. However, these rules generally lead to useful improvements in practice, enabling new optimizations by bringing the construction and use of `v` in to the same context.

5.6 Additional Optimizations

Beyond the rewrites kind described in previous sections, our optimizer implements several other program transformations that help to improve code quality. One of the most important of these in practice—because it also performs a strongly-connected components analysis on the program to prioritize the order in which rewrites are applied—is a "tree-shaking" analysis. This automatically removes definitions from a program if they are not reachable from the program's entry points. In addition to sections of library code that are not used in a given application, this also helps to clean up after other optimizations by eliminating single-use blocks whose definitions have been inlined or blocks that have been replaced with new derived versions. The optimizer also attempts to recognize and merge duplicated definitions, and to rewrite block definitions (and all corresponding uses) to eliminate unused parameters. Unused stored fields in closure definitions can also be eliminated in this way, but we cannot remove arguments in closure definitions: even if they are not used in the code for a particular closure, they must be retained for compatibility with other closures of the same type.

One other detail in our implementation is that we run the MIL type checker after every use of the optimizer. This has two practical benefits: (1) All of our optimizations are required to preserve typing, so running the type checker provides a quick sanity check and may help to detect errors in the optimizer. (2) The type checker will automatically reconstruct type information for each part of the program, so we do not need to deal with those details in the implementations of individual rewrites.

5.7 Reflections on Optimization

The design of an optimizer compiler inevitably requires some judicious compromises. After all, the problem that it is trying to solve—to generate truly optimal versions of any input program—is uncomputable, and so, at some point, it must rely instead on heuristics and incomplete strategies. Even if an optimizer delivers good results on a large set of programs, there is still a possibility that it will perform poorly on others. Subtle interactions between different optimization techniques may prevent the use of key transformations in some situations and instead lead to expanded code size or degraded performance. With those caveats in mind, we have, so far, been very satisfied with the performance of our optimizer for MIL.

As a concrete example, the diagrams in Figure 3 outline the structure of a MIL implementation of the Habit "prioset" example [15, Section 5]. Part (a) here is for the original MIL implementation, generated directly from 56 lines of LC code; it is too small to be readable, but does convey that the original program—910 lines of MIL code—is quite complex. Part (b) shows the result obtained after 1,217 separate rewrite steps in the MIL optimizer, resulting in 140 lines of MIL code. This version of the program still has non-trivial control flow, but its overall structure is much simpler and we can start to see details such as loop structures and a distinction between the blue nodes (representing blocks) and the red nodes (representing closure definitions). By comparison, there are red and blue nodes scattered throughout the diagram in Part (a), which suggests that our optimizations have been effective in eliminating many uses of closures in the original program. (The remaining red nodes in Part (b) are only there because the program exports the definitions of two functions, `insertPriority` and `removePriority` as first-class values; in a program that only uses fully-applied calls to these functions, even those nodes would be eliminated.)

The results that we see with this example are representative of our experience across a range of test programs, and suggest that the MIL optimizer can work will in practice. That said, we plan to do more extensive studies, including performance benchmarking, and to use those results to further tune and refine our implementation. (As a new language, we do not currently have a pre-existing set of benchmarks, but we do hope to grow such a library as our implementation matures and gains users.)

One particularly effective aspect of our implementation that is already clear is the decision to perform optimization at multiple levels throughout the compiler pipeline. An initial use of the MIL optimizer takes care of relatively high-level rewrites, such as inlining of higher-order functions to eliminate the costs of constructing closures. It would be harder to apply this kind of optimization at later stages once the operations for function application and closure construction have been decomposed in to sequences of lower level instructions. A subsequent use of the optimizer, after representation transformations have been applied, enables the compiler to find newly exposed opportunities for optimization, and to further simplify the generated MIL code before it is translated in to LLVM, as described in the next section. Finally, the use of LLVM itself allows us to exploit the considerable effort that has been invested in that platform to perform lower-level optimizations, and—although our focus to date has been on IA32-based systems—also provides a path for targeting other architectures.

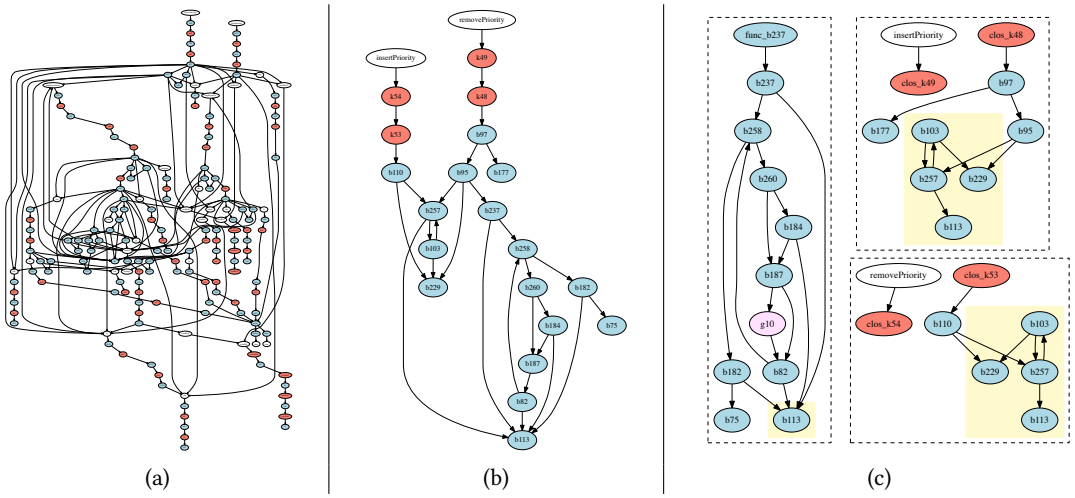


Figure 3: Control flow graph examples.

6 COMPILING MIL TO LLVM

In this section, we explain how MIL programs can be translated in to corresponding LLVM programs, which can then be subjected to further optimization and used to generate executable binaries. Beyond the specific practical role that it serves in our Habit compiler, this also provides another test for MIL's suitability as an intermediate language: it is important, not only that we are able to generate executable programs from our IL, but also that we are able to do so without introducing overhead or undoing any of the improvements that were made as a result of optimizations on the IL.

6.1 Translating MIL Types to LLVM

Before generating LLVM code, we use representation transformations to provide `Word`-based implementations for bitdata types (Section 4.2) and to eliminate polymorphism and parameterized datatypes (Section 4.3). In the resulting programs, MIL types like `Word` and `Flag` are easily mapped to LLVM types such as `i32` and `i1`. The only types that require special attention are for functions (which we cover in this section) and algebraic datatypes (which are handled in a similar manner).

Every MIL value of type $[d_1, \ldots, d_m] \twoheadrightarrow [r_1, \ldots, r_n]$ will be a closure that can be represented by a block of memory that includes a code pointer and provides space, as needed, for stored fields:

entry	···

We can describe structures of this form using three LLVM types with mutually recursive definitions:

```
%clo = type { %fun }
%fun = type {r1, ..., rn} (%ptr, d1, ..., dm)
%ptr = type %clo*
```

Here, `%clo` is a structure type that describes the layout of the closure. Its only component is the code pointer of type `%fun`: no additional components are listed because the number and type of fields is a property of individual closure definitions, not the associated function type. In the generated code, functions of this type will be represented by pointers of type `%ptr`. Given such a pointer, the implementation can read the code pointer from the start of the closure and invoke the function, passing in the closure pointer and the argument values corresponding to the domain types d_i. If that function needs access to stored fields, then it can cast the `%ptr` value to a more specific type that reflects the full layout for that specific type of closure. Finally, the function can return a new structure containing values for each of the range types r_j. (If there is only one result, then it can be returned directly, without a structure; if there are no results at all, then we can use a void function.)

The techniques described here are standard, but there are still many details to account for. Among other things, this reinforces the importance of performing closure optimizations in MIL, rather than generating LLVM code directly and then hoping, unrealistically, that the LLVM tools will be able to detect the same opportunities for improvement. Instead, we divide the responsibilities for optimization between MIL and LLVM, with each part making important contributions to the overall quality of generated code.

6.2 Translating MIL Code to LLVM

The process of translating MIL statements to LLVM instructions is relatively straightforward. As examples: an `and` primitive in MIL maps directly to the (identically named) `and` instruction in LLVM; a closure allocation in MIL is implemented by a call to a runtime library function to allocate space for the closure, followed by a sequence of `store` instructions to initialize its fields; and so on. As such, we will not discuss the fine details of this translation here.

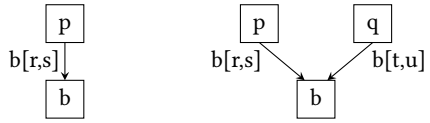
There are, however, some key, higher-level structural mismatches between MIL and LLVM—specifically, parameterization and sharing of blocks—that do need to be addressed. As we have seen, MIL programs are collections of (parameterized) basic blocks that are connected together either by regular block calls or tail calls (in the middle, or at the end, respectively, of a code sequence). By contrast, LLVM programs consist of a collection of (parameterized) functions, each of which has a *control flow graph* (CFG) comprising a single (parameterized) entry point, and a body that is made up from a

collection of (unparameterized) basic blocks. How then should we approach the translation of an arbitrary MIL programs into LLVM?

One approach would be to generate an separate LLVM function for each MIL block (and closure definition). Although LLVM does provide support for tail calls, those features are difficult to use and it might be difficult to ensure that the compiled loops, encoded as tail recursive blocks in MIL, run in constant space.

Our strategy instead is to compile mutually tail recursive blocks directly into loops, accepting that there will be some (small) duplication of blocks in the process. In Figure 3(b), for example, there are several blocks that are reachable from either of the two distinct entry points at the top of the diagram. Our code generator uses some simple heuristics to generate a set of LLVM CFG structures from MIL programs with the following properties: (1) There must be a distinct CFG for every closure definition and for every block that is a program entry point or the target of a non-tail call; (2) If one block is included in a CFG, then all other blocks in the same strongly connected component (SCC) should also be included in the same CFG (this ensures that tail calls can be compiled to jumps); (3) If a single block has multiple entry points from outside its SCC, then it is a candidate entry point for a new CFG (this attempts to reduce duplication of blocks). The result of applying our algorithm to this particular example is shown in Figure 3(c) and is typical of the behavior we see in general: there is some duplication of blocks (the portions highlighted with a yellow background) but the amount of duplicated code is small and has not been a concern in practice.

Our second challenge is in dealing with the mapping from parameterized blocks in MIL to unparameterized blocks in LLVM. It turns out that the number of predecessors is key in determining how to generate code for the body of each block. To understand this, consider the following two diagrams:



In both diagrams, we assume a block `b` defined by `b[x,y] = c` for some code sequence `c`. For the diagram on the left, there is exactly one predecessor, `p`, which ends with a call to `b[r,s]`. In this situation, there is actually no need for the parameters to `b` because we already know what values they will take at the only point where `b` is called. All that it needed is to apply a substitution, replacing the formal parameters, `x` and `y`, with the actual parameters `r` and `s`, respectively, so that we use the code sequence `[r/x, s/y]c` for the body of `b`. (Of course, we also need to account for this substitution on any edges from `b` to its successors.) For the diagram on the right, there are two predecessors, each of which ends by calling `b` with (potentially distinct) parameters. In this case, the phi functions that are part of LLVM's SSA representation provide exactly the functionality that we need to 'merge' the incoming parameters and we can generate code of the following form for `b`:

```
x = phi [r,p], [t,q]
y = phi [s,p], [u,q]
... LLVM code for c goes here ...
```

Generating code in SSA form is sometimes considered to be a tricky or complicated step in the construction of an optimizing compiler. It is fortunate that the translation is relatively straightforward and that we are able to take advantage of phi functions—a key characteristic of the SSA form—quite so directly.

7 CONCLUSIONS

We have described the MIL language and toolset, demonstrating (1) that it has the fundamental characteristics needed to qualify as an effective intermediate language for the compilation of functional languages; and (2) that it can enable new techniques for choosing efficient data representations. The tools are available from our source code repository at https://github.com/habit-lang/mil-tools. The design and implementation of any new intermediate language requires considerable engineering effort. As we continue to develop the MIL system ourselves—for example, to explore its potential for compilation of lazy languages—we hope that it will also serve as useful infrastructure for other functional language implementors.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their helpful feedback. This work was supported in part by funding from the National Science Foundation, Award No. CNS-1422979.

REFERENCES

[1] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA.
[2] Justin Bailey. 2012. *Using Dataflow Optimization Techniques with a Monadic Intermediate Language.* Master's thesis. Department of Computer Science, Portland State University, Portland, OR.
[3] Adam Chlipala. 2015. An Optimizing Compiler for a Purely Functional Web-application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015).* ACM, New York, NY, USA.
[4] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. High-level views on low-level representations. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005.* ACM, 168–179.
[5] Matthew Fluet and Stephen Weeks. 2001. Contification Using Dominators. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01).* ACM, New York, NY, USA, 2–13.
[6] Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, 201).* Springer-Verlag, 190–203.
[7] Mark P. Jones. 1994. Dictionary-free Overloading by Partial Evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94).*
[8] Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07).* ACM, New York, NY, USA, 177–190.
[9] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[10] LLVM 2018. The LLVM Compiler Infrastructure. http://llvm.org.
[11] Luke Maurer, Zena Ariola, Paul Downen, and Simon Peyton Jones. 2017. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17).* ACM, 482–494.
[12] E. Moggi. 1989. Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science.* IEEE Press, Piscataway, NJ, USA, 14–23.
[13] Alan Mycroft. 1984. Polymorphic Type Schemes and Recursive Definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming.* Springer-Verlag, London, UK, UK, 217–228.
[14] Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press.
[15] The Hasp Project. 2010. The Habit Programming Language: The Revised Preliminary Report. http://github.com/habit-lang/language-report.
[16] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90).* 61–78.
[17] Stephen Weeks. 2006. Whole-program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML (ML '06).* ACM, New York, NY, USA.

Task Oriented Programming and the Internet of Things

Mart Lubbers
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
mart@cs.ru.nl

Pieter Koopman
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
pieter@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

In the omnipresent Internet of Things (IoT), tiny devices sense and alter the environment, process information and communicate with the world. These devices have limited amounts of processing power and memory. This imposes severe restrictions on their software and communication protocols. As a result, applications are composed of parts written in various programming languages that communicate in many different ways. This impedance mismatch hampers development and maintenance.

In previous work we have shown how an IoT device can be programmed by defining an embedded Domain Specific Language (eDSL). This paper shows how IoT tasks can be seemlessly integrated with a Task Oriented Programming (TOP) server such as iTasks. It allows the specification on a high level of abstraction of arbitrary collaborations between human beings, large systems, and now also IoT devices. The implementation is made in three steps. First, there is an interface to connect devices dynamically to an iTasks server using various communication protocols. Next, we solve the communication problem between IoT devices and the server by porting Shared Data Sources (SDSs) from TOP. As a result, data can be shared, viewed and updated from the server or IoT device. Finally, we crack the maintenance problem by switching from generating fixed code for the IoT devices to dynamically shipping code. It makes it possible to run multiple tasks on an IoT device and to decide at runtime what tasks that should be.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Software and its engineering → Client-server architectures**; **Functional languages**; **Domain specific languages**.

KEYWORDS

Internet of Things, Functional Programming, Distributed Applications, Task Oriented Programming, Clean

IFL'18, August 2019, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310239

ACM Reference Format:
Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Task Oriented Programming and the Internet of Things. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310239

1 INTRODUCTION

1.1 Internet of Things (IoT)

The term IoT stands for a whole network of (smart) devices that interact with each other and — most of all — interact with the world. IoT is booming, Gartner estimated that there will be around 21 billion IoT devices online in 2020[1]. IoT devices are already entering our households in the form of smart electricity meters, thermostats, weather stations, door locks and so on. IoT technology is emerging rapidly and is transforming the way people interact with technology and with each other.

There are typically severe limitations on the processing power of IoT devices. Because they must be cheap, very tiny computers are used with limited memory, e.g. Microcontroller Units (MCUs). The programs are usually stored in flash memory that only withstands a fairly limited number of write cycles. IoT devices communicate with the internet to share information such as sensor data and to act on demand with actuators all while using as little power, bandwidth, and memory as possible [Da Xu et al. 2014].

IoT not only encompasses the devices but all components of the system including the server, devices, and communication. There is an impedance mismatch between these which leads to isolated logic to integration problems. Every component has to be programmed separately in different languages and on different platforms resulting in high update roll-out costs. For example, rolling out updates in a device is relatively expensive since reprogramming MCUs in the field often requires physical access, while updating an app on the server is as simple as deploying an updated app.

Reprogramming devices automatically is beneficial when devices are often reprogrammed. It allows the creation of dynamic systems in which programs can be moved on demand between devices; e.g. in case of a failing device. In a compiled setting, deploying a different program on a device requires a complete reprogramming.

Interpretation can mitigate this limitation but comes with downsides as well. Sending serialized general purpose programs causes a big communication overhead and they have to be stored in the already scarce memory.

[1] Gartner (November 2015)

1.2 Task Oriented Programming (TOP)

The TOP paradigm and the corresponding iTasks implementation offer a high abstraction level for defining real world workflow tasks [Plasmeijer et al. 2007]. These tasks are described in an eDSL hosted in the purely functional programming language Clean [Brus et al. 1987; Plasmeijer et al. 2011]. Tasks are the basic building blocks of the language; they resemble actual work that needs to be done. The language contains combinators — arising from workflow modelling — to combine tasks in a sequential, parallel or conditional way. The iTasks system generates a multi-user web application to coordinate the tasks that the end users and the computer systems have to do in collaboration.

The iTasks eDSL is type-driven and built on generic functions that are created on the fly for the given types. These generic functions provide the basic TOP functionality which means the programmer has to do little to no implementation work on details such as the user interface. If needed, functions can be specialized to offer fine-grained control over this generated functionality.

1.3 Integrating IoT Devices with TOP

With TOP, one can describe arbitrary complex distributed collaborations between end users and systems without the need to worry about the technical details. In this paper we show how to program all layers of IoT from one single source and thus incorporate IoT devices seamlessly in TOP/iTasks. Adding IoT devices to the current iTasks system is difficult as it was not designed to cope with devices which are that tiny and that restricted in their communication bandwidth.

A natural way of adding clients to a server in iTasks is to use the distributed extension. Oortgiese et al. lifted iTasks from a single server model to a distributed server architecture[Oortgiese et al. 2017]. For example, Android apps can be created that run an entire iTasks core and are able to receive tasks from a different server and execute them. Although their system is suitable for dynamically sending tasks over platforms with different types of processors, their solution cannot be ported to MCUs because they are simply not powerful enough to run or store an iTasks core. Devices that run Android are still a lot more powerful than the typical IoT MCU. Moreover, sending serialized iTasks tasks over an Low Power Low Throughput Network (LTN) requires too much bandwidth.

1.4 Research Contribution

In this paper we present a novel way of controlling IoT devices in TOP/iTasks using restricted tasks for IoT devices and special interfaces to SDSs. It presents the following research contributions.

(1) Extensions for the mTask-eDSL by Koopman and Plasmeijer [2016] are given to create a language for describing imperative IoT tasks. (2) The novel backend for the eDSL generates specialized bytecode programs. (3) A Runtime System (RTS) is shown for the devices that can dynamically receive and execute this bytecode so that can be repurposed without reprogramming. The RTS is modular just as the eDSL and easily portable. (4) A method for integrating the devices with a TOP server is shown by giving an implementation in iTasks. With this glue, IoT tasks can be executed as if they were regular TOP tasks and communication with these tasks is transparently achieved via SDSs. Device and communication specific information is hidden for the programmer and user.

1.5 Structure of this Paper

In Section 2, the basic concepts of TOP as offered by the iTasks system are introduced together with an IoT application that is used as a running example. Section 3 explains the eDSL techniques and the actual eDSL used to express IoT tasks. The glue needed for the interaction between iTasks and IoT devices is discussed in Section 4. Moreover, the example from Section 2 is finished to illustrate the process of building IoT applications. Section 5 shows the bytecode compilation backend for the eDSL to dynamically generate code that can be executed on IoT devices. Section 6 covers the details of the run-time system for the IoT devices. In Section 7, the server-side implementation and integration is discussed, Section 8 describes related work and Sections 9 and 10 conclude with the conclusion and discussion.

2 BRIEF OVERVIEW OF TOP

Here we present a brief overview of the main concepts of TOP. The details are specific to the TOP implementation iTasks. More detailed information can be found in [Plasmeijer et al. 2012].

2.1 Tasks

A task is a statefull event processor that returns a value of type `:: TaskValue a = NoValue | Value a Bool` in which the `Bool` represents the stability. A `TaskValue` is special since it may change over time because its event handling function is re-evaluated on every event. Once a value is `Stable`, it does not change again. A `TaskValue` can be observed by other tasks and it can affect which other tasks are to be started. There are basic tasks and combinators to compose tasks in familiar workflow patterns.

Basic tasks come in two flavours: interactive and non-interactive. Non-interactive basic tasks consist of processing, task value manipulation, external connections, and communication with the host system.

Interactive tasks provide interaction with a user via a type driven generated web interface. A type used in iTasks must have instances for a collection of generic functions that is captured in the class `iTask`[2]. One of these generic functions is the generic web form generation that allows the user to edit a value of that type using the web browser. Basic types have specialization instances for these generic functions and they can be derived for any first-order user-defined type. When desired, derived interfaces can be fine-tuned or specialized instances can be defined.

The main interactive tasks for entering, viewing and updating values of arbitrary types are called the `*Information` tasks and shown below. These tasks spawn a web form for the user to the work with. The first argument is the title, the second argument contains the display options on the data and the third argument of the `view` and `update` variant are the initial value.

```
enterInformation :: String [EnterOption m ]   → Task m | iTask m
viewInformation  :: String [ViewOption  m ] m → Task m | iTask m
```

[2]In Clean, class constraints on overloaded functions are placed after the signature denoted by a bar, separated by ampersands

```
updateInformation :: String [UpdateOption m m] m → Task m | iTask m
```

2.2 Task Combinators

Tasks can be combined with task combinators to express sequential, parallel and conditional workflows. With these combinators, one can define how tasks depend on each other and how the information is passed between them. The resulting combination delivers a new task. Some of the — for this paper — relevant combinators are explained below.

The parallel combinators combine two or more tasks in such a way that they are offered to the user at the same time, possibly combining the result. For example the -|| emits the first task with a value and stabilizes when either one of the task has a stable value. The -&&- emits a task value only when both sides have a value and stabilizes only when both sides are stable. Specialized versions of the -|| exist that executes the two tasks in parallel but only regards the value of one side.

```
(-||-)  infixr 3 :: (Task a) (Task a) → Task a       | iTask a
(-&&-)  infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a & iTask b
(-||)   infixr 3 :: (Task a) (Task b) → Task a       | iTask a & iTask b
(||-)   infixr 3 :: (Task a) (Task b) → Task b       | iTask a & iTask b
allTasks         :: [Task a] → Task [a]              | iTask a
```

Instead of running tasks at the same time, sequential task combinators compose tasks sequentially. The value of the left-hand side is fed to the right-hand side if one of the `TaskCont` predicates hold. These predicates can be based on the stability of the value, the actual value, an action or an exception. Actions are presented to the user as buttons in the generated web form. Exceptions are thrown by tasks and can be caught using the `try` construction. The bind combinator (>>=) is implemented as a step that continues only when either the left-hand side is stable or the user presses the continue button.

```
(>>=) infixl 1 :: (Task a) (a → Task b)          → Task b | iTask a & iTask b
(>>|) infixl 1 :: (Task a) (Task b)              → Task b | iTask a & iTask b
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)] → Task b | iTask a & iTask b

:: TaskCont a b
    =     OnValue         ((TaskValue a) → Maybe b)
    |     OnAction String ((TaskValue a) → Maybe b)
    | ∃e: OnException     (e → b)                 & iTask e

try :: (Task a) (e → Task a) → Task a | iTask a & iTask e & toString e
```

2.3 Shared Data Sources

Non sequential data sharing happens in TOP via SDSs. SDSs are solely defined by their stateful read and write functions and are an abstraction on data in the broadest sense. For example, an SDS can be a file on disk, the system time, a place in memory, lenses on other SDSs, or an external database. There is a publish-subscribe system attached to SDSs which means that a task reading an SDS is automatically notified when the value has changed. This results in low resource usage because tasks do not need to poll. Lenses on SDSs can be used to map functions, combine multiple SDSs, or apply data or notification filters. An SDS is typed by three types; the read, the write type and the parametric lens type. The parametric lens is ignored for now and is fixed to `()` in all access tasks anyway. However, different parameter types are later used (see Section 7.4).

There are four atomic tasks to interact with SDSs. The `get` function retrieves the value, the `set` value sets the value and the `upd` changes the value. Finally the `watch` function is a task that constantly returns the value of the SDS when it is changed.

```
:: SDS p r w
:: Shared a :== SDS () a a

get   ::          (SDS () r w) → Task r | iTask r
set   :: w        (SDS () r w) → Task w | iTask w
upd   :: (r → w) (SDS () r w) → Task w | iTask r & iTask w
watch ::          (SDS () r w) → Task r | iTask r
```

Moreover, the interactive `*Information` (see Section 2.1) tasks are available for SDSs as well. In this way, one can interact with shared data using the generated web forms. Below are the type signatures for these functions. The view on the SDS data in the web page is automatically updated when the SDS is updated. These functions support the same lenses as their counterparts.

```
viewSharedInformation   :: String [...] (SDS r w) → Task r | iTask r
updateSharedInformation :: String [...] (SDS a a) → Task a | iTask a
```

There are some extra functions available in the realm of SDSs that need some introduction. The >*< operator combines two SDSs. The `mapRead` function embeds a transformation function atomically in the given SDS. Derived from the `watch` function is the `whileUnchanged` function that — given an SDS and a task — executes the task every time the SDS value changes.

A way to create an SDS is by using the `withShared` function. This function creates a memory mapped SDS that is only available within scope.

```
(>*<) infixl 6 :: (SDS () rx wx) (SDS () ry wy) → SDS () (rx, ry) (wx, wy)
mapRead        :: (SDS () r w) (r → r') → (SDS r' w)
whileUnchanged :: (SDS () r w) (r → Task b) → Task b | iTask b
withShared     :: a ((SDS () a a) → Task b) → Task b | iTask b
```

2.4 Thermostat Example: iTasks

As an illustrative running example we introduce a thermostat application written in iTasks.

The user can set the limits through the generated web interface. If the temperature drops below the lower limit, the heater turns on. If it rises over the upper limit, the fan turns on.

The iTasks system provides the interaction with the data sources to the user. All of the communication is going through these data sources which are represented by SDSs. An SDS is created for the current temperature, upper and lower limit. Moreover, virtual SDSs define the on/off state of the heater and the cooler. The current temperature and the corresponding limit are combined to an SDS lens yielding a `Bool` defining the limit status.

Now suppose we are also able to execute tasks on an IoT device and suppose we have all the actual device interaction captured in the `iotThermostat` task (Section 4.3). Then we can program a thermostat as follows.

```
1  thermostat :: Task Int
2  thermostat =
```

```
3       withShared 0  (λcurrentTemp →
4       withShared 18 (λlowerTarget →
5       withShared 22 (λupperTarget →
6       let coolerSDS = mapRead (uncurry (>)) (currentTemp>*<lowerTarget)
7           heaterSDS = mapRead (uncurry (<)) (currentTemp>*<upperTarget)
8       in viewSharedInformation   "Current"     [ViewAs viewAsCelcius] currentTemp
9       -|| updateSharedInformation "Lower limit" [] lowerTarget
10      -|| updateSharedInformation "Upper limit" [] upperTarget
11      -|| viewSharedInformation   "Cooler"      [] coolerSDS
12      -|| viewSharedInformation   "Heater"      [] heaterSDS
13      -|| iotThermostat currentTemp cooler heater)))
14  where
15      viewAsCelcius s = toString s +++ "[+½degree+]C"
```

Lines 3-5 instantiates SDSs in memory representing the current temperature and the limits. The initial value for the current temperature (type `Shared Int`) is set to 0 and we will discuss later how it is updated. The initial temperature limits are 18 and 22 degrees and are both represented by a `Shared Int`. All communication between the tasks goes via these SDSs.

Lines 6-7 contain lenses on SDSs to create virtual SDSs representing the status for the heater and the cooler. The current temperature and limit are combined using a comparison operator.

Lines 8-12 contain the tasks that generate the user interface. The interface depends on the type of the SDS. Hence, it is used to change the target temperatures, view the current temperature and view the status of the cooler and the heater. The view option from line 15 makes sure the temperature — a plain `Int` — is decorated with a unit.

Line 13 contains the IoT logic which is defined in Section 4.3. The compound task uses the SDSs to operate the thermostat. It creates an IoT task, compiles it to bytecode and executes this code on the device. The device measures the temperature and controls the cooler and the heater and communicate via the SDSs.

This code — modulo some aesthetic options — results in the interface in Figure 1. Every change in the IoT system — server *or* device — is automatically propagated to tasks watching it. This results in an interface that automatically updates when for example the temperature on the device updates. Moreover, if the user modifies one of the limits, this is automatically propagated to the device so that the IoT tasks can respond to it.

Figure 1: Thermostat user interface

3 AN eDSL FOR IoT TASKS

Regular iTasks tasks are not suitable to run on small devices because of their resource usage. However, a subset of the TOP tasks are natural to the IoT domain and we still want to express them in a type safe and extendible way. EDSLs offer a solution for creating new languages in a host language while benefiting from properties of the host language such as the type system.

3.1 Class-Based Shallow Embedding

There are several basic embedding techniques, such as shallow and deep embedding [Gibbons 2015]. Class-based shallow embedding — or tagless embedding — has the advantages of both shallow and deep embedding [Carette et al. 2009; Svenningsson and Axelsson 2012]. Here, the language constructs are defined as type classes and a backend is a type with an instance for some of the classes. This means that adding backends is easy and a backend only needs to implement the classes it needs. Moreover, type safety is guaranteed 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 backends do not need to be updated when an extension is added in a new class. Naturally, if the extension is added in an existing class, the backends implementing the class need to be updated.

3.2 IoT EDSL

The mTask eDSL is a class-based shallowly eDSL hosted in Clean [Koopman and Plasmeijer 2016]. Their backend generates C-code for complete TOP-like programs that can run on an Arduino. The language itself is imperative of nature and programs written in it are suited to run on an MCU. However, this backend generates a self-contained system and is not suitable for our purpose because there is no connection whatsoever with the regular iTasks system.

In this paper, the mTask eDSL is extended with a new bytecode generation backend and language extensions that allow run-time assignment of tiny IoT tasks to IoT devices as well as integration of these IoT tasks with regular iTasks tasks. To avoid confusion, the extended mTask eDSL with the novel backend is called the IoT eDSL.

The IoT eDSL is a collection of classes implementable by types with two type variables. The type implementing the classes is called the backend (`b`). The first type variable (`t`) represents the type of the construction and the second type variable (`r`) the role of the construction. Type constraints are used to make sure the expressions are well typed and to disallow expressions like: `lit True +. lit 1`. Roles can be `Expr`, `Stmt` and `Upd` to denote expressions, statements and updatables. The roles form a hierarchy that is expressed in class constraints, for example, an updatable can be used as an expression but not the other way around. This type and class definitions for this hierarchy follows.

```
:: Expr = Expr
:: Stmt = Stmt
:: Upd  = Upd

class isExpr a :: a
instance isExpr Upd, Expr
class isStmt a :: a
instance isStmt Upd, Expr, Stmt
```

The constructions in the IoT language can be grouped by roles and extra categories for device access and SDS operations. The tasks are small imperative programs that are executed continuously by the RTS and therefore there is no need for loop control. Therefore we only need conditional and sequential statements.

3.2.1 Expressions. There are two classes of expressions, namely boolean expressions and arithmetic expressions. The class of arithmetic language constructs also contains the function `lit` that lifts a host language value into the IoT eDSL domain. All operators are suffixed with a full stop to avoid have name clashes with Clean's builtin operators. All standard arithmetic functions are included in the eDSL, but some are omitted for brevity.

```
class arith b where
    lit          :: t              → b t Expr
    (+.) infixl 6 :: (b t r) (b t q) → b t Expr   | + t & isExpr r & isExpr q
    ...
```

3.2.2 Statements. Both the sequence operator (`:.`) and the conditional (`IF`, `?`) statements are shown below. The `?` is a variant of the standard conditional operation where the else clause is empty.

```
class IF b where
    IF          :: (b Bool p) (b t q) (b s r) → v () Stmt | isExpr p
    (?) infix 1 :: (b Bool r) (b t q)         → v () t    | isExpr p
class seq b where
    (:.) infixr 0 :: (b t r) (b u q)          → b u Stmt
```

3.2.3 Assignables. The IoT eDSL offers an imperative language and therefore an assignment construction is very natural. Only constructs with the `Upd` role can be assigned to. Examples of constructs with the `Upd` role are variables and General Purpose Input/Output (GPIO) pins. Variables — and other decorations — can only be defined at the top level. The `Main` type statically ensures this. The type signature is complex; to illustrate the usage, an implementation example for a variable written to an analog pin is given below.

```
:: In   a b = In infix 0 a b
:: Main a   = {main :: a}
:: DigitalPin = D0 | D1 | D2 | D3 | D4 | D5 | ...

class dIO b :: DigitalPin → b Bool Upd
class var b :: ((b t Upd) → In t (Main (b c r))) → (Main (b c r)) | ...
class assign b where
    (=.) infixr 2 :: (b t Upd) (b t r) → b t Stmt | isExpr r

writeAnalog :: Main (b Bool Stmt)
writeAnalog = var λx=True In {main = dIO D3 =. x}
```

4 THE GLUE BETWEEN TOP AND IoT TASKS

With a language to express IoT tasks we still need glue to actually put the bytecode on the device at run-time for execution as well as a way of integrating them with a TOP server. From the iTasks program, IoT tasks can be executed on a device transparently as if it were iTasks tasks. The IoT tasks and iTasks tasks can communicate via SDSs that are synchronized between the device and the server. The glue functions and tasks to achieve this can be divided into three categories, namely *devices*, *tasks* and *SDSs*.

4.1 Glue Functions and Tasks

We introduce the `withDevice` task that is needed to interact with an IoT device. This task — given a specification — connects to a device and sets it up for usage with the iTasks system. The task only requires a specification that implements the `Duplex` class. The `Duplex` class' only member is the iTasks task that, given the specification, synchronizes the communication channels. Implementations of this class have been made for TCP and Serial devices. When the device is connected, the further interaction is communication agnostic. The `Device` is an abstract type representing an IoT device and passed to functions interacting with devices. It can be seen as a reference to the device and IoT/iTasks programmers should not use the structure directly. If the programmer wants to use another type of device or communication method, they only need to change the value they pass to `withDevice`.

```
withDevice :: a (Device → Task b) → Task b | Duplex a & iTask a & iTask b
class Duplex a where synFun :: a Device → Task ()
```

IoT tasks that are expressed in the IoT eDSL have to be compiled to bytecode first. Next, they are sent to the device for execution. All of this is captured in the `liftIOTTask` function. This function compiles the IoT task, sends it to the device and handles the communication. When the IoT task terminates, the lifted task becomes stable. In TOP, there is no hard limit in the number of tasks that are assigned to the same system. In analogy, multiple IoT tasks that are sent to the same device, are executed after each other. The scheduling of the IoT tasks is done by the RTS on the device, hence the lack of looping functionality in the IoT tasks. An IoT task is always accompanied by a scheduling strategy that is either a oneshot execution or a repeated execution (see Section 4.2).

```
:: Interval = OneShot | OnInterval Int
liftIOTTask :: (Device, Interval) (Main (ByteCode a Stmt)) → Task ()
```

All interaction of the iTasks system with the running IoT tasks happens via SDSs. To accommodate this, a class has been added that looks similar to the `var` class that allows iTasks SDS to be used in IoT tasks. SDSs in iTasks are automatically published to all readers when it is written. Applying this strategy to IoT SDSs could cause a large communication overhead. To mitigate this overhead, a lifted SDS that is written on the server is passed on to the device immediately, but a device writing an SDS needs to publish this explicitly using the added `pub` class.

```
class sds v :: ((v t Upd)→In (SDS t t) (Main (v c s))) → (Main (v c s)) | ...
class pub v :: (v t Upd) → v t Stmt
```

4.2 Scheduling

Tasks sent to an IoT device are accompanied by a scheduling strategy. With this strategy they behave like TOP tasks in the sense that they are continuously executed and their values can be observed, albeit through SDSs. Two scheduling strategies are available for different types of workflow.

The `OneShot` strategy can be used to execute a 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, a thermostat might read the temperature every hour but, the user might want to know the temperature at an exact moment. Then they can just send a `OneShot` task probing the temperature. If the temperature sensor is connected to GPIO analog pin 7, such a task looks like

```
sharePin :: Device (Shared Pin) → Main (ByteCode () Stmt)
sharePin dev someShare
    = liftIOTTask (dev, OneShot)
        (sds λpin=someShare In {main = pin =. aIO A7 :. pub pin})
```

Secondly, tasks accompanied with the `OnInterval Int` strategy are executed every given number of milliseconds. This strategy most closely resembles tasks as in the iTasks system and fits the use case of periodic measurements. The task shown previously can be used to measure the temperature constantly. Moreover, the strategy can be (ab)used to simulate recursion because variables and SDS are kept between executions on the device. The repeated execution can be terminated with a `return`. The following example shows this with the factorial function.

```
IOTFac :: Device Int → Main (ByteCode () Stmt)
IOTFac dev n
    = liftIOTTask (dev, OnInterval 500)
        (var λresult=1 In var λy=n In {main =
            IF (y ==. 0) return (result =. y *. result :. y =. y -. lit 1)})
```

4.3 Thermostat Example: IoT

Now that we have provided all the tooling, we can finish the example from Section 2.4. It is possible to program the thermostat with only a single task and a single device but to make the example a bit more interesting, we divided the work over two devices: `sensorDevice` and `coolerDevice`.

The first device — connected through TCP — measures the temperature and operates the heater. The temperature is read from analog GPIO pin `A0` and written in the `currentTemp` SDS. The heater is connected to digital GPIO pin `D1` and is set according to the value in the `heater` SDS.

The second device — connected via a serial connection — operates the cooling fan. The cooler is connected to digital GPIO pin `D5` and the state of the cooler is read from the `cooler` SDS. The sensor is executed every 500 milliseconds and the cooler and heater IoT tasks every 1000 milliseconds.

```
1  iotThermostat :: (Shared Int) (Shared Bool) (Shared Bool) → Task ()
2  iotThermostat currentTemp cooler heater = readTempHeat -||- operateCooler
3  where
4      sensorDevice :: TCPSettings
5      sensorDevice = {host="192.168.0.12", port=8888}
6
7      coolerDevice :: TTYSettings
8      coolerDevice = {devicePath="/dev/ttyUSB0", baudrate=B9600, ...}
9
10     readTempHeat :: Task ()
11     readTempHeat
12         = withDevice sensorDevice λsensor→
13             liftIOTTask (sensor, OnInterval 500)
14                 (sds λx=currentTemp In {main= x =. analogRead A0 :. pub x})
15         -||- liftIOTTask (sensor, OnInterval 1000)
16                 (sds λf=heater      In {main=dIO D1 =. f})
17
18     operateCooler :: Task ()
19     operateCooler
20         = withDevice coolerDevice λcoolerOper→
21             liftIOTTask (coolerOper, OnInterval 1000)
22                 (sds λf=cooler In {main= dIO LED1 =. f :. dIO D5 f})
```

Line 2 is the parallel combination of the two tasks representing the work that needs to be done on each device. Lines 4-8 give the specification for the devices.

Lines 10-16 show the work that is done on the TCP device. First the device is instantiated with the `withDevice` task. Then, two IoT tasks are sent to the device that run in parallel. The first IoT task reads and publishes the current temperature and the second operates the heater.

Lines 18-22 describe the work that is done on the cooler device. The IoT task operates the cooler through GPIO pin `D5` and shows the status on `LED1`.

5 COMPILING IoT TASKS

Sending an IoT task to a device is a multi step process under the hood.

First, the class functions from the IoT eDSL are implemented for the `ByteCode` type. This type is a boxed Reader Writer State Transformer (RWST) [Jones 1995] that transforms a compiler state while writing bytecode instructions when evaluated.

```
:: ByteCode t r = BC (RWS () [BC] BCState ())
```

Secondly, the RWST generates the appropriate bytecode but it also keeps track of the used SDSs and variables in the state. An IoT task is not only defined by its bytecode but also by its SDSs and variables that are stored in the state together with fresh identifier streams. The state is kept between compilations to not have common identifiers between tasks.

Finally, all the aforementioned data must be converted to messages that the device can understand. To keep the communication overhead small and the execution fast, the bytecode is assembled. The assembly consists of converting the instructions to bytes and resolving the labels. A device receives two types of messages for an IoT task. First, it receives the specification for all the SDSs and variables. Then, it receives the task containing the bytecode.

5.1 Instruction Set and Representation

The instruction set is defined by the `BC` type and contains basic instructions for a stack machine such as stack operations, labels, jumping and arithmetics. IoT specific instructions are included to allow interaction with peripherals. Moreover, it contains instructions for variables and SDSs (prefixed with `BCSds`). There is no typing on the instruction level, all types are boxed in the `BCValue` type. This box can contain any type for which the `IOTType` class is defined. The context restriction contains all functions needed to interact with the values (e.g. serialization and de-serialization) to send them to the device and the `iTask` class to interact with them via a web interface. Instances for the serialization classes are given for the basic types, IoT specific types — e.g. GPIO pins and LEDs — and for the box type itself.

```
:: BC
    = BCLab Int      | BCJmp Int      | BCJmpT Int | BCJmpF Int
    | BCPop          | BCPush BCValue
    | BCAdd          | BCMult         | ...
    | BCSdsStore Int | BCSdsFetch Int | BCSdsPub Int
    | BCAnalogRead AnalogPin | ...
:: BCValue =∃e: BCValue e & IOTType e
```

```
class toByteCode   a :: a → String
class fromByteCode a :: String → (Either String (Maybe a), String)
class IOTType a | toByteCode a & fromByteCode a & iTask a
```

Generating the stack machine bytecode is straightforward for the basic imperative language constructs. The next listing shows some implementation for the arithmetic and conditional classes.

```
tell :: w → RWST r w s m () //From MonadWriter

instance arith ByteCode where
    lit x             = BC (tell [BCPush (BCValue x)])
    (+.) (BC x) (BC y) = BC (x ≫| y ≫| tell [BCAdd])
    …

instance IF ByteCode where
    (?) b t = …
    IF  (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]

//Fetch a label from the state
freshLabel :: RWS () [BC] BCState Int
```

5.2 Shared Data Sources (SDSs)

SDSs and variables used in an IoT task are related concepts. They are represented differently in the compiler and in the iTasks system but on the device they are the same thing. They are both stored in a list of `BCShares` that is stored in the compiler's state. In the compilation process, the method of getting the initial value is different and the `sdss` are synchronized with the referenced iTasks SDSs on execution. For a `var`, the initial value is available but for an `sds` this initial value must be queried using the `get` iTasks task. A `BCShare` consists of a device unique identifier and either the initial value or the iTasks reference.

```
:: BCShare = {sdsi :: Int, sdsval :: Either BCValue (Shared BCValue)}
```

An `sds` definition is always of the form `sds λ x=someShare In ↪ {main = …}`. The compiler adds a `BCShare` to the list and the lambda variable — named `x` in this case — is the RWST writing the `BCSdsFetch` instruction with the identifier embedded. The `BCShare` is initialized with the default value in the `var` case. The `sds` case requires some more work because IoT SDSs in a `BCShare` record are not typed anymore by the Clean type system in the compiler state but boxed in the `BCValue` type. Therefore, a lens on the linked iTasks SDS is created to map the original type to the box and the other way around. This is a potentially unsafe cast, but the `BCValue` SDS is not accessible from the outside. It is used to process publications coming from the device. The device cannot change the type and therefore this is safe.

```
mapReadWriteError :: (r → MaybeError String r`, w` r → MaybeError String (Maybe
      ↪ w)) (SDS r w) → SDS r` w`

lens :: (SDS t t) → SDS BCValue BCValue | IOTType t
lens s
    = mapReadWriteError
        (λ           t → Ok (BCValue t)
        ,λ(BCValue v) t → case fromByteCode (toByteCode v) of
            (Right (Just t), "") = Ok (Just t)
            _ = Error "Mismatch in BCValue type"
        ) v
```

5.3 Assignables

Assignables — e.g. the `dIO` construct — result in an RWST writing a *fetch* instruction. Therefore, an the left hand side of an assignments needs to be rewritten to their *store* counterpart. The `censor` function from the `Writer` monad is used to rewrite the instruction accordingly. This technique is applied for all assignables and the technique is used for the `pub` function as well to transform the `BCSdsFetch` instruction to a `BCSdsPublish` instruction.

```
instance dIO ByteCode    where dIO (BC p) = BC (tell [BCDigitalRead p])
instance assign ByteCode where (=.) (BC v) (BC e) = BC (e ≫| censor makeStore v)

makeStore [BCSdsFetch i]    = [BCSdsStore i]
makeStore [BCDigitalRead i] = [BCDigitalWrite i]
makeStore […]               = […]
```

5.4 Compilation Example

To demonstrate the compilation, the following code shows a room monitoring program that reports if the temperature is too high. The report is done by setting the alarm SDS to `True`, this will trigger an iTasks task for handling the alarm. If the temperature (read from the given pin) is over the `panic` value, the alarm will sound. Moreover, it will also set the alarm if the temperature is over the `limit` value for longer than 10 ticks. The IoT task is parametrized as a Clean function and requires a temperature pin, a limit value, a panic value and an alarm SDS. The values of the arguments can easily be obtained through an editor in iTasks. This really shows that Clean is a macro language that you can use to construct IoT tasks dynamically and according to a runtime specification. It returns a tailor made IoT task that can be sent to the device.

```
temp :: AnalogPin Int Int (Shared Bool) → Main (ByteCode () Stmt)
temp pin limit panic alarmShare =
    sds λalarm = alarmShare In
    var λcount = 0          In
    {main =
        IF (aIO pin > . lit panic)
            (alarm =. lit True :.  pub alarm )
            ( IF (aIO pin > . lit limit) (
                count =. count +. lit 1 :.
                IF ( count > . lit 10)
                    ( alarm =. lit True :. pub alarm)
                    ( noOp )
            ) ( count =. lit 0)
    }
```

Using the bytecode backend, this program is compiled to the bytecode given. The bytecode is numbered with the program memory offset. The labels are already resolved to actual addresses but for clarity the conditional structure is displayed next to the instructions. The compiler state returning includes two `BCShare` values, the second — identifier 2 initialized with the `var` construct — has the initial value `0 :: Int`. The first — identifier 1 initialized with the `sds` construct — has no initial value because it is a referenced iTasks SDS. The initial value is retrieved from the iTasks system upon sending the IoT task to the device.

```
 0. BCAnalogRead A0
 2. BCPush (panic :: Int)
 6. BCGre
 7. BCJmpF 20              / if (aIO pin >. lit panic)
 9. BCPush (True :: Bool)  |
12. BCSdsStore 1           |
15. BCSdsPublish 1         |
18. BCJmp 69               |
20. BCAnalogRead A0        + else
22. BCPush (limit :: Int)  |
26. BCGre                  |
27. BCJmpF 62              |   / if (aIO pin >. lit limit)
29. BCSdsFetch 2           |   |
32. BCPush (1 :: Int)      |   |
36. BCAdd                  |   |
37. BCSdsStore 2           |   |
40. BCSdsFetch 2           |   |
43. BCPush (10 :: Int)     |   |
47. BCGre                  |   |
48. BCJmpF 60              |   |   / if (x >. lit 10)
50. BCPush (1 :: Bool)     |   |   |
53. BCSdsStore 1           |   |   |
56. BCSdsPublish 1         |   |   |
58. BCJmp 60               |   |   + else
60. BCJmp 69               |   |   \ endif
62. BCPush (0 :: Int)      |   + else
66. BCSdsStore 2           |   |
69. END OF PROGRAM         \   \ endif
```

6 RUNTIME SYSTEM

The RTS is the single program/firmware that needs to be executed on the device for the system to be able to execute IoT tasks and integrate with iTasks. On startup, the allocated memory is initialized, followed by running a device specific setup function. In this specific setup function, peripherals can be initialized and communication can be established. If this function returns, a connection with the server has been made and the main loop is continuously executed. Only in case of a shutdown request, the memory is cleared and the device specific setup function is executed again so that the device is in the initial state again waiting for a connection.

The main loop (1) checks if there is input on the communication channel. If input is available, it reads and parses this to a message and process the message. All messages — such as new tasks, SDSs or a specification request — are processed immediately. (2) The RTS executes tasks that are ready. This means that all one shot tasks and all interval tasks for which the interval has passed are executed. The execution happens in a round robin fashion. If a one shot task is executed, it is removed from the memory. If an interval task is executed, its last run time is set to the current time. Interpretation always starts from the first cell in the program memory and ends when the program counter exceeds the size of the program or a return instruction is encountered in which case the task is also removed from the memory. (3) When one entire loop is finished, the program waits for a — compile time determined — time after which it continues. During this waiting the device idles.

6.1 Interface

The RTS is written in C and only uses standard C functions such that the same code can be used for all devices. All device specific functions are hidden in a single header file called the interface. It contains functions for accessing device specific peripherals, communication functions, setup and tear-down and it defines the specification. The interface header file is created in a modular way using conditional macros. This means that — similar to the eDSL — parts of the interface are optional. Every device has to implement the communication functions but peripherals are optional. The specification of the device is generated from the implemented functions to tell the server upon startup what its capabilities are. Therefore, porting the RTS to a new device only requires implementing the interface. The device specific interface is very simple. For example, there are only three functions regarding communication as shown below.

```
bool    input_available(void);
uint8_t read_byte(void);
void    write_byte(uint8_t b);
```

Implementations are made for *POSIX*, *mbed*[3], *ChibiOS*[4] and *Arduino*[5] compatible platforms using either TCP or Serial connections.

6.2 Memory Management

The RTS has to store both statically and dynamically allocated data. The static data contains the interpreter state, the interpreter stack and the communication buffers. The dynamic data consists of the tasks and SDSs. Tasks consists of their bytecode, an identifier and the scheduling information. An SDS is made up of the identifier and their value.

Some devices have very little memory and therefore space needs to be used optimally. While almost all MCUs support heaps nowadays, the functions for allocating and freeing memory on the heap are not very space optimal and often leave holes if allocations are not freed in a last in first out order. SDSs and tasks may be dynamically added and removed. To mitigate this problem, the RTS manages its own — compile time configurable sized — memory in the global data segment. Tasks are stored from the top down and SDSs are stored from the bottom up.

When a task or an SDS is removed, this managed space is compacted immediately so that there are no holes left. In practice this means that if the first received task is removed, all tasks received later are moved up until the hole is filled completely. Obviously, this is quite time intensive, but it cannot be permitted to leave holes in the memory since the memory space is so limited. With this aggressive memory management technique, even an Arduino UNO R3 with just $2K$ RAM can execute several tasks accessing several SDSs concurrently.

7 SERVER INTEGRATION

The server part of the system — given as an iTasks implementation — is responsible for housekeeping the IoT devices. Accessing the functionality only happens via two glue functions shown in Section 4.1. The following sections give details detail on what these functions actually do under the hood.

[3] https://mbed.com
[4] https://chibios.org
[5] https://arduino.cc

7.1 Communication with Devices

Every IoT device has an iTasks memory based SDS assigned to it that contains their communication channels. These channels form the communication-agnostic interface for all communication to-and-fro the device. The type signature for this is given below. The channels SDS is a triple containing an incoming channel, an outgoing channel and a stop flag. The synchronization task — started on device connection — synchronizes the communication channels with the device. If messages appear in the first list, the synchronization task relays them to the device. Moreover, if the device sends a message, the synchronization task places it in second list. When the stop flag is set, the synchronization function terminates because the connection with the device is closed. If a new communication method is to be added, a programmer only has to implement the synchronization function for the whole system to work.

```
:: Channels :== Shared ([MSGRecv], [MSGSend], Bool)
:: MSGRecv = MTTaskAck Int Int | MTSDSAck Int | MTPub Int BCValue | ...
:: MSGSend = MTTask Interval String | MTSds Int BCValue | ...
```

7.2 Device Storage

All devices are stored in one global SDS containing a list of `Device` records. Each record contains everything a device encompasses. Storing all devices in a single SDSs has the advantage that one can inspect all devices from anywhere in the iTasks program. This facilitates debugging and error handling. The system knows which devices are connected and which IoT SDSs are available on a device. Moreover, this approach also allows the creation of systems that reconnect devices after a restart because a persistent SDS can be used to store the devices.

Every record stores a reference to the communication channels, the compiler state, the information to setup the synchronization function, a list of IoT tasks and a list of IoT SDSs. If the device is connected, it also stores the task identifier for the synchronization function, and the hardware specification. If the device is erroneously disconnected it can set a descriptive error.

The programmer can only add devices to the system through the `withDevice` function. This function is a wrapper around several asynchronous functions that interact with the device records in the global SDS. This function (1) adds the device to the global device list (2) connects the device (3) executes the task requiring the device (4) waits for a stable value for the given task (5) requests a shutdown for the device (6) removes the device from the global device list.

Connecting a device is also a multi step process in itself. It (1) clears the channels. (2) starts a message processing task. (3) sends a device specification request and stabilizes when this request has been honored.

The message processing task is a compound task consisting of the device specific synchronization function and a device agnostic message processing function. The message processing function acts upon new messages in the incoming channels. For example, when an `MTPub` is received, the corresponding SDS in the device record is updated. Similarly, when an `MTTaskAck` is received, the task in the device record is updated with the appropriate task identifier.

7.3 Synchronizing Shared Data Sources

The server stores a proxy value for every IoT SDS in the form of an iTasks SDS. This proxy iTasks SDS stores the latest value from the device as a cache. If it is written, it will send an SDS write request to the device. Watchers are notified when the device published a new value. This cached value is stored in the device record in the `IOTShare` type. This `IOTShare` type contains the identifier, the current value and possibly the reference to the iTasks SDS.

```
:: IOTShare = { identifier:: Int
              , value     :: BCValue
              , iTaskRef  :: Maybe (Shared BCValue)
              }
```

If the device publishes a new value for an IoT SDS, the processing task updates the proxy value stored in the device record. Moreover, the processing task writes the new value to the reference iTasks SDS lens. This lens automatically translates the `BCValue` box to the correct value and writes the actual referenced iTasks SDS.

The other way around, when a task updates the referenced iTasks SDS, a watcher task — started in the `liftIOTTask` function (Section 7.4) is notified. The watcher task can then update the proxied value in the device record accordingly. The synchronizing of this proxied value with the actual device is explained in Section 7.5.

7.4 Executing Tasks

The programmer can only execute tasks on a device through the `liftIOTTask` task. In the same fashion as the `withDevice` function, it wraps several device record modifying tasks.

The function (1) compiles the IoT task to messages (2) places the messages in the channels SDS (3) adds the task to the device record (4) sets the task identifier when the task acknowledgement is received (5) waits for the task to stabilize while watching all the reference iTasks SDSs watching a reference iTasks task is done by using the `whileUnchanged` function on the `Shared BCValue` lens. Moreover, it removes the task from the device by sending a `MTTaskDel` message when the iTasks task is terminated. Termination of iTasks tasks happens for example if the task is part of a step (`>>*`) and one of the conditions matches or when one of its siblings throws an exception.

7.5 Lenses on the Device SDSs

All the device information is stored in a single iTasks SDS for reasons mentioned in Section 7.2. Unfortunately, there are also two issues with this approach.

First, it is not convenient to edit the global SDS containing the devices if you are only interested in a part of it. For example, updating a single IoT SDS value for a single device requires a complicated update function.

Secondly, a task watching the global SDS might only be interested in a very small section of it but is notified on all changes. For example, watcher tasks are launched in the `liftIOTTask` function. These tasks watch only a single reference iTasks SDS to make sure the value is proxied in the device record. However, if another task writes in the device SDS to a different place, the watchers are notified either way.

To solve the first type of problems, parametric lenses were introduced [Domoszlai et al. 2014]. The `p` that was fixed to `()` in Section 2

represents the parameter and this parameter is available during reading and writing. In the SDS access tasks, this parameter must be fixed to `()` which can be achieved using the `sdsFocus` function. This function fixes the parameter and casts it to `()` so that the SDS is usable for the standard tasks. The SDS can specify or refine the data read or written according to the given value of p. Moreover, it can place a filter on the notifications using this parameter. For example, it can be used to create a lens on a tuple — e.g. only giving write access to the first element. If the second element is written, a watcher on the first element is not notified.

```
sdsFocus :: p (SDS p r w) → SDS () r w | iTask p
```

And the type used for the parametric lens and the device SDS is defined as follows.

```
:: IOTParam = Global | Local Device | Share Device Int
deviceStore :: SDS IOTParam [Device] [Device]
```

Every constructor denotes a different type of view on the root SDS. First, the `Global` constructor is only interested in the entire list of devices. Secondly, the `Local` lens only looks at a single device. Finally, the `Share` view only focusses on a single SDS on a single device. For these SDS lenses, functions are available to access the parts of the global SDS.

```
deviceStoreNP ::                    Shared [Device]
deviceShare   :: Device             → Shared Device
shareShare    :: Device IOTShare    → Shared BCValue
```

Focussing the `deviceStore` to `Global` gives access to the global SDS. Global watchers are only be notified if the structure of the list changes, i.e. if a device is added or removed.

Accessing a single device is done with the `deviceShare` function. The SDS requires a `Device` record to know on which device to focus. This record is available within the task given to `withDevice`. Watchers of a local SDS are notified when something in the device changes. Writers can change something in the device such as the specification.

Share SDSs can be accessed through the `shareShare` function. This function focusses the global SDS on a single SDS on a single device. It is only notified when that specific proxy SDS value changes.

This brings us to the last unsolved part of the extension, namely synchronizing the proxy values with the device. Albeit not designed for it, parametric lenses can also be used to solve this problem. The parameter is known in the stateful write function. The state can be used to write to other — e.g. device's channels — SDSs.

When a task writes to this SDS, the global SDS knows this through the parameter and propagates the value to the device. When the server or the device changes the SDS, this view is notified. The SDS requires a `Device` and a `IOTShare` record. The `IOTShare` record and therefore the lens is only used internally, for example by the processing function watching the IoT SDSs to synchronize them with their iTasks references.

7.6 Task Migration and Task Construction

The thermostat example is kept simple for illustration purposes. However, it does not show the full potential of the extension. For example, it is possible to extend the thermostat with a redundant cooler. If the cooler IoT device would stop working it will be automatically reassigned to another device. The only function we need to change is the `operateCooler` function, note that the `cooler` iTasks SDS is in scope here. The `withDevice` function throws an exception if a device terminates the connection and the connection cannot be re-established. This exception can be caught and the work that is done on the device can be moved to another device.

```
operateCooler :: Task ()
operateCooler
    = try
        (withDevice coolerDevice runCoolerTask)
        (λexc → viewInformation "Exception" [] (toString exc)
            ≫| withDevice coolerDevice2 runCoolerTask)
where
    runCoolerTask dev
        = liftIOTTask (dev, OnInterval 1000)
            (sds λf=cooler In {main= dIO LED1 =. f :. dIO D5 =. f})

    coolerDevice2 :: TTYSettings
    coolerDevice2 = {devicePath="/dev/ttyACM0", baudrate=B19200, ...}
```

Moreover, IoT tasks can be sent dynamically at runtime to the device. To illustrate this, we show a task in which the user can send tasks to a device. The user selects the task from a list of predefined tasks and provides the execution strategy via the web interface as well. If this task is executed in parallel with the sensor and cooling tasks with the device, the user can use the device as well for miscellaneous other tasks. For example to blink a light, or to open window blinds on demand while the thermostat is operating.

```
interact :: Device → Task ()
interact device
    =    enterChoice "Choose a task" [ChooseFromList fst] taskList
    -&&- enterInformation "Execution Strategy" []
    >^* [OnAction (Action "Send") $ withValue λ((_, task), strat)→
            Just (task ≫=λiottask→liftIOTTask (device, strat) iottask)]
    @! ()
where
    taskList :: [(String, Task (Main (ByteCode () Stmt)))]
    taskList =
        [("faculty", enterInformation "Faculty of what?" []
            ≫=λn → var λx=1 In var λy=n In {main =
                IF (x ==. 0) return (x =. y *. x :. y -. lit 1)})
        ,("count", return $
            sds λx=0 In {main = x =. x +. lit 1 :. pub x})
        ,("blink", enterInformation "Led on which pin?" []
            ≫=λl → var λx=True In {main = x =. Not x :. dIO l =. x})
        ,("...", ...)
        ]
```

8 RELATED WORK

Related research has been conducted on the subject arising from academia and the industry. For example, MCUs such as the Arduino can be remotely controlled very directly using the Firmata-protocol[6]. 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 communication bandwidth since no code is executed on the device itself, only the

[6] https://github.com/firmata/protocol

peripherals are queried. A Haskell implementation of the protocol is also available[7]. The hardware requirements for running a Firmata client are very low because all the logic is on the server. However, the bandwidth requirements are high and therefore it is not suitable for IoT applications that communicate through LTN networks. Similarly, Grebe and Gill [2016] created *HaskIno*, a monadic interface over *hArduino* that allows remote code execution on Arduinos. Their initial tethered solution is based on Firmata but they also propose an untethered approach that is similar to compilation by storing the program in EEPROM. However, there is no communication between the device and the server that programmed it and the solution is very specific to the Arduino ecosystem. An extension has been proposed where explicit threading is supported [Grebe and Gill 2017] in which the code executed is similar to mTask's IoT tasks. It differs in the execution model and in data access between threads and there is no shared data with the server.

There are also some more general OS/RTS solutions for MCUs that allow the programmer to program MCUs on a more abstract level. They generate a static image to flash on the MCU for operation and do not support dynamic task sending and there is no out of the box typed transparent data sharing between the server and the client. For example, Levis et al. [2005] created TinyOS, which is an OS that can compile a static program for a lot of MCUs and supports threading and has a similar execution model as the new system — namely slicing programs and lacking a blocking API. Furthermore, Elsts et al. [2015] proposed ProFUN, a — similar to TOP — declarative language using Task Graphs (TGs) to create sensor networks. The TGs can be created in a graphical interface accessible with a web browser. Functionality exists for automatic logging via a server application.

Clean has a history of interpretation, for example, there is a lot of research happening on the intermediate language SAPL. SAPL is a purely functional intermediate language that can be efficiently interpreted. It has interpreters written in C++ [Jansen et al. 2007] and a compiler to JavaScript [Domoszlai et al. 2011]. Compiler backends exist for Clean and Haskell which compile the respective code to SAPL [Domoszlai and Plasmeijer 2012]. The SAPL language is 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. 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 MCU 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 MCU environments. This work uses parts of the existing mTask-eDSL which generates C code to run a TOP-like system on MCUs [Koopman et al. 2018; Koopman and Plasmeijer 2016]. Again, this requires a reprogramming cycle every time the task-specification is changed and there is no interaction with the server. The nature of the embedding technique allows additional backends to be written without touching existing ones. Hence, the eDSL is used for this solution but with a novel backend.

Another eDSL designed to generate low-level programs is called Ivory and uses Haskell as a host language [Elliott et al. 2015]. 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 [Hickey et al. 2014; Pike et al. 2014]. 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 Abstract Syntax Tree (AST) that is sent to a chosen backend. In our 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.

Not all IoT devices run solely compiled code, e.g. the ESP8266 powered NodeMCU is able to run interpreted Lua code. Moreover, there is a variation of Python called micropython that is suitable for running on MCUs. 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 an IoT device because implementing this extensibility in the interpreted language leaves no room for the actual programs. Also, some devices only have $2K$ of ram, which is not enough for this.

9 CONCLUSION

The IoT is growing and gaining popularity very fast. However, programming the IoT is cumbersome. The devices in the IoT are programmed individually using a plethora of programming languages and communication protocols. Previously we introduced an eDSL to program the IoT devices from a single source. In this work we modify the approach to support creating dynamic IoT applications.

First, an iTasks extension is shown that allows us to dynamically connect devices to a running program. The communication is physical connection method and protocol agnostic. The devices need to be programmed once with an appropriate RTS to function in this system. The RTS is available for several MCUs. The device specific part of the RTS is small and modular. This makes porting it to a new architecture easy and keeps the footprint small. Adding peripherals is easy. It does not even require recompilation of clients due to the modular setup of the RTS code.

Secondly, we add high level communication between the eDSL on the device and the iTasks server via SDSs. The iTasks SDSs on the server can be shared with specific devices. These devices publish SDS changes on command to limit the amount of communication.

Lastly, we tackle the maintenance problem of programs on the devices in the field. A new backend for the mTask eDSL is created that compiles to tailor-made bytecode. This bytecode is shipped at runtime to the device for execution. The RTS interprets the bytecode using a stack machine. The IoT tasks are then dynamically loaded to a device for execution. This dynamic allocation of tasks to devices makes the system very flexible. For example, when an IoT task is assigned to a device and that device becomes unusable, the iTasks system can reassign it to another device automatically to add redundancy to a system.

We think that these contributions make it suitable for real world applications. We are currently testing this. Nevertheless, the technique can be improved in umpteen ways.

[7] https://leventerkok.github.io/h\gls{Arduino}

10 FUTURE WORK

10.1 Extensions

An additional simulation view to the IoT eDSL can be added that works similar to the existing C-backend simulation. The first option is to simulate a device completely, the simulator is an instance of the `Duplex` class. Secondly, it can simulate symbolically by implementing the eDSL classes. At the time of writing, work is done on an iTasks simulator device of the former kind.

True multitasking can be added to the client software. IoT tasks get slices of execution time and have their own stack, this allows IoT tasks to run truly concurrent. Multitasking allows tasks to be truly interruptible by other tasks. Furthermore, this allows for more fine-grained timing control of tasks. However, it influences memory requirements.

Many research topics can be explored in the field of resource management and analysis, both statically at compile time and dynamically at runtime for both peripheral requirements and memory requirements.

10.2 Further Improvements

The current implementation offers a subset of the TOP combinators in IoT. The subset can be extended to allow for more fine-grained control flow between IoT tasks. Furthermore, more logic can be moved to the device instead of it residing on the server reducing the communication overhead.

At the moment, all data is sent as plain text over the wire and the device cannot know whether it talks to a legitimate server or an attacker. Due to the nature of the system, namely sending code that is executed, security needs to be investigated. Only bytecode for very specific IoT tasks can be sent to the device at the moment which mitigates the risk somewhat. As the language will become more expressive, the security risk increases.

Finally, the robustness of the system can be improved. Tasks residing on a device that disconnects should be kept on the server to allow a swift reconnect and restoration of the tasks. Moreover, an extra specialization of the shutdown can be added that drops the connection but keeps the tasks in memory. This can be extended by allowing devices to send their tasks back to the server. In this way devices can even connect to different servers. Tasks can be stored in EEPROM or on external memory to be able to access them even after a reboot or to save memory. EEPROM can be written about ten to a hundred times more often than flash memory.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments that helped improve this paper.

REFERENCES

Tom Brus, Marko van Eekelen, Maarten Van Leer, and Marinus Plasmeijer. 1987. Clean – a language for functional graph rewriting. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 364–384.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

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.

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.

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.

László Domoszlai and Rinus Plasmeijer. 2012. Compiling Haskell to JavaScript through Clean's core. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica* 36 (2012), 117–142.

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.

Atis Elsts, Farshid Hassani Bijarbooneh, Martin Jacobsson, and Konstantinos Sagonas. 2015. ProFuN TG: A tool for programming and managing performance-aware sensor network applications. In *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th*. IEEE, 751–759.

Jeremy Gibbons. 2015. Functional programming for domain-specific languages. In *Central European Functional Programming School*. Springer, 1–28.

Mark Grebe and Andy Gill. 2016. Haskino: A remote monad for programming the arduino. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 153–168.

Mark Grebe and Andy Gill. 2017. Threading the Arduino with Haskell. In *Post-Proceedings of Trends in Functional Programming*.

Patrick Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. 2014. Building embedded systems with embedded DSLs. In *ACM SIGPLAN Notices*. ACM Press, 3–9. https://doi.org/10.1145/2628136.2628146

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.

Mark Jones. 1995. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*. Springer, 97–136.

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

Pieter Koopman and Rinus Plasmeijer. 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.

Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

Arjan Oortgiese, John van Groningen, Achter Peter, and Plasmeijer Rinus. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29nd 2017 International Symposium on Implementation and Application of Functional Languages (IFL '17)*. ACM, New York, NY, USA, 12.

Lee Pike, Patrick Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. 2014. Programming languages for high-assurance autonomous vehicles: extended abstract. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages meets Program Verification*. ACM Press, 1–2. https://doi.org/10.1145/2541568.2541570

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.

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.

Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2011. Clean language report version 2.2 (2011). https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf

Josef Svenningsson and Emil Axelsson. 2012. Combining deep and shallow embedding for EDSL. In *International Symposium on Trends in Functional Programming*. Springer, 21–36.

A Staged Embedding of Attribute Grammars in Haskell

Marcos Viera
Instituto de Computación
Universidad de la República
Uruguay
mviera@fing.edu.uy

Florent Balestrieri
fbalestrieri@orange.fr

Alberto Pardo
Instituto de Computación
Universidad de la República
Uruguay
pardo@fing.edu.uy

ABSTRACT

In this paper, we present an embedding of attribute grammars in Haskell, that is both modular and type-safe, while providing the user with domain specific error messages.

Our approach involves to delay part of the safety checks to runtime. When a grammar is correct, we are able to extract a function that can be run without expecting any runtime error related to the EDSL.

KEYWORDS

Attribute Grammars, EDSL, Staging, Dynamics, Haskell

ACM Reference Format:
Marcos Viera, Florent Balestrieri, and Alberto Pardo. 2018. A Staged Embedding of Attribute Grammars in Haskell. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310235

1 INTRODUCTION

In [4], de Moor et.al. showed that the combination of attribute grammars [9] and functional programming results in a powerful toolset for the rapid prototyping of domain specific languages. They provide a compositional semantics of the structuring mechanisms of attribute grammars which is given as a Haskell program that can be easily extended with new structuring features. The implementation of the attribute grammar operators is given in such a way that it makes it possible to define attribute grammars modularly by combining smaller attribute grammar fragments together, hence enabling code reuse. The main drawback of de Moor et.al. approach is that it fails to capture in Haskell type system important invariants (e.g. to check that the used attributes are already defined and they have the needed type) concerning the correct definition of attribute grammars. As a result, many of the operations on attribute grammars result to be partial.

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310235

Previous attempts to use Haskell type system to make structural checks on attribute grammars relied on complex types that make use of several (type system) extensions of GHC [16][2, § 8.4,p. 121]. The deal breaker however is that the error messages related to the attribute grammars reveal implementation details that should not be shown to the users of the library: types that are private to the library are leaked out when printing error messages, and are extremely difficult to understand for someone who is not familar with the implementation.

In this paper we present an implementation of first-class attribute grammars in Haskell that does not compromise on safety, nor on the simplicity of the public interface nor on the clarity of error messages. It relies on a combination of static and dynamic type-checking. The dynamic type-checking is all done in one single global verification of the attribute grammar. Upon success, it returns a function that can be executed as many times as necessary, without any further dynamic checks. Upon failure, it returns a precise error message related to the attribute grammar being constructed, and by using one of GHC extensions, may even point to the relevant position in the source code where the error is produced (instead of where it was detected).

The paper is organized as follows. In Section 2 we introduce the public interface of the library through a simple example. In Section 3 we explain the internals of the library, while in Section 4 we show how some common patterns of attribute grammars can be encoded using its public interface. In Section 5 we discuss related work and in Section 6 we conclude.

2 PUBLIC INTERFACE OF THE LIBRARY

An attribute grammar (AG) is a context free grammar to which we associate attributes and semantic rules (that define how to compute the value of the attributes) to each nonterminal of the grammar.

In the following subsections we explain the interface of the library through a running example consisting of a simple expressions language, with integers, variables and addition, represented by the following context-free grammar:

$$\begin{array}{lcl} expr & \to & ival \\ expr & \to & vname \\ expr_1 & \to & expr_2 + expr_3 \end{array}$$

The grammar contains three productions. The first production consists of an integer *ival* terminal. The second production, with a string *vname* terminal, represents variables.

The third production, for the addition, has two non-terminal children. In the last production we add sub-indices to the non-terminals to distinguish the father ($expr_1$) from the respective children ($expr_2$ and $expr_3$).

Attributes provide semantics to a grammar. In our case, the semantics we want to define is the evaluation of an expression, where the values of the variables are provided in a given context. For example, the evaluation of the expression $2 + x$ in the context $x = 7$ results in 9.

We define two attributes:

env the environment, which is distributed from the root to the leaves of the tree described by the grammar;

eval the result of the evaluation, computed from the leaves to the root.

The former is said to be an *inherited attribute* because it is computed in a top-down traversal where the information flows from the root to the leaves, while the latter is a *synthesized attribute* because it is computed in a bottom-up traversal, with the information flowing from the leaves to the root.

An AG decorates a grammar with attributes and their computations, that we call semantics rules. In our example:

$$
\begin{array}{lcl}
expr & \to & ival \\
 & & [expr.eval = ival] \\
expr & \to & vname \\
 & & [expr.eval = slookup\ vname\ expr.env] \\
expr_1 & \to & expr_2 + expr_3 \\
 & & [expr_1.eval = expr_2.eval + expr_3.eval \\
 & & , expr_2.env = expr_1.env \\
 & & , expr_3.env = expr_1.env]
\end{array}
$$

The synthesized attribute *eval* is computed in the following way: in the first production we return the value of the terminal *ival*, in the second production the value associated to the variable is looked-up into the inherited environment, and in the last production we sum the values of the synthesized *eval* attributes from its children ($expr_2$ and $expr_3$). The inherited attribute *env* is distributed to the children non-terminals $expr_2$ and $expr_3$ unchanged.

The definition of an AG in terms of our library involves the following steps.

(1) Define the elements of a context free grammar: non-terminals, productions, non-terminal children of a production, and terminals.
(2) Define attributes and their semantic rules.
(3) Combine attributes to form a complete AG.
(4) Associate grammar elements to a concrete datatype and obtain, after validity checks, a function that evaluates attributes on a value of such datatype.
(5) Apply the generated function.

The whole process is modular. Only the last two steps require us to fix the datatype and the corresponding grammar elements on which the attributes will be computed as well as the concrete rules to compute them. In addition, the error messages produced by the AG's well-formedness checks are clear and do not leak implementation details. This is achieved through the use of staging; after a first compilation stage performed by the Haskell compiler, we compile the AG to a Haskell function (step 4) in a second stage which is performed at runtime[1]. The second stage generates either the semantic function or error messages due to failures in the AG checks. In case the function is obtained, it can be applied to different values of the grammar and input attributes (step 5) without the necessity of further dynamic checks.

```
expr     = non_terminal "Expr"
val      = production expr "Val" [] (ival `consT` nilT)
var      = production expr "Var" [] (vname `consT` nilT)
add      = production expr "Add" [leftAdd, rightAdd] nilT
leftAdd  = child add "leftAdd" expr
rightAdd = child add "rightAdd" expr
ival     = terminal "ival"  pInt
vname    = terminal "vname" pString
```

Figure 1: Grammar

2.1 Context free grammar

There is a direct correspondence between context free grammars and mutually recursive algebraic datatypes. When we want to evaluate an AG, we attach the grammar to a concrete datatype. Unlike Haskell datatypes, our grammars are modular and extensible; they can be defined through different compilation units (modules). We can add new datatypes (non-terminals) and new constructors (productions).

We define a *Grammar* as a set of productions.

```
type Grammar = Set Production
grammar :: [Production] → Grammar
```

The types *NonTerminal*, *Production*, *Child*, *Terminals* are abstract, and represent the different elements of a grammar. The following functions are their smart constructors, which we use in Figure 1 to define the grammar for the expression language:

- *non_terminal* :: *Name* → *NonTerminal*

  Defines a non-terminal given its name.

- *production* :: *NonTerminal* → *Name* → [*Child*] → *Terminals* → *Production*

  Defines a production given the non-terminal to which it belongs, the name of the production and the lists of children (non-terminals) and terminals that compose it.

- *child* :: *Production* → *Name* → *NonTerminal* → *Child*

  Defines a child out of the production to which it belongs, its name and non-terminal.

- *terminal* :: *Typeable a* ⇒ *Name* → *Proxy a* → *Terminal a*

[1] In section 2.7 we show that this stage can be moved to compile time by using Template Haskell.

```
eval = attr S "eval" tEval
env  = attr I "env"  tEnv
```

Figure 2: Attributes

Defines a terminal given its name, and its type. *Proxy* is an empty type with a phantom parameter that is used to provide type information. For instance, *pInt* has type *Proxy Int*. The use of Typeable will become clear in Section 3, and it has to be with the use of Dynamic to store values of different types in the same collection.

Notice that we provide names[2] to all the ingredients of a grammar. Also notice that the grammar definition is modular, since all its ingredients are defined separately. Despite that, typically a production and its children are mutually recursive, like the production *add* and its children *leftAdd* and *rightAdd* of Figure 1[3].

The list of terminals (*Terminals*) is abstract and constructed using the functions *nilT* and *consT*.

```
nilT  :: Terminals
consT :: Typeable a ⇒ Terminal a → Terminals → Terminals
```

In the example, the production *add* has two children (*leftAdd* and *rightAdd*) and does not have non-terminals, while *val* has a terminal *ival*, representing the integer value, and *var* has a terminal *vname*, for the name of the variable.

2.2 Attributes

The type (*Attr k a*) of attributes of kind *k* (e.g. inherited) and type *a* is abstract. We declare new attributes with:

```
attr :: Typeable a ⇒ Kind k → Name → proxy a → Attr k a
```

Kinds are given by three abstract types *I* (inherited), *S* (synthesized) and *T* (terminal), and are specified using the homonymous constructors of the singleton type *Kind k*.

```
data Kind k where
  I :: Kind I
  S :: Kind S
  T :: Kind T
```

In our approach terminals are treated as a special kind of attributes, called *terminal attributes*.

```
type Terminal a = Attr T a
```

Thus, the function *terminal*, that defines terminals, applies *attr* to the Kind *T*:

```
terminal = attr T
```

We declare the attributes of our example in Figure 2; *tEval* and *tEnv* have types *Proxy Int* and *Proxy* (*String* :-> *Int*), respectively. We use the type operator (:->) as a synonym of *Map*.

[2]The type *Name* is a synonym of *String*.
[3]The library provides alternative constructors to define a production at the same time as its children

```
evalA = syn eval [ add |- (+) <$> leftAdd ! eval
                             <*> rightAdd ! eval
                 , val |- ter ival
                 , var |- slookup <$> ter vname
                                  <*> par env ]
envA  = copyPs env [add]
asp   = evalA <> envA
```

Figure 3: Aspects

2.3 Aspects

Aspects are collections of attribution rules. They are constructed using the functions *inh* and *syn* which define the attribution rules for a single inherited and synthesized attribute, respectively.

```
inh :: Typeable a ⇒ Attr I a → [(Child,          AR a)]
    → Aspect
syn :: Typeable a ⇒ Attr S a → [(Production, AR a)]
    → Aspect
```

Inherited attributes are computed top-down, thus for a given attribute (*Attr I a*), we declare the attribution rules (*AR a*) to compute the values for each child (*Child*). In the case of synthesized attributes (*Attr S a*), since the information flows bottom-up, the rules are defined per production.

Attribution Rules. *AR* is the applicative functor [12] of *attribution rules*. The following *AR* primitives are used to access attributes:

- (!) :: *Typeable a ⇒ Child → Attr S a → AR a*

  projects a synthesized attribute from a child.

- *par* :: *Typeable a ⇒ Attr I a → AR a*

  projects an inherited attribute from the parent.

- *ter* :: *Typeable a ⇒ Attr T a → AR a*

  projects a terminal attribute.

For example, in Figure 3 *evalA* declares the set of rules to define the way the *eval* attribute of our example is computed: The operator (|-) is a synonym of the pair constructor (,) that provides a more readable syntax for association lists.

The *env* attribute is declared using *inh*.

```
envA = inh env [ leftAdd  |- par env
               , rightAdd |- par env ]
```

For both children *leftAdd* and *rightAdd* of the *add* production, the value is just the inherited *env* of the parent. This is a common pattern when defining inherited attributes, called *copy rule*. We capture this pattern with the *copyPs* combinator, which defines an aspect given an attribute and the list of productions where it has to be applied. Thus we can define *envA* as shown in Figure 3.

```
elet    = production expr "Let" [exprLet, bodyLet]
                                (vlet `consT` nilT)
exprLet = child elet "exprLet" expr
bodyLet = child elet "bodyLet" expr
vlet    = terminal "vlet" pString
```

Figure 4: Grammar Extension

```
evalA2 = syn eval [elet |- bodyLet ! eval] <> evalA
envA2  = inh env  [exprLet |- par env
                  , bodyLet |- insert <$> ter vlet
                                      <*> exprLet ! eval
                                      <*> par env
                  ] <> envA
asp2 = evalA2 <> envA2
```

Figure 5: Attributes of the extension

Merging Aspects. Aspects form a *Monoid*: *mempty* defines an aspect with no rule and (<>) combines the rules of two aspects. The aspects of our example (*asp* in Figure 3) are composed by *evalA* and *envA*.

Merging aspects requires their domains to be disjoint, we provide primitives to delete some attribution rules from an aspect.

```
delete_I :: Typeable a
         ⇒ Attr I a → Child      → Aspect → Aspect
delete_S :: Typeable a
         ⇒ Attr S a → Production → Aspect → Aspect
```

Such operations are useful for *redefining attributes*. For example, suppose that, for debugging purposes, we want to print messages tracing all the additions performed when evaluating an expression. This can be easily done, without changing the code of the existing attribute definitions, by defining a new behavior for the attribute *eval* at the production *add* and deleting the previous one:

```
asp' = syn eval [add |- trace "add" ∘ (+) <$> leftAdd ! eval
                                          <*> rightAdd ! eval]
     <> delete_S eval add asp
```

2.4 Extending the Grammar

An important design goal of our library is modular extensibility, both at the attribute and grammar level. Suppose we want to extend the expression language with a **let** construct.

$$expr_1 \quad \rightarrow \quad \textbf{let}\ var\ =\ expr_2\ \textbf{in}\ expr_3$$

In Figure 4 we show how the new production can be defined, declaring that it belongs to the non-terminal *expr*, it has two children (*exprLet* and *bodyLet*), and one terminal attribute (*vlet*) of type *String*.

Figure 5 shows the definition of the attributes for the extension.

2.5 Contexts

Attribute rules and aspects generate a set of constraints that must be satisfied by a grammar to be evaluated. The context of an aspect contains information about which attributes it needs and which ones it provides. Each use of (!), *par* and *ter* generates a *Require* constraint to make sure that the corresponding attribute will be defined when we run the AG. The aspect constructors *syn* and *inh* generate *Ensure* constraints for synthesized and inherited attributes, respectively.

We can inspect the constraints associated with an aspect using the function *print_context* in the interactive Haskell evaluator.

```
print_context :: Aspect → IO ()
```

For example, the context of *envA* denotes that the inherited attribute *env* is defined for *leftAdd* and *rightAdd*, and that the definition of the same inherited attribute is required for the non-terminal *Expr*.

```
$ print_context envA
Require Inherited   : Expr.env
Require Synthesized : -
Require Terminal    : -
Ensure Inherited    : leftAdd.env, rightAdd.env
Ensure Synthesized  : -
```

Similarly, the context of *evalA* denotes that the synthesized attribute *eval* is defined for the productions *Add*, *Val* and *Var*, given that: both *eval* and *env* are defined for the non-terminal *Expr* and the terminal attributes *ival* and *vname* are defined for the productions *Val* and *Var*, respectively.

```
$ print_context evalA
Require Inherited   : Expr.env
Require Synthesized : Expr.eval
Require Terminal    : Val.ival, Var.vname
Ensure Inherited    : -
Ensure Synthesized  : Add.eval, Val.eval, Var.eval
```

If we merge *evalA* with the aspect *envA*, then we have the same requirements, adding the definition of the inherited attribute *env* for the children *leftAdd* and *rightAdd*.

```
$ print_context asp
Require Inherited   : Expr.env
Require Synthesized : Expr.eval
Require Terminal    : Val.ival, Var.vname
Ensure Inherited    : leftAdd.env, rightAdd.env
Ensure Synthesized  : Add.eval, Val.eval, Var.eval
```

Not all aspects are valid, for instance merging two aspects with overlapping domains yields an invalid aspect. In such cases the function *print_context* shows the errors of the aspect. For example, if we merge *evalA* with itself we get an error for each definition of *eval*.

```
$ print_context (evalA # evalA)
AG:Error:   # at <location>
merging conflict: Add.eval
merging conflict: Val.eval
merging conflict: Var.eval
```

where the location (<location>) is the position in the source code where the aspects containing the duplicated attributes are merged.

Modular attribute grammars allow us to pick and choose any set of productions to form a grammar. An attribute grammar is complete if all the inherited and synthesized attributes *required* by the aspect are *ensured* by the same aspect, and all the terminal aspects are provided by the grammar. The function *print_missing* performs this checks, displaying the unmet requirements.

```
print_missing :: Grammar → Aspect → IO ()
```

For instance, if we check if the aspect *evalA* is complete for our grammar, we get that there is no rule to compute the way the inherited attribute *env*, used to compute *eval* at the production *Var*, is distributed.

```
$ print_missing (grammar [add,val,var]) evalA
Missing Inherited   : leftAdd.env, rightAdd.env
Missing Synthesized : -
Missing Terminal    : -
```

2.6 Running the Attribute Grammar

An attribute grammar denotes a function on a tree that takes the inherited attributes of the root and returns the synthesized attributes of the root.

```
type Check a = Either Error a

run :: Typeable t
    ⇒ GramDesc t → InhDesc i → SynDesc s → Aspect
    → Check (t → i → s)
```

To run the attribute grammar in a type-safe way, the function *run* returns, in case it is possible, a function of type $t \to i \to s$ given:

- a description *GramDesc t* of how the grammar can be extracted from a concrete tree t;
- a mapping *InhDesc i* between the inherited attributes and a value of type i representing them;
- a mapping *SynDesc s* between the synthesized attributes and a value of type s representing them; and
- the *Aspect* containing the rules.

If any of the mappings is not correct or the aspect is not complete, then the function is not constructed and the errors are returned. It is important to notice that a given grammar (and aspect) can be associated to different concrete tree types.

In Figure 6 we show how the semantic function of the example is extracted for a type of expressions *Expr*. To map the concrete tree type with the grammar, in *exprDesc* we have to define for each production how to obtain its children and terminal values from the different terms of the concrete type. The overloaded operator `|=` is used to define such associations. The mapping of the inherited attributes *InhDesc* is a monoid; the function *embed* creates a mapping of an attribute from a value of type i, given the mapping function. For instance, in Figure 6 we embed a list $[(String, Int)]$ as the attribute *env* using the function *fromList*.

```
embed :: Typeable a ⇒ Attr I a → (i → a) → InhDesc i
```

The mapping of the synthesized attributes *SynDesc* is an attribute functor. The function *project* projects the value of an attribute to create a value of type s. In the example we project the *eval* attribute to a *Maybe Int*.

```
project :: Typeable a ⇒ Attr S a → SynDesc a
```

Thus, we can apply the semantics of our example to a given expression and environment.

```
$ (fromRight runExpr) (Add (Add (Val 2) (Val 3))
                           (Var "x"))
                      [("x",20)]
Just 25
```

However, if we try:

```
$ (fromLeft $ run exprDesc (embed env id)
                           (project eval)
                           evalA)
ERRORS
- missing inherited = leftAdd.env,rightAdd.env
$ (fromLeft $ run exprDesc emptyInDesc
                           (project eval)
                           asp)
ERRORS
- missing InhDesc = Expr.env
```

2.7 Moving Generation to Compile Time

By using the staging capabilities of Template Haskell, we can easily move the AG generation, and thus all the error checks, to compile time. The idea is to use quotation to treat the *run* generator as a meta function, that either produces a *safe* evaluator or does not compile, showing the errors of the attribute grammar.

```
safeRun = $(case runExpr of
              Right _ → [| fromRight runExpr |]
              Left  e → error $ show e)
```

3 DESIGN OF THE LIBRARY

In this section we introduce the design of the library by showing fragments of its implementation. The complete implementation of the library is available at [3].

3.1 Grammars

An important goal on the design of the library is that the grammars have to be modular. A *Grammar* can be given by a set of productions; this fully specifies a grammar and the representation is unique (up to set equality).

```
type Grammar = Set Production
```

Productions are identified by their name and non-terminal to which they belong. A *Production* includes a list of children and a set of terminal attributes.

```
type ProdName = (NonTerminal, Name)
data Production = Production { prod_name      :: ProdName
                             , prod_children  :: [Child]
                             , prod_terminals :: Set (Attribute T) }
  deriving (Eq, Ord)
```

Non-terminals are identified by their name.

```
type NonTerminalName = Name
```

```
data Expr = Val Int | Var String | Add Expr Expr
  deriving (Show, Typeable)
exprDesc = gramDesc $ expr |= [add |= [leftAdd    |= λcase { Add x _ → Just x; _ → Nothing }
                                  , rightAdd |= λcase { Add _ x → Just x; _ → Nothing }] & []
                           , val  |= [] & [ival     |= λcase { Val x → Just x; _ → Nothing }]
                           , var  |= [] & [vname |= λcase { Var x → Just x; _ → Nothing }]
                           ]
runExpr :: Check (Expr → [(String, Int)] → Maybe Int)
runExpr = run exprDesc (embed env Map.fromList) (Just <$> project eval) asp
```

Figure 6: Running the Attribute Grammar with a concrete tree *Expr*

```
newtype NonTerminal = NT NonTerminalName
  deriving (Eq, Ord)
```

A *Child* has a name, a production to which it belongs, and a non-terminal.

```
type ChildName = (Production, Name)
data Child = Child { child_name :: ChildName
                   , child_nt   :: NonTerminal }
  deriving (Eq, Ord)
```

Notice that the flexibility obtained by this representation comes with the cost of not statically assuring the grammar correctness. For instance, a production could contain a child that is defined to belong to another production. We will see later in this section that the checks we make before generating the semantic function prevent incorrect grammars.

3.2 Pure Rules

Our encoding of attributes, rules and their evaluation to be presented in this subsection is based on the embedding for modular AGs defined in [4]. We deviate from the original proposal in that we use *String*s and *Dynamic* values, instead of closed universes, in order to obtain a greater extensibility.

The type *Attr k a*, to represent attributes of kind *k* and type *a*, is a phantom type, since the parameter *a* does not appear on the right-hand side of the definition.

```
data Attr k a = Attr { attr_name :: Name
                     , attr_kind :: Kind k }
```

Attributes are collected in finite maps. We hide the types of the attributes to use them as keys of the maps.

```
data Attribute k where
  Attribute :: Typeable a ⇒ Attr k a → Attribute k
```

In order to be able to store the attribute computations in a *Map* (**:->**) from the attributes to their values, we hide their types using *Dynamic*.

```
type AttrMap k       = Attribute k :-> Dynamic
type ChildrenAttrs k = Child :-> AttrMap k
```

It is important to emphasize that the use of *Dynamic* and *Attribute k* are not exposed by the library. When using the library primitives the user will always use concrete types and will refer to the attributes with type *Attr k a*. This lets us guarantee that the dynamic casting will not fail. Thus, we could have saved the cost associated with using *Dynamics*, simply by using the type *Any*[4] and *unsafeCoerce#* to and from it. We decided to use the well-known *Dynamics* just for clarity.

A rule is a function that computes the attributes of a production; it is a mapping from *input attributes* to *output attributes*. These kinds of rules are called *pure rules*, in contrast to the monadic rules that we will define in the following subsections.

```
type PureRule = InAttrs → OutAttrs
```

The input attributes of a production are the inherited attributes of the parent, the synthesized attributes of the children and the terminal attributes of the production.

```
type InAttrs = (AttrMap I, ChildrenAttrs S, AttrMap T)
```

The output attributes are the synthesized attributes of the parent and the inherited attributes of the children.

```
type OutAttrs = (AttrMap S, ChildrenAttrs I)
```

As we said in the previous section, an AG denotes a function on a tree that takes the inherited attributes of the root and returns the synthesized attributes of the root. In our encoding this is represented by the following type:

```
type SemTree = AttrMap I → AttrMap S
```

The semantics of a production can be modeled as a function that takes the semantic functions of the children (subtrees) and the values of the terminal attributes, and returns the semantic function (*SemTree*) of the tree with this production as root.

```
type SemProd = Child :-> SemTree → AttrMap T → SemTree
```

The function *sem_prod* ties the knot of attribute computation; it takes a rule and produces the semantics of a production.

```
sem_prod :: PureRule → SemProd
sem_prod rule semch terminals inh = syn
  where
    (syn, inh_children) = rule (inh, syn_children, terminals)
    syn_children = applyMap semch extended_inh
    extended_inh = Map.union inh_children
                             (constantMap emptyAttrs
                                          (Map.keysSet semch))
```

[4] *GHC.Prim.Any*

Given the rule (*rule*), the semantics of the children (*semch*), the terminal attributes (*terminals*) and the inherited attributes of the parent (*inh*), :

- Compute the synthesized attributes (*syn*) and the inherited attributes of the children (*inh_children*) by applying the rule to the inherited attributes of the parent, the synthesized attributes of the children (*syn_children*) and the terminal attributes.
- Compute the synthesized attributes of the children (*syn_children*) by applying the semantics of the children to their inherited attributes (*inh_children*). Since the rule may not produce attributes for all the children of the production, we need to extend the domain of *inh_children* (with empty attributions) to cover all of them.

Notice that the definition is cyclic, since we use *syn_children* to produce *inh_children* and we use *inh_children* to produce *syn_children*.

A *pure aspect*, that groups together rules that define related attributes across multiple productions, is defined as a mapping from productions to pure rules.

```
type PureAspect = Production :-> PureRule
```

Thus, the semantics of a pure aspect is defined by applying *sem_prod* to its rules.

```
sem_asp = Map.map sem_prod
```

3.3 Contexts

In order to check that the rules are compatible with a grammar, we must collect information about the rules, like which attributes are used and of which type. While building attribution rules, we build a system of inference rules that we call *Context*. The context is formed of a set of premises and a set of conclusions. The meaning of the context is that the conjunction of premises entails the conjunction of conclusions.

```
data Context = Ctx { ensure_I  :: Set Ensure_I
                   , ensure_S  :: Set Ensure_S
                   , require_I :: Set Require_I
                   , require_S :: Set Require_S
                   , require_T :: Set Require_T }
```

The premises are composed by the inherited (*require_I*), synthesized (*require_S*) and terminal (*require_T*) attributes that are used by the rules. The conclusions are the inherited (*ensure_I*) and synthesized (*ensure_S*) attributes that are defined by the rules. The premises are captured by the *Require* types and the conclusions by the *Ensure* types, which are all constraints that keep track of the used/defined attribute (hiding its type a) and the place of the grammar where it occurs.

```
data Constraint k t where
  Constraint :: Typeable a ⇒ Attr k a → t → Constraint k t

type Ensure_I  = Constraint I Child
type Ensure_S  = Constraint S Production
type Require_I = Constraint I NonTerminal
type Require_S = Constraint S NonTerminal
type Require_T = Constraint T Production
```

Contexts form a monoid, where *mempty* is a *Context* with all its *Constraint* sets empty and *mappend* performs the union of such sets. We define a smart constructor *cstr* to construct singleton constraint sets.

```
cstr :: Typeable a ⇒ Attr k a → t → Set (Constraint k t)
cstr a x = Set.singleton (Constraint a x)
```

3.4 Checking the Contexts

The check if a *Context* is complete with respect to a *Grammar* is to check whether all *require* constraints are met by matching *ensure* constraints, and all terminals are defined in the grammar.

We define a type *Missing* to represent the unmet ensure constraints.

```
newtype Missing = Missing (Set Ensure_I
                          , Set Ensure_S
                          , Set Ensure_T)
```

For instance, given a set of children *chn* and a set of ensured inherited attributes *ensure*, the function *missing_I* returns the set of missing ensured inherited attributes to meet a given required inherited attribute r. That is, an ensure constraint for each child with the same non-terminal of the required attribute that does not define it.

```
missing_I :: Set Child → Set Ensure_I → Require_I
          → Set Ensure_I
missing_I chn ensure r@(Constraint a n) =
        Set.difference match_chn match_ensure
  where
    match_chn    = Set.map (λc → Constraint a c) $
                           Set.filter ((≡ n) ∘ child_nt) chn
    match_ensure = Set.filter ((≡ n) ∘ child_nt ∘ constr_obj)
                              ensure
```

The missing ensure constraints are computed as the difference between the following sets:

- *match_chn*, with an ensure constraint for the attribute a for each child whose non-terminal matches with the non-terminal (n) of the require constraint.
- *match_ensure*, the subset of the ensure constraints for children with the same non-terminal n of the require constraint r.

Thus, in the function *missing*, that returns all the unmet ensure constraints, the set of not ensured inherited attributes is computed by applying the function *missing_I* for all the children of the grammar and the ensured inherited attributes of the context to every required inherited attribute of the context. Since every ensure constraint of the context implies a require constraint, the function *require_all_I* returns the set of required inherited attributes (*Set Require_I*) of the context, including all its *Require_I* and *Ensure_I* constraints with the last ones are transformed to *Require_I*.

```
missing :: Grammar → Context → Missing
missing g c = Missing
  (foldMap (missing_I (gram_children g) (ensure_I c))
           (require_all_I c)
  ,foldMap (missing_S g (ensure_S c)) (require_all_S c)
  ,missing_T g (require_T c))
```

The set of not ensured synthesized attributes is computed by the function *missing_S*, which is similar to *missing_I*. This case of terminal attributes is different, since they are not associated with non-terminals but with productions.

```
missing_T :: Grammar → Set Require_T → Set Ensure_T
missing_T g required =
  Set.difference (Set.filter is_prod required) (ensure_T g)
  where is_prod r = constr_obj r `Set.member` g
```

We also remove the constraints that mention productions that are not in the grammar.

3.5 Aspects

To construct a *Context* in parallel with the rule, instead of directly defining pure rules, rules are defined in an applicative *AR*, that comes with primitives to project terminal attributes or attributes from either the parent or a child of the production. Those primitives generate *require* and *ensure* constraints, or throw errors if necessary.

In order to compute rules, we must first check that they are valid. Rules are defined in the context of a production, and may fail if some constraints are not met; e.g. using a child that is not a valid child of the current production. And lastly, we collect constraints. We define the *Aspect Monad*: a *MonadReader* that reads from a *Production*, a *MonadError* that reports errors of type *Error* and a *MonadWriter* that constructs a *Context*.

```
newtype A a = A (ReaderT Production
                         (ExceptT Error (Writer Context))
                         a)
```

Given a production, the monadic computation can be *run* to get the computed value (or an *Error*[5]) and the context constructed so far out of the monad.

```
runA :: A a → Production → (Check a, Context)
runA (A a) = runWriter ∘ runExceptT ∘ runReaderT a
```

Within this monad we can, for example, define a function to add a constraint to the context requiring that a given terminal attribute has to be provided at the current production.

```
require_terminal :: Attr T a → A ()
require_terminal a = do p ← ask
                        tell $ mempty {require_T = cstr a p}
```

We generate errors if a child is not valid in the current production.

```
assert_child :: HasCallStack ⇒ Child → A ()
assert_child c = do
  p ← ask
  unless (c ∈ prod_children p)
         (throwErrorA (Error_Rule_Invalid_Child c p))
```

If the child *c* does not belong to the children of the production *p*, the function *throwErrorA* throws the given error, including the most recent call-site off the call stack. The *CallStack* is used to point to the location in the source code where the error occurs. Then we can define a function to add a constraint requiring an inherited attribute in a given child.

```
require_child :: HasCallStack ⇒ Child → Attr S a → A ()
require_child c a = do
  assert_child c
  tell $ mempty {require_S = cstr a (child_nt c)}
```

Similarly, we can add an ensure constraint.

```
ensure_child :: Child → Attr I a → A ()
ensure_child c a = do
  assert_child c
  tell $ mempty {ensure_I = cstr a c}
```

In order to gather information from the use of input attributes, we define the rules in a specific monad in which the input attributes are accessed through primitives. The *Rule Monad* is a *MonadReader* reading from the input attributes.

```
newtype R a = R {runR :: Reader InAttrs a}
```

By combining an *Aspect Monad* and a *Rule Monad* we define the *Attribution Rule* applicative functor.

```
newtype AR a = AR {runAR :: A (R a)}
```

The *AR* primitives we introduced in the previous section are implemented using the *Aspect Monad* to collect the constraints and the *Rule Monad* to access to the input attributes. For instance, the function *ter* to project a terminal attribute imposes a *require_terminal* of the attribute and looks it up at the terminal attributes mapping.

```
ter :: HasCallStack ⇒ Attr T a → AR a
ter a = AR $ do
  require_terminal a
  return $ fromJust ∘ lookupAttr a <$> asks (λ(_, _, t) → t)
```

When searching the attribute we apply *fromJust* to the result, assuming that it will be found. Also in the function *lookupAttr* we do an unsafe conversion. These are not problems, because the AG will be checked before we can evaluate it.

```
lookupAttr :: Attr k a → AttrMap k → Maybe a
lookupAttr a@Attr { } m =
  fromJust ∘ fromDynamic <$> Map.lookup (Attribute a) m
```

Another example of *AR* primitive is the operator (!), to project a synthesized attribute from a child. In this case we require that the child must belong to the production and that it has to include the attribute.

```
(!) :: HasCallStack ⇒ Child → Attr S a → AR a
  c ! a = AR $ do
    require_child c a
    return $ fromJust ∘ lookupAttr a ∘ fromJust ∘ Map.lookup c <$>
             asks (λ(_, ch, _) → ch)
```

[5]The datatype *Error* includes a constructor for each possible error to report.

A *Rule* is an attribution rule *AR* that computes output attributes.

```
type Rule = AR OutAttrs
```

Empty rules are rules that produce no output; i.e. an empty mapping of synthesized attributes and an empty mapping of inherited attributes of the children. An empty rule has no effect on the context.

```
emptyRule :: Rule
emptyRule = pure (empty, empty)
```

The rules are merged by the union of their attribute mappings. Merging rules whose domain overlap is an error.

```
mergeRule :: HasCallStack ⇒ Rule → Rule → Rule
mergeRule left_rule right_rule
  | ¬ (Set.null duplicate_inh) = AR $ throwErrorA $
        Error_Rule_Merge_Duplicate_I duplicate_inh
  | ¬ (Set.null duplicate_syn) = AR $ throwErrorA $
        Error_Rule_Merge_Duplicate_S duplicate_syn
  | otherwise = liftA2 mergeOutAttrs left_rule right_rule
  where duplicate_inh = ...
        duplicate_syn = ...
mergeOutAttrs (s1, ic1) (s2, ic2) =
  (Map.union s1 s2, Map.unionWith Map.union ic1 ic2)
```

An *Aspect* maps productions to rules.

```
newtype Aspect = Aspect (Production :-> Rule)
```

Aspects form a monoid, where the binary operator combines the rules of the two aspects by merging them.

```
instance Monoid Aspect where
  mempty = Aspect empty
  Aspect a1 <> Aspect a2 = withFrozenCallStack $
                  Aspect $ Map.unionWith mergeRule a1 a2
```

The primitives to define an aspect for a single attribute, given an *AR* with the attribute computation, add to the context the respective ensure constraints. For example, the function *syn1*, to define a synthesized attribute *attr* for a production *prod* with the computation *rule*, returns an *Aspect* that contains a singleton mapping from the production *prod* to an *AR* that adds an *Ensure_S* of *atr* to the context produced by *rule*. Note that we set the production of the rule using *local*. The rule results in a singleton mapping from the attribute *attr* to the result of its computation, which is stored as a *Dynamic* object.

```
syn1 :: Typeable a ⇒ Attr S a → Production → AR a → Aspect
syn1 atr prod rule
  = Aspect $ Map.singleton prod $ AR $ do
    rule' ← local (const prod) (ensure_parent atr ≫ runAR rule)
    return $ do r ← rule'
                return (Map.singleton (Attribute attr) (toDyn r)
                       , empty)
```

Thus, the primitive *syn* of Section 2.3 merges the aspects resulting from applying *syn1 a* to all the elements of the list.

```
syn :: Typeable a ⇒ Attr S a → [(Production, AR a)] → Aspect
syn a = foldl (λrs (p, r) → rs # syn1 a p r) emptyAspect
```

3.6 Checking the Rules

When the rule set is deemed complete, we can check its context with respect to a grammar. The function *checkAspect* takes a grammar and an aspect, returning an *Error* monad that computes the pure aspect and the produced context if no errors are found.

```
checkAspect :: Grammar → Aspect → Check (PureAspect, Context)
checkAspect gram rule = do
  check_grammar gram
  let (check_aspect, ctx) = runAspect rule
  pure_asp ← check_aspect
  check_missing (missing gram ctx)
  return (pure_asp, ctx)
```

First, we use *check_grammar* to check that the children have unique names for each production. Then we obtain the pure rule (if it has no errors) and context with *runAspect*. With the function *missing*, from Section 3.4, we check that the context is complete with respect to the grammar. If some constraints are unmet, *check_missing* throws an error.

3.7 Running the Attribute Grammar

In order to run the AG in a type-safe way, we must check that a concrete type is compatible with the context free grammar. We do this verification at runtime, but once and for all. If everything is ok we obtain a safe semantic function. Recall the type of the function *run* of Section 2.6.

```
run :: Typeable t ⇒ GramDesc t → InhDesc i → SynDesc s
                  → Aspect → Check (t → i → s)
```

The type *GramDesc t* associates a grammar to a family of concrete types, where *t* is associated to the start symbol of the grammar; i.e. the root of the tree will have type *t*.

```
newtype GramDesc t = GramDesc { runGramDesc ::
  Check (NonTerminal, Grammar, Child :-> TypeRep, GramMap) }
```

A *GramDesc* is an *Error* monad that computes a quadruple containing: the root non-terminal of the grammar, the grammar, a mapping from the children of the grammar to the representation of their corresponding concrete type[6], and a *GramMap* mapping each non-terminal with the information of its associated type.

```
type GramMap = NonTerminal :-> (TypeRep, Dynamic → Match)
```

For each non-terminal we have to provide the representation of its associated type, and a function that *destructs* a *Dynamic* that wraps a value of this type into a *Match*.

```
data Match = Match { match_prod :: Production
                   , match_child :: Child :-> Dynamic
                   , match_terminals :: AttrMap T }
```

The *Match* information includes the associated production, a map from the children of the production to their corresponding values (into *Dynamic*), and an *AttrMap* containing the non-terminals.

[6] This is used to check that type of the non-terminal associated with each child corresponds to the type of the child.

The library provides a function *gramDesc* and a couple of combinators (|= and &) to construct a value of type *GramDesc* as we showed in Section 2.6.

In order to hide the *AttrMap* mappings from the user, we have to specify an association between the inherited and synthesized attributes of the root and two given types i and s, respectively. With *SynDesc* and *InhDesc* we provide an interface to define this association and build conversion functions ($i \to AttrMap\ I$) and ($AttrMap\ S \to s$). Both are *Writer* monads that accumulate the mentioned attributes, in order to be able to check that they are correct with respect to the rule.

In the case of the synthesized attributes, *SynDesc* accumulates a set of synthesized attributes and returns a conversion function ($AttrMap\ S \to a$).

```
newtype SynDesc a = SynDesc {runSynDesc ::
  Writer (Set (Attribute S)) (AttrMap S → a)}
```

When we *project* a synthesized attribute, the conversion function is to look it up into the *AttrMap*. Notice that again we use *fromJust* because we know that its existence will be checked.

```
project :: Attr S a → SynDesc a
project a = SynDesc $ do
  tell (Set.singleton (Attribute a))
  return $ fromJust ∘ lookupAttr a
```

In the case of the inherited attributes, *InhDesc* accumulates a list of inherited attributes and returns a conversion function ($a \to AttrMap\ I$).

```
newtype InhDesc t = InhDesc {runInhDesc ::
  Writer ([Attribute I]) (t → AttrMap I)}
```

To *embed* an inherited attribute is to apply a function that *extracts* the value of the attribute from i and insert it as a *Dynamic* into the *AttrMap*.

```
embed :: Typeable a ⇒ Attr I a → (i → a) → InhDesc i
embed a p = InhDesc $ do
  tell [Attribute a]
  return $ Map.singleton (Attribute a) ∘ toDyn ∘ p
```

To get the conversion functions we have to run the monads and return their result.

```
proj_S :: SynDesc a → AttrMap S → a
proj_S = fst ∘ runWriter ∘ runSynDesc
proj_I :: InhDesc i → i → AttrMap I
proj_I = fst ∘ runWriter ∘ runInhDesc
```

The function *check* returns a computation that checks the whole attribute grammar.

```
check :: GramDesc t → InhDesc i → SynDesc s → Aspect
      → Check (NonTerminal, PureAspect, GramMap)
check g i s r = do
  (root, grammar, gmap) ← check_gramDesc g
  (pure_asp, ctx)       ← checkAspect grammar r
  check_inh_unique i
  check_inh_required i root (require_I ctx)
  check_syn_ensured s grammar root (ensure_S ctx)
  return (root, pure_asp, gmap)
```

We perform the following checks:

- *check_gramDesc*, the concrete types associated to every non-terminal correspond to the types of the children of these non-terminals.
- *checkAspect*, the aspect is complete with respect to the grammar.
- *check_inh_unique*, there are no duplicated inherited attributes specified for the root.
- *check_inh_required*, all the required inherited attributes have been specified for the root.
- *check_syn_ensured*, all the synthesized attributes accessed from the root are ensured by the rules.

The function run performs the checks and returns the semantic function.

```
run :: Typeable t ⇒ GramDesc t → InhDesc i → SynDesc s
                  → Aspect → Check (t → i → s)
run g i s a = do
  (root, pure_asp, gmap) ← check g i s a
  let coalg = Map.map snd gmap
  let sem = sem_coalg (sem_asp pure_asp) coalg root ∘ toDynG g
  return $ λx → proj_S s ∘ sem x ∘ proj_I i
where
  toDynG :: Typeable t ⇒ GramDesc t → t → Dynamic
  toDynG _ = toDyn
```

Once the checks are passed, we proceed to extract the coalgebra of the *Dynamic* tree from the GramMap of the description of the grammar. The coalgebra has the following type.

```
type Coalg = NonTerminal :-> (Dynamic → Match)
```

Then we construct the semantic function using the pure aspect and the coalgebra. To apply the semantic function we convert the tree to *Dynamic*. We also apply the conversion functions to and from *AttrMap*.

The function *sem_coalg* iterates the AG-algebra on the tree-coalgebra.

```
sem_coalg :: Production :-> SemProd → Coalg
          → NonTerminal → Dynamic → SemTree
sem_coalg alg coalg = sem
  where
    sem nt dyn = (alg ! prod) children terms
      where
        Match prod cmap terms = (coalg ! nt) dyn
        children = Map.mapWithKey (λk d → sem (child_nt k) d)
                                  cmap
```

4 GENERIC RULES

An advantage of having first class attribute grammars hosted in a functional language like Haskell, is that we can use it to capture common patterns. Some examples of common patterns in attribute grammars are the copy, collect and chain rules.

The copy rule simply passes down an inherited attribute from the parent to a children. Using the public primitives

```
errors = attr S "errors" tErrors
errorsA = collect errors concat add [leftAdd, rightAdd]
      <> syn errors [val |- pure []]
      <> syn errors [var |- notDefined <$> ter vname <*> par env]
      <> asp
notDefined nm env = if member nm env then [] else [nm]
```

Figure 7: Expressions with errors

```
stmt      = non_terminal "Stmt"
seq       = production stmt "Seq" [leftStmt, rightStmt] nilT
leftStmt  = child seq "leftStmt" stmt
rightStmt = child seq "rightStmt" stmt
asg       = production stmt "Asg" [body] (consT vname nilT)
body      = child asg "body" expr
```

Figure 8: Grammar of the imperative language

```
errorsA' = copyN env [leftStmt, rightStmt, body]
      <> collect errors concat asg [body]
      <> collect errors concat seq [leftStmt, rightStmt]
      <> errorsA
```

Figure 9: Imperative language with errors

```
sState = attr S "sState" tEnv
iState = attr I "iState" tEnv
stateA = chain iState sState seq [leftStmt, rightStmt]
      <> syn sState [asg |- insert <$> ter vname
                                   <*> body ! eval
                                   <*> par iState]
      <> inh env    [body |- par iState]
      <> delete_Is env [leftStmt, rightStmt, body] errorsA'
```

Figure 10: Imperative language with state

of the library we can implement a function *copy* that passes down an inherited attribute to a given child.

```
copy :: Typeable a ⇒ Attr I a → Child → Aspect
copy a c = inh a c (par a)
```

We can also apply *copy* to a list of children for which the attribute is to be copied.

```
copyN :: Attr I a → [Child] → Aspect
copyN = many1 copy
```

Or we can copy the inherited attribute of the parent to all the children that have the same non-terminal.

```
copyP :: Attr I a → Production → Aspect
copyP a p = copyN a cs
  where cs = [c | c ← prod_children p, child_nt c ≡ prod_nt p]
```

Or apply the copy rule for a list of productions.

```
copyPs :: Attr I a → [Production] → Aspect
copyPs a = foldr (λp r → copyP a p <> r) emptyAspect
```

We have already used the *copyPs* copy rule in the definition of the attribute *env* in Section 2.3, Figure 3.

The collect rule applies a function to collect the values of the synthesized attributes of the children. For instance, we can define a function *collect* that applies a function to the attributes of a list of children to compute a synthesized attribute.

```
collect :: Attr S a → ([a] → a) → Production → Children
      → Aspect
collect a reduce p cs = syn a p $ reduce <$> traverse (!a) cs
```

As an example of the use of the *collect* rule, let us extend the semantics of the expression language with an *errors* attribute, that verifies that the variables used exist in the environment. The result of computing such attribute is a list with the names of the undefined variables. In Figure 7 we show the implementation of this extension, where we use *collect* to define the computation of *errors* for the production *add* as the concatenation of the values of *errors* of its children.

Now suppose we want to define a (very simple) imperative language, that consists of sequences of assignments, where variables are considered as declared if a value have been assigned to them. We define the grammar of the language in Figure 8. Notice that we use the non-terminal *expr* of the expression language. We show in Figure 9 how by using *copy* and *collect* rules, the *errors* semantics can be extended to the language. In this extension we are assuming a global environment (the attribute *env*), and the assignments have no effect.

If we want to define the semantics of the imperative language as a state transformer, then we need to *chain* the computation of the state through the sequence of statements. The chain rule takes a pair of an inherited and a synthesized attribute and threads them through the children of a production: the parent attribute is given to the first child, the attribute of the first child is given to the second and so on, the attribute of the last child is given back to the parent. So, this rule defines the inherited attribute for all the children and the synthesized attribute for the parent.

```
chain :: Typeable a ⇒
  Attr I a → Attr S a → Production → Children → Aspect
chain i s p cs = (inh i $ zip cs $ par i : ((!s) <$> init cs))
            <> syn s p [last cs ! s]
```

In Figure 10 we show how by defining a pair of attributes *sState* and *iState* to represent the state, and using the *chain* rule to pass the state through the children of the sequences, we can implement the semantics of the language. Notice that in this case the environment *env* of each expression is the actual (inherited) state, and that we had to delete the previous definitions of *env*.

5 RELATED WORK

Starting with Johnson [8], there is a long list of works on encoding AGs in pure functional languages with lazy evaluation. Our embedding improves the compositional semantics of AGs given by de Moor et.al. [4] by adding extensibility and type-safety.

Other embeddings of AGs in Haskell are based on the use of extensible records [5] or type-level programming techniques [2, 16] that use advanced type system extensions of GHC. In those embeddings it is possible to perform correctness checks at compile time, but with the cost of poor quality error messages. In our approach, since we implement ourselves the checks of the AG, we have more control over the error messages. Moreover, by using Template Haskell we are able to perform such checks at compile time.

Some other embeddings use a *Zipper* [7] to model AGs [1, 6, 10, 11, 15]. All of them work over grammars represented by concrete datatypes, not being able to solve the expression problem[17]. In fact, once the datatype representation of the grammar is fixed it is not possible to extend the grammar with new productions. In contrast, in our embedding we solve the expression problem, because we allow to both extend the grammar (by adding productions and nonterminals) and the semantics (by adding attributes).

While staging is a well-known program generation technique usually based on quotation operators [13, 14], we rely on the idea of stage separation, but focusing on the implementation of a domain specific nontrivial type system and the generation of good quality error messages.

6 CONCLUSIONS

We have introduced an embedding of AGs as a domain specific language in Haskell. Embedding AGs in a host language has many advantages over the alternative of using a specific tool. In particular the host language and its libraries are readily available to the grammar writer for use when expressing the computation rules of attributes. Another benefit of the embedding is that it becomes possible to run an AG within a program, and hence compute the semantics of a syntax tree. This extends the host language with the AG paradigm for writing recursive tree computations.

Our embedding of modular AGs does not compromise on safety, on the simplicity of the public interface and the clarity of error messages. The approach is to have two phases of typechecking: (1) the static phase uses the host typesystem to capture as many invariants as possible, (2) the runtime phase implements a *global* verification of the domain specific program (the AG) before running it. This global verification captures the remaining constraints that were not caught at compile time, and allows us to precisely identify the source of the problem. In addition, after the runtime global checks, the embedded program may be run without dynamic checks, thus speeding up execution.

We believe that our approach to domain specific errors may be successfully applied to define other domain specific language embeddings, being this a possible direction of future research.

It would be also interesting to measure the impact in performance of moving the checks to runtime, and compare our library with other embbedings. Another interesting line of future research, concerning the embedding or AGs, is to perform more checks, like detecting cycles or scheduling the evaluation. Since our embedding is a *shallow embedding*, is not trivial to perform such kind of analysis.

REFERENCES

[1] E. Badouel, R. Tchougong, C. Nkuimi-Jugnia, and B. Fotsing. 2013. Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition. *Theoretical Comp. Science* 480, 0 (2013), 1 – 25.

[2] Florent Balestrieri. 2014. *The Productivity of Polymorphic Stream Equations and the Composition of Circular Traversal.* Ph.D. Dissertation. University of Nottingham.

[3] Florent Balestrieri, Alberto Pardo, and Marcos Viera. 2018. SafeAG library. https://github.com/balez/Safe-AG. (2018). Accessed: 2019-02-10.

[4] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. 2000. First Class Attribute Grammars. *Informatica: An International Journal of Computing and Informatics* 24, 2 (June 2000), 329–341. Special Issue: Attribute grammars and Their Applications.

[5] Oege de Moor, L. Peyton Jones, Simon, and Van Wyk, Eric. 2000. Aspect-Oriented Compilers. In *Proceedingsof the 1st Int. Symposium on Generative and Component-Based Software Engineering*. Springer-Verlag, London, UK, 121–133.

[6] João Paulo Fernandes, Pedro Martins, Alberto Pardo, João Saraiva, and Marcos Viera. 2019. Memoized zipper-based attribute grammars and their higher order extension. *Science of Computer Programming* 173 (2019), 71 – 94. Brazilian Symposium on Programming Languages (SBLP '15+16).

[7] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554.

[8] Thomas Johnsson. 1987. Attribute grammars as a functional programming paradigm. In *Proceedingsof the Functional Programming Languages and Computer Architecture*. Springer-Verlag, London, UK, 154–173.

[9] Donald Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (June 1968). *Correction: Mathematical Systems Theory* 5 (1), March 1971.

[10] Pedro Martins, João Paulo Fernandes, and João Saraiva. 2013. Zipper-Based Attribute Grammars and Their Extensions. In *Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings (Lecture Notes in Computer Science)*, André Rauber Du Bois and Phil Trinder (Eds.), Vol. 8129. Springer, 135–149.

[11] Pedro Martins, João Paulo Fernandes, João Saraiva, Eric Van Wyk, and Anthony Sloane. 2016. Embedding attribute grammars and their extensions using functional zippers. *Sci. Comput. Program.* 132 (2016), 2–28.

[12] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

[13] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75.

[14] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211 – 242. PEPM'97.

[15] Tarmo Uustalu and Varmo Vene. 2005. Comonadic functional attribute evaluation *(Trends in Functional Programming)*. Intellect Books (10), 145–162.

[16] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. 2009. Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell. In *Proceedingsof the 14th Int. Conf. on Functional Programming*. ACM, New York, USA, 245–256.

[17] Phil Wadler. 1998. The Expression Problem. (1998). http://www.daimi.au.dk/~madst/tool/papers/expression.txt E-mail available online.

Extended Memory Reuse

An Optimisation for Reducing Memory Allocations

Hans-Nikolai Vießmann
Heriot-Watt University
Edinburgh, UK
hv15@hw.ac.uk

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, UK
a.sinkarovs@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, UK
S.Scholz@hw.ac.uk

ABSTRACT

In this paper we present an optimisation for reference counting based garbage collection. The optimisation aims at reducing the total number of calls to the heap manager while preserving the key benefits of reference counting, *i.e.* the opportunities for in-place updates as well as memory deallocation without global garbage collection. The key idea is to carefully extend the lifetime of variables so that memory deallocations followed by memory allocations of the same size can be replaced by a direct memory reuse. Such memory reuse turns out particularly useful in the context of innermost loops of compute-intensive applications. It leads to a runtime behaviour that performs pointer swaps between buffers in the same way it would be implemented manually in languages that require explicit memory management, *e.g.* C.

We have implemented the proposed optimisation in the context of the Single-Assignment C compiler tool chain. The paper provides an algorithmic description of our optimisation and an evaluation of its effectiveness over a collection of benchmarks including a subset of the Rodinia benchmarks and the NAS Parallel Benchmarks. We show that for several benchmarks with allocations within loops our optimisation reduces the amount of allocations by a few orders of magnitude. We also observe no negative impact on the overall memory footprint nor on the overall runtime. Instead, for some sequential executions we find mild improvement, and on GPU devices we observe speedups of up to a factor of 4×.

CCS CONCEPTS

• **Software and its engineering → Memory management**; *Compilers*; Functional languages;

KEYWORDS

memory management, reference counting, compiler optimisation

ACM Reference Format:
Hans-Nikolai Vießmann, Artjoms Šinkarovs, and Sven-Bodo Scholz. 2018. Extended Memory Reuse: An Optimisation for Reducing Memory Allocations. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310242

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09.
https://doi.org/10.1145/3310232.3310242

1 INTRODUCTION

Many modern programming languages use implicit memory management. Languages such as SISAL [4, 5], Python [10], Swift [18, 27], or Single-Assignment C (SaC) [12] use reference counting to implement non-delayed garbage collection. The key idea of reference counting is to associate each data structure with a counter that keeps track of the number of shared references that exist in the system. Once the last reference to a data structure is no longer needed the corresponding memory can be freed. For the price of maintaining such a reference counter for each data structure, this approach not only enables earliest possible asynchronous garbage collection, it also provides an elegant solution to the aggregate update problem [17]: whenever a reference counter is one, the corresponding data structure can be updated in place.

Several optimisations to reference counting have been proposed in the literature. They predominantly focus on avoiding reference counter maintenance overheads [7, 21, 25, 30] and on maximising the potential for in-place updates of aggregate data structures [8, 13]. Some other works aim at improving the actual memory handling process, be it in the context of sequential or parallel executions [19, 31].

This paper proposes a novel optimisation for reference counting based garbage collection named *extended memory reuse* (*EMR*). It extends the lifetime of memory allocations to avoid sequences of deallocations and reallocations in favour of a direct memory reuse. While the overheads of such dual calls to the memory manager on shared-memory systems usually has little impact on the overall runtime performance, in the context of accelerator systems, like GPUs, they can have devastating effects on the overall performance.

Accelerator systems typically have separate memory contexts, one for the host and one on the accelerator, and all memory management, including memory allocations on the accelerator, has to be performed by the host [24]. Any memory deallocation or reallocation that has to happen between two executions on the GPU requires a control transfer from GPU to the host and back. Moreover, the change in allocated memory on the GPU may induce superfluous data-transfers between host and device which are known to be the primary source for poor GPU performance [14].

The main contributions of this paper are:

(1) Design of a code-transformation that implements the inference of extended reuse opportunities. We present it as a series of rewrites for a first-order functional language with array comprehensions which resembles the core of SaC.
(2) A full-fledged implementation of the proposed optimisation for the SaC compiler.
(3) Evaluation of the effects of the proposed transformation at the example of benchmarks from the Livermore Loops [23],

Rodinia [6], and NAS [1] Benchmark Suites. We present the effects of *EMR* on these benchmarks with respect to the number of memory operations performed, memory footprint, and overall runtime for two different architectures: a shared memory system and a GPU accelerated system.

Our results suggest that the object lifetime extension pattern that we make use of can be picked up in a wider contexts like static analysers for languages with manual memory allocations like C. Specifically, this becomes desirable in the context of GPU or FPGA accelerator programming using languages like CUDA or OpenCL.

The rest of the paper is organised as follows: in section 2 we briefly describe the SaC language and explain how reference counting works within the language by giving examples. In section 3 we extend on these examples to demonstrate the problem we tackle and how we intend to solve it. In section 4 we provide an overview of the code-transformations and what they are meant to achieve. This is followed by explanations on how the code-transformations work, with some examples given to demonstrate this. In section 5 we present and evaluate the results from applying the *EMR* optimisation on various benchmarks. In section 6 we give a discussion on certain aspects of the optimisation, such as its effect on heap usage during runtime. In section 7 we present and compare some related work, and finally in section 8 we give our conclusion and ideas for future work.

2 BACKGROUND

We start with a brief overview of the SaC language and its reference counting model. The main goal here is to introduce a large enough subset of the language so that we can conveniently illustrate the underlying problem and our solution. For a full introduction consider [11, 29].

2.1 The Core of SaC

SaC is a first-order functional language with built-in arrays. In SaC, arrays are predominantly constructed by an array comprehension construct called the *with*-loop. In this paper we assume the sole construct for array construction to be a restricted form of the *with*-loop which has the following form[1]:

$$\{\ \mathtt{x}\ \texttt{->}\ e_1\ \mid\ \mathtt{x}\ \texttt{<}\ e_2\ \}$$

where x is a variable and e_1 and e_2 are expressions. `x -> `e_1 can be seen as a mapping from indices x to element values e_1 which define the array elements for all its legal indices. The range of legal indices is delimited by e_2 which evaluates to a vector that defines the shape of the resulting array.

A few instances of such *with*-loops are contained in our running example shown in listing 1. Consider line 4, where we use a *with*-loop to implement element-wise addition of the 5-element 1-d arrays a and b. The index iv ranges over the indices {[0],[1],[2],[3],[4]}, and we compute a 5-element 1-d array c where at every index iv the value is computed by the expression a[iv] + b[iv]. The square brackets here denote element selections from a and b, respectively.

The *with*-loops in lines 8 and 12 compute rotations of the array a using the built-in modulo operator denoted by the %-symbol.

Other components of the language are first-order functions and conditionals. As shown in our running example, we use SaC syntax for function definitions and function applications. Partial applications of functions are not supported. Conditional expressions are restricted to the ternary ?: operator, whose syntax is identical to the corresponding construct in C. In contrast to full-fledged SaC, we do not support special syntax for loops. We resort to recursive functions instead in order to keep our language core concise. The function stencil in lines 19–25 together with its call in line 15 of our running example constitutes such a loop. It represents a do-loop that performs 10 iterations of a two-point stencil with cyclic boundary conditions.

```
1  int[5] fun (int[5] a, int[5] b)
2  {
3    // element-wise add
4    c = { iv -> a[iv] + b[iv] | iv < [5] };
5    print (c);
6
7    // rotate right by one element
8    d = { iv -> a[(iv-1) % 5] | iv < [5] };
9    print (d);
10
11   // rotate left by one element
12   e = { iv -> a[(iv+1) % 5] | iv < [5] };
13
14   // 10 iterations 2 point stencil
15   f = stencil (e, 10);
16   return f;
17 }
18
19 int[5] stencil (int[5] e, int n)
20 {
21   f = { iv -> e[(iv-1) % 5] + e[(iv+1) % 5]
22         | iv < [5] };
23
24   r = n == 0 ? f : stencil (f, n-1);
25   return r;
26 }
```

Listing 1: SaC code example

2.2 Basic Reference Counting in SaC

Arrays defined by *with*-loops initially have a reference count of 1. New references to arrays are generated whenever they are passed as arguments to abstractions. At that point one reference is consumed and as many references are generated as there are references to the formal parameter within the body of the abstraction. Our core language here supports three forms of abstractions: functions, assignments to variables which constitute nested let-constructs, and *with*-loops themselves as they replicate their element-expression for each array element. Built-in operations like +, selection, *etc.*,

[1] In SaC, a similar syntactical form exists as a notational short-cut for *with*-loops named set-expression [26]. The form we use here can be expanded into a proper *with*-loop by applying the following expansion rule:

$[\![\{\ \mathtt{x}\ \texttt{->}\ e_1\ \mid x < e_2\ \}]\!] = \mathtt{with}\ \{(0 * e_2 <= x < e_2)\texttt{:}\ e_1;\ \}\texttt{: genarray}\ (e_2, 0)$

consume their arguments. At runtime, when the reference count of an array is decreased to 0, the object is deallocated.

Consider our running example from listing 1 again, we extend it in listing 2 to show function fun annotated with explicit reference count operations (*e.g.* incrc).

```
1  int WLBc (int[1] iv, int[5] a, int[5] b)
2  {
3    incrc (iv, 1); incrc (a, 0); incrc (b, 0);
4    return a[iv] + b[iv];
5  }
6  int WLBd (int[1] iv, int[5] a) { /* ... */ }
7  int WLBe (int[1] iv, int[5] a) { /* ... */ }
8  int[5] fun (int[5] a, int[5] b)
9  {
10   incrc (a, 2);          // 3 references
11   incrc (b, 0);          // 1 reference
12   // ——— WL c ———
13      incrc (a, 4);       //  a s in c with-loop body
14      incrc (b, 4);       //  b s in c with-loop body
15      c = {
16        iv -> WLBc (iv, a, b)
17        | iv < [5]
18      };                  // rc (c) == 1
19   incrc (c, 0);          // referenced in print-call
20   print (c);
21
22   // ——— WL d ———
23   // ...                 // rc (d) == 1
24   incrc (d, 0); print (d);
25
26   // ——— WL e ———
27   // ...                 // rc (d) == 1
28   incrc (e, 0);          // referenced in stencil-call
29   f = stencil (e, 10);
30   incrc (f, 0);          // referenced in return
31   return f;
32 }
```

Listing 2: Example from listing 1 with reference counting

At the beginning of the function we adjust the reference counts of the formal parameters of fun. Array a is referenced 3 times and array b is referenced once. As fun on application consumes its arguments we increment the reference counts of a and b by 2 and 0, respectively. This is because all function arguments have at least a reference count of 1. After each assignment, the reference counts of the let-bound variables are adjusted. In our example, all variables are used exactly once, resulting in increments by 0, *i.e.* no adjustments at all. Every *with*-loop allocates new memory for its result with a reference count of one.

In lines 10–16 we show the reference counting of the first *with*-loop. Each element computation acts like a function call. The conceptual arguments of those functions are the index variable (here iv) and the relatively free variables (here a and b). We make this idea explicit by abstracting the body of all *with*-loops into corresponding functions WLBc, WLBd and WLBe.

Before starting the computation of the array c, the reference counts of the relatively free variables are adjusted by the number of array elements minus one. Within the element computation (the body of WLBc function), as iv occurs twice in a[iv] + b[iv], we increment the reference count of iv by 1. Similarly, the reference counts of a and b are left unmodified as they occur exactly once in the element expression.

Consider an execution where fun is called with reference counts of 1 for both a and b. When reaching the *with*-loop computation in line 14, the reference counts of a and b have been changed to 7 and 5, respectively. Within each element computation, the reference counts of a and b are decremented by one at the end of each corresponding selection. Since we have 5 element computations, the reference count of a is going to be reduced to 2 and b is going to be freed directly after the last element selection.

The *with*-loops in lines 20 and 25 are treated in the same way; so we have omitted the details in listing 2.

Note here, that in this example the arrays c and d will be freed immediately after they are printed.

Reference counting for conditionals requires reference count adjustments of all relatively free variables whenever the individual alternatives are being computed. A more detailed account of reference counting in SaC can be found in [12].

2.3 Improved Reference Counting in SaC

The basic reference counting in SaC has been significantly extended by several optimisations [2, 28]. Our implementation is an extension of the proposed mechanisms that builds on the memory reuse optimisations in [12, 13].

As can be seen from the naïve reference counting in listing 2, we always allocate fresh memory for the result of a *with*-loop. Looking at the example of the *with*-loop in lines 10–16, we can observe that the array b would actually be an excellent choice for memory reuse in all those cases where fun is called with an argument b with reference count 1. Specifically, we could avoid allocating arrays c, d, and e and reuse the memory of array b instead. Not only would it avoid an extra allocations and deallocations, it would also improve the cache locality of the overall code.

To enable such an optimisation we identify arrays that are referenced in the *with*-loop body that

(1) have the same shape as the *with*-loop,
(2) have reference count one in the beginning of the *with*-loop, and
(3) have a well-behaved access pattern.

Techniques to identify the latter can be found in [12, 13]. We refer to such arrays as *reuse candidates* (*RC*s). Once identified, the reuse candidates are kept until runtime. Once the *with*-loop is executed, the reference counts of the reuse candidates are inspected. As soon as a reference count of 1 is found the corresponding memory is reused. Only if no reuse candidate with reference count 1 is found, is new memory allocated.

3 MOTIVATION FOR *EXTENDED* MEMORY REUSE

In the previous section, we have seen how memory can be reused for computing new arrays from existing ones. While this works

fine for the first *with*-loop of our running example, it does not apply for the three other *with*-loops in listing 1. There the array `c` is deallocated during the call to `print` in line 5 although an array of the same size is needed for the computation of the array `d` in line 8. Similarly, `d` is discarded in line 9 just before an array of the same size is needed in line 12.

The loop function `stencil` in lines 19–25 exposes a similar pattern. While the argument array `e` cannot be reused for `f`, which is passed into the next iteration, it could serve as memory for the next instance of `f` to be computed in the next iteration. If we look at an unrolled version of the stencil operation we can see exactly the same situation as in the function body of the function `fun`:

```
f0 = { iv -> e[(iv-1) % 5] + e[(iv+1) % 5]
       | iv < [5] };
f1 = { iv -> f0[(iv-1) % 5] + f0[(iv+1) % 5]
       | iv < [5] };
f2 = { iv -> f1[(iv-1) % 5] + f1[(iv+1) % 5]
       | iv < [5] };
// ...
```

The memory of `e` can be reused for f_1, the memory of f_0 can be reused for f_2, *etc.* Effectively, this would lead to two memory locations being used for the *with*-loop computation in an alternating fashion.

Our goal here is to achieve the same effect without the need for explicitly unrolling of the loop. This can be achieved by identifying loop functions that contain such *with*-loops and extending their signatures. For our running example, we aim at an extended stencil function of the form:

```
int[5] stencil1 (int[5] e, int n, int[5] f0)
{
  f = { iv -> e[(iv-1) % 5] + e[(iv+1) % 5]
        | iv < [5] };

  r = n == 0 ? f : stencil1 (f, n-1, e);
  return r;
}
```

The extra parameter f_0 effectively provides a pointer to memory that can be reused for the computation of `f`. In the subsequent iteration, this memory is provided from the array `e` whose lifetime is now extended into the next iteration.

Note here that this completely elides the need to interact with the heap manager within this loop. In particular in the context of executions on accelerators such as GPUs we expect to see a noticeable impact for this optimisation.

In the next section, we provide an algorithmic description which systematically transforms programs like our running example into a form where memory is being reused within the scope of individual functions and across loops.

4 EXTENDED MEMORY REUSE

The basic idea for implementing *EMR* is to extend the idea of reuse candidates as outlined in section 2.3. Instead of restricting the compiler to only use arrays as reuse candidates whose last reference is within the with-loop body in a suitable way, we would like to also include arrays that are neither referenced in the loop body nor within the remainder of the function body. We call such arrays *Extended Reuse Candidates* or *ERC*s for short. Once these are identified, the idea is to inspect the reference counts of such ERC at runtime and, similar to the reuse candidates from section 2.3, reuse their memory in case the reference count is 1.

To do so, we first collect arrays that are of suitable size and that have been defined prior to the with-loop in question. This happens in a phase we call *ERC inference*. Once we have collected such potential *ERC*s, we filter out non-suitable candidates in a second phase named *ERC filtering*. Finally, we adjust the function signatures and function calls to the tail-end recursive functions that represent our loops. We call this phase *ERC loop optimisation*.

4.1 Overview

Before providing a more formal description of these three phases, we sketch their intended behaviour by using our running example.

4.1.1 ERC inference. First of all, we annotate every *with*-loop with the set of potential *ERC*s. These are variables that have been defined before the *with*-loop, whose shape is identical to that of the *with*-loop result, and which are not referenced within the *with*-loop itself. For our running example from listing 1, this leads to the following annotations of the *with*-loops with *RC*s and *ERC*s:

```
1  int[5] fun (int[5] a, int[5] b)
2  {
3    c = { /*.*/ a[iv] + b[iv] /*.*/ }; // RC={b} ERC={}
4    print (c);
5
6    d = { /*.*/ a[(iv-1) % 5] /*.*/ }; // RC={} ERC={c,b}
7    print (d);
8
9    e = { /*.*/ a[(iv+1) % 5] /*.*/ }; // RC={} ERC={d,c,b}
10
11   f = stencil (e, 10);
12   return f;
13 }
```

As can be seen, for the *with*-loop in line 3, `b` is a *RC*, as it has a well-behaved access and is not referenced in the function body any further. The array `a` cannot serve as *RC* in line 3 since it is referenced later. In the remaining *with*-loops the array `a` cannot serve as a *RC* due to the access patterns that stem from the rotations.

The *ERC* inference now traverses the functions top to bottom, collects the variables that denote arrays of suitable shape and attaches those as *ERC*s to *with*-loops that are not contained in the corresponding *with*-loop body.

4.1.2 ERC filtering. The *ERC*s inferred by the previous step require further filtering as the inference tends to over-approximate and can sometimes pick candidates which are referenced at a later stage. When an *ERC* is selected for a given *with*-loop, from that point onward it *is* a reference to the resulting array. If at a later stage we select the same *ERC*, we cause a conflict that can lead to an additional allocation. Consequently, an array that serves as an *ERC* for the *with*-loop in line 9 cannot be used as an *ERC* for the *with*-loops in lines 3 or 6; an array that serves as an *ERC* for the *with*-loop in line 6 cannot be used as an *ERC* for the *with*-loop in

line 3. We achieve this by performing a bottom-up traversal that filters out those *ERC*s that are referenced within the remainder of the function body. Once this is done we pick the first remaining *ERC*, provided the *with*-loop does not have an *RC*. All other *ERC*s are filtered out as well. Then we add the chosen *ERC* to the set of variables that are referenced and continue traversing bottom-up. By picking the first *ERC* we ensure that the most recently defined array is always chosen as the *ERC*, which implicitly minimises the extension of array lifetimes. For our running example this results in the following *ERC* annotations:

```
1  int[5] fun (int[5] a, int[5] b)
2  {
3    c = { /*.*/ a[iv] + b[iv] /*.*/ }; // RC={b} ERC={}
4    print (c);
5
6    d = { /*.*/ a[(iv-1) % 5] /*.*/ }; // RC={} ERC={c}
7    print (d);
8
9    e = { /*.*/ a[(iv+1) % 5] /*.*/ }; // RC={} ERC={d}
10
11   f = stencil (e, 10);
12   return f;
13 }
```

4.1.3 ERC loop optimisation. When applying the traversals explained so far to the loop function `stencil`, the *with*-loop within that function remains without an *RC* or *ERC* since there simply is no array that possibly can be reused. The task of this phase is to identify *with*-loops without *RC*s and *ERC*s within loop functions, invent new identifier names, and add these identifiers as *ERC*s to the corresponding *with*-loops. This traversal also adjusts the function signatures of the loop functions as well as all applications of these functions accordingly. For our running example we obtain for the loop function:

```
1  int[5] stencil (int[5] e, int n,
2                  int[5] f0 /* Fresh variable */)
3  {
4    /* RC={}, ERC={f0} */
5    f = { /*.*/ e[(iv-1) % 5] + e[(iv+1) % 5] /*.*/ };
6
7    r =  n == 0
8      ? f
9      : stencil (f, n-1, e /* shape(e) == shape(f0) */);
10   return r;
11 }
```

Note that the choice of variable to expand the recursive function call only needs to match the type and shape of what is needed and it must not be used as existing argument within the recursive call. In what order this variable is created in the function body does not matter.

After all loop functions are extended, we need to extend their call sites as well. Given the type and shape information of the newly added function arguments, we can either find variables in the current context or create new arrays before the function call. For our example, we obtain:

```
9   // ...
10  f0 = { iv -> 0 | iv < [5] };
11  f = stencil (e, 10, f0);
12  // ...
```

4.2 Algorithmic Formulation of the Transformations

We base our transformations on the language core delineated in section 2. More specifically, we make the following assumptions:

(1) a program is a list of functions where each function consists of a return type, arguments and body;
(2) the body of each function is a list of assignment statements in static single assignment form and a single return statement at the end of that list (note that functions like `print` assign the unit type value to some temporary variable);
(3) the right-hand side of each assignment is either a *with*-loop, a function application, or a conditional (all conditionals are in the form $x\ ?\ e_1 : e_2$ where x is a variable of type bool and e_1 and e_2 are expressions);
(4) loop functions are tail-recursive functions where the recursive call has been identified;

Note here, that although this core language is designed to reflect SaC, the only SaC-specific construct is the *with*-loop. Conceptually, this language construct can be replaced by any other language construct that creates a reference-counted data structure. With such a modification our transformation is immediately applicable to any other programming language that uses reference counting as the garbage collection mechanism.

4.2.1 ERC Inference. The *ERC* inference is shown in algorithm 1. It expects a function definition (F_d) and computes *ERC* annotations for all *with*-loops within that function. We represent these annotations as **EC**, a mapping from expressions to variable lists.

During the top-down traversal of the function body we create a local variable list C which we use to store all candidates found so far. We define a local recursive function T that operates on expressions: for *with*-loops it computes the corresponding *ERC*s; for conditionals it recursively inspects both branches. In the former case we filter the current C by removing all the variables that are free in the body of the *with*-loop or whose shapes do not match the shape of the return value of the *with*-loop. As conditionals cannot have local variable bindings, we simply annotate the *with*-loops in the individual branches with the *ERC*s.

The actual top-down traversal is defined in lines 11–12. We iterate through all assignments in F_d, and apply T to *expr* looking for assignments made by either *with*-loops or function applications. Once done with the expression of the assignment, we add the defined variable *var* to the set of candidates C. Once the entire function body has been traversed, all found *ERC*s are stored in **EC** which we will refer to in the subsequent transformations.

4.2.2 ERC Filtering. The *ERC* filtering is shown in algorithm 2. It relies on mappings **EC** and **RC** which hold the *ERC*s inferred by

Algorithm 1: Extended Reuse Candidate Inference

```
   Input: A function definition F_d
1  begin
2      Let EC be an empty mapping
         (expression ↦ variable list)
3      Let C be the found candidates of F_d
4      function T(expr) is
5          switch expr do
6              case expr is a with-loop do
7                  ℰℛC = Filter(
                     x ⇒ x ∉ FreeVariables(expr)
                     and Shape(x) == Shape(expr), C)
8                  Add expr ↦ ℰℛC to EC
9              case expr is a conditional x ? e_1 : e_2 do
10                 T(e_1); T(e_2)
11     foreach (var = expr) in F_d do          /* top-down */
12         T(expr); C = var ++ C
   Output: The updated mapping EC
```

Algorithm 2: Extended Reuse Candidate Filtering

```
   Input: A function definition F_d
          A mapping EC (expression ↦ variable list)
          A mapping RC (expression ↦ variable list)
1  begin
2      Let 𝒱ℛ be the return value of F_d
3      function T(expr) is
4          switch expr do
5              case expr is a with-loop do
6                  ℰℛC = Filter(x ⇒ x ∉ 𝒱ℛ, EC[expr])
7                  𝒱 = FreeVariables(expr body)
8                  if RC[expr] = ∅ ∧ ℰℛC ≠ ∅ then
9                      EC[expr] = ℰℛC_0
10                     𝒱 = ℰℛC_0 ∪ 𝒱
11                 else
12                     EC[expr] = ∅
13                 return 𝒱
14             case expr is conditional x ? e_1 : e_2 do
15                 return T(e_1)++T(e_2)
16             otherwise do
17                 return FreeVariables(expr)
18     foreach (var = expr) in F_d do          /* bottom-up */
19         𝒱ℛ = T(expr) ++ 𝒱ℛ
   Output: The updated mapping EC
```

algorithm 1 and *RC*s for all expressions, respectively. As a result of this bottom-up traversal, the mapping **EC** is being updated.

During the traversal, a list of referenced variables $\mathcal{VR}$ is maintained which is initialised with the function's return value. This list is used to do the actual filtering of *ERC*s for *with*-loops. Similar to the previous algorithm, we define a local function T which we map over all assignments in a bottom-up fashion (lines 18–19). When encountering a *with*-loop, we first filter out all *ERC*s that are referenced in the code below the current assigment (line 6). We then collect the free variables contained in the *with*-loop body in $\mathcal{V}$ which we will append to our global list $\mathcal{VR}$ once we are done with the *with*-loop. If our filtering of **EC**[*expr*] left an *ERC* and there is no *RC*, we pick the first *ERC*, update **EC**[*expr*] accordingly, and we add it to the list of referenced variables in $\mathcal{V}$. Otherwise, we set **EC**[*expr*] to empty. Upon return of T the variables referenced in the *with*-loop including the remaining *ERC* contained in $\mathcal{V}$ are added to $\mathcal{VR}$ before continuing with the bottom-up traversal.

4.2.3 *ERC Loop Optimisation.*

ERC loop optimisation is shown in two algorithms, algorithm 3 and algorithm 4. This split is purely for presentational purposes as we describe our algorithms on a per-function basis and *ERC* loop optimisation requires changes of two functions, the actual loop function and the function that calls the loop-function. The former is described in algorithm 3 and it has to happen before the corresponding loop function call is modified as described in algorithm 4.

Transformation of Loop Functions. This top-down traversal utilises the mappings **EC** and **RC** for *ERC*s and *RC*s for each *with*-loop. Similar to the previous algorithms, we map a local function T successively on all assignments of the function F_d, as shown in algorithm 3. We use a list $\mathcal{TR}$ of variable-expression pairs to store fresh variables that will be used as *ERC*s for *with*-loops without *RC*s or *ERC*s, if possible. When we come across a *with*-loop, we check if it has any *RC*s or *ERC*s. If it has *none*, we create a variable for a new array with the same type and shape as the return value of the *with*-loop and put this jointly with a reference to the *with*-loop into $\mathcal{TR}$. After the body of F_d is traversed, we use $\mathcal{TR}$ to extend the F_d arguments with the variables in $\mathcal{TR}$.

When traversing the recursive call, we iterate over $\mathcal{TR}$ and search the body of F_d for the variables of the corresponding type and shape that can be used as arguments of the extended function call. This search happens in the `FindMatching` helper function. If the search is not successful, we elide the variable-expression pair from $\mathcal{TR}$. Otherwise, we append the found variable $\mathcal{N}_a$ to the arguments of the function application and we add the variable from $\mathcal{TR}$ to *ERC*s of the *with*-loop referred in the variable-expression pair of $\mathcal{TR}$. This way, we make sure the number of added arguments to the recursive call matches the number of entries in $\mathcal{TR}$.

Adjustment of Loop Function Calls. The adjustment of the external loop function calls is shown in algorithm 4. It assumes that all loop functions have already been adjusted through algorithm 3. Again, we iterate through the assignments within the body of F_d again searching for a function call to a loop function. Once found, we check whether the function signature has changed during the application of algorithm 3. If it has, we use a helper function `CreateArgs` to create the missing number of arrays of the right type and shape and append these to the loop function call.

Algorithm 3: *ERC* Loop Optimisation for Loop Functions

Input: A loop function definition F_d
A mapping **EC** ($expression \mapsto variable\ list$)
A mapping **RC** ($expression \mapsto variable\ list$)

begin
  Let $\mathcal{TR}$ be a empty list tuples ($variable \times expression$)
  function $T(expr)$ **is**
    switch *expr* **do**
      case *expr* **is** *a* with-*loop* **do**
        if **EC**[*expr*] = ∅ **and** **RC**[*expr*] = ∅ **then**
          V_{rc} = *fresh variable to store expr*
          $\mathcal{TR}$ ++ $[(V_{rc}, expr)]$
      case *expr* **is** *a* function application **do**
        if *expr* **is** *the* recursive call **then**
          foreach $(V_{rc}, expr')$ **in** $\mathcal{TR}$ **do**
            $\mathcal{N}_a$ = `FindMatching`(V_{rc}, F_d)
            if $\mathcal{N}_a = \emptyset$ **then**
              Delete $(V_{rc}, expr')$ **from** $\mathcal{TR}$
            else
              Append $\mathcal{N}_a$ to *arguments* of *expr*
              EC$[expr'] = V_{rc}$
      case *expr* **is** conditional $x\ ?\ e_1\ :\ e_2$ **do**
        $T(e_1); T(e_2)$
  foreach $(var = expr)$ **in** F_d **do** `/* top-down */`
    $T(expr)$
  foreach $(V_{rc}, expr')$ **in** $\mathcal{TR}$ **do**
    Append V_{rc} to *arguments* of F_d

Output: The updated mapping **EC**
The updated function F_d

Algorithm 4: *ERC* Loop Optimisation - Adjust Loop Calls

Input: A function definition F_d

begin
  function $T(expr)$ **is**
    switch *expr* **do**
      case *expr* **is** *a* loop function f application **and** *expr no. arguments* < f *no. arguments* **do**
        $\mathcal{N}_a$ = `CreateArgs`$(\Delta$ *no.* arguments$)$
        Append $\mathcal{N}_a$ to *arguments* of *expr*
      case *expr* **is** conditional $x\ ?\ e_1\ :\ e_2$ **do**
        $T(e_1); T(e_2)$
  foreach $(var = expr)$ **in** F_d **do** `/* top-down */`
    $T(expr)$

Output: The updated function F_d

5 EXPERIMENTAL EVALUATION

We implemented the *EMR* optimisation in a feature branch of the `sac2c` compiler based on `sac2c` 1.3.2. We evaluate the effects of *EMR* on a variety of benchmarks from different benchmark suites which represent various types of computations.

5.1 Experimental Setup

The benchmarks are compiled to run both sequentially on an AMD Opteron 6376 and on an NVIDIA K20 GPU (driver version 384.81) using GCC 4.8.5 and CUDA 9.0, respectively. Memory measurements are generated using compiler-instrumented SAC code through `sac2c` flag '`-profile m`' as well as NVIDIA's profiling tool `nvprof`. Runtime measurements are wall-clock times observed by the GNU `time` command. We present the average of five runs, and show the extreme values we found as well, in order to obtain an idea of the measurement noise.

The benchmarks include basic ones such as matrix multiplication and matrix relaxation. We also use a kernel from a real-world application by the British Geological Survey (BGS) called the Global Geomagnetic Model, which was translated to SAC and evaluated in [32]. Additionally we include benchmarks[2] from the Livermore Loops suite [23], the Rodinia Benchmark suite [6], and the NAS Parallel Benchmark suite [1]. A list of benchmarks and their respective problem sizes is given in table 1.

Some of the benchmarks we use have several different implementations, and we indicate these by adding suffixes to their names. For implementations that closely reflect their C counterpart, which use no *with*-loops, we use *C*. The suffix *APL* indicates an APL-like implementation that uses SAC implementations of APL operators. The suffix *N* indicates a naïve SAC implementation whereas the suffix *SC* is a more optimal SAC implementation. Additionally, for the Rodinia Pathfinder benchmark we use two implementations as one of them performs better on a GPU than on a shared-memory architecture [3]. These are suffixed with *nCD* and *CD*, respectively.

Our first experiment analyses the overall impact of *EMR* on the number of memory allocations that are being performed during the lifetime of the chosen benchmarks. Figure 1 shows for each benchmark the total number of allocations made in four different columns: the first two columns show the number of allocations during sequential executions, first without *EMR* then with *EMR* being applied. The second set of columns presents the corresponding numbers for executions on the GPU.

Note here that the y-axis is a logarithmic scale. For the first benchmark, the BGS-Kernel, we can see that for both, sequential execution and execution on the GPU, the number of memory allocation shrinks by more than two orders of magnitude, from roughly 2.5×10^6 to 10^4.

We observe that 10 out of the 37 benchmarks benefit from *EMR*. The other 27 benchmarks essentially remain the same or have small improvements only. 9 out of the 10 benchmarks that benefit from *EMR* have improvements of more than 2 orders of magnitude. In 8 out of the 10 cases the number of allocations decreases to levels that are very similar to those of manually written code that have explicit memory management.

Only Gauss-Jordan seems to benefit less significantly. Closer inspection reveals that, nevertheless, in the sequential case the

[2] More information on the SAC implementations of the Livermore Loops, Rodinia Benchmarks, and the NAS Parallel Benchmarks (as well as other benchmarks) are available at https://github.com/SacBase.

Suite	Benchmark	Problem Size
Livermore Loops	Loop01	100001 vector
	Loop02	100001 vector
	Loop03	100001 vector
	Loop04	10001 vector
	Loop05	100001 vector
	Loop07	100001 × 64 array
	Loop08-split	1001 × 64 array
Rodinia Benchmarks	Back-propagation	65536 input units
	Hotspot	1024 × 1024 grid
	Kmeans	34 features, 5 clusters
	Leukocytes	640 × 480 image, 88 cells
	LUD	2048 × 2048 matrix
	Needleman-Wunsch	24 × 24 grid
	Particle Filter	10 frames at 128 × 128, 1000 particles
	Pathfinder	100 × 512000 grid
	SRAD	2048 × 2048 matrix
NAS Parallel	Conjugate Gradient	14000, 75000, and 150000
	Embarrassingly Parallel	2^{24}
	Multigrid	128^3
Other	Matrix Multiplication	1000 × 1000 matrices
	Matrix Relaxation	10000 × 10000 matrix
	Gauss-Jordon	500 × 500 matrix
	BGS Kernel	1000 × 1000 matrix

Table 1: Benchmarks and their Problem Sizes

number of allocations is reduced to 50% of the unoptimised version and for the GPU case, the number of allocations is reduced by 30%.

The fact that we see different effects for the two architectures has two reasons: Firstly, we only measure the memory allocations on the machine that executes the data-parallel kernels. Consequently, the GPU figures are potentially smaller. Secondly, we perform different code optimisations depending on the target platform which may impact the number of intermediate arrays materialised in memory.

In this context, two benchmarks stick out: Relax-Expo only gains on the sequential platform and the Rodinia implementation of Needleman-Wunsch benefits on the GPU only. In the case of Relax-Expo, the reason for this behaviour lies in the use of a reduction operation in the innermost loop which cannot be computed on the GPU platform but must be done on the host. This prevents the application of *EMR*, thereby leading to significant communication between the host and the GPU device. In the case of Needleman-Wunsch, a variance in the applied optimisations is responsible for the difference in effectiveness of *EMR*.

5.2 Impact of *EMR* on Memory Pressure

While the total number of allocations gives us an idea of how many calls to the heap manager could be eliminated, it is not clear how relevant these are in terms of the sizes of memory allocations that could be avoided. In order to quantify this aspect, we measured the sum of all memory allocation requests. Figure 2 shows these numbers, again using a logarithmic scale. For these results, we only show the numbers for the sequential execution in order to make sure that we capture all allocations rather than only those on the architecture where the data-parallel computation is being executed, as is the case with using a GPU.

Here, we can observe that all benchmarks where benefits were found in terms of the total number of allocations, also benefit by a reduction to the total amount of memory allocation requests. The reductions are typically bigger than 2 orders of magnitude. An excellent example is the Livermore Loop 8, where the total sum of allocations shrinks by almost 6 orders of magnitude from 380GB down to 500KB. This suggests that the optimisation of innermost loops is effective in these examples.

Another interesting aspect is the observation that the NAS Conjugate Gradient actually benefits from *EMR* as well, which was not evident from the total number of allocations. From this experiment we learn that the total amount of memory that is requested by NAS CG is cut by roughly a factor of 4.

5.3 Impact of *EMR* on the Runtime

After verifying that *EMR* actually reduces the number of memory allocations and that these are of relevant size, we investigate the impact of *EMR* on the runtime for both architectures.Our measurements are presented in fig. 3 which shows the speedup of *EMR* for all benchmarks on both architectures. The baseline for both architectures is the corresponding unoptimised runtime. The horizontal bars indicate the maximum and minimum speedups obtained from the maxima and minima of 5 runs each.

The first observation we can make is that the runtime gains are mainly realised on the GPU platform. This does not really come as a surprise as runtime experiments with our GPU backend triggered this line of research in the first place. The need to perform allocations and deallocations of GPU memory from the host has a very noticeable effect and the change of memory often also generates the need for superfluous data-transfers between host and device. 7 out of the 9 applications that benefited in terms of memory allocations on the GPU show speedups between a factor of 2 and a factor of 4 against the unoptimised versions.

The two applications that do not benefit on GPUs despite savings in terms of memory allocations are Gauss-Jordan and Rodinia's Pathfinder. For these two examples the optimisations apply to parts of the application that are not the main hotspots. Additionally, we observe a slowdown in Relax-Expo although the overall allocations in the GPU didn't decrease. As previously mentioned this is due to the additional communication caused by a reduction operation.

Finally, we observe some speedups for the sequential CPU executions as well. 4 of the 10 applications that benefit from fewer memory allocations expose performance gains between 10% and 50%.

5.4 Impact of *EMR* on the Memory Footprint

Our final experiment concerns the memory footprint of the benchmarks. Since *EMR* extends the lifetime of variables it can potentially lead to an increase in the maximum of memory that is allocated at some time during execution. Our measurements indicate that none of the examples suffered from any increment of the memory footprint.

6 DISCUSSION

The evaluation in the previous section shows that *EMR* decreases the number of memory allocations by several orders of magnitude for more than a quarter of the compute intensive kernels we investigated. Most of these come down to levels of allocations that are similar to what codes with hand-written explicit memory management would look like. About half of the examples we investigated

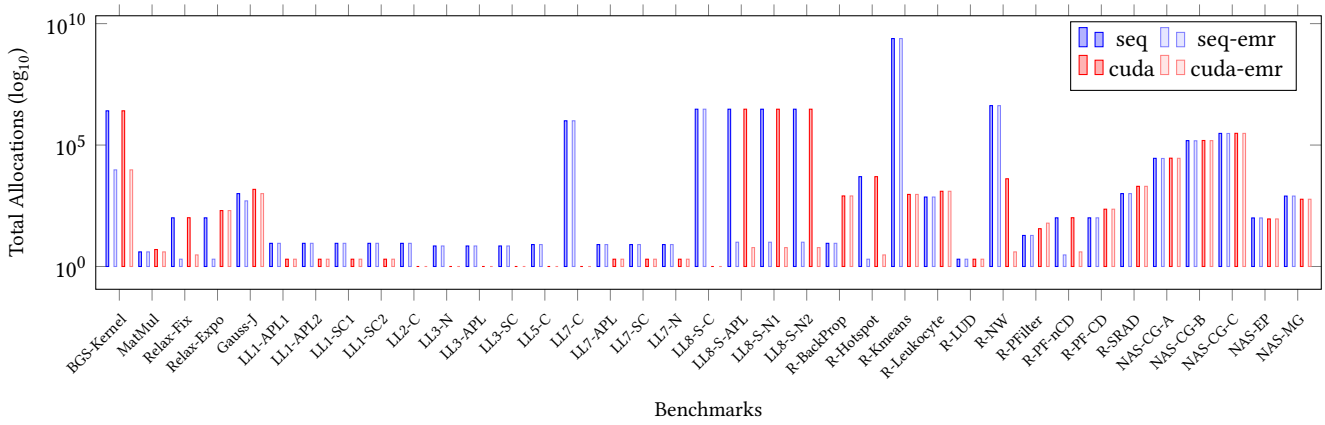


Figure 1: Allocation Count (normal vs. optimised)[+]

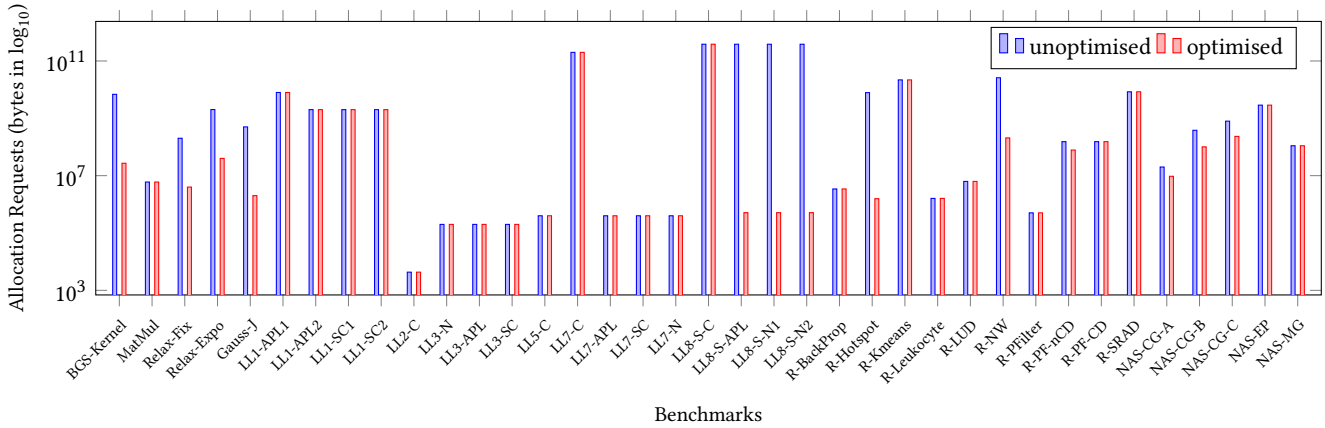


Figure 2: Total Memory Allocation Requests for *sequential* Target (normal vs. optimised)[+]

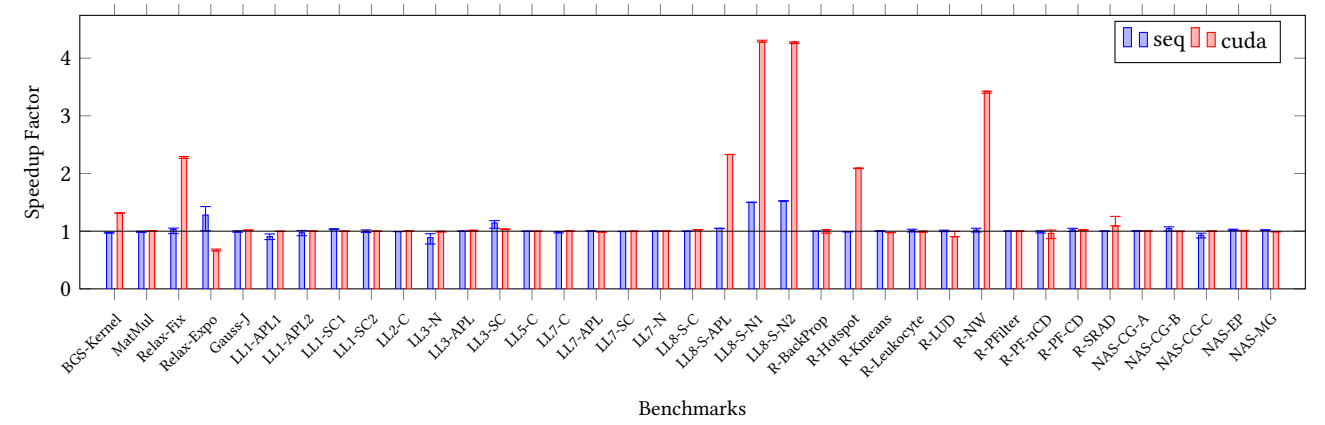


Figure 3: Runtime Speedup (normal vs. optimised)[+]

[+] Some of the benchmarks have more than one implementation-style. These alternatives are marked with suffixes: *-SC* for SaC, *-N* for naïve SaC, *-APL* for APL, *-C* for C/C++.

do not benefit *EMR* simply because the pre-existing reuse analysis explained in section 2.3 already brings down the allocations to the hand-written level.

An analysis of the remaining quarter of applications identifies two main reasons why deallocations and subsequent allocations remain within innermost loops.

The first reason is that some of the benchmarks are written in a C-style using *for*-loops rather than *with*-loops. For those cases to benefit from *EMR*, we need an extension of *EMR* to deal with these loops in the same way as we deal with *with*-loops. While this is conceptually easy, given that the underlying operation is a scalar modification of the array, the implementation effort that would be required is non-trivial. Since most SaC programs are defined in terms of *with*-loops whenever possible, we decided to leave this extension as future work.

The second reason lies in the fact that several of our benchmarks operate on differently sized arrays within the innermost loops. Examples for these are Gauss-Jordan and Conjugate Gradient. Trying to capture these cases poses a bigger challenge. It would require the memory system to allow memory chunks to be larger than what is strictly needed at allocation time. This immediately increases the potential danger of memory footprint increases. Furthermore, in case of growing memory demands, it would require further analyses to predict the maximum memory need over the lifetime of loops. Given these major challenges and it not being clear whether these applications would benefit significantly, support for varying-size memory reuse remains outside of the scope of this paper.

Though the evaluation of *EMR* shows improvements and effectiveness throughout, two aspects remain as possible concerns. The first relates to the way *ERC* filtering is done and the second relates to the change of memory footprint of applications.

6.1 Choice of Extended Reuse Candidates

The filtering of *ERC*s from section 4.2 currently picks the most recent possible candidate. While our experimentation shows that this works well in practice, in theory, this can lead to sub-optimal choices. If we have more choices than we need for memory reuse the choice of the first candidate may undermine the potential success of memory reuse. Consider the following example:

```
int[10] strange (int[10] a, int[10] b)
{
   c = { iv -> 0 | iv < [10] };
   return c;
}
```

Here, after inference and initial filtering, we have *ERC*s `a` and `b` to choose from. Let us assume *EMR* has chosen `b`, if it turns out at runtime that the reference count of `a` is 1 and that of `b` is 2, we forgo the opportunity for reuse. While it would be possible to maintain both choices until runtime in this case, a slight variation of the example shows that this cannot be done in general. Consider this variant:

```
int strange(int[10] a, int[10] b)
{
   c = { iv -> 0 | iv < [10] };
   d = { iv -> 1 | iv < [10] };
   r = f (c, d);
   return r;
}
```

Now let us assume we would keep both, `a` and `b`, as *ERC*s for both *with*-loops. Furthermore, let us assume `a` and `b` at runtime both come in with a reference count of 1. If the *with*-loop for `c` chooses to reuse `a`, then after the computation of `c`, array `c` will have a reference count of 1. Since `c` reuses `a`, an inspection of the reference count of `a` which conceptually is no longer exists will still yield 1. From the perspective of the *with*-loop that computes `d` this suggests that it can use `a` for memory reuse as well which would simply be wrong.

Consequently, we either need to establish a mechanism that communicates the choice made by the first *with*-loop to the second *with*-loop, or we need to ensure that the sets of *ERC*s after filtering are disjoint. Our proposed approach opts for the latter. We consider loosing out on options in situations like our first example here less of a concern since, in the worst case, it only results in the same reuse behaviour as without using *EMR* in the first place.

6.2 Memory Footprint

By reusing memory we increase the lifetime of allocated memory. This potentially translates into a larger utilisation of memory over the runtime of the program. Consider the following example:

```
{
   // ...
   c = { iv -> 0 | iv < [1000] };
   print (c);
   d = { iv -> 1 | iv < [1001] }
   print (d);
   e = { iv -> 2 | iv < [1000] };
   print (e);
   // ...
}
```

Assuming that we have no further references to the arrays `c` and `d`, compilation without *EMR* would result in a memory footprint of this code snippet of `1001×sizeof (int)`. Using *EMR*, this figure increases to `2001×sizeof (int)`. If we construct this within a non-tail-end-recursive function, we can construct an example where the footprint increases linear with the number of recursive calls:

```
int elephant (int n)
{
   c = { iv -> 0 | iv < [1000] };
   print (c);
   res = n == 0 ? 0 : elephant (n-1);
   e = { iv -> 2 | iv < [1000] };
   print (e);
   return res;
}
```

Here the memory footprint of the function `elephant` without *EMR* is constant while the application of *EMR* renders the memory footprint linear to the value of the parameter `n` as all instances of the body will keep the memory allocated for `c` for reuse in `e` while the recursive calls happen.

Although we have not seen this effect in any of the examples we ran, it would be better if we could rule out such a memory footprint increase by construction. One way to tackle this problem would be to rule out all *ERC*s whose scope would need to be extended across any code that potentially allocates memory. Unfortunately, this would rule out almost all examples, even the most simple ones. The reason is that we cannot guarantee that a function call such as `print (a)` does not allocate some memory *after* deallocating the array `a`. This would require sophisticated analyses and might still turn out to be too conservative in many practical cases.

7 RELATED WORK

Most of the literature that deals with improving the performance of reference counting based garbage collection focuses on decreasing overheads or improving the inference of in-place update opportunities.

In the work by Park and Goldberg [25], the authors introduce a compile-time based analysis to track the lifetime of references. With this information, the authors can determine points within the code where a reference counter need not be updated. For instances, when a reference is made and then quickly discarded before the total number of references for the memory object reaches zero. By avoiding these reference counter updates, the author's analysis reduces the runtime overheads of reference counting.

Shahriyar et al. [30] propose a solution to make reference counting more amenable to high performance environments. In the context of Java, they looking at an existing reference count implementation that uses cyclical garbage collection. They perform a series of analyses looking at intrinsic properties of the reference counting mechanism such as counter storage and reference count operations at runtime. From this they introduce and implement several optimisations which, *a.* change how memory objects are tracked for cyclical garbage collections, and *b.* lazily allocate memory objects at the point where their counter is initially incremented. Their experiments demonstrate a positive speedup for their optimisations.

For GPU based applications, reference counting can offer a possible way to handle communication between the host and GPU device, as is done by Jablin et al. [19]. Here the authors introduce a runtime-library and serious of code-transformations that they call the CPU-GPU Communication Manager. The runtime-library provides wrappers for `malloc` and `free` which do reference counting, and the code-transformations move communication primitives around to minimise communication. For a set of benchmark, the authors solution results in speedups.

Outside of reference counting based garbage collection, other techniques exist which try to reuse heap allocated memory. For instance Hamilton and Jones [16] introduce a compile-time based solution for a functional program that searches the code for in-place update opportunities. They do this by applying *necessity* analysis to determine which parts of a list are needed and where. Whenever they find a list part that has no further necessity, that list part can be safely reused.

Similarly, Kågedal and Debray [20] extend on this idea in a single-assignment context. They provide a code-transformation that introduces runtime-time primitives to the code that either perform a new allocation or do an in-place update. Their code-transformation is able to deal with iterative constructs, and can move copy-operations out of inner loops. Their experiments demonstrate large speedups with this technique.

Alternatively, Hage and Holdermans [15] do not do any compile-time analysis but instead introduce a new construct to a lazy functional programming language. The construct is used to mark opportunities for memory reuse, and what structure to use for this. They provide a set of semantics and type rules to ensure that reuse only occurs when it is safe to do so. Lee et al. [22] do something similar for a ML-like functional language. Here though they perform a static memory analysis, after which the inferred information from this is used to place free primitives which destructively update nodes in a list.

8 CONCLUSION AND FUTURE WORK

In this paper we propose a novel compiler optimisation to increase memory reuse. One of the main insights that made this work possible is the observation that small increases in the lifetime of an object offers large opportunities for memory reuse.

We implement the proposed optimisation in the context of the SaC compiler. The underlying memory management of SaC is based on reference counting. This makes the entire analysis easier, but in no way restricts us to either SaC or reference counting to apply the proposed technique in other contexts.

We evaluate the effect of the proposed transformation by running a set of 37 benchmarks on CPUs and GPUs. Most of benchmarks come from Livermore Loops, Rodinia and NAS benchmark suites which are typically used when evaluating high-performance compilers.

In our experiments we look for changes in the number of memory allocations, the amount of totally allocated memory and runtime performance. On all the three fronts the majority of benchmarks show either no changes or significant improvements. In 8 benchmarks, the number of memory allocations comes down by several orders of magnitude, ending up close to what we might find in hand-written codes. The runtime performance on GPUs for six benchmarks improves between 2× to 4× while only one benchmark shows a slowdown when running on a GPU due to the interplay of *EMR* with the code generation scheme for reductions on GPUs.

We also see some improvements for executions on CPUs albeit on a more moderate level leading to speedups between 10% and 50%.

The observed results suggest that the first step towards better memory reuse is successful. At the same time this opens up a lot of opportunities for future work.

There are two main directions of the future work we would like to pursue. First, we would like to further improve the applicability of *EMR*. As explained in section 6, our optimisation is limited to the case where we are dealing with identical array sizes. It just naturally happens that in high-performance codes data objects of the same size are used over and over again. If this is not the case, it would be desirable to develop some form of cost model that allows an extended version of *EMR* to decide whether to reuse memory that is larger than needed or to allocate more memory than needed, just for later reuse.

Secondly, we believe that the underlying memory reuse pattern that we build on can be used in other contexts. For example we could envision a static analyser for languages with explicit memory allocations, *e.g.* C/C++ family. In our experiments we have seen that memory reallocation is relatively cheap on CPUs, but on GPUs it can have a significant overhead. Furthermore, memory allocations on GPUs are always done from the host, and they cannot be done asynchronously. So, given a CUDA/OpenCL program where deallocation/reallocation of the same size happens between the kernels, we can apply our mechanism and replace the `free (x); y = malloc (sizeof (x))` with `y = x`.

At the same time, languages like C++ have smart pointers, some of which use reference counting which makes our technique immediately applicable.

Finally, explicit lifetime annotations in Rust may be used to manually control the lifetime of the memory object. We can envision a static analyser that identifies the cases that are described in the paper, and uses lifetime annotations to implement the proposed reuse technique.

ACKNOWLEDGMENTS

This work is supported by the Engineering and Physical Sciences Research Council through grants EP/L00058X/1 and EP/L016834/1. We also made use of Edinburgh Centre for Robotics' Robotarium Cluster [9] located at Heriot-Watt University, Edinburgh, UK for our experiments.

REFERENCES

[1] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[2] Robert Bernecky, Stephan Herhut, Sven-Bodo Scholz, Kai Trojahner, Clemens Grelck, and Alex Shafarenko. 2007. Index Vector Elimination: Making Index Vectors Affordable. In *Implementation and Application of Functional Languages, 18th International Symposium (IFL'06), Budapest, Hungary, Revised Selected Papers (Lecture Notes in Computer Science)*, Zoltan Horváth, Viktória Zsók, and Andrew Butterfield (Eds.), Vol. 4449. Springer, 19–36. https://doi.org/10.1007/978-3-540-74130-5_2

[3] Robert Bernecky and Sven-Bodo Scholz. 2015. Abstract Expressionism for Parallel Performance. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 54–59. https://doi.org/10.1145/2774959.2774962

[4] David C. Cann. 1989. *Compilation Techniques for High-performance Applicative Computation*. Ph.D. Dissertation. Fort Collins, CO, USA. AAI9007070.

[5] David C. Cann and Paraskevas Evripidou. 1995. Advanced array optimizations for high performance functional languages. *IEEE Transactions on Parallel and Distributed Systems* 6, 3 (March 1995), 229–239. https://doi.org/10.1109/71.372771

[6] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC '09)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[7] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (01 Dec 2002), 255–271. https://doi.org/10.1007/s00446-002-0079-z

[8] Steven M. Fitzgerald and Rodney R. Oldehoeft. 1996. Update-in-place Analysis for True Multidimensional Arrays. *Sci. Program.* 5, 2 (July 1996), 147–160. https://doi.org/10.1155/1996/493673

[9] Edinburgh Centre for Robotics. 2014. Robotarium Cluster. https://doi.org/10.5281/zenodo.1455754 The cluster is part of the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems (RAS) in Edinburgh grant (EP/L016834/1) funded by The Engineering and Physical Sciences Research Council (EPSRC) (UK).

[10] Python Software Foundation. 2018. Python 3.7 Language Documentation. https://docs.python.org/3/c-api/index.html Online, accessed 14 August 2018.

[11] Clemens Grelck. 2012. Single Assignment C (SaC): High Productivity Meets High Performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary (Lecture Notes in Computer Science)*, V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.), Vol. 7241. Springer, 207–278. https://doi.org/10.1007/978-3-642-32096-5_5

[12] Clemens Grelck and Kai Trojahner. 2004. Implicit Memory Management for SaC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Clemens Grelck and Frank Huch (Eds.). University of Kiel, Institute of Computer Science and Applied Mathematics, 335–348. Technical Report 0408.

[13] Jing Guo, Robert Bernecky, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2014. Polyhedral Methods for Improving Parallel Update-in-Place. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.

[14] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the Gpu Programming Barrier with the Auto-parallelising Sac Compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24. https://doi.org/10.1145/1926354.1926359

[15] Jurriaan Hage and Stefan Holdermans. 2008. Heap recycling for lazy languages. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*. 189–197. https://doi.org/10.1145/1328408.1328436

[16] G. W. Hamilton and S. B. Jones. 1991. Compile-Time Garbage Collection by Necessity Analysis. In *Functional Programming, Glasgow 1990*, Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst (Eds.). Springer London, London, 66–70.

[17] Paul Hudak and Adrienne Bloss. 1985. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*. ACM, New York, NY, USA, 300–314. https://doi.org/10.1145/318593.318660

[18] Apple Inc. 2018. Swift Language Documentation. https://docs.swift.org/swift-book/ Online, accessed 14 August 2018.

[19] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU Communication Management and Optimization. *SIGPLAN Not.* 46, 6 (June 2011), 142–151. https://doi.org/10.1145/1993316.1993516

[20] Andreas Kågedal and Saumya Debray. 1997. A Practical Approach to Structure Reuse of Arrays in Single Assignment Languages. In *Proceedings of the 14th International Conference on Logic Programming*. MIT Press, 18–32.

[21] Akash Lal and G. Ramalingam. 2010. Reference Count Analysis with Shallow Aliasing. *Inform. Process. Lett.* 111, 2 (Dec. 2010), 57–63. https://doi.org/10.1016/j.ipl.2010.08.003

[22] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. 2003. Inserting Safe Memory Reuse Commands into ML-Like Programs. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 171–188.

[23] Frank H. McMahon. 1986. *The Livermore Fortran Kernels: A computer test of the numerical performance range*. Technical Report UCRL-53745. Lawrence Livermore National Lab., CA, USA.

[24] NVIDIA Corporation 2018. *CUDA C Programming Guide* (9.0.176 ed.). NVIDIA Corporation. https://docs.nvidia.com/cuda/archive/9.0/ Online, accessed 12 Aug. 2018.

[25] Young Park and Benjamin Goldberg. 1995. Static analysis for optimizing reference counting. *Inform. Process. Lett.* 55, 4 (1995), 229 – 234. https://doi.org/10.1016/0020-0190(95)00096-U

[26] SaC Development Team 2016. *SaC EBNF Grammar*. SaC Development Team. Available online at http://www.sac-home.org/doku.php?id=docs:syntax.

[27] Kazuki Sakamoto and Tomohiko Furumoto. 2012. *Life Before Automatic Reference Counting*. Apress, Berkeley, CA, 1–29. https://doi.org/10.1007/978-1-4302-4117-1_1

[28] Sven-Bodo Scholz. 1997. An Overview of Sc Sac – a Functional Language for Numerical Applications. In *Programming Languages and Fundamentals of Programming, Technical Report 9717*, R. Berghammer and F. Simon (Eds.). Institut für Informatik und Praktische Mathematik, Universität Kiel.

[29] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[30] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the Count? Getting Reference Counting Back in the Ring. *SIGPLAN Not.* 47, 11 (June 2012), 73–84. https://doi.org/10.1145/2426642.2259008

[31] H. Sundell. 2005. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*. 10 pp.–. https://doi.org/10.1109/IPDPS.2005.451

[32] Hans-Nikolai Vießmann, Sven-Bodo Scholz, Artjoms Šinkarovs, Brian Bainbridge, Brian Hamilton, and Simon Flower. 2015. Making Fortran Legacy Code More Functional. *27th Symposium on Implementation and Application of Functional Languages (IFL '15)* (2015). https://doi.org/10.1145/2897336.2897348

A DSL embedded in Rust

Kyle Headley
University of Alabama at Birmingham
kheadley@uab.edu

ABSTRACT

Rust includes two "languages" that are not as commonly used as the main one: a sophisticated macro system and a type-level language utilizing the trait system. The type-level language can be used in both a functional style and a logic style. We explore the capabilities of these languages, focusing on the functional type-level language, where our main contribution is showing a way to define first-class type functions in Rust.

Additionally, to show off these languages, we use them to create a eDSL. We use a simple variant of the lambda calculus. This embedded language is parsed by the Rust parser (and macros) and type checked by the Rust type checker at compile time.

ACM Reference Format:
Kyle Headley. 2018. A DSL embedded in Rust . In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3310232.3310241

1 INTRODUCTION

The Rust language[1] reached version 1.0 in mid-2015, bringing together high-performance, thread-safety, and a minimal runtime system. We ignore those features in this paper, concentrating instead on macros and on trait-based generics, the type-level language of Rust. Both of these are expressive enough to be used as their own general-purpose programming languages. However, their use does not seem to be as common as their utility would suggest. This paper explores those languages, demonstrating features that will be valuable to anyone looking to expand their usage of Rust. These features will be especially useful for creating alternative syntax (macros), extending polymorphism (type functions), and guaranteeing program properties (extended type system).

1.1 Macros

Macros are generally used for syntactic abstraction. They can reduce code size when patterns of characters are present, but they cannot be written as a common function. They are often expanded into code before compiler features like type-checking are run. In Rust, macros are fairly advanced, with multiple rounds of expansion, different levels of parsing, hygienic variables, and pattern matching. Though we will be using the original macro system, Rust has been enhanced with procedural macros, which allow runtime code to handle the expansion.

These advanced features allow us to do more than write syntactic functions that take parameters, we can use macros to separate Rust code from DSL code written in another language. The parameter to the macro in this case would be arbitrary text that the macro parses into a syntax tree before transforming into Rust code to be handled by the type-checker and compiler.

This use of macros is available in other languages as well. Racket, notably, is based on a philosophy of language-based programming. Racket macros are even more sophisticated than those of Rust, and Racket programmers are encouraged to build DSLs with them.

Rust programmers are not usually encouraged to build new languages, but a simple one may be appropriate for a project. This paper demonstrates some techniques for parsing them into an AST with Rust macros, which can then be passed to a custom type checker.

1.2 Traits

Types provide languages with a simple form of static verification that compilers may use to assist programmers in their work. Most typed languages allow users to create new types, often to define complex data structures that need to maintain certain invariants. For example, a binary tree must always have two or fewer branches at each node, and each node contains data of the same form. Once defined, the compiler will generate errors when the tree is used inappropriately, just as it would when built-in types are misused.

But users often want to create abstract types just as they create abstract code by writing functions. Most languages allow users to abstract the data in binary trees, but few allow them to also abstract the links between branches.

Rust provides users some additional flexibility of types with traits. When used, these restrict use of types to those with a particular set of properties, like the ability to add two terms together. The restriction gives us a guarantee, which can be used to, for example, provide a tree with the additional functionality of adding together all of its data, regardless of the type of data the user had chosen.

The ability to abstract the links between branches of a tree (to specialize them for performance or parallel processing) is often called higher-kinded types, or HKT. There are a number of discussions online about how to get around the fact that Rust has no explicit support for them. People have simulated HKT if a few different ways [4, 7]. These are often complex and must be re-implemented for new data types. HKT is a feature often requested from the Rust community, and there is work towards it by the Rust development team.

However, Rust does have the ability to have type functions, a more general technique than HKT. In this paper we go into detail about how to deal with type functions in Rust: techniques for creating them, passing them as parameters, and restricting them with

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310241

the rest of the type system. While there is no explicit syntactic support, simple type functions like those of HTK require only a few lines of code to set up, and about twice as many characters to use as a regular function call would.

One goal of this paper is to share these features with the developers of Rust, so that they may take them into account as development proceeds. They will have knowledge beyond the scope of this work, and can choose to integrate it into plans, or discourage its use as appropriate.

1.3 Contributions

In this paper we explore secondary features of the Rust language in the context of language implementation. We make the following contributions:

- Provide techniques for creating type functions
- Provide techniques for type-checking type functions
- Demonstrate parsing a simple language with macros
- Demonstrate type-checking a DSL at compile time

This paper is divided into two parts, the first shows off advanced techniques, and the second makes use of some of them to parse and type-check a simple DSL. Each part is further divided into two subparts. The first deals with macro features and the second deals with trait features.

We introduce Rust macros in Section 2. These are defined with a name and a list of rewrite rules. We use this to mirror BNF grammars, with a different macro for each component of the grammar.

Since higher-order functions essentially form a language on their own, and we will rely on traits for implementation of our type-level functions, we refer to them as "TraitLang" for the remainder of this paper. We describe our usage of TraitLang in Section 3. To avoid confusion when discussing type-level values, we refer to one as a "struct", the keyword used when defining a type in Rust. We introduce the basic constructions in Section 3.1

TraitLang is interpreted by the Rust trait resolution algorithms, which are expected to be enhanced in the future. In this paper we use the original semantics from version 1.0, though the Rust team almost never introduces breaking changes (until the next major version). We also rely on the Rust type-checker to verify that our programs are well-formed. To verify correctness, we can define variables of our output types, which are all singletons.

Like types in a common language, traits classify structs, but unlike types, a struct can "implement" an unlimited number of traits. Each of these traits may contain associated types specific to its implementation by a struct. This implementation therefore acts as a mapping from one struct to another, one of the ways to define a function. However, to allow first-class functions, we prefer a different technique, described in Section 3.2, that uses a struct as a first-class function, and an implemented trait as the function's expression.

Another use for a mapping is to map values to their types. We introduce a trait called "Typed" in Section 3.3, and expand its use to functions in Section 3.4. Providing constraints on structs and functions allows us to define our own type system. Both here and in our DSL example later we set up a standard one, but there's no reason why something more exotic couldn't be done.

```
macro_rules! expr {
    (0) => (Num(Zero));
    ($a:tt $p:tt) => (App(expr![$a],expr![$p]));
    ({^$e:expr}) => ($e);
    (($($e:tt)+)) => (expr![$($e)+]);
    ($n:tt + $($ns:tt)+)
      => (Plus(expr![$n],expr![$($ns)+]));
}
```

Figure 1: Selected macro rules (out of order)

The next part of the paper walks through our implementation of the lambda calculus with addition as a DSL. We describe parsing in Section 4, type checking in Section 5. Both are rather elegantly implemented, since the techniques used mimic the grammar and operational semantics used to define languages. We do however need some supporting functions for our operational semantics, especially for dealing with a context. We have not yet developed an elegant way to implement functions with nested branching.

We conclude with some discussion of additional concerns in Section 6 and related work in Section 7.

2 RUST MACROS

Each Rust macro is an identifier and a list of rewriting rules, from a pattern matcher to a template. The first rule whose pattern matches is used to expand into the template. Macros are commonly used to transform code snippets, but they have a mode that deals with arbitrary tokens.

The matcher may include literals, pattern variables, and repeaters. Pattern variables are prefixed with a `$` and include a "fragment specifier". We will mostly be using token trees (`tt`), which macro invocations are initially parsed into. Token trees are either a single token, or a parenthesized (`()`,`{}`,`[]`) sequence of tokens. We also make use of `expr`, which signals the Rust parser to fully parse the match as a Rust expression. Repeaters `$( ... )+` match multiple instances of their inner pattern.

The template may also include literals, variables, and repeaters. They may also include macro invocations (but not definitions), allowing recursive calls, even through variables. Some selected rules from our later example are in Figure 1, explained below.

Later we will use the `expr!` macro to parse syntax into an AST. For now, we use selected portions shown in Figure 1 to introduce Rust macros. The first line contains only a literal in the matcher, to transform a number into its AST representation. (Our AST distinguishes raw natural numbers from syntax.) There are no pattern variables, so the `0` must be matched exactly. The second line shows two pattern variables specified as token trees. This pattern matches any two tokens, and the expander recursively invokes the `expr!` macro on each, placing them within an `App` node in our AST. The third line is used to insert pre-created expressions into our AST. The pattern variable is parsed and used directly. The other tokens are literals and must be matched exactly. Using one of the forms of parenthesis to surround the match allows it to be treated as a token tree before reaching this rule.

The final two rules of Figure 1 show off the macro repeaters. The first is a minimal repeater surrounded by parentheses, for parsing parenthesized expressions. The contents of the parentheses are

```
trait Nat {}

struct Zero;
impl Nat for Zero {}

struct Succ<N>(N);
impl<N:Nat> Nat for Succ<N> {}

type One = Succ<Zero>;
```

Figure 2: Declaration of Natural numbers in Rust's type-level language

copied into a recursive invocation. The final rule is a more complex version of the same principal, used to put everything after the first `+` into the second section of the `Plus` AST node (after a recursive call). If the matched pattern contained multiple `+`'s, they would evaluate left to right. Our DSL doesn't need to deal with order of operations, but one that did would need a more complex matcher.

3 TRAITLANG

TraitLang is a lazy, untyped, interpreted language with some features similar to both logic and functional languages. It is declarative and order of declaration doesn't matter, as all items are fully recursive. TraitLang is pure, since Rust's type-level items do not have access to the object language at all. It is not even possible to get output from a TraitLang program directly, instead, it will be used to support polymorphism and invariant checking for the object language. Because TraitLang is lazy, the well-formed check also requires that type aliases be used. We assume a "`fn main() { let x:TypeAlias1; ... }`" with each alias used at least once.

This section describes the use of TraitLang as a functional language. The syntax and programming style are very different from traditional languages, so we take some care in walking through a series of progressively more complex examples. TraitLang is interesting on its own, so we go a bit beyond what is needed to implement our DSL, describing a method for using first-class functions, and providing additional type systems for them. Type checking our later example mostly makes use of a logical style, but does use supporting functions. The membership function for contexts (described in Section 5.1) is rather verbose, since TraitLang is not well-suited for functions with multiple branches.

3.1 A Hidden Language

When we ignore Rust's main language and focus on the trait language, we are left with four items: declaration of a trait, declaration of a struct, implementation of a trait for a struct, and declaring a type alias, which functions like a let-binding. The basic syntax of these items in shown in Figure 2, which gives the standard definition of natural numbers. Here we define `Nat` as a trait, which works well at first, but is not sophisticated enough for a formal definition. Structs may implement multiple traits, allowing a later "crate" (Rust package) to implement e.g. trait `Real` for the same `Zero`. We return to this issue later.

Figure 2 continues by declaring a struct called `Zero` and implementing `Nat` for it. This is the simplest form of the declarations. More complex is `Succ`, which requires a parameter when the struct

```
trait AddOne : Nat { type Result:Nat; }
impl<N:Nat> AddOne for N { type Result = Succ<N>; }

trait SubOne : Nat { type Result:Nat; }
impl<N:Nat> SubOne for Succ<N> { type Result = N; }

type Two = <One as AddOne>::Result;
```

Figure 3: Using traits as mappings

is used. In this case `N` may be any other struct, including a recursive `Succ` (though infinite sequences cannot be defined). The second to last line is read "For all `N` such that `N` implements `Nat`, implement `Nat` for `Succ<N>`". Using this definition, the compiler will not give an error when using e.g. `Succ<Red>` (assuming a struct `Red` has been declared), but it would not implement `Nat`. We could have given a trait bound when declaring `Succ`, that is, "`struct Succ<N:Nat>(N);`". Doing so would cause a compiler error on use of `Succ<Red>`. We can use a struct by creating an alias like in the last line. The struct must be concrete, with no type variables.

There are a few syntactic peculiarities in Figure 2. Trait definitions end in curly braces, which are usually filled with object-level function definitions. We will add associated types here later. Formal type parameters, which can appear in any of the four syntactic items, are placed between angle braces and separated by commas. Each one may be required to implement any number of traits, placed after a colon and separated by a "`+`". Struct definitions must include each type parameter in parens, which is required for the object-level language, but we will not use it anywhere else. In the second to last line of Figure 2, the formal type parameters are after the `impl`, and their use is after the `Succ`. Usage does not include trait bounds.

3.2 A Functional Language

The full power of a functional language requires having functions. Figure 3 presents the simplest form available, using a trait as a mapping. Like defining `Nat` as a trait above, this form is simpler but limited, and we mainly use it for DSL meta-functions. We describe the syntax and semantics of this form first before moving on to one that allows first-class functions. In the figure, we define addition and subtraction by one.

Figure 3 introduces bounds for trait declarations, associated types, and how to access them. The first line declares a trait `AddOne` that requires any struct it's implemented for to also implement `Nat`. It includes a single associated type named `Result` that must also implement `Nat`. The implementation on the next line shows off the power of variables, implementing `AddOne` for every `Nat`, and providing an associated type dependent upon it. `SubOne` is similar, but note that it is not implemented for every `N`. Every associated type must be defined in order to implement a trait, but traits need not be implemented for every struct. This can be useful to ensure that suitable values are provided to computations. If appropriate, we could implement `SubOne` for every `Nat` by including the line "`impl SubOne for Zero { type Result = Zero; }`"

The last line in Figure 3 uses the unfortunate syntax for accessing an associated type. Both of the traits here have the same associated type name, so we must disambiguate by naming the struct, the trait implemented on the struct, and the associated type of that trait.

```
trait Func2<A,B> { type Result; }

struct Add;
impl<N:Nat> Func2<Zero,N> for Add { type Result = N; }
impl<N1,N2> Func2<Succ<N1>,N2> for Add where
    N1:Nat, N2:Nat,
    Add : Func2<N1,N2>
{ type Result = Succ<<Add as Func2<N1,N2>>::Result>; }

type Three = <Add as Func2<One,Two>>::Result;
```

Figure 4: Structs that can be used as functions

The syntax is slightly better in Figure 4 where we use structs as functions.

Rust generics use type variables, but there are no trait variables. If we were to continue to use traits as we did above, we would run into problems with generics and first-class functions in our type-level language. Below, we use traits to represent higher-level concepts. For example, there is a trait to mean that a struct is a function, rather than using a trait as a function. Figure 4 declares a trait representing a function of two variables. It then declares a struct `Add` and implements the inductive algorithm for adding two numbers.

Figure 4 introduces trait parameters which are similar to struct parameters. It also introduces the "where" clause, which can be used to add arbitrary requirements to any item. Here, the inductive case for defining `Add` requires `Add` be defined on a smaller structure. Since it is guaranteed by the where clause, we can look up its associated type to use in the definition of the result. Since `Add` is a struct, it can be passed as a parameter to functions just like `One` and `Two` were in the last line. Since the function parameters are declared on the trait, it can be called by any code with a where clause recognizing it as a function. We do this at the end of the next section, in Figure 6.

3.3 A Constraint Language

In this section we describe the final piece of syntax that will allow us to emulate a standard type system in our language. So far, we have been using traits as if they were types. But structs can implement multiple traits, so our functions can be applied to multiple "types". In order to have one type per struct, we need a mapping. Figure 5 demonstrates using a trait to declare types.

Figure 5 repeats functionality defined above, but in our form with types. The fourth line can be read: "For all `N` of type `Natural`, the type of `Succ<N>` is `Natural`". Note that we now have multiple levels of constraint, since we do not need to constrain the associated type. `Func1` requires its argument and result to be `Typed`, but doesn't require a specific type. Applying it to `Three` in the last line works the same as our prior example, but now the compiler is checking the associated type (required for arguments of `Next`) as well as the trait of `Three`.

We now know all the features we need to use TraitLang as a general-purpose language. Our language is untyped, but uses traits both to add and remove capabilities. Functions were added from mappings in traits, and the ability for `Succ` to take any parameter was removed by a constraint. Structs can be used as any value in the language, even when that value is acting as a different feature,

```
trait Typed { type Type; }

struct Natural;
impl Typed for Zero { type Type = Natural; }
impl<N:Typed<Type=Natural>> Typed for Succ<N>
{ type Type = Natural; }

trait Func1<A:Typed> { type Result:Typed; }

struct Next;
impl<N:Typed<Type=Natural>> Func1<N> for Next
{ type Result = Succ<N>; }

type Four = <Next as Func1<Three>>::Result;
```

Figure 5: The typing trait, allowing us to emulate a standard type system

```
struct Apply;
impl<A,B,R> Func2<A,B> for Apply where
    B: Typed, R: Typed,
    A: Func1<B,Result=R>
{ type Result = R; }

type Five = <Apply as Func2<Next,Four>>::Result;
```

Figure 6: A function that takes another as an argument

like a type or a function. For example, Figure 6 shows a use of first-class functions. Using structs as types allows type-based operations, as we will see next. We also see a syntactic optimization in the third line. We can constrain the associated type as well as type parameters. This in effect "binds" `R` to the result of `A` applied to `B`, allowing us to use it rather than the longer form used in Figure 4.

3.4 A Typed Language

We now take a step beyond the needs of our DSL to show how to constrain TraitLang to be a typed language. This requires function types and their use to constrain the trait that implements our functions. But our types are structs and Rust uses traits for constraints. We also do not have access to for-all variables in trait (or struct) definitions like we do in implementations. So we need an intermediate trait that picks out the relevant structs from our type and passes them along as usable constraints. This is what the first code block in Figure 7 does.

The first line of Figure 7 defines the type of our functions of three variables, `Arrow3`. The next few lines define our intermediate trait, `TypedFunc3`, and implement it on all structs that have type `Arrow3`, extracting the inner structs as associated types. The trait used to express functions, `Func3`, is then defined and can constrain its parameters to the associated types of its `TypedFunc3` trait. Now to define a function in TraitLang, we also need to provide its struct with a type, and that type will be enforced in the function's implementation, as can be seen in the next two code blocks.

The second and third code blocks in Figure 7 are examples of functions typed as explained above. Each of them defines the function name a struct, then gives them a type before implementing the function. The first shows how easy a simple function is to

```
struct Arrow3<T0,T1,T2,T3>(T0,T1,T2,T3);
trait TypedFunc3 {type T0; type T1; type T2; type T3;}
impl<T0,T1,T2,T3,A> TypedFunc3 for A where
    A:Typed<Type=Arrow3<T0,T1,T2,T3>>
{ type T0=T0; type T1=T1; type T2=T2; type T3=T3; }
trait Func3<A,B,C> : TypedFunc3 where
    A:Typed<Type=Self::T0>,
    B:Typed<Type=Self::T1>,
    C:Typed<Type=Self::T2>,
{ type Result:Typed<Type=Self::T3>; }

struct AddOneTwo;
impl Typed for AddOneTwo
{ type Type=Arrow3<Natural,Natural,Natural,Natural>; }
impl Func3<One,Two,Zero> for AddOneTwo { type Result=Three; }
impl Func3<One,Two,One> for AddOneTwo { type Result=Four; }
impl Func3<One,Two,Two> for AddOneTwo { type Result=Five; }

struct Add3;
impl Typed for Add3
{ type Type=Arrow3<Natural,Natural,Natural,Natural>; }
impl<A,B,C,R0,R1> Func3<A,B,C> for Add3 where
    A:Typed<Type=Natural>, B:Typed<Type=Natural>,
    C:Typed<Type=Natural>, R1:Typed<Type=Natural>,
    Add : Func2<A,B,Result=R0>,
    Add : Func2<R0,C,Result=R1>,
{ type Result = R1; }

type Six = <Add3 as Func3<One,Two,Three>>::Result;
```

Figure 7: The typing constraint and typed functions

define when using concrete inputs and outputs. There is no difference in the implementation from an untyped version. The second shows that, unfortunately, when variables remain abstract, their types must be made explicit. The Rust team has plans to implement constraint inference, so this may not be an issue in the future.

4 PARSING OUR DSL

The DSL we're implementing is the simply-typed lambda calculus with numbers and addition. Our grammar is standard and shown in Figure 8. The AST result is defined in Figure 12, discussed along with well-formedness checks in Section 5.2. Using macro rules for parsing means that we can follow our grammar very closely. We create one macro for each syntax class, and one rule for each syntax form. Our only deviations are in representing numbers and variables, and adding an injection point for easier composition, as described in Section 2. The full parser is shown in Figure 9.

Representing numbers and variables is a pain point of this method. Since we're working in TraitLang, we don't have access to any runtime functionality, only logic and induction. Integers and arithmetic are not available, so we use inductively-defined natural numbers (nats). The parser needs to map number literals to nats, so we need a rule for each number. Variables are available as additional structs, which would still add lines to the code. Also, we need to abstract over variables in our type checking later, but Rust does not give us an easy way to check both equality and inequality. To overcome this, we use nats as variables as well, with AST nodes that distinguish them from numbers.

Many of the rules in Figure 9 were shown previously or are similar to those. We describe some additional complexity here. The

```
e :=                expressions
    (e)             parentheses
    n               number
    v               variable
    lam (v:t) e     abstraction
    lam (v1:t1)(v2:t2)... e
                    multiple abstraction
    e1 + e2         addition
    e1 e2           application
    e1 e2 e3 ...    multiple application
t :=                types
    (t)             parentheses
    Number          base type
    t1 -> t2 -> ... arrow type
```

Figure 8: The grammar for our DSL

```
macro_rules! expr {
    (($($e:tt)+)) => (expr![$($e)+]);
    ({^$e:expr}) => ($e);
    (0) => (Num(Zero));
    (1) => (Num(Succ(Zero)));
    ...
    (x) => (Var(Zero));
    (y) => (Var(Succ(Zero)));
    ...
    (lam ($x:ident : $($t:tt)+)
        $(($($ts:tt)+))+ $($e:tt)+
    ) => (Lam(
        expr![$x],
        typ![$($t)+],
        expr![lam $(($($ts)+))+ $($e)+]
    ));
    (lam ($x:ident : $($t:tt)+) $($e:tt)+) => (Lam(
        expr![$x],
        typ![$($t)+],
        expr![$($e)+]
    ));
    ($n:tt + $($ns:tt)+)
    => (Plus(expr![$n],expr![$($ns)+]));
    ($a:tt $p:tt) => (App(expr![$a],expr![$p]));
    ($a:tt $p:tt $($ps:tt)+)
    => (expr![{^App(expr![$a],expr![$p])} $($ps),+]);
}
macro_rules! typ {
    (($($ts:tt)+)) => (typ![$($ts]);
    (N) => (Number);
    ($t:tt -> $($ts:tt)+)
    => (Arrow(typ![$t],typ![$($ts)+]));
}
```

Figure 9: The parser for our DSL

multi variable lambda rule has a nested repeater. The inner matches all the var and type tokens, and the outer matches the parenthesized groups. Before it is the first variable and type, which are used to create the lambda AST node. The repeater represents additional variables, which are used to create a nested lambda node with a recursive call. The lambda nesting pattern is convenient in this way, but the application nesting pattern is not. It is the reason we created the injection rule above. The nesting of applications must be as deep initially as the number of parameters, which we don't know. So we create the first AST node and pass it unchanged into

```
trait NatEq<N> { type Eq; }
impl NatEq<Zero> for Zero { type Eq=True; }
impl<N> NatEq<Succ<N>> for Zero { type Eq=False; }
impl<N> NatEq<Zero> for Succ<N> { type Eq=False; }
impl<N1,N2,E> NatEq<Succ<N1>> for Succ<N2> where
    N2: NatEq<N1,Eq=E>
{ type Eq=E; }
```

Figure 10: Equality function for natural numbers

the recursive call. The type rules are simple because arrow nests the lame way that lambda does.

With this macro definition, we may now write code such as `let e = expr![(lam (y:N)(x:N->N) x y) 2 (lam (x:N) 1+x)];`, and `e` will hold an AST for a reversed parameter application that can reduce to a number. This is typed-checked by the empty function `is_typed(&e,&typ![N])` that activates the trait resolution described in the next section.

5 TYPE CHECKING OUR DSL

This section describes the code for our type checker, divided into two parts. The first deals with functions for context lookup, and is done in the functional style introduced in Section 3. The next part handles static checks of our AST, and are written in a more logical style. This is valuable because, like for parsing, our code can mirror the rules from the notation of the theory.

5.1 Supporting functions

Context lookup appears in type checking rules, but often as a function over the data structure for simplicity. We mirror that here, but still need a full description of the algorithm. Context lookup involves comparing variables to find our target. As seen above, we have implemented our variables as natural numbers, so we need an equality function for them. This is shown in Figure 10. It follows the method from Figure 3, since we don't make use of first-class functions. There are three base cases for equality with zero, followed by an inductive case. This code is rather elegant, but the membership function that makes use of it is not.

The context membership function (called `Contains` in code) requires a data structure and two branch points, one for checking if we're reached the end of the list, and one for checking if we've reached our target variable. TraitLang only allows one branch point and one return value per branch. To get around this, we create two functions, the first one passing the results of its check to the second as parameters. The second then branches once based on all the information. Even for this simple function, the code is difficult to read. We work through it below.

Figure 11 shows the code of the context membership function. The first two lines are the data structure, implemented like a linked list. It can be empty or contain the natural number id of a variable, a type, and the next node. The `Contains` function takes an id and returns an optional value, in the case of calling it on an empty context, it returns `None`. When called on a non-empty context, `Contains` checks for equality with the target, passes that result, along with the type, as parameters to `Contains2` called on the rest of the context and returns the result of `Contains2`. `Contains2` has enough information

```
struct EmptyCtx;
struct TypeCtx<Id,Typ,Next>(Id,Typ,Next);

trait Contains<Id> { type Result; }
impl<N> Contains<N> for EmptyCtx
{ type Result=None; }

impl<Check,First,Typ,Next,Eq,R>
Contains<Check> for TypeCtx<First,Typ,Next> where
    Check: NatEq<First,Eq=Eq>,
    Next: Contains2<Eq,Typ,Check,Result=R>,
{ type Result=R; }

trait Contains2<Eq,Map,Check> { type Result; }

impl<Map,C,Cxt> Contains2<True,Map,C> for Cxt
{ type Result=Some<Map>; }

impl<Map,C> Contains2<False,Map,C> for EmptyCtx
{ type Result=None; }

impl<Check,First,T,Typ,Next,Eq,R>
Contains2<False,T,Check> for TypeCtx<First,Typ,Next> where
    Check: NatEq<First,Eq=Eq>,
    Next: Contains2<Eq,Typ,Check,Result=R>,
{ type Result=R; }
```

Figure 11: Membership function for contexts

to chose one of the three end-points of the algorithm. If the prior equality check was true, it returns the prior type (called `map` in the code) regardless of the rest of the context. If the check was false and the rest of the context is empty, it returns `None`. If their is more context to process, it does an equality check on the next value and calls itself recursively the same way `Contains` did.

5.2 Type checking

Type checking starts by checking that the AST is well-formed. The code is in Figure 12. Most rules define the syntax that we're using as well-formed if its sub-syntax is well-formed. The exception is the `Lam` case, which requires a variable as its first item. There are different traits used for different parts of the syntax, like `WFNat` and `WFType`, to make sure they are used in the proper places. There is little complexity to the code, it mostly tags some constructions as appropriate.

Our type checking code in Figure 14 is among the most simple and elegant in this paper, because we are able to directly mirror the type checking rules. We use a trait called `Typed` parametrized by a context. Premises are found in the "where" clauses with the syntax form preceding them. The resulting type is an associated type, to make sure that there is only one type per value. Otherwise, the rules are direct translations of the typing rules for the lambda calculus. For example, the last rule, `App`, requires that the first expression (`E1`) be an `Arrow` type (from `T1` to `T2`) in the current context (`Ctx`), and the second expression (`E2`) be of the type at the front of the arrow (`T1`), also in the current context. The type of the `App` expression is the type of the end of the arrow (`T2`).

The last piece of or type checker is the code to invoke it, requiring a type for our AST. Figure 13 shows what looks like runtime functions, but they contain no code. Instead, each requires that

```
trait WFNat {}
impl WFNat for Zero {}
impl<N:WFNat> WFNat for Succ<N> {}

trait WFType {}

struct Number;
impl WFType for Number {}

struct Arrow<T1,T2>(T1,T2);
impl<T1:WFType,T2:WFType> WFType for Arrow<T1,T2> {}

trait Expr {}

struct Num<N>(N);
impl<N:WFNat> Expr for Num<N> {}

struct Plus<N1,N2>(N1,N2);
impl<N1:Expr,N2:Expr> Expr for Plus<N1,N2> {}

struct Var<N>(N);
impl<N:WFNat> Expr for Var<N> {}

struct Lam<V,T,E>(V,T,E);
impl<N:WFNat,T:WFType,E:Expr> Expr for Lam<Var<N>,T,E> {}

struct App<E1,E2>(E1,E2);
impl<E1:Expr,E2:Expr> Expr for App<E1,E2> {}
```

Figure 12: Well-formedness checking logic

```
fn is_wf_expr<E:Expr>(e:&E) {}
fn is_wf_type<T:WFType>(t:&T) {}
fn is_typed<E,T>(e:&E,t:&T) where
        E:Typed<EmptyCtx,T=T>
{}
```

Figure 13: Functions to invoke the type checker

the parameters satisfy some trait. This will activate the compiler's trait resolution, type-checking our macro-generated AST. The first two functions check the well-formedness of an expression and a type, respectively. The last checks that the given expression has the given type, in the empty context.

6 DISCUSSION

Running our code may require an initial conversion, but would otherwise be standard. When defining a struct in Rust, it generates a singleton constructor (parametrized as appropriate) with the same name. This is what we've been using in our AST nodes, while the struct itself is used in our implementation of traits. From the runtime perspective, each AST node is a different type, which can make coding up the evaluation difficult. A conversion to an AST using tagged variants of types (Rust's `enum`) would simplify the eval code.

The ability to define and use a type system (for the type-level functions) seems really powerful, but ultimately must support the more mundane code that is more commonly written. A type-level type system may be too far removed to be useful. We imagine that a dependent type system may be useful here to prove properties about code as it's compiled. We have experimented with such a system, but not thoroughly enough for this paper, and without

```
trait Typed<Ctx> { type T; }

impl<N,Ctx> Typed<Ctx> for Num<N> { type T=Number; }

impl<N1,N2,Ctx> Typed<Ctx> for Plus<N1,N2> where
    N1:Typed<Ctx,T=Number>,
    N2:Typed<Ctx,T=Number>,
{ type T=Number; }

impl<N,Ctx,T> Typed<Ctx> for Var<N> where
    Ctx:Contains<N,Result=Some<T>>
{ type T=T; }

impl<Ctx,N,T1,T2,E> Typed<Ctx> for Lam<Var<N>,T1,E> where
    E:Typed<TypeCtx<N,T1,Ctx>,T=T2>,
{ type T=Arrow<T1,T2>; }

impl<Ctx,E1,E2,T1,T2> Typed<Ctx> for App<E1,E2> where
    E1:Typed<Ctx,T=Arrow<T1,T2>>,
    E2:Typed<Ctx,T=T1>
{ type T=T2; }
```

Figure 14: Type checking rules for our DSL

Rust syntactic support, it seems too complex for all but the most important tasks.

7 RELATED WORK

A similar approach to higher-kinded types is [2]. In that gist, a trait is used to represent the HKT. It is very similar to the author's equivalent [5], but it has additional complexity so that a built-in type can be passed to a function directly as a type constructor. Working off of an existing type rather than a new type function doesn't allow for specializing for its use case.

A similar project is "turnstile" [3] for the Racket language. The authors similarly take advantage of a compile-time algorithm to do type checking. In their case, they use Racket's macro expander, adding typing annotations to the syntax objects it creates. Rust traits provide a declarative way to add meta-data to types, allowing much simpler use, and the ability to follow the typing rules more directly. On the other hand, Racket has more advanced capability in its macro system, allowing a layer of abstraction that lets the user follow typing rules as well. Racket also provides a mechanism for generating useful error messages.

8 CONCLUSION

We have shown how to use Rust traits to define first-class type functions. We have shown the implementation of a DSL with a shallow embedding in Rust. The Rust parser, through the macro system, was used to parse it. The Rust compile-time algorithms were used to type check it. And we suggested a way for the Rust runtime system to run the code, since that is a far more common task. A full demo can be run and modified from [6].

It is our hope that these explorations will inform further language design. Rust's traits were not originally intended to be used this way, as is obvious looking at error messages of some programs that fail to type check. We hope that type-level programming becomes more valuable in the future, and use cases like those demonstrated will highlight areas to work on.

REFERENCES

[1] Rust language. https://www.rust-lang.org/, 2015.

[2] 14427. Higher-kinded type trait. https://gist.github.com/14427/af90a21b917d2892eace, 2018.

[3] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 694–705, 2017.

[4] freebroccolo. Simulated higher-kinded types for rust. https://github.com/freebroccolo/hkt.rs, 2015.

[5] Kyle Headley. hkt tree. https://gist.github.com/kyleheadley/dbc2469182c61eedb481fc034a55223f, 2018.

[6] Kyle Headley. Traitlang full code demo. https://play.rust-lang.org/?gist=a0f5ec8999cb08de3842500a9aa959a7&version=stable&mode=debug&edition=2015, 2018.

[7] Joshua Liebow-Feeser. Rust has higher kinded types already... sort of. https://joshlf.com/post/2018/10/18/rust-higher-kinded-types-already/, 2018.