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.

$$\frac{read\ ro\ \langle rf, ps, h\rangle\ v \quad bb, \langle rf, p, v :: s, h\rangle \to_{bb} S}{push\ ro : bb, \langle rf, p, s, h\rangle \to_{bb} S} \quad \text{(Push)}$$

$$\frac{write\ wo\ w \langle rf, p, s, h\rangle S' \quad bb, S' \to_{bb} S}{pop\ wo : bb, \langle rf, p, w :: s, h\rangle \to_{bb} S} \quad \text{(Pop)}$$

$$\frac{\forall i < n, f + i \notin \mathrm{dom}\,(h) \quad write\ wo\ f \langle rf, p, s, zeroes\ n\ f +\!\!+ h\rangle S' \quad bb, S' \to_{bb} S}{new\ n\ wo : bb, \langle rf, p, s, h\rangle \to_{bb} S} \quad \text{(New)}$$

$$\frac{read\ ro\ s\ v \quad write\ wo\ v\ S\ S' \quad bb, S' \to_{bb} S''}{mov\ ro\ wo : bb, S \to_{bb} S''} \quad \text{(Mov)}$$

$$\frac{read\ ro\ S\ 0 \quad write\ ip\ k\ S\ S'}{jump\ \ (ro, k)\ \ j, S \to_{bb} S'} \quad \text{(Jump 0)}$$

$$\frac{l > 0 \quad read\ ro\ S\ l \quad read\ j\ S\ k \quad write\ ip\ k\ S\ S'}{jump\ \ (ro, k')\ \ j, S \to_{bb} S'} \quad \text{(Jump S)}$$

$$\frac{read\ ro\ S\ l \quad write\ ip\ l\ S\ S'}{jump\ ro : S \to_{bb} S'} \quad \text{(Jump)}$$

$$\frac{read\ ip\ \langle rf, p, s, h\rangle\ k \quad p\,[k] = bb \quad bb, \langle rf, p, s, h\rangle \to_{bb} S'}{\langle rf, p, s, h\rangle \to S'} \quad \text{(Enter)}$$

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 fÃijr 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 userâĂŹs 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.