Brack: A Verified Compiler for Scheme via CakeML

Pascal Y. Lasnier
University of Cambridge
Cambridge, United Kingdom
pyl37@cantab.ac.uk

Jeremy Yallop
University of Cambridge
Cambridge, United Kingdom
jeremy.yallop@cl.cam.ac.uk

Magnus O. Myreen
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden
myreen@chalmers.se

Abstract

This paper describes Brack, which is a new verified compiler for Scheme. Brack compiles a substantial subset of Scheme, including first-class continuations, recursive bindings, first-class functions, mutable local variables, and lists, to CakeML, from where programs can be compiled to machine code. Compilation from Scheme to CakeML is based around a continuation-passing-style (CPS) transformation that naturally arises from Scheme's small-step semantics. We have formally established the correctness of Brack in the HOL4 theorem prover.

CCS Concepts: • **Software and its engineering** → **Software verification**; *Compilers*; Control structures; • **Theory of computation** → Abstract machines.

Keywords: compiler verification, Scheme, CPS transform, first-class continuations, HOL4

ACM Reference Format:
Pascal Y. Lasnier, Jeremy Yallop, and Magnus O. Myreen. 2026. Brack: A Verified Compiler for Scheme via CakeML. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779031.3779098

1 Introduction

Compilers play a key role in software development: almost all code that is executed by a computer is the result of compilation. To be confident that the programs we run reflect our intentions, we must be confident in the correctness of compilers. Unfortunately, realistic compilers typically contain bugs [39], and many compiler bugs silently introduce security problems into programs [38].

Compilers are often extensively tested by their developers, by researchers, and by users. However the space of programs is extremely large and complex, with subtle interactions between language features, and even thorough testing can only explore a tiny fraction of the space. In contrast, formal proof can guarantee that programs are never compiled incorrectly [27]. In particular, a mechanised proof that compilation preserves semantics provides the highest level of assurance that compilation will not introduce bugs.

Given the clear advantages of verified compilation, it is unfortunate that developing verified compilers is very costly. Realistic modern compilers do not typically generate executable code directly from the source; instead, they translate between a series of intermediate languages. In a verified compiler, each of these intermediate languages must be given a formal semantics, and each translation step must be formally proved to preserve semantics. Developing these proofs requires substantial resources, and there are consequently very few complete verified compilers: the most notable ones are the C compiler CompCert [27] and the ML compiler CakeML [24].

Reducing the Cost of Verified Compilation. Techniques that reduce the cost of developing verified compilers are likely to improve their adoption, ultimately leading to a general increase in software quality. This article considers two techniques that make it easier to develop a verified compiler for a high-level language:

Backend reuse Large compilers typically include many components that are language independent, such as lowering, middle-end and peephole optimisations, and instruction selection. Reusing the language-independent portions of an existing verified compiler can lower the cost of development.

Refunctionalisation Ager et al. [1] systematically relate transition semantics to higher-order functional programs. One challenging part of developing compilers for high-level languages is the treatment of control operators such as **call/cc**. Refunctionalisation offers a straightforward way to reuse the backend of a compiler for a higher-order language to verify compilation for a language with arbitrary control operators.

We have used these techniques to build a new verified Scheme compiler, Brack[1]. The Brack compiler is complete: compilation is verified from the Scheme source language down to machine code. Brack achieves completeness at low cost by targeting CakeML [24], relying on CakeML's

[1]Brack, or barmbrack, is a traditional Irish fruitcake often baked as a part of Halloween celebrations.

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779098

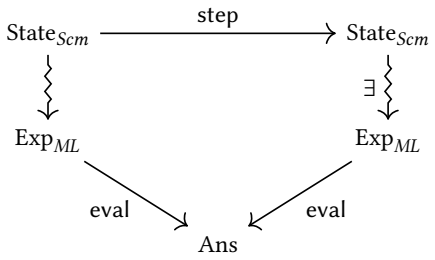


Figure 1. Bʀᴀᴄᴋ semantic preservation as a commutative diagram. Scheme states relate $\rightsquigarrow$ to ML expressions, which have equal evaluations after a Scheme small step.

existing backend for most of compilation. The definition of Bʀᴀᴄᴋ is systematically derived from the operational semantics in the Scheme specification via refunctionalisation, resulting in a compiler which produces ML code in continuation-passing style and faithfully preserves the meaning of programs. Bʀᴀᴄᴋ supports a substantial subset of Scheme that includes first-class continuations, recursive let-bindings, first-class functions, mutable local variables, and lists. We believe that Bʀᴀᴄᴋ is the first compiler with verified support for *dynamic typing* and *first-class continuations*.

Our work has been carried out in the HOL4 theorem prover [34], and the resulting proof scripts are available as supplementary material [25].

2 Approach

This work combines the following technical ingredients.

1. The Scheme Formal Semantics. We define the semantics of Bʀᴀᴄᴋ as a CESK abstract machine, closely following the formal operational small-step semantics in R^{6}RS, the Sixth Revised Report on Scheme (Section 3).

2. A New Compiler from Scheme to ML. Bʀᴀᴄᴋ compiles Scheme using a continuation-passing style (CPS) transform, targeting ML. Following Ager et al. [1], we systematically derive a CPS transform by refunctionalising the CESK-machine semantic definition of Scheme (Section 4).

3. The CakeML Verified Compiler. Our work builds on the CakeML verified compiler for ML. Building on CakeML allows us to focus on the distinctive features of Scheme, avoiding the need to implement and verify the language-independent compiler back-end (Section 4.1).

4. The HOL4 Theorem Prover. Since CakeML is written in HOL4, we also implement Bʀᴀᴄᴋ in HOL4, making use of the fact that HOL4 definitions are also written in ML in the derivation of our compiler (Section 4.3).

5. A New Proof of Semantic Preservation. We establish semantic preservation for the CPS transform by defining a relation between Scheme small-step abstract machine states and CPS ML expressions capturing those states, and then showing that the Bʀᴀᴄᴋ implementation satisfies the relation. The semantic preservation proof shows that, for an ML expression of a Scheme program, there exists an ML expression of that program after an abstract machine step that has equal evaluation (Figure 1).

A key finding of our work is that the semantics-directed compiler derivation makes the semantic preservation proof easy to complete, and the case for first-class continuations is surprisingly trivial to prove (Section 5).

6. The CakeML Executable Generation Tool. CakeML provides a translation tool that generates an executable file from HOL4 definitions [23, 29], and we use this tool to generate an executable for Bʀᴀᴄᴋ.

Bʀᴀᴄᴋ supports a substantial subset of Scheme, allowing us to compile some interesting data-processing programs (Section 6).

3 Scheme and First-Class Continuations

Scheme is a dynamically-typed, impure dialect of Lisp, with support for first-class continuations using the `call-with-current-continuation`, or **call/cc**, procedure. First-class continuations enable interesting programming patterns not otherwise available to functional languages, by offering direct manipulation over Scheme's control flow. For example, first-class continuations may be used to implement backtracking:

```
(begin
  (set! x (call/cc (lambda (ec) (begin
                                  (set! checkpoint ec)
                                  (rand)))))
  (let ((y (compute x)))
    (if (pred y)
        y
        (checkpoint (rand)))))
```

In this code fragment, **call/cc** captures the continuation at the point of assigning some value to x. The captured continuation can be thought of a snapshot of the evaluation context at that point, which in this case has the form (**begin** (**set!** x [ ]) ...), where the square brackets denote the *hole* of the evaluation context, which gets filled by a value. Scheme exposes this continuation to the program in the form of an escape procedure, which it passes to the argument of **call/cc**. Here, that escape procedure is assigned to the variable *checkpoint*, which can be invoked further along in program execution, before a random value chosen by *rand* is returned to be assigned to x.

With a checkpoint escape procedure established, it may be invoked with a new value to assign to x in order to restart

computation from the checkpoint. In this example, we simply check that a value computed from x satisfies some predicate, and restart from the checkpoint with a new value if it does not. This backtracking pattern is even more helpful for more complex computations based on x, possibly with multiple points of failure in nested expressions, where invoking the escape procedure is simpler than unrolling the call stack.

As this example shows, the often imperative nature of patterns using control operators such as **call/cc**, though powerful, make programs using them behave in complex ways. This complexity makes implementing and verifying these operators appear challenging. However, Scheme is a relatively small language, and the semantics of **call/cc** is actually straightforward. Much of the complexity of Scheme programs using **call/cc** arises from its combination with dynamic typing and mutable variables. This paper seeks to show that, since the complex behaviour of control operators is emergent, not intrinsic, verified compilation of continuation-based languages like Scheme is more feasible than it may seem at first glance.

3.1 Semantics

To mechanise a semantic preservation proof, we must first establish a semantic definition of Scheme. Bracк's semantics are based on the formal small-step semantics of the Sixth Revised Report on Scheme (R^{6}RS) [35], the most recent operational semantics for Scheme[2]. Since Bracк's semantics follow the formal R^{6}RS semantics, all programs in the supported subset have fully specified behaviour when compiled with Bracк. Some such programs, e.g. those involving let-binding reinitialisation using **call/cc**, are treated inconsistently by industrial Scheme compilers, which follow the underspecified informal R^{6}RS semantics rather than the formal semantics. We include a program demonstrating the inconsistency in the supplementary material for this paper [25].

Figure 2 shows the subset of R^{6}RS supported by Bracк. Besides continuations, the subset supports booleans, integers, and lists, allowing Bracк to support interesting data-processing programs. In future, we plan to support additional features from R^{6}RS including exception handling and other control operators (`apply`, `call-with-values`, `dynamic-wind`), where exception handlers and `dynamic-wind` in particular a require finer granularity of representation of continuations than is necessary for **call/cc**.

Data is primarily inserted into programs through self-quoting values (*sqv*): integers, booleans, or the empty list **null**. Primitive procedures (*pproc*) consist of arithmetic procedures on integers (*aproc*), data equivalence (**eqv**?), list/pair operations (**cons**, **car**, **cdr**, **null**?, **pair**?), and **call/cc**. Expressions e consist of these atomic terms, variables x, if-expressions, procedure application (as a list of expressions), expression sequencing (**begin**), variable assignment (**set!**), lambda expressions, and recursive let-bindings (**letrec**/**letrec***). Lambda expression parameters take the form of formals f, which capture some or all of the passed arguments as a list.

Internally, Bracк's semantics are defined as a CESK abstract machine (**C**ontrol, **E**nvironment, **S**tore, (**K**)ontinuation) [14]. Figure 3 gives the grammar of machine states.

Evaluation contexts E directly correspond to continuations as a nested sequence of *stack frames*, with attached environments (denoted by a prefixed $\{\rho\}$), corresponding to different partially evaluated expressions. The evaluating term

$$
\begin{array}{lcl}
x & \in & \text{Variable} \\
f & ::= & (x\ x\ ...) \mid (x\ x\ ...\cdot x) \mid x \\
n & \in & \mathbb{Z} \\
sqv & ::= & n \mid \#\mathbf{t} \mid \#\mathbf{f} \mid \mathbf{null} \\
aproc & ::= & + \mid - \mid * \\
pproc & ::= & aproc \mid \mathbf{call/cc} \mid \mathbf{cons} \mid \mathbf{car} \mid \mathbf{cdr} \\
 & \mid & \mathbf{eqv?} \mid \mathbf{null?} \mid \mathbf{pair?} \\
 & & \\
e & ::= & x \mid sqv \mid pproc \mid (\mathbf{if}\ e\ e\ e) \mid (e\ e\ ...) \\
 & \mid & (\mathbf{begin}\ e\ e\ ...) \mid (\mathbf{lambda}\ f\ e) \mid (\mathbf{set!}\ x\ e) \\
 & \mid & (\mathbf{letrec}\ ((x\ e)\ (x\ e)\ ...)\ e) \\
 & \mid & (\mathbf{letrec*}\ ((x\ e)\ (x\ e)\ ...)\ e)
\end{array}
$$

Figure 2. Expression grammar.

$$
\begin{array}{lcl}
\ell & \in & \mathbb{L} \\
\rho & : & \text{Variable} \rightharpoonup \mathbb{L} \\
 & & \\
proc & ::= & pproc \mid (\mathbf{proc}\ f\ \{\rho\}e) \mid (\mathbf{throw}\ E) \\
v & ::= & sqv \mid proc \mid (\mathbf{pp}\ \ell) \mid \mathbf{unspecified} \\
 & & \\
E & ::= & [\,] \mid \{\rho\}(\mathbf{if}\ E\ e\ e) \mid \{\rho\}(v\ ...\ E\ e\ ...) \\
 & \mid & \{\rho\}(\mathbf{begin}\ E\ e\ e\ ...) \mid \{\rho\}(\mathbf{set!}\ x\ E) \\
 & \mid & \{\rho\}(\mathbf{letrec}\ ((x\ v)\ ...\ (x\ E)\ (x\ e)\ ...)\ e) \\
 & & \\
sv & \in & \text{StoreVal} \\
sv & ::= & v \mid (\mathbf{cons}\ v\ v) \mid \mathbf{bh} \\
\sigma & : & \mathbb{L} \rightharpoonup \text{StoreVal} \\
 & & \\
s & \in & \text{String} \\
c & ::= & \{\rho\}e \mid v \mid (\mathbf{raise}\ s)
\end{array}
$$

Figure 3. Evaluation grammar.

[2]The most recent Scheme revision R^{7}RS uses formal denotational semantics, which is harder to extract a small subset from. Both specifications are equivalent for the subset of Scheme supported by Bracк.

c lies in the hole [ ] of an evaluation context E and corresponds to the control string of the CESK machine. This evaluating term may be an expression e, value v, or exception (**raise** s). The expression has an environment ρ attached to it, denoted by $\{\rho\}e$. The environment ρ is a partial function from variables to store locations $\mathbb{L}$.

Values v are irreducible terms that may fill the hole of an evaluation context. Values may be either the atomic *sqv* or *pproc* terms from the expression grammar, lambda expressions as closures (**proc**) with a captured environment, continuations captured by **call/cc** as escape procedures (**throw**), or immutable pair pointers (**pp**) to store locations ℓ. The special **unspecified** value is the result of a **set!** expression, which has an unspecified value in R^{6}RS.

The locations in the range of an environment ρ are mapped to store entries by another partial function σ representing the store. Store entries are either values v or black holes **bh** used for **letrec** initialisation.

A CESK machine state consists of a combination of these components:

$$\langle \sigma, E[c] \rangle$$

and small steps in the operational semantics are defined by transitions between these states.

3.1.1 Correspondence with R^{6}RS.

In contrast with the CESK machine semantics of Brack, the semantics specified by R^{6}RS is more like a CS machine (control, store) reduction semantics under Felleisen & Friedman's formulation, with substitution rules rather than an environment ρ, and combined control strings with contexts. This section recapitulates the well-understood equivalence between these definitions [15].

First, there is a natural correspondence between environment and substitution semantics [14, 15]. Substitutions propagate to all sub-expressions, e.g. when evaluating the conditional expression e in $\{x_1 \mapsto bp_1 ...\}$(**if** e e_1 e_2), the substitution implicitly applies to the conditional expression $\{x_1 \mapsto bp_1 ...\}e$.

To satisfy this propagation condition in a CESK machine, an environment must be attached to every stack frame in a continuation corresponding to levels of expression nesting, and environments are propagated from expressions to their sub-expressions. For example, here are the abstract machine steps for an if-expression, with the environment explicitly propagated:

$$\langle \sigma, E[\{\rho\}(\textbf{if } e\ e_1\ e_2)] \rangle \longrightarrow \langle \sigma, E[\{\rho\}(\textbf{if } [\{\rho\}e]\ e_1\ e_2)] \rangle$$
$$\langle \sigma, E[\{\rho\}(\textbf{if } [\#\textbf{f}]\ e_1\ e_2)] \rangle \longrightarrow \langle \sigma, E[\{\rho\}e_2] \rangle$$
$$\langle \sigma, E[\{\rho\}(\textbf{if } [v]\ e_1\ e_2)] \rangle \longrightarrow \langle \sigma, E[\{\rho\}e_1] \rangle \quad v \neq \#\textbf{f}$$

The explicit attachment of environments to terms in the CESK machine is also seen in the Brack rule for **lambda**, which captures the environment ρ:

$$\langle \sigma, E[\{\rho\}(\textbf{lambda } f\ e)] \rangle \longrightarrow \langle \sigma, E[(\textbf{proc } f\ \{\rho\}e)] \rangle$$

R^{6}RS does not explicitly capture the environment when evaluating **lambda**, but does so implicitly, because substitutions propagate to all sub-expressions, including values. Since these substitutions only affect lambdas, not other values, Brack represents an environment attached to a lambda as a **proc** value, rather than attaching environments to every value.

Also, Brack always allocates arguments passed to lambda expressions in the store, omitting the R^{6}RS rule that directly maps immutable variables to values. Consequently, while R^{6}RS substitutions substitute variables for store locations or immutable values, Brack environments exclusively map variables to store locations.

Finally, the explicit decomposition step for an if-expression above specifies how the evaluation context is extended with a conditional stack frame. Such steps are not present in the R^{6}RS semantics and are implicit from the form of evaluation contexts, effectively combining the control string and evaluation context in the abstract machine state [15]. The two forms of abstract machine are equivalent by the notion of *unique decomposition* of the evaluation contexts [37].

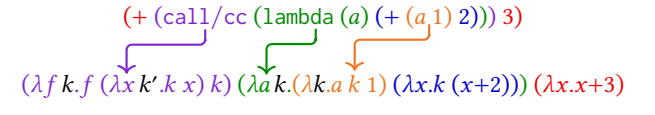
Brack's semantics otherwise follows the R^{6}RS formal semantics, with left-to-right evaluation order.

3.2 CPS Transform

Continuation-passing style (CPS) is a program representation in which functions do not return values, but instead pass values to continuations by tail-call. All functions must consequently take a continuation as an argument and eventually call into a continuation with some value. Brack performs a CPS transform during compilation, turning expressions into CPS.

Plotkin's seminal work on the CPS transform uses it as a means to unify call-by-name and call-by-value semantics [32]. One important aspect of CPS in this context is that it makes order of evaluation explicit, hence offers a means to *simulate* call-by-value lambda calculus in call-by-name calculus using continuations.

The more relevant simulation property for Scheme, however, is that continuations can simulate control operators in a calculus without them [17]. Here is a (slightly simplified) CPS transform of a simple example program using **call/cc**:



In CPS, continuations are represented as functions (e.g. $(+\ [\ ]\ 3)$ is represented by $\lambda x.x+3$), and the behaviour of **call/cc** is made explicit by its manipulation of these continuations captured into arguments k. **call/cc** passes its argument an escape procedure $(\lambda x\, k'.k\ x)$ which captures the current continuation, and escapes the existing continuation when called (in this case, corresponding to $(+\ (+\ [\ ]\ 2)\ 3)$) by

discarding it and applying its captured continuation instead. In this example, the final result is $(+\ [1]\ 3) = 4$.

CPS transforms are one of several approaches to compiling continuation-based language features, others largely being forms of runtime stack manipulation techniques [3, 21]. These alternatives are often preferred to CPS because unoptimised CPS code can be slow [13].

However, the CPS transform has a distinct advantage for verified compilation: its naturality in representing continuations for implementing control operators. To show this, we elaborate on a deep connection between small-step semantics and CPS which lends itself to compilation to a lambda-calculus-like language (ML) and to semantic preservation proof using simulation.

4 Compiler Implementation

Brack takes advantage of an intrinsic property relating operational small-step semantics (by which Scheme is specified) and CPS, which is that small-step semantics are a *defunctionalisation* of CPS, a notion first explored by Reynolds [33]. Particularly, there is a well-established transform [1, 4, 5, 7] between small-step abstract machines, which are commonly used to define continuation semantics, and higher-order CPS definitional interpreters, which are similar to functional big-step semantics but are defined in CPS and hence pass around a continuation representing the rest of evaluation.

We note that a CPS definitional interpreter may be rewritten, with some changes, into a compositional denotational transform from Scheme programs to HOL4 CPS programs. However, because HOL4 terms correspond to pure ML, we may instead embed the result of the transform into ML and construct an AST, resulting in a compiler. We then make additional optimisations, including direct compilation of mutable Scheme variables into ML `ref` variables.

Hence, the CPS transform performed by Brack is derived directly from the Scheme semantics, and requires little additional consideration. This construction becomes especially useful once we consider non-linear continuations such as those involving exceptions and **call/cc**, which are otherwise challenging to correctly implement in a compiler because of their non-linearity. The fact that the compiler is derived from the semantics immediately benefits the semantic preservation proof, because the proof largely becomes a reconciliation of the continuation definitions and the abstract machine definitions they directly correspond to.

4.1 CakeML

Brack is built upon the existing verified ML compiler CakeML [24], and uses ML as a compilation target. Compiling to ML allows us to focus on semantic preservation proofs for high-level transformations close to the source semantics, without the burden of proving semantic preservation to machine code, which has already been established for CakeML. This approach has previously been used for PureCake [22], a compiler for a Haskell-like language.

$$
\begin{array}{lcl}
x & \in & \text{Variable} \\
n & \in & \mathbb{Z} \\
C & \in & \text{Constructor} \\
P & ::= & x \mid n \mid C\ P\ \ldots \mid \_ \\
\\
e & ::= & x \mid n \mid e + e \mid e - e \mid e * e \mid C\ e\ \ldots \mid \lambda x.\, e \mid e\ e \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{letrec}\ x\ x = e, \ldots\ \texttt{in}\ e \\
 & \mid & e;\ e \mid \texttt{case}\ e\ \texttt{of}\ \mid P \Rightarrow e \mid \ldots \\
 & \mid & \texttt{ref}\ e \mid\ !e \mid e := e \\
\\
\ell & \in & \mathbb{L} \\
\rho & : & \text{Variable} \rightharpoonup \text{Value} \\
\sigma & : & \mathbb{L} \rightharpoonup \text{Value} \\
\\
v & \in & \text{Value} \\
v & ::= & n \mid C\ v\ \ldots \mid \ell \mid \Lambda x.\ \{\rho\}\, e \mid \texttt{fix}_x\ \rho\ \langle\langle x, x, e\rangle, \ldots\rangle
\end{array}
$$

Figure 4. ML AST and evaluation grammar.

Targeting ML for compilation means that Brack must use CakeML's existing ML semantics to state the Scheme-to-ML semantic preservation theorem. CakeML uses a *functional big-step semantics* [30], where the evaluation of ML expressions is processed through a recursively-defined eval_{ML} function. This functional definition provides a notion of equality between expression evaluations, regardless of whether they terminate or diverge, and so may be used to equate simulated Scheme programs that may not terminate.

Figure 4 gives a grammar for the subset of ML used by Brack, including variables x, integers n, arithmetic operations, constructors C with zero or more term arguments, lambda abstraction, unary application, non-recursive and recursive let-bindings, sequencing with '; ', case match expressions, and mutable references. Programs evaluate with a store σ and environment ρ using the function $\text{eval}_{ML}(\sigma, \rho, e)$, which, if the program terminates, returns a store and value $\langle\sigma, v\rangle$. Values include integers, constructors with zero or more value arguments, references to store locations ℓ, and closures Λ and recursive closures $\texttt{fix}_x$, both of which capture an environment ρ. Recursive closures may contain multiple mutually recursive functions defined by a `letrec` expression, and the subscript attached to `fix` selects one of them.

CakeML's evaluation largely follows Standard ML, but with right-to-left evaluation order.

4.1.1 HOL4. Definitions for HOL4 terms are written in a purely functional variant of ML, and so the compiler implementations for CakeML, Brack, etc. are ML programs in HOL4 definitions. The CakeML framework includes a translation tool that produces compiler executables from these pure ML implementation definitions [29].

A property of HOL4 that is especially relevant to the present work is the fact that the pure ML used in HOL4 terms, as a *defining language* for the Scheme semantics, is strictly a subset of the *target language* of compilation: CakeML. This property enables the derivation of a compiler to CakeML directly from a HOL4 definition for a definitional interpreter, as explained in Section 4.3.

4.2 Refunctionalisation

One way to succinctly model the behaviour of control operators in continuation-based languages such as Scheme is to use a CEK machine-based small-step semantics [15]. For example, the core reduction rules that capture the behaviour of **call/cc** in Scheme (without `dynamic-wind`, which introduces entry and exit procedures to continuations) are cunningly simple compared to the emergent program complexity they introduce:

$$\begin{array}{lcl} E[(\textbf{call/cc}\ v)] & \longrightarrow & E[(v\ [(\textbf{throw}\ E)])] \\ E[((\textbf{throw}\ E')\ v)] & \longrightarrow & E'[v] \end{array}$$

Felleisen et al. have shown that CPS can similarly capture control operator behaviours with a simple CPS transform to lambda calculus [17]. The similarity of representation between reduction semantics and CPS is not a coincidence; Danvy has thoroughly documented the nature of evaluation context-based reduction semantics as a *defunctionalisation*[3] of CPS interpretation [4, 5], building upon Reynolds' seminal work on definitional interpreters [33].

In one particularly relevant instance of this relationship, Ager et al. demonstrate that a CEK machine may be *refunctionalised* into a CPS interpreter, based on the reduction steps corresponding to particular continuations defined by the abstract machine [1]. Ager et al.'s refunctionalisation technique may be similarly applied to the CESK-machine semantic definition of Scheme in Brack, with some key differences from the original technique. To demonstrate, we define the small steps for evaluating an assignment expression in Scheme:

$$\begin{array}{l} \mathsf{step}_{eval}\ (\sigma, E, \{\rho\}\ (\textbf{set!}\ x\ e)) = \\ \quad \langle \sigma, E[\{\rho\}\ (\textbf{set!}\ x\ [])], \{\rho\}\ e\rangle \end{array}$$

$$\begin{array}{l} \mathsf{step}_{cont}\ (\sigma, E[\{\rho\}\ (\textbf{set!}\ x\ [])], v) = \\ \quad \langle \sigma\{\rho(x) \mapsto v\}, E, \textbf{unspecified}\rangle \end{array}$$

[3]Defunctionalisation is a transformation from higher-order programs to first-order programs [7], where "*a first-class function is introduced with a constructor* [...] *and it is eliminated with a case-expression dispatching over the corresponding constructors*".

The notation $f\{x \mapsto y\}$ denotes an update of the partial function f to map x to y. For visual consistency, we denote small steps using a function step which operates on a store, evaluation context, and control string as a parameters. From this point onwards, we also colour Scheme syntax and runtime constructs in blue, to distinguish them from HOL4 meta-language terms in black. We also divide the cases for the reduction function step into those which *decompose* an expression into a stack frame and an inner expression (step_{eval}), and those which *continue* with a value in an evaluation context (step_{cont}), though they are both part of the definition for step.

In Ager et al.'s technique, the evaluation context E of a CEK machine is refunctionalised as a closure k that serves as a continuation. In our adaption, continuations must take an additional argument representing the store. The purpose of this change is to track the state of mutable variables in Scheme; it reflects the presence of the store in a continuing state $\langle \sigma, E, v\rangle$ in the small-step semantics. For example, here is the refunctionalisation of the semantic definition for assignment above, which produces one case in a CPS definitional interpreter eval_{CPS}:

$$\begin{array}{l} \mathsf{eval}_{CPS}\ (\textbf{set!}\ x\ e)\ \sigma\ \rho\ k = \\ \quad \mathsf{eval}_{CPS}\ e\ \sigma\ \rho\ (\lambda\sigma'\ v.\ k\ (\sigma'\{\rho(x) \mapsto v\})\ \textbf{unspecified}) \end{array}$$

In general, in the application of Ager et al.'s technique to a small-step semantics defined as a CESK machine, the cases in the resulting definitional interpreter eval_{CPS} are based on the decomposition steps step_{eval}. The translation of a step step that produces another expression state of the form $\langle \sigma, E, \{\rho\}\ e\rangle$ is a nested call to eval_{CPS} with the corresponding arguments.

Importantly, when an evaluation context is extended by a decomposition step step_{eval}, the nested evaluator call is passed an extended refunctionalised continuation. The body of this continuation matches the body of the continuation step step_{cont} case for the correspond evaluation context.

Lastly, returning a *value* state of the form $\langle \sigma, E, v\rangle$ from step corresponds to applying the refunctionalised continuation to the value. This situation arises, for example, with expressions that immediately return values, such as lambda expressions:

$$\begin{array}{l} \mathsf{step}_{eval}\ (\sigma, E, \{\rho\}\ (\textbf{lambda}\ f\ e)) = \langle \sigma, E, (\textbf{proc}\ f\ \{\rho\}\ e)\rangle \\ \mathsf{eval}_{CPS}\ (\textbf{lambda}\ f\ e)\ \sigma\ \rho\ k = k\ \sigma\ (\textbf{proc}\ f\ \{\rho\}\ e) \end{array}$$

Other logic within the semantics is preserved in refunctionalisation, such as the arity checking of applications imposed by Scheme's dynamic typing. Here are some cases of the semantic definition of application, showing evaluation of the function expression, evaluation of an argument, and application of a **proc** value to evaluated arguments:

$$\text{step}_{eval}\ (\sigma, E, \{\rho\}\ (e\ e_1\ ...)) =$$
$$\langle \sigma, E[\{\rho\}\ ([]\ e_1\ ...)], \{\rho\}\ e\rangle$$

$$\text{step}_{cont}\ (\sigma, E[\{\rho\}\ (...\ []\ e_i\ ...)], v) =$$
$$\langle \sigma, E[\{\rho\}\ (...\ v\ []\ ...)], \{\rho\}\ e_i\rangle$$
$$\text{step}_{cont}\ (\sigma, E[\{\rho\}\ (v\ v_1\ ...\ v_{N-1}\ [])], v_N) = \text{case } v \text{ of}$$
$$\quad |\ (\textbf{proc}\ f\ \{\rho'\}e) \Rightarrow \text{param}\ f\ e\ \sigma\ \rho'\ E\ [v_1, ..., v_N]$$
$$\quad\quad ...$$

The arity checking and variable binding for a **proc** value are handled by an auxiliary function param:

$$\text{param}\ (x \cdot f)\ e\ \sigma\ \rho\ E\ vs = \text{case } vs \text{ of}$$
$$\quad |\ [] \quad\Rightarrow\quad \langle \sigma, E, (\textbf{raise}\ \text{“Arity mismatch”})\rangle$$
$$\quad |\ v::vs' \quad\Rightarrow\quad \text{let } \ell = \text{fresh\_loc } \sigma$$
$$\quad\quad \text{in param}\ f\ e\ \sigma\{\ell \mapsto v\}\ \rho\{x \mapsto \ell\}\ E\ vs'$$

$$\text{param}\ ()\ e\ \sigma\ \rho\ E\ vs = \text{case } vs \text{ of}$$
$$\quad |\ [] \quad\Rightarrow\quad \langle \sigma, E, \{\rho\}\ e\rangle$$
$$\quad |\ \_ \quad\Rightarrow\quad \langle \sigma, E, (\textbf{raise}\ \text{“Arity mismatch”})\rangle$$

$$\text{param}\ x\ e\ \sigma\ \rho\ E\ vs = \text{let}$$
$$\quad \langle \sigma', \ell\rangle = \text{alloc\_list}\ \sigma\ vs;\ \ \rho' = \rho\{x \mapsto (\textbf{pp}\ \ell)\}$$
$$\text{in } \langle \sigma', E, \{\rho'\}\ e\rangle$$

The auxiliary alloc_list function allocates a linked list of fresh cons cells (pairs) containing the values vs and returns its location ℓ along with an updated store σ', with $\sigma \sqsubseteq \sigma'$.

When refunctionalising the application expression case, we also refunctionalise param to be mutually recursively defined with the evaluator. The logic of the semantic definition is preserved, and it is only its return values which are substituted:

$$\text{param}_{CPS}\ (x \cdot f)\ e\ \sigma\ \rho\ k\ vs = \text{case } vs \text{ of}$$
$$\quad |\ [] \quad\Rightarrow\quad (\textbf{raise}\ \text{“Arity mismatch”})$$
$$\quad |\ v::vs' \quad\Rightarrow\quad \text{let } \ell = \text{fresh\_loc } \sigma$$
$$\quad\quad \text{in param}_{CPS}\ f\ e\ \sigma\{\ell \mapsto v\}\ \rho\{x \mapsto \ell\}\ k\ vs'$$

$$\text{param}_{CPS}\ ()\ e\ \sigma\ \rho\ k\ vs = \text{case } vs \text{ of}$$
$$\quad |\ [] \quad\Rightarrow\quad \text{eval}_{CPS}\ e\ \sigma\ \rho\ k$$
$$\quad |\ \_ \quad\Rightarrow\quad (\textbf{raise}\ \text{“Arity mismatch”})$$

$$\text{param}_{CPS}\ x\ e\ \sigma\ \rho\ k\ vs = \text{let } \langle \sigma', \ell\rangle = \text{alloc\_list}\ \sigma\ vs$$
$$\quad \text{in eval}_{CPS}\ e\ \sigma'\ \rho\{x \mapsto (\textbf{pp}\ \ell)\}\ k$$

Brack does not yet support exception handling, so the definitional interpreter immediately terminates on a raised exception. Auxiliary functions which do not take an evaluation context such as alloc_list remain unchanged.

With these changes, the application expression case for the evaluator becomes:

$$\text{eval}_{CPS}\ (e\ e_1\ ...)\ \sigma\ \rho\ k =$$
$$\quad \text{eval}_{CPS}\ e\ \sigma\ \rho\ (\lambda\sigma'\ v.$$
$$\quad\quad \text{eval}_{CPS}\ e_1\ \sigma'\ \rho\ (\lambda\sigma''\ v_1.$$
$$\quad\quad\quad ...\ \text{case } v \text{ of}$$
$$\quad\quad\quad\quad |\ (\textbf{proc}\ f\ \{\rho'\}e) \Rightarrow \text{param}_{CPS}\ f\ e\ \sigma'^{N\prime}\ \rho'\ k\ [v_1, ...]$$
$$\quad\quad\quad\quad\quad ...))$$

4.3 HOL4 Term Definitions to ML AST

We may pull the store, environment, and continuation parameters of the evaluator out into lambda abstractions to arrive at a denotation for Scheme expressions of the form $\text{eval}_{CPS}\ e = \lambda\sigma\ \rho\ k. ....$ At this point we use the property that HOL4 term definitions embed into a pure subset of ML, and so transform the denotation of a Scheme expression into a generated ML AST (denoted in pink). In particular, continuations k and their internal logic are constructed as ML AST lambda abstractions $\lambda\sigma\ t. ...$, and continuation applications as applications $k\ \sigma\ t$. The store and environment are blue to reflect their direct ML representations as Scheme runtime constructs. For example, the CPS transform for assignment becomes:

$$\text{compile}_{CPS}\ (\textbf{set!}\ x\ e) = \lambda\sigma\ \rho\ k.$$
$$(\text{compile}_{CPS}\ e)\ \sigma\ \rho\ (\lambda\sigma'\ t.\ k\ (\sigma'\{\rho(x) \mapsto t\})\ \textbf{unspecified})$$

where recursive calls to compile_{CPS} compose to construct a full ML AST from an arbitrary Scheme program.

Some extra care is needed to ensure that compile_{CPS} is properly defined in this transition to the ML AST. For the CPS transform for a Scheme application expression, naively swapping HOL4 terms to ML AST leaves an inner Scheme expression which cannot be transformed in the case of applying a lambda expression:

$$\text{compile}_{CPS}\ (e\ e_1\ ...) = \lambda\sigma\ \rho\ k.\ (\text{compile}_{CPS}\ e)\ \sigma\ \rho\ (\lambda\sigma'\ t.$$
$$\quad ...\ \text{case } t \text{ of}$$
$$\quad\quad |\ \text{Proc}\ \rho'\ f\ e \Rightarrow (\text{cparam}_{CPS}\ f\ e)\ \sigma'^{N\prime}\ \rho'\ k\ [...]$$
$$\quad\quad\quad ...)$$

In this example, cparam_{CPS} cannot expand over the lambda expression body e because it exists as an ML value at runtime which was introduced verbatim by the **lambda** compile case. The solution is to define a special CPS transform for lambda expressions and move the cparam_{CPS} expansion there instead:

$$\text{compile}_{CPS}\ (\textbf{lambda}\ f\ e) = \lambda\sigma\ \rho\ k.$$
$$k\ \sigma\ (\text{Proc}\ \rho\ (\text{cparam}_{CPS}\ f\ e))$$

The resulting application CPS transform may then use the transformed lambda expression body:

$$\begin{array}{l}\texttt{case } t \texttt{ of}\\ \quad \mid \text{Proc } \rho' \ \mathit{proc} \Rightarrow \mathit{proc}\ \sigma'^{N'}\ \rho'\ k\ [...]\\ \quad \ldots\end{array}$$

To derive cparam_{CPS}, terms are converted to ML AST as with the evaluator, but importantly there is no equivalent derivation of an ML-generating alloc_list. The definition of alloc_list cannot be expanded at compile time, as it is recursively defined over an argument list whose length is arbitrary at runtime. Therefore, the definition of alloc_list itself must be introduced into the compiled ML program, and we assert that it is accessible through the variable *alloc_list* in the runtime environment of the program, which the code generated by cparam_{CPS} will call. Other auxiliary semantic functions such as those handling arithmetic are similarly introduced into the compiled ML, constituting a Scheme runtime.

4.4 Shared Lexical Structure and Store Semantics

The compiler we derived in Section 4.3 directly simulates the Scheme store and environment, just as it does with the continuations. It is possible, however, to simplify the compiler by inspecting the similarities between Scheme and ML, allowing the store and environment to be compiled directly rather than passed around. More precisely, since Scheme is lexically scoped in the same way as ML, we may substitute the Scheme environment ρ with ML variables. Also, since we are no longer confined to the pure subset of ML, the Scheme store σ may be replaced by usage of the ML `ref` syntax to take advantage of the equivalent mutable store semantics of Scheme and ML.

With these notions in mind, we eliminate the Scheme store σ and environment ρ from our constructed ML AST expressions, and make appropriate use of mutable ML `ref` variables, e.g. in our new definition for cparam_{CPS}:

$$\begin{array}{l}\text{cparam}'_{CPS}\ (x \cdot f)\ e = \lambda k\ ts.\ \texttt{case } ts \texttt{ of}\\ \quad \mid [] \quad \Rightarrow \quad \text{Ex "}\mathit{Arity\ mismatch}\text{"}\\ \quad \mid t::ts' \quad \Rightarrow \quad \texttt{let } x = \texttt{ref } t \texttt{ in } (\text{cparam}'_{CPS}\ f\ e)\ k\ ts'\end{array}$$

and compile_{CPS} simplifies:

$$\text{compile}'_{CPS}\ (e\ e_1\ ...) = \lambda k.\ (\text{compile}'_{CPS}\ e)\ (\lambda t.\ ...)$$

We colour the Scheme variable x blue even when it is used in the ML AST, to indicate that Scheme variable names are tagged to avoid clashing with ML variables such as k and t that are introduced by the transform. In practice, Brack prepends Scheme variables with "var" and asserts that no introduced variable names start with "var".

This resultant CPS transform is equivalent to Plotkin's original transform [32]; indeed, applying this refunctionalisation technique similarly to the call-by-value lambda calculus results in its CPS transform derived by Plotkin.

4.5 Redundant Redexes and Proof Simplification

The CPS transform we have developed so far, based on compile and its auxiliary transforms such as cparam, produces redexes which could be eliminated during compilation. For example, when a CPS expression such as $(\text{compile}'_{CPS}\ e)$ is applied to a continuation variable k, the result is a reducible application $(\lambda k.\ ...)\ k$. We can eliminate these redexes by factoring the continuation variable back into the CPS transform expanded at compile time, provided continuations are always captured in a variable which may be let-bound, leading to the final transform:

$$[\![(\textbf{set!}\ x\ e)]\!]_k = \texttt{let } k' = \lambda t.\ x := t;\ k \text{ Unspecified } \texttt{in } [\![e]\!]_{k'}$$

where $[\![\cdot]\!]_k$ is the CPS transform of a Scheme program with its current continuation bound to the ML variable k. This definition propagates the continuation variable statically rather than passing it dynamically, eliminating redexes in expressions where the continuation is not modified.

The auxiliary transform cparam_{CPS} and others like it similarly include the continuation variable as part of the transform, and also factor out the argument list variable:

$$\begin{array}{l}\mathscr{P}[\![(x \cdot f), e]\!]_{k,ts} = \texttt{case } ts \texttt{ of}\\ \quad \mid [] \quad \Rightarrow \quad \text{Ex "}\mathit{Arity\ mismatch}\text{"}\\ \quad \mid t::ts' \quad \Rightarrow \quad \texttt{let } x = \texttt{ref } t \texttt{ in } \mathscr{P}[\![f, e]\!]_{k,ts'}\end{array}$$

The arguments k and ts must now be bound by a lambda abstraction in the CPS transform for a Scheme lambda expression, which captures the environment at the point of evaluation consistent with Scheme's lexical scoping:

$$[\![(\textbf{lambda}\ f\ e)]\!]_k = k\ (\text{Proc } (\lambda k\ ts.\ \mathscr{P}[\![f, e]\!]_{k,ts}))$$

Factoring out the continuation variable also makes proving semantic verification easier, because it becomes possible to relate arbitrary Scheme CESK machine states to CPS expressions in ML where the current continuation is always bound to some variable k, which we show in Section 5.

A complete Scheme program p compiles to the CPS transform of p with the identity continuation bound to its continuation variable $\texttt{let } k = \lambda t.\ t \texttt{ in } [\![p]\!]_k$.

4.6 First-Class Continuations

Deriving the compiler from the refunctionalised small-step semantics captures the behaviour of first-class continuations with **call/cc** without much additional work. Here are

the semantic specifications of **call/cc** and of escape procedures **throw**, which appear as cases of the application continuation case match introduced in Section 4.2:

$$
\begin{array}{l}
\text{step}_{cont}\ (\sigma, E[\{\rho\}\,(v\ v_1\ \ldots\ v_{N-1}\ [])], v_N) = \text{case } v \text{ of} \\
\quad \mid (\textbf{proc}\ \rho'\ f\ e) \Rightarrow \text{param } f\ e\ \sigma\ \rho'\ E\ [v_1, \ldots, v_N] \\
\quad \mid \textbf{call/cc} \Rightarrow \text{case } [v_1, \ldots, v_N] \text{ of} \\
\qquad \mid [] \Rightarrow \langle \sigma, E, (\textbf{raise}\ \text{“Arity mismatch”})\rangle \\
\qquad \mid v :: vs' \Rightarrow \text{case } vs' \text{ of} \\
\qquad\quad \mid [] \Rightarrow \langle \sigma, E[\{\emptyset\}\,(v\ [])], (\textbf{throw}\ E)\rangle \\
\qquad\quad \mid \_ \Rightarrow \langle \sigma, E, (\textbf{raise}\ \text{“Arity mismatch”})\rangle \\
\quad \mid (\textbf{throw}\ E') \Rightarrow \text{case } [v_1, \ldots, v_N] \text{ of} \\
\qquad \mid [] \Rightarrow \langle \sigma, E, (\textbf{raise}\ \text{“Arity mismatch”})\rangle \\
\qquad \mid v :: vs' \Rightarrow \text{case } vs' \text{ of} \\
\qquad\quad \mid [] \Rightarrow \langle \sigma, E', v\rangle \\
\qquad\quad \mid \_ \Rightarrow \langle \sigma, E, (\textbf{raise}\ \text{“Arity mismatch”})\rangle \\
\quad \ldots
\end{array}
$$

This semantic definition is refunctionalised and embedded into ML, just like the rest of the semantics. The CESK machine states returned by these cases of step_{cont} correspond to values, so they transform into applications of continuations in ML: **throw** simply applies its captured continuation to its argument, and **call/cc** constructs a new partially evaluated application continuation with its argument as the new function to be applied. This nested application continuation requires that we factor out the logic for application into a recursive function, defined as follows:

```
letrec apply k = λt ts. case t of
  | Proc proc ⇒ proc k ts
  | CallCC ⇒ case ts of
      | []    ⇒ Ex "Arity mismatch"
      | t::ts' ⇒ case ts' of
          | []  ⇒ let k' = λt1. apply k t [t1]
                  in k' (Throw k)
          | _   ⇒ Ex "Arity mismatch"
  | Throw k' ⇒ case ts of
      | []    ⇒ Ex "Arity mismatch"
      | t::ts' ⇒ case ts' of
          | []  ⇒ k' t
          | _   ⇒ Ex "Arity mismatch"
    ...
```

The CPS transform for an application expression becomes:

$$
[\![(e\ e_1\ \ldots)]\!]_k = \texttt{let}\ k' = \lambda t.\ \texttt{let}\ k'' = \lambda t_1.\ \ldots\ \textit{apply}\ k\ t\ [t_1, \ldots]\ \ldots\ \texttt{in}\ [\![e_1]\!]_{k''}\ \texttt{in}\ [\![e]\!]_{k'}
$$

Because **call/cc**, like other Scheme features, is implemented into Brack using Ager et al.'s refunctionalisation technique and embedding into ML, its semantic preservation proof is not difficult.

Felleisen et al. previously have informally described a similar semantics-directed compiler derivation method for a calculus with control operators [16, 18]. We have generalised their approach, using Danvy and Ager et al.'s formal notion of refunctionalisation, and added an additional step of embedding the denotational form of the resulting interpreter into ML. This embedding step is possible because the defining language of the interpreter is a subset of the target language of our compiler.

5 Semantic Preservation as Simulation

Verifying the construction of Brack described in Section 4 involves establishing a semantic preservation property which states that evaluating the generated ML code simulates the language defined by its CESK machine. This simulation proof technique is derivative of Plotkin's work on the call-by-value lambda calculus CPS transform [32].

We first derive a definition for the *CPS relation*, which we denote $\rightsquigarrow$, between Scheme CESK machine states and ML expressions with corresponding store and environment configuration. The CPS relation mirrors the compiler derivation from the CESK machine to ML AST in Section 4, with machine states corresponding either to the CPS transform of an expression $[\![e]\!]_k$ or to the application $k\ e$ of a continuation $k\ e$ to some expression e that trivially evaluates to a value:

$$
\frac{\sigma \rightsquigarrow \sigma \qquad \rho \rightsquigarrow \rho \qquad E \rightsquigarrow \rho(k)}{\langle \sigma, E, \{\rho\}\,e\rangle \rightsquigarrow \langle \sigma, \rho, [\![e]\!]_k\rangle}
$$

$$
\frac{\begin{array}{ll}\sigma \rightsquigarrow \sigma & v \rightsquigarrow v \\ E \rightsquigarrow \rho(k) & \forall\, \sigma'\,.\, \text{eval}_{ML}\ (\sigma', \rho, e) = \langle \sigma', v\rangle\end{array}}{\langle \sigma, E, v\rangle \rightsquigarrow \langle \sigma, \rho, k\ e\rangle}
$$

The relation on stores dictates a direct correspondence between Scheme mutable variable store entries and ML mutable `ref`s, and the relation on environments matches the Scheme and ML lexical scoping. We also require that the ML environment ρ has bindings to Scheme runtime functions like *apply* and *alloc_list*, and incorporate this requirement into the environment relation.

The relations between values and between continuations are derived directly from the compiler derivation based on refunctionalisation and ML AST construction in Section 4. An evaluation context E is inductively related to an ML continuation closure $\rho(k)$ by rules for different stack frames, e.g. the assignment stack frame:

$$
\frac{E \rightsquigarrow \rho(k) \qquad \rho \rightsquigarrow \rho}{E[\{\rho\}\,(\textbf{set!}\ x\ [])] \rightsquigarrow \Lambda t.\ \{\rho\}\,(x\ :=\ t\,;\ k\ \text{Unspecified})}
$$

Using the CPS relation, we prove semantic preservation from Scheme to ML by showing that ML expressions related to Scheme states before and after a Scheme small step have

the same big-step ML evaluation (Figure 1). Formally, this property is stated as follows:

Theorem 5.1 (Step preservation).

$$\begin{aligned}
&\vdash \forall\, \sigma\ E\ c\ \sigma'\ E'\ c'\ \sigma\ \rho\ e\,.\\
&\quad \mathsf{step}\,(\sigma, E, c) = \langle \sigma', E', c' \rangle \wedge\\
&\quad \langle \sigma, E, c \rangle \rightsquigarrow \langle \sigma, \rho, e \rangle \wedge \sigma \vDash \langle E, c \rangle\\
&\quad \Longrightarrow\\
&\quad \exists\, \sigma'\ \rho'\ e'\,.\\
&\qquad \mathsf{eval}_{ML}(\sigma, \rho, e) = \mathsf{eval}_{ML}(\sigma', \rho', e') \wedge\\
&\qquad \langle \sigma', E', c' \rangle \rightsquigarrow \langle \sigma', \rho', e' \rangle
\end{aligned}$$

Assuming equivalence of environment and store semantics, the simulation evaluates equivalently to eval_{CPS}. By the correctness of refunctionalisation, the simulation is then valid with respect to the CESK machine specification.

The functional big-step definition of the ML evaluator [30] allows us to use equality between the evaluations of ML programs without having to consider their final values, or even whether they terminate, because we may symbolically evaluate eval_{ML} in the proof to partially reduce the program.

We define a validity relation $\vDash$ on Scheme state representing the condition that (1) all expressions in the control string, evaluation context, and closure bodies are statically scoped, and (2) all store locations in all environments and pair pointers are valid, i.e. correspond to real entries in the store. This condition is required to ensure that the reduction of Scheme programs, including their simulation in ML, cannot get stuck.

The key verification component for Brack is a proof of Theorem 5.1. Using a lemma that states that Scheme states in a CESK machine maintain validity, we apply Theorem 5.1 for multiple steps to arrive at a statement of equivalent evaluation:

Corollary 5.2 (Equivalent evaluation).

$$\begin{aligned}
&\vdash \forall\, p\ \sigma\ v\ \sigma\ \rho\ p\,.\\
&\quad \left(\exists\, N\,.\,\mathsf{step}^N\,(\emptyset, [], \{\emptyset\}\,p) = \langle \sigma, [], v \rangle\right)\ \wedge\\
&\quad \langle \emptyset, [], \{\emptyset\}\,p \rangle \rightsquigarrow \langle \sigma, \rho, p \rangle\ \wedge \emptyset \vDash \langle [], \{\emptyset\}\,p \rangle\\
&\quad \Longrightarrow\\
&\quad \exists\, \sigma'\ v\,.\,\mathsf{eval}_{ML}(\sigma, \rho, p) = \langle \sigma', v \rangle \wedge \sigma \rightsquigarrow \sigma' \wedge v \rightsquigarrow v
\end{aligned}$$

Theorem 5.1 also allows us to prove that diverging Scheme programs will also diverge as compiled ML programs, by taking advantage of the construction of the functional big-step semantics used by CakeML, which tracks ML reductions using a clock [30]. We establish a lexicographic ordering on the clock value and the size of the Scheme program, which strictly decreases with each Scheme step, hence we can prove by a well-founded measure that diverging Scheme programs always exhaust the ML evaluator clock, for all clock values. We omit the clock in our notation for clarity.

The final property established for Brack states that the compiler-generated ML code $\texttt{let}\ k\ =\ \lambda t.t\ \texttt{in}\ [\![p]\!]_k$ for a valid Scheme program p evaluates to an ML expression corresponding by the CPS relation to the Scheme state $\langle \emptyset, [], \{\emptyset\}\,p \rangle$. Taken together, this theorem, the preservation of divergence, and Corollary 5.2 establish semantic preservation: a valid Scheme program shares the same observable trace of execution as the ML program that it compiles to, for both terminating and diverging executions.

We have composed the Brack semantic preservation proof with the CakeML backend correctness proof, establishing an end-to-end semantic preservation proof from Scheme to machine code, making Brack a complete verified compiler.

5.1 Proof

Our most significant, and perhaps surprising result is that, due to the semantics-derived CPS transform, proof of semantic preservation of Scheme first-class continuations is trivial. The proof for the relevant case of Theorem 5.1 amounts to a proof of correctness of refunctionalisation and embedding into ML, where we must simply reconcile the logic from the semantic definition of **call/cc** with its equivalent logic in ML:

Case (call/cc).

$$\begin{aligned}
&\vdash E \rightsquigarrow \rho(k) \wedge v_1 \rightsquigarrow v_1 \wedge \ldots \wedge\\
&\quad \rho(ts) = [v_1, \ldots] \wedge \sigma \rightsquigarrow \sigma \wedge \rho \rightsquigarrow \rho \wedge\\
&\quad e = \texttt{case}\ ts\ \texttt{of}\\
&\qquad\qquad \mid [] \quad \Rightarrow \quad \mathrm{Ex}\ \text{"}\textit{Arity mismatch}\text{"}\\
&\qquad\qquad \mid t :: ts' \quad \Rightarrow \quad \texttt{case}\ ts'\ \texttt{of}\\
&\qquad\qquad\qquad \mid [] \quad \Rightarrow \quad \texttt{let}\ k' = \lambda t_1.\ \mathit{apply}\ k\ t\ [t_1]\\
&\qquad\qquad\qquad\qquad\qquad\ \texttt{in}\ k'\ (\mathrm{Throw}\ k)\\
&\qquad\qquad\qquad \mid _ \quad \Rightarrow \quad \mathrm{Ex}\ \text{"}\textit{Arity mismatch}\text{"}\\
&\quad \Longrightarrow\\
&\quad \exists\, e'\,.\,\mathsf{eval}_{ML}\,(\sigma, \rho, e) = \mathsf{eval}_{ML}\,(\sigma, \rho, e') \wedge\\
&\qquad \left(\begin{array}{l}
\texttt{case}\ [v_1, \ldots]\ \texttt{of}\\
\quad \mid [] \quad \Rightarrow \quad \langle \sigma, E, (\textbf{raise}\ \textit{"Arity mismatch"}) \rangle\\
\quad \mid v :: vs' \quad \Rightarrow \quad \texttt{case}\ vs'\ \texttt{of}\\
\qquad \mid [] \quad \Rightarrow \quad \langle \sigma, E[\{\emptyset\}\,(v\ [])], (\textbf{throw}\ E) \rangle\\
\qquad \mid _ \quad \Rightarrow \quad \langle \sigma, E, (\textbf{raise}\ \textit{"Arity mismatch"}) \rangle
\end{array}\right)\\
&\qquad \rightsquigarrow \langle \sigma, \rho, e' \rangle
\end{aligned}$$

The arity checks evaluate equivalently, either raising terminating exceptions or constructing the application continuation.

The interesting parts of the proof lie in the points of divergence from the pure compiler derivation method. First,

we must prove that the Scheme and ML stores and environments remain synchronised:

Lemma 5.3 (Store synchronisation)**.**

$$\vdash \sigma \rightsquigarrow \sigma \wedge v \rightsquigarrow \rho(t) \wedge \text{fresh\_loc}\ \sigma = \ell \implies \exists \sigma' .\, \text{eval}_{ML}\ (\sigma, \rho, \texttt{ref}\ t) = \langle \sigma', \ell \rangle \wedge \sigma\{\ell \mapsto v\} \rightsquigarrow \sigma'$$

The synchronised store simply depends on the ML `ref` construct allocating to the store with the fresh_loc definition used by the Scheme semantics. With this lemma, the combined store and environment relation follows:

Lemma 5.4 (Store and environment synchronisation)**.**

$$\vdash \sigma \rightsquigarrow \sigma \wedge \rho \rightsquigarrow \rho \wedge v \rightsquigarrow \rho(t) \wedge \text{fresh\_loc}\ \sigma = \ell \implies \exists \sigma'\ \rho' .\, \text{eval}_{ML}\ (\sigma, \rho, \texttt{let}\ x = \texttt{ref}\ t\ \texttt{in}\ e) = \text{eval}_{ML}\ (\sigma', \rho', e) \wedge \sigma\{\ell \mapsto v\} \rightsquigarrow \sigma' \wedge \rho\{x \mapsto \ell\} \rightsquigarrow \rho'$$

It is also trivial to prove that continuation and value bindings to the ML store do not affect the environment relation, because the compiler tags Scheme variables to prevent name collisions, denoted again in blue:

Lemma 5.5 (Environment monotonicity with respect to Scheme variables)**.**

$$\vdash k \notin \text{Variable} \wedge \rho \rightsquigarrow \rho \implies \rho \rightsquigarrow \rho\{k \mapsto \mathit{cont}\} \wedge t \notin \text{Variable} \wedge \rho \rightsquigarrow \rho \implies \rho \rightsquigarrow \rho\{t \mapsto \mathit{val}\}$$

Proof of the Theorem 5.1 case for correct procedure application compilation then amounts to reconciling the param function in HOL4 with the evaluation of its expanded $\mathscr{P}[\![\cdot,\cdot]\!]_{k,ts}$ transform in ML:

Case (Procedure application)**.**

$$\vdash E \rightsquigarrow \rho(k) \wedge v_1 \rightsquigarrow v_1 \wedge \ldots \wedge \rho(ts) = [v_1, \ldots] \wedge \sigma \rightsquigarrow \sigma \wedge \rho \rightsquigarrow \rho \implies \exists \sigma'\ \rho'\ e' .\, \text{eval}_{ML}\ (\sigma, \rho, \mathscr{P}[\![f, e]\!]_{k,ts}) = \text{eval}_{ML}\ (\sigma', \rho', e') \wedge \text{param}\ f\ e\ \sigma\ \rho\ E\ [v_1, \ldots] \rightsquigarrow \langle \sigma', \rho', e' \rangle$$

Proof. By induction over f. □

Another interesting point of the proof is where runtime functions must be introduced into the ML program for semantic functions which are recursively defined over a runtime argument list, such as alloc_list. Such functions are direct embeddings of their HOL4 term definitions, so semantic preservation proof follows, as with the other cases, of the functions evaluating equivalently:

Lemma 5.6 (alloc_list semantic preservation)**.**

$$\vdash v_1 \rightsquigarrow v_1 \wedge \ldots \wedge \rho(ts) = [v_1, \ldots] \wedge \sigma \rightsquigarrow \sigma \wedge \text{alloc\_list}\ \sigma\ [v_1, \ldots] = \langle \sigma', \ell \rangle \implies \exists \sigma' .\, \text{eval}_{ML}\ (\sigma, \rho, \mathit{alloc\_list}\ ts) = \langle \sigma', \text{PairP}\ \ell \rangle \wedge \sigma' \rightsquigarrow \sigma'$$

Proof. By induction over the argument list. The HOL4 function alloc_list is assumed to use fresh_loc, hence using the same store allocation strategy as ML. Like the param proof, this proof depends on Lemma 5.3. □

5.2 Difficulty of Dynamic Typing

Dealing with Scheme's dynamic typing in our development was not technically difficult, but it involved a significant amount of work. A value passed to a continuation may be of any type, and so each continuation in the proof involves a great many cases. This large number of cases increases the amount of rewriting that HOL4 performs, slowing down development.

5.3 Room for Optimisation

The CPS transform used in Brack follows Plotkin's definition, and so it includes many *administrative redexes*, i.e. abstractions that are artifacts of the translation. One example may be seen in the transform for **call/cc**, where the application continuation is introduced in a let-binding before immediately being applied to a value:

$$\texttt{let}\ k' = \lambda t_1 .\, \mathit{apply}\ k\ t\ [t_1]\ \texttt{in}\ k'\ (\text{Throw}\ k)$$

It would be more efficient to directly use the Throw value in the body of the continuation without allocating a closure, but that optimisation would result in CPS expression reductions which no longer fit the CPS relation we have constructed for the proof. Other similar changes, such as directly using literal values in continuation bodies, add complexity to the transform and also do not fit our current CPS relation.

Brack's CPS transform uses the semantics-directed derivation described in this paper, with the goal of making verification of its control features as simple as possible. Using a Plotkin-style transform also avoids α-equivalence issues which other CPS correctness mechanisations have encountered when implementing Danvy & Nielsen's optimised transform [28, 31]. However, we believe that it may be possible to adjust the semantics-directed compiler derivation in a way that enables the verification of an optimised transform whilst avoiding α-equivalence [26], and we hope to implement the optimised version in Brack in the future.

6 Example Programs

The subset of Scheme supported by Brack is sufficient to compile interesting data-processing programs. The source code for the following programs is available in the supplementary material for this paper [25].

Fibonacci Numbers. Brack can compile arithmetic algorithms over integers, including calculating the Fibonacci numbers. We have implemented several variants of the Fibonacci number calculation, including the classic (but inefficient) recursive implementation and a tail-recursive implementation with an accumulator, both of which we take from the Larceny Scheme benchmark suite.

We also include an imperative-style Fibonacci implementation which calculates the Fibonacci numbers using mutable variable accumulators, and which has no recursive calls, instead using **call/cc** to jump back to the start of the algorithm. As shown in the following, the jump case makes a call to the escape continuation held in *j*, which jumps to the **letrec*** initialisation, and reinstates the same escape procedure *j* to be used again:

```
(lambda (n)
  (letrec* ((a 0) (b 1) (c 0) (j (call/cc (lambda (cc) cc))))
    (if (eqv? n 0) a
        (begin
          (set! c a) (set! a (+ a b)) (set! b c)
          (set! n (- n 1))
          (j j)))))
```

List Operations. In Scheme, lists are built from pairs (cons cells). Brack can support higher-order list operations such as `map` and `fold`, which are implemented using Scheme's basic pair operations. In turn, these operations can be used to implement more complex list operations such as the list concatenation function `cat`.

Non-determinism. One application of first-class continuations is non-determinism. Non-deterministic programs can be emulated by deterministic programs using backtracking [19, Chap. 22]. Brack can compile an implementation of the `choose` and `fail` operators to enable non-deterministic patterns, where possible solutions to some algorithm may be passed to `choose` and a resultant correct solution arrived at if the algorithm does not `fail`, for example choosing matching numbers from lists:

```
(let ((x (choose '(2 4 6 8)))
      (y (choose '(3 6 9 12))))
  (if (eqv? x y) x (fail)))
```

For clarity, we write the code using **let**, but our Brack implementation uses **letrec**.

7 Related Work

Compilation via CakeML. The approach of targeting CakeML has been used previously in the PureCake compiler for a lazy Haskell-like language PureLang [22], which supports lazy evaluation via translation into `delay` and `force` primitives. We see targeting CakeML as a promising approach to verified compilation for other high-level languages, lowering development cost by avoiding the duplicated effort involved in verifying language-independent portions of the back-ends of compilers.

Mechanised Proofs of CPS correctness. An early proof of the correctness of the CPS transform is given by Plotkin [32], who introduced the transform to simulate the call-by-value lambda calculus with the call-by-name lambda calculus. Plotkin's simulation proof and its variants have been mechanised many times, e.g. by Minamide & Okuma [28], Tian [36], Dargaye & Leroy [10], and Paraskevopoulou & Grover [31]. Brack proves simulation similarly to these works, and the CPS transform Brack employs, which is derived from Plotkin's, makes it possible to use strict equality of reduced CPS terms over eval_{ML} in the semantic preservation proof. The earlier mechanisations also prove correctness of Danvy & Nielsen's optimised CPS transform which generates fewer administrative redexes [9]; simulation using this particular transform results in reductions of CPS terms which are not necessarily equal over eval_{ML}, and so require an additional notion of equivalence. However, an alternative formulation of Danvy & Nielsen's optimised transform using evaluation contexts [26] could be used to prove simulation in Brack without needing an equivalence lemma.

Verified Compilation for Scheme. Verified compilation for Scheme and similar languages has been previously developed for small, demonstrative systems. Guttman et al. handwrote a verified compiler from VLISP Scheme to a virtual machine byte code, proving semantic preservation of the denotational semantics between both languages [20]. Chlipala built a verified compiler in Coq for a small impure functional programming language to an idealised assembly language, using a CPS transform as part of the compilation [2]. Dold et al. [11, 12] describe a particularly impressive work, a verified bootstrap compiler from ComLisp, a small subset of Common Lisp, to the Transputer architecture. None of these compilers support first-class continuations, and we believe Brack is the first verified compiler for Scheme to do so.

8 Conclusion

We have described Brack, a new verified compiler for a subset of Scheme featuring **call/cc**, and the first fully verified compiler to support dynamic typing and first-class continuations. Brack is written in 4,000 non-whitespace

lines of HOL4 proof scripts. Our semantics-directed compiler derivation technique produces a CPS transform from the small-step operational semantics of an abstract machine, and facilitates the proof of semantic preservation for continuation-based languages compiled to ML. Applying this construction technique to Scheme produces the CPS transform used in Bracк.

As a consequence of our use of the CPS transform, first-class continuations were the simplest aspect of Bracк both to implement *and* to verify, despite the emergent complexity of program behaviours that first-class continuations introduce. Surprisingly, dynamic typing (including arity checking), an apparently simpler language feature, was comparatively harder to implement and verify. Dynamically-typed Scheme values are represented at runtime using an ML sum-type which must be pattern matched at every point where a value is dynamically type-checked, and dealing with these frequent case splits makes the HOL4 proofs cumbersome.

In common with ML, Scheme also features static scoping and mutability. The fact that both languages share these features allows Bracк to directly compile mutable Scheme variables to ML `ref` variables, rather than passing around a store and environment along with the continuation at runtime. Sharing the store and environment semantics between Scheme and ML simplifies implementation of the compiler, but this simplification is traded for complexity in the semantic preservation proof, which must reconcile the Scheme and ML stores and environments. This complexity made the preservation of lambda expression and recursive let-binding semantics the hardest part of the semantic preservation proof.

The simplicity of the **call/cc** implementation in Bracк is a testament to how naturally the continuation-passing style captures the behaviour of continuation-based control operators, and demonstrates that the semantics-directed compiler derivation technique that we used for Bracк is a viable approach to the verification of compilation of language features based on continuations.

Future Work. Bracк is currently based on a Plotkin-style CPS transform, which introduces many administrative redexes. There exist mechanised proofs [10, 26, 28, 36] for optimised CPS transforms [6, 8, 9] which eliminate these administrative redexes, and we believe it is possible to implement such an optimised CPS transform in a verified compiler like Bracк. Using these techniques, the CPS transform might serve as a basis for efficiently supporting other control operators and language features such as effect handlers in verified compilers.

Data-Availability Statement

The artifact of the supplementary material for this paper can be found at [25], and up-to-date source code is available online at https://github.com/CakeML/cakeml/tree/master/compiler/scheme.

Acknowledgements

We thank the anonymous reviewers for helpful comments. The third author was partially funded by Swedish Research Council grant 2021-05165.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (Uppsala, Sweden) *(PPDP '03)*. Association for Computing Machinery, New York, NY, USA, 8–19. doi:10.1145/888251.888254

[2] Adam Chlipala. 2010. A verified compiler for an impure functional language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 93–106. doi:10.1145/1706299.1706312

[3] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. 1999. Implementation Strategies for First-Class Continuations. *Higher Order Symbol. Comput.* 12, 1 (April 1999), 7–45. doi:10.1023/A:1010016816429

[4] Olivier Danvy. 2004. On Evaluation Contexts, Continuations, and the Rest of Computation. (02 2004).

[5] Olivier Danvy. 2008. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 131–142. doi:10.1145/1411204.1411206

[6] Oliver Danvy and Andrzex Filinski. 1992. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. doi:10.1017/S0960129500001535

[7] Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Florence, Italy) *(PPDP '01)*. Association for Computing Machinery, New York, NY, USA, 162–174. doi:10.1145/773184.773202

[8] Olivier Danvy and Lasse R. Nielsen. 2001. A Higher-Order Colon Translation. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS '01)*. Springer-Verlag, Berlin, Heidelberg, 78–91.

[9] Olivier Danvy and Lasse R. Nielsen. 2003. A first-order one-pass CPS transformation. *Theoretical Computer Science* 308, 1 (2003), 239–257. doi:10.1016/S0304-3975(02)00733-8

[10] Zaynah Dargaye and Xavier Leroy. 2007. Mechanized verification of CPS transformations. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (Yerevan, Armenia) *(LPAR'07)*. Springer-Verlag, Berlin, Heidelberg, 211–225.

[11] Axel Dold, Friedrich Wilhelm von Henke, Vincent Vialard, and Wolfgang Goerigk. 2005. A mechanically verified compiling specification for a realistic compiler. (2005).

[12] Axel Dold and Vincent Vialard. 2001. A Mechanically Verified Compiling Specification for a Lisp Compiler. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, Ramesh Hariharan, V. Vinay, and Madhavan Mukund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 144–155.

[13] Kavon Farvardin and John Reppy. 2020. From folklore to fact: comparing implementations of stacks and continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing

Machinery, New York, NY, USA, 75–90. doi:10.1145/3385412.3385994

[14] Mattias Felleisen and D. P. Friedman. 1987. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 314. doi:10.1145/41625.41654

[15] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts*. https://api.semanticscholar.org/CorpusID:57760323

[16] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. 1987. A syntactic theory of sequential control. *Theoretical Computer Science* 52, 3 (1987), 205–237. doi:10.1016/0304-3975(87)90109-5

[17] Matthias Felleisen, Daniel P Friedman, Eugene E Kohlbecker, and Bruce F Duba. 1986. Reasoning with continuations. In *LICS*, Vol. 86. 131–141.

[18] Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992), 235–271. doi:10.1016/0304-3975(92)90014-7

[19] P. Graham. 1994. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall.

[20] Joshua Guttman, John Ramsdell, and Vipin Swarup. 1995. The VLISP verified Scheme system. *Lisp and Symbolic Computation* 8 (03 1995), 33–110. doi:10.1007/BF01128407

[21] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. 1990. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 66–77. doi:10.1145/93542.93554

[22] Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. PureCake: A Verified Compiler for a Lazy Functional Language. *Proc. ACM Program. Lang.* 7, PLDI, Article 145 (June 2023), 25 pages. doi:10.1145/3591259

[23] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 362–369.

[24] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. doi:10.1145/2535838.2535841

[25] Pascal Y. Lasnier, Jeremy Yallop, and Magnus O. Myreen. 2025. *CPP '26 Artifact – Brack: A Verified Compiler for Scheme via CakeML*. doi:10.1145/3747413

[26] Pascal Y. Lasnier, Jeremy Yallop, and Magnus O. Myreen. 2026. A One-Pass CPS Transform with Simulation on the Nose. In *Proceedings of the 28th International Symposium on Practical Aspects of Declarative Languages* (Rennes, France) *(PADL '26)*. Springer Nature Switzerland, Cham.

[27] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

[28] Yasuhiko Minamide and Koji Okuma. 2003. Verifying CPS transformations in Isabelle/HOL. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding* (Uppsala, Sweden) *(MERLIN '03)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/976571.976576

[29] Magnus O. Myreen and Scott Owens. 2014. Proof-producing Translation of Higher-order logic into Pure and Stateful ML. *Journal of Functional Programming (JFP)* 24, 2-3 (May 2014), 284–315. doi:10.1017/S0956796813000282

[30] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 589–615. doi:10.1007/978-3-662-49498-1_23

[31] Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with continuations, correctly. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 114 (Oct. 2021), 29 pages. doi:10.1145/3485491

[32] G.D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159. doi:10.1016/0304-3975(75)90017-1

[33] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. doi:10.1145/800194.805852

[34] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics* (Montreal, P.Q., Canada) *(TPHOLs '08)*. Springer-Verlag, Berlin, Heidelberg, 28–32. doi:10.1007/978-3-540-71067-7_6

[35] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19, S1 (2009), 1–301. doi:10.1017/S0956796809990074

[36] Ye Henry Tian. 2006. Mechanically verifying correctness of CPS compilation. In *Proceedings of the Twelfth Computing: The Australasian Theory Symposium - Volume 51* (Hobart, Australia) *(CATS '06)*. Australian Computer Society, Inc., AUS, 41–51.

[37] Yong Xiao, Amr Sabry, and Zena M. Ariola. 2001. From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition. *Higher Order Symbol. Comput.* 14, 4 (Dec. 2001), 387–409. doi:10.1023/A:1014408032446

[38] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3655–3672. https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. doi:10.1145/1993498.1993532

Received 2025-09-13; accepted 2025-11-13