PureCake: A Verified Compiler for a Lazy Functional Language

HRUTVIK KANABAR, University of Kent, United Kingdom
SAMUEL VIVIEN, École Normale Supérieure PSL, France and Chalmers University of Technology, Sweden
OSKAR ABRAHAMSSON, Chalmers University of Technology, Sweden
MAGNUS O. MYREEN, Chalmers University of Technology, Sweden
MICHAEL NORRISH, Australian National University, Australia
JOHANNES ÅMAN POHJOLA, University of New South Wales, Australia
RICCARDO ZANETTI, Chalmers University of Technology, Sweden

We present PureCake, a mechanically-verified compiler for PureLang, a lazy, purely functional programming language with monadic effects. PureLang syntax is Haskell-like and indentation-sensitive, and its constraint-based Hindley-Milner type system guarantees safe execution. We derive sound equational reasoning principles over its operational semantics, dramatically simplifying some proofs. We prove end-to-end correctness for the compilation of PureLang down to machine code—the first such result for any lazy language—by targeting CakeML and composing with its verified compiler. Multiple optimisation passes are necessary to handle realistic lazy idioms effectively. We develop PureCake entirely within the HOL4 interactive theorem prover.

CCS Concepts: • Software and its engineering → Software verification; Compilers; • Theory of computation → Higher order logic.

Additional Key Words and Phrases: compiler verification, Haskell, interactive theorem proving, HOL4

1 INTRODUCTION

High-level languages often claim to provide strong guarantees for the programmer. For example: Haskell’s strong typing demarcates stateful and pure computations, and its lazy evaluation reduces unnecessary computation; Rust’s borrow checker prevents use-after-free bugs and thread-unsafe behaviour. Compilers for these languages reject programs which are semantically ill-defined, removing the burden of safety from the programmer. By contrast, it is non-trivial to avoid undefined behaviour in more low-level languages such as C++. However, high-level languages require longer compilation paths to generate efficient code, providing greater scope for miscompilations which could compromise intended guarantees.

End-to-end verification prevents such miscompilations: any bugs in a compiled binary must come from the input source code. CompCert [Leroy 2009] first showed that this was feasible for featureful, optimising compilers by verifiably compiling a subset of C99. Inspired by CompCert, the verified CakeML [Kumar et al. 2014; Myreen 2021a] compiler compiles a Standard ML-like language. Note that CakeML is a functional programming language in which memory safety is guaranteed by type safety at the source and verified garbage collection.

Compilation of Haskell-like languages presents unique challenges. These languages implement lazy evaluation: expressions are computed only when needed, and never recomputed on reuse. This necessitates purity: computations are free of side-effects (stateful operations and I/O) by default, leading to referential transparency. That is, equal expressions give rise to equal values in all contexts, corresponding to the programmer model of equational reasoning. However, programmers can still...
access stateful features and interact with the surrounding execution environment using monads. Therefore, compiling these languages with some semblance of realism requires at least:

- efficient call-by-need evaluation without compromising equational reasoning;
- demand analysis to compute values eagerly when they are unconditionally required; and
- monadic reflection to generate idiomatic imperative code for stateful monadic operations.

It is extremely difficult to achieve these desiderata correctly, efficiently, and simultaneously: the Glasgow Haskell Compiler (GHC) has over 5000 open issues at the time of writing.¹

We create PureCake: to the best of our knowledge, the most realistic certified compiler for a lazy functional language to date, and the first which verifiably implements the features above.

Contributions

In this paper, we make the following contributions.

- We implement the first end-to-end verified compiler for a non-strict, purely functional programming language, with:
  - sound equational reasoning;
  - Haskell-like indentation-sensitive parsing;
  - *verified two-phase constraint-based Hindley-Milner type inference;
  - *verified demand analysis;
  - verified optimisations for the compilation of non-strict code; and
  - *verified monadic reflection.

- The items marked (*) above have not been mechanically verified before, even in isolation.
- We demonstrate that CakeML is a convenient back end for verified, optimising compilation of high-level languages.

All of the work described in this paper has been developed within the HOL4 theorem prover.

2 OVERVIEW

In this section, we give a high-level tour of the PureCake development.


It also borrows several features from Haskell: indentation-sensitive parsing (§ 4.1), which accepts a simple Haskell-like syntax; mutually recursive top-level definitions; the seq operator; and a built-in IO monad, which provides stateful arrays, exception-handling, and a foreign function interface (FFI).

PureLang’s features target a significant subset of CakeML’s mature language. Formally, PureLang is specified using two ASTs (§ 3.2): high-level compiler expressions are used in compilation, and are considered syntactic sugar for simpler semantic expressions, which provide ground truth for semantics. PureLang’s compiler expressions are inspired by Core in the Glasgow Haskell Compiler (GHC) [Peyton Jones and Launchbury 1991]. Semantics of PureLang is cleanly defined in stages (§ 3.3): translation from compiler expressions to semantic expressions; pure, call-by-name evaluation to weak-head normal form; and stateful interpretation of monadic operations. We use a variant of interaction trees [Xia et al. 2020] as our semantic domain to model PureLang’s observable behaviours: termination, runtime type errors, and FFI calls.

We use applicative bisimulation to mechanise an equational theory over PureLang’s semantic expressions (§ 3.4), proving it congruent via Howe’s method [Howe 1996]. Standard contextual equivalence (equality of observable events under all contexts) coincides with our equational theory.

¹A recently-fixed bug sent type-checking into an infinite loop.
Hindley-Milner typing rules for PureLang’s compiler expressions (§ 3.5) do not satisfy subject reduction, preventing a standard proof of type soundness. We establish soundness using an alternative syntactic sugar which does satisfy subject reduction, using our equational theory to prove unconditional equivalence with compiler expressions.

Compiler front end (§ 4). Our indentation-sensitive parsing expression grammar (PEG) is inspired by Adams [2013] and the CakeML parser, and parses concrete syntax that is largely a subset of Haskell’s. Before compilation, top-level functions are sorted to minimise mutual recursion (§ 4.2).

We implement and verify non-standard, two-phase type inference (§ 4.3) inspired by the Helium teaching compiler [Heeren et al. 2003]: typing constraints are separately generated then solved. Our approach is open to the addition of language features and high-quality error messages.

Demand analysis (§ 4.4) uses \texttt{seq} to precompute expressions whose values are unconditionally required. Binding group and demand analyses are verified entirely within our equational theory.

Compiler back end (§ 5). We define three intermediate languages: \texttt{ThunkLang}, \texttt{EnvLang}, and \texttt{StateLang} (§§ 5.2 to 5.4 respectively). Like PureLang, each has two ASTs: compiler and semantic expressions. Compilation to the call-by-value \texttt{ThunkLang} introduces thunks, and optimisation passes minimise their usage in key locations. In \texttt{EnvLang}, semantic definitions begin using environments rather than substitutions. \texttt{StateLang} implements both thunks and monadic operations as stateful primitives; the latter is monadic reflection [Filinski 1994, 2010]. Therefore, its semantics does not require separate stages for pure evaluation and stateful interpretation.

Targeting CakeML (§ 6). Compilation targets CakeML, but CakeML’s functional big-step semantics [Owens et al. 2016] uses an oracle to model FFI behaviour where PureLang’s interaction tree semantics models all possible FFI behaviours. We verify an interaction tree semantics for CakeML, deriving the corresponding novel compiler correctness theorem. Verifiably bootstrapping [Myreen 2021b] PureCake transports its end-to-end correctness theorem down to its compiler binary.

3 \textsc{PureLang}

3.1 Features

Figure 1 showcases a small PureLang program accepting an integer \( n \) on the command line and printing the first \( n \) numbers of the factorial sequence. PureLang syntax is indentation-sensitive and inspired by Haskell; GHC accepts this program with minimal tweaks. Features common to

\begin{figure}
\begin{verbatim}
1 numbers :: [Integer]
2 numbers =
3   let num n = n : num (n + 1)
4     in num 0
5
6 factA :: Integer -> Integer -> Integer
7 factA a n =
8   if n < 2 then a
9     else factA (a * n) (n - 1)
10
11 factorials :: [Integer]
12 factorials = map (factA 1) numbers
13

app :: (a -> IO b) -> [a] -> IO ()
app f l = case l of
    [] -> return ()
    h:t -> do h ; app f t

main :: IO ()
main = do
    arg1 <- read_arg1
    -- fromString == 0 on malformed input
    let i = fromString arg1
    facts = take i factorials
    app (\i -> print $ toString i) facts
\end{verbatim}
\caption{A small PureLang program printing a user-specified prefix of the factorial sequence. We have removed boilerplate definitions for brevity.}
\end{figure}
Fig. 2. PureLang operations op, compiler expressions ce, and semantic expressions e.

functional languages are supported: first-class functions (map on line 12), general recursion (factA on lines 6-9), algebraic data types and pattern matching ([1:h:t on lines 16-17).

PureLang borrows several other features from Haskell. Top-level declarations are mutually recursive and can be reordered freely. Evaluation is call-by-need: expressions are computed only as deeply as they are inspected and never recomputed, so infinite data structures can be constructed (e.g., numbers on lines 1-4). Eager evaluation can be forced using Haskell’s seq operator.

The built-in monad permits effectful computation (main on lines 19-25), inspired by its namesake. Aside from the standard return/bind (which are masked by do-notation), it provides the ability to create/update/query mutable arrays, raise/handle exceptions, and perform FFI calls (e.g., for I/O).

3.2 Formal syntax

Figure 2 defines PureLang operations op, compiler expressions ce, which are inspired by GHC’s Core, and semantic expressions e. Compiler expressions consist of variables, operations, multi-arity λ-abstractions/-applications, let-statements, recursive bindings, sequencing, and pattern matches. Operations include data construction, primitives over built-in types, and the monadic operations: return/bind; exception-handling (raise/handle); mutable array operations (alloc/len/deref/update); and FFI interaction (action). Note that operations tuple and monadic mop are formalised as cons cname using reserved cnames, but we present them separately for clarity. Pattern matching is shallow, considering only constructor name and arity.

Semantics of compiler expressions is defined via desugaring (exp_of) to semantic expressions e, which removes multi-arity λ-abstraction/-application by nesting single-arity ones, and removes case by introducing if-, eq-., and proj-statements (eq. (1), below). To desugar case \( x = ce \) of \( \text{row}_n \) we assign exp_of ce to x and test it against each row of the pattern match \( \text{row} = \text{cname}[\ y_n \] \rightarrow ce' \) with if/eq. If a row matches both constructor name (cname) and arity (n), proj statements extract constructor arguments, which are assigned to the corresponding pattern variables \( y_n \) before desugaring the continuation. Failed pattern matches produce runtime type errors (fail); the last row in a case can optionally be a catch-all to avoid these.

\[
\begin{align*}
\text{exp}_\text{of} (\text{case } x = \text{ce of } \text{row}_n) & \overset{\text{def}}{=} \text{let } x = \text{exp}_\text{of} \text{ce in } \text{expand}_x [\ \text{row}_n] \\
\text{expand}_x [\ \text{cname}[\ y_n \] \rightarrow ce', \ \text{row}_m] & \overset{\text{def}}{=} \text{if } (\text{eq: } \text{cname } n \ (\text{var } x)) \text{ then } \\
& \text{let } y_n = \text{proj}_n \text{cname } (\text{var } x) \text{ in } (\text{exp}_\text{of} ce') \quad (1) \\
\text{expand}_x [\ ] & \overset{\text{def}}{=} \text{fail} \\
\text{else } & \text{expand}_x [\ \text{row}_m]
\end{align*}
\]
3.3 Semantics

We use a variant of interaction trees [Xia et al. 2020] for our semantic domain, cleanly modelling non-termination and interactions with the surrounding execution environment. In this section we review interaction trees and their implementation in HOL4, before examining PureLang semantics.

**Interaction trees (ITrees).** ITrees are a data type for representing the interactions of computations with their environments. Intuitively (but not technically), they are a coinductive variant of the free monad, i.e., a potentially infinite series of uninterpreted interactions. Each interaction is a pair: a computational output and a continuation, which accepts the environment’s response and produces the rest of computation. In Coq, ITrees are the coinductive interpretation of the following grammar:

\[
\text{itree } E R ::= \text{Ret (} r : R \text{)} \mid \text{Div} \mid \text{Vis } (A : \text{Type}) (e : E A) (k : A \rightarrow \text{itree } E R)
\]

That is, an ITree with event type \(E\) and return type \(R\) is either: Ret \(r\), an immediate halt to produce \(r\); Tau \(t\), a silent step which carries on as \(t\); or Vis \(A e k\), an output \(e\) which expects a response \(a : A\), before continuing as \(ka\). Tau nodes permit expression of silently diverging computations without violating Coq’s guardedness condition. However, ITrees must be equated via weak bisimulation (they can differ by a finite number of silent steps).

**Modified ITrees in HOL4.** The definition above is not expressible in HOL4’s simple type theory (\(E\) is a type-level function; Vis quantifies over the type \(A\))—we must modify it. We require \(E\) to be simply typed and lift quantification of \(A\) to the top-level. Foster et al. [2021] faced the same issue when defining ITrees in Isabelle/HOL; instead they chose to remove \(A\), forcing environment responses to share the same type as program outputs (\(E\)).

As HOL4 has no guardedness condition when writing co-recursive functions, we further remove Tau; ITrees can be equated by strong bisimulation once more (which coincides with HOL4 equality). To recover the ability to express silent divergence, we add a nullary Div constructor, which we can straightforwardly produce using non-constructivity of HOL4’s logic (eq. (2)).

Our final definition is below. Note that it is less expressive than the original: the (simple) types of program outputs \(E\) and environment responses \(A\) are fixed at the top-level. Fortunately, PureLang semantics requires only fixed types for both (see paragraph below). Note also that we use our modified ITrees as a convenient semantic domain only, as discussed further in § 8.

\[
\text{itree } E A R ::= \text{Ret (} r : R \text{)} \mid \text{Div} \mid \text{Vis (} e : E \text{)} (k : A \rightarrow \text{itree } E A R)
\]

**Semantics of PureLang.** We must define a semantics function for PureLang: \([e] : \text{itree } E A R\) for appropriate instantiations of \(E, A, R\). Figure 3 shows the key definitions required: weak-head normal forms \(wh\), and our instantiations of \(E, A, R\).

The only externally observable effects of PureLang programs are communications with the surrounding environment via FFI. Visible program outputs \(E\) consist only of FFI calls: a pair of an FFI channel name \(ch\) and argument \(s\) (both of type \text{string}\(^2\)). Environment responses \(A\) consist only of FFI outputs: successful return of a string, failure, or lack of return (i.e., divergence). As in CakeML, PureLang FFI functions are written in C: \(ch\) is effectively a function name, and the various strings are proxies for the low-level byte arrays used to interface with the C functions. PureLang’s observable return values \(R\) are therefore successful termination, runtime type error, or FFI failure/divergence.

We build our semantics function using three intermediate stages, shown here with their types:

1. \(\text{eval}_{wh}^n : (n : \text{num}) \rightarrow e \rightarrow wh\)
2. \(\text{eval}_{wh} : e \rightarrow wh\)

3. \((\cdot, \cdot, \cdot) : wh \rightarrow \kappa \rightarrow \sigma \rightarrow \text{itree } E A R\) for stacks \(\kappa\) and mutable stores \(\sigma\).

\(^2\)Unlike Haskell’s \text{String}, PureLang’s string is not a list of characters—rather, an efficient representation using packed bytes (like Haskell’s \text{Text}.)
First, pure evaluation produces weak-head normal forms in a functional big-step [Owens et al. 2016] style ($\text{eval}_{\text{wh}}^n e = \text{wh}$), i.e., a recursive, fuelled, call-by-name interpreter for semantic expressions. The weak-head normal form $\text{diverge}$ indicates running out of fuel. Note that monadic operations $\text{monadic mop}[ e_n ]$ are both weak-head normal forms and expressions ($e$).

Second, we lift to non-fuelled evaluation ($\text{eval}_{\text{wh}} e = \text{wh}$) by classically quantifying over clock $n$, and derive clean semantic rules, e.g., for $\text{eq}$ and $\text{proj}$ (eq. (1), pg. 4):

$$
\text{eval}_{\text{wh}} e \overset{\text{def}}{=} \text{wh} \begin{cases} 
\text{diverge} & \text{if } \exists n. \text{eval}_{\text{wh}}^n e = \text{wh} \land \text{wh} \neq \text{diverge}, \\
\text{diverge} & \text{otherwise, i.e., } \forall n. \text{eval}_{\text{wh}}^n e = \text{diverge}.
\end{cases}
$$

Third and last, we statefully interpret monadic operations $\text{monadic mop}[ e_n ]$ using a stack machine with mutable state. Machine states consist of a weak-head normal form $\text{wh}$, a stack of continuations $\kappa$, and mutable state $\sigma$: $(\text{wh}, \kappa, \sigma)$. We define this stack machine as a coinductive function $\langle \cdot, \cdot, \cdot \rangle$ that produces ITrees. We omit a formal definition here, instead showing descriptive equations we have derived in fig. 4 (note the absence of Tau nodes during silent steps). Machine states with diverging or ill-behaved expressions emit Div and Ret nodes respectively. A $\text{bind}$-statement continues as its left-hand expression, pushing a $\text{bind}$-continuation onto the stack; conversely a $\text{return}$ pops a $\text{bind}$-continuation off the stack. Raising an exception pops successive continuations off the stack until a $\text{handle}$-continuation is reached. Array operations interact with mutable state, e.g., $\text{len}$ queries the length of an array in the state. An $\text{action}$-statement produces a visible (Vis) node which both contains the output message of the program and accepts response $a$ from the environment to construct the remainder of the ITree coinductively. We elide the full definition of the Vis construction due to its verbose checking of FFI errors; suffice it to say that a well-formed environment response of $\text{ok} s'$ generates the ITree $\langle \text{return} (\text{str} s'), \kappa, \sigma \rangle$.

We can then define the semantics of expressions straightforwardly using the empty continuation stack and initial empty state: $\langle e \rangle \overset{\text{def}}{=} \langle \text{eval}_{\text{wh}} e, \epsilon, \emptyset \rangle$.

Note that the equation for $\text{bind} e_1 e_2$ implies that monadic operations are strict: $e_1$ is always weak-head normalised and statefully interpreted regardless of its usage in $e_2$. PureLang I/O is therefore not lazy; lazy I/O is known to break referential transparency.$^3$

3.4 Equational reasoning

Haskell programmers rely on equational reasoning, “stepping through” program execution by unfolding function definitions. Here we describe our formalisation of equational reasoning and prove its coincidence with contextual equivalence.

We adopt untyped applicative bisimilarity from Abramsky’s lazy λ-calculus [Abramsky 1990] as an equivalence relation on expressions. That is, a relation satisfies applicative simulation if it is closed under weak-head reduction and application of any resulting weak-head normal form. Each possible weak-head normal form (wh) imposes a restriction; for example functions require applicative simulations to satisfy (for closed $e_1, e_2, e$):

$$e_1 \mathcal{R} e_2 \Rightarrow \text{eval}_{wh} e_1 = \lambda x_1 e'_1 \Rightarrow \exists x_2 e'_2. \text{eval}_{wh} e_2 = \lambda x_2 e'_2 \ \land \ \forall e. e'_1[e/x] \mathcal{R} e'_2[e/x]$$

Applicative bisimulations are symmetric applicative simulations, and applicative bisimilarity ($e_1 \approx e_2$) is the greatest applicative bisimulation. We derive strong proof principles for applicative bisimilarity using up-to techniques [Pous 2016], and so show it is an equivalence.

We must also prove it is a congruence: expressions formed from bisimilar sub-expressions are themselves bisimilar. Howe’s method [Howe 1996] is a well-studied technique for establishing congruence. A short summary of the technique follows; Pitts [2012] gives a detailed account.

Applicative (bi)similarity is extended to open terms using closing substitutions (open (bi)similarity, $\Gamma \vdash e_1 \leq e_2$ and $\Gamma \vdash e_1 \equiv e_2$ for free variables $\Gamma$). We define Howe’s construction, an inductive definition of a relation $\Gamma \vdash \mathcal{R}^H$ from a relation $\Gamma \vdash \mathcal{R}$. By construction, $\mathcal{R}^H$ is closed under substitution and is a congruence. To demonstrate congruence of open bisimilarity, we prove $\Gamma \vdash e_1 \leq^H e_2$ iff $\Gamma \vdash e_1 \equiv e_2$. Expression equivalence ($e_1 \equiv e_2$) is straightforwardly defined in terms of open bisimilarity. We turn our attention to uses of expression equivalence.

We define $\alpha$-equivalence using $\text{perm}_{x,y}$, which swaps all instances of variables $x$ and $y$ in an expression (whether bound or free). In particular, $\alpha$-equivalence is the transitive closure of a relation $\mathcal{R}_\alpha$ which swaps the variable bound at a single site while carefully avoiding free variables, e.g.:

$$y \notin \text{freevars} e \Rightarrow (\lambda x. e) \mathcal{R}_\alpha (\lambda y. \text{perm}_{x,y} e)$$

We show that $\alpha$-equivalence is contained within expression equivalence. We define capture-avoiding substitution ($e(\ell/x)$) as a freshening of bound variables followed by ordinary substitution. By noting that freshening is a special case of $\alpha$-conversion, we prove the following standard $\beta$-equivalence:

$$(\lambda x. e) \cdot e' \equiv e(\ell/x)$$
Expression equivalence also coincides with contextual equivalence \((e_1 \sim e_2)\), defined as equality of observable events (i.e., ITree equality) under all closing contexts:

\[ e_1 \sim e_2 \iff e_1 \equiv e_2 \]

To prove the right-to-left direction, we use congruence of expression equivalence and strong bisimulation on ITrees. To prove the converse, we take the contrapositive: given two inequivalent expressions, we construct a context which distinguishes them.

### 3.5 Type system

**PureLang** has a standard Hindley-Milner type system [Hindley 1969; Milner 1978]. Typing judgements are defined over compiler expressions \((ce)\), but soundness must be proved with respect to desugared expression \((e)\) semantics (§§ 3.2 and 3.3). Therefore, we consider a semantic expression \(e\) well-typed if there is some well-typed compiler expression \(ce\) which desugars to it. However, type preservation does not hold of case-statements, which desugar into nested if-/eq-/proj-statements (eq. (1), pg. 4). A successful pattern match substitutes a bare proj-statement into the continuation expression, e.g.:

\[
\text{(evalwh} \circ \text{exp_of)} \left( \text{case } x = \text{cons } cname[ce] \text{ of } cname[y] \rightarrow ce' \right) = \\
\text{evalwh} \left( \text{exp_of } ce' \right) \left( \text{cons } cname[\text{exp_of } ce]/x \right) \left[ \text{proj}_0 \text{ cname } \left( \text{cons } cname[\text{exp_of } ce] \right)/y \right]
\]

But a proj-statement can only be produced by desugaring a case-statement, and well-typed case-statements must be exhaustive; in other words, proj-statements are only well-typed when several of them are found together (representing an exhaustive pattern match). Therefore, a bare proj-statement is ill-typed in general and the reduction above violates type preservation.

Our proof of type soundness is therefore non-standard. We define a syntax of “typing expressions” \(tce\) and associated typing rules. Typing expressions satisfy type preservation by construction, due to the introduction of safeproj, which desugars as follows:

\[
\text{exp_of (safeproj}_m \text{ cname } tce) \overset{\text{def}}{=} \text{if (eq? cname } m \text{ (exp_of } tce) \text{) then proj}_n \text{ cname } (\text{exp_of } tce) \text{ else } \bot
\]

Use of the always-diverging \(\bot\) ensures that a bare safeproj never produces a runtime type error, unlike a bare proj-statement. Therefore, desugaring tce-flavoured case to produce safeproj instead of proj permits proof of type preservation once more.

Using our equational theory (§ 3.4), we show that any compiler expression \(ce\) is equivalent to its injection into typing expressions \((texp_of)\):

\[
\vdash \text{wf? } ce \Rightarrow \text{exp_of } ce \equiv \text{exp_of } (texp_of } ce)
\]

The precondition \(\text{wf?}\) asserts that expressions are syntactically well-formed (e.g., no empty case-statements), and is guaranteed by our typing rules. We use this result to lift type soundness to compiler expressions. Note that indirection through typing expressions \(tce\) is just a proof technique; they do not appear in the compiler implementation.

### 4 COMPILER FRONT END

#### 4.1 Parsing expression grammar (PEG) parsing

Parsing and lexing convert textual source files into PureLang compiler expressions \(ce\) (fig. 2, pg. 4). Input files should be complete programs, with a definition of main. A file’s definitions \((def_1, \ldots, def_n)\) are combined into a single letrec-statement, i.e., as if they had been written \(\text{let } \{ def_1; \ldots; def_n \} \text{ in } \text{main}\). Data type declarations (using data, only at the top-level) are passed separately to type inference (§ 4.3); the compiler only needs constructor names and arities.
To support Haskell-like indentation-sensitive syntax, we augment CakeML’s PEG support with the indentation relations of Adams [2013]: each $X_i$ in a traditional context-free grammar production $N \to X_1X_2 \ldots X_n$ gains an annotation describing how its indentation relates to that of the non-terminal $N$. The indentation of a terminal is a column number in the input string; the indentation of a non-terminal is governed by the annotations of the $X_i$ on the right-hand side of its production.

We use the same set of relations chosen by Adams [2013]: \{=, \geq, \@\}, where the first and second arguments are the indentations of an $X_i$ and the non-terminal $N$ respectively (\@ is the universal relation: $\forall n \ m. \ n @ m$). For example, our PEG rule for type signatures is: $|\text{Decl}| \to |\text{Ident}| = ':' > \text{Ty}$, where $|N|$ is a naming convention indicating that the indentation of non-terminal $N$ is equal to that of its first token. This rule requires the double-colon token (':' ) and the type (Ty) to appear to the right of the Decl, which itself appears in the same column as the Ident.

We adapt CakeML’s PEG-parsing algorithm by recording the set of possible indentations $N_C$ for the current non-terminal during PEG-evaluation. To initialise this set, we combine the indentation set $N_P$ of the parent non-terminal with the indentation relation associated with $N_C$. When exiting a production, we pass the final set back to $N_P$ and adjust it with respect to $N_C$’s indentation relation.

We represent indentation sets symbolically, as every indentation is one of four forms: a closed interval $[i \ldots j]$, a lower-bounded set $[i \ldots ]$, anywhere ($\langle N\rangle$), or nowhere ($\emptyset$). These forms are closed with respect to operation of the parsing algorithm: no further forms are required to represent indentation sets as the parser manipulates and combines them when entering, traversing, and exiting productions (rule right-hand sides).

Patterns in case-expressions consist of a constructor name (or tuple) applied to arguments which are all variables, or an underscore as a catch-all in the last branch. In particular, nested patterns are not supported. There is a tension here due to our compilation to CakeML’s source language: CakeML supports richly nested patterns, which it flattens and compiles away in an early pass of its compiler. But to target these, each intermediate language of the PureCake compiler would also have to support nested patterns, significantly complicating their semantics.

Parsing is verified to terminate on all inputs. Testing shows that it accepts many well-formed Haskell-like programs (§ 7).

### 4.2 PureLang

We first transform the program from a single letrec-statement into a series of nested let-/letrec-statements, using a dependency analysis to partition bindings into minimally mutually recursive groups such that no binding group requires variables from later groups.

Immediate dependencies ($x$ uses $y$) of each binding $x$ are apparent from its free variables, and computing a transitive closure ($x$ uses $y$) gives all dependencies. Bindings can then be sorted with respect to an ordering derived from the dependency relation: $x \leq y$ iff $y$ uses $x$ and $x = y$ iff $x \leq y \land y \leq x$. Our sorting algorithm is "pseudo-topological": it permits cyclic dependencies for equal elements, i.e., mutually dependent elements can remain in the same binding group. Correctness of the algorithm states that for any dependency $x$ uses $y$, $y$’s binding group is no later than $x$’s.

We transform code using the resulting nested binding groups, and clean it up after type inference (§ 4.3): we convert non-recursive, singleton letrec-bindings to let, and delete unused bindings.

This pass is proven sound entirely within our equational theory (§ 3.4). A valid split of a collection of recursive bindings $\chi_n = \epsilon_n$ into $\gamma_m = \epsilon_{1m}$ and $\omega_j = \epsilon_{2j}$ ensures all $\epsilon_{1m}$ do not depend on any $\omega_j$. We show that the pseudo-topological sort produces nested valid splits. It then suffices to show that a valid split permits nesting of a single letrec:

$$\text{valid_split } \chi_n = \epsilon_n \gamma_m = \epsilon_{1m} \omega_j = \epsilon_{2j} \Rightarrow$$

$$\text{letrec } \chi_n = \epsilon_n \text{ in } \epsilon \Rightarrow \text{letrec } \gamma_m = \epsilon_{1m} \text{ in } (\text{letrec } \omega_j = \epsilon_{2j} \text{ in } \epsilon)$$
4.3 Constraint-based type inference

Hindley-Milner type systems are popular due to their decidable type inference. However, well-studied algorithms $\mathcal{W}$, $\mathcal{J}$, and $\mathcal{M}$ [Lee and Yi 1998] are notorious for impenetrable type errors.

Instead, we pursue a two-phase algorithm which separates generation of typing constraints from their solving. HM$(X)$ [Odersky et al. 1999] is a well-studied example of constraint-based inference, parametrised on a signature $X$ to specify and prove sound two-phase inference for a variety of ML-like type systems at once. This generality is unnecessary for our purposes. Instead, we mechanise a proof-of-concept subset of the Haskell-tailored Top: a framework used to produce clear, precise error messages in the Helium teaching compiler [Heeren 2005; Heeren et al. 2003]. Algorithms $\mathcal{W}$ and $\mathcal{M}$ are equivalent to specific solving strategies within Top. Helium’s open-source, near-complete implementation of Haskell 98 provides a roadmap for high-quality error messages and additional language features (e.g., typeclasses) in future versions of PureCake. Separating the two phases also modularises their implementations and proofs. To the best of our knowledge, we are the first to mechanise such an approach and demonstrate its applicability to verified compilation. Note that CakeML’s verified Hindley-Milner type inference uses a traditional, one-pass algorithm.

We assume familiarity with Hindley-Milner type schemes ($\sigma ::= \forall \alpha \cdot \tau$), typing judgements $\Gamma \vdash ce : \tau$, parametric polymorphism, and unification. Polymorphism is introduced by rule HMLET (fig. 5) only for type variables not free in the typing context. Using this rule, standard inference algorithms first infer the type of $ce_1$ fully and generalise it with respect to $\Gamma$ to produce $\forall \alpha_n \cdot \tau_1$, before using the result to infer the type of $ce_2$. Two-phase inference must generate constraints for $ce_2$ without type information for $ce_1$. Then when solving constraints, it must soundly generalise $\tau_1$ with respect to $\Gamma$. Therefore, constraint syntax must be expressive enough to “remember” the monomorphic type variables found in $\Gamma$. We give an intuitive overview of Top’s solution, referring interested readers to prior work [Heeren 2005] for full details.

Judgements are of the form $M \vdash ce : \tau \Rightarrow A \land C$, which reads “for monomorphic type variables $M$, $ce$ has type $\tau$ subject to assumptions $A$ and constraints $C$”. Inference is mostly bottom-up: free term variables are assigned fresh type variables which are recorded in the assumptions multiset $A$ (TopVar, fig. 5). Assumptions bubble up to the binding which introduced their term variable; as all these assumptions must represent the same type, unification constraints ($\tau_1 \equiv \tau_2$) are generated per-assumption (e.g., TopLam, fig. 5). Monomorphic type variables are introduced by non-let bindings and passed on to sub-expressions top-down (e.g., TopLam, fig. 5).

The rule TopLet (fig. 5) uses implicit instance constraints $\tau_1 \leq_M \tau_2$ to record monomorphic type variables $M$ for constraint solving: $\tau_1$ must specialise the scheme obtained by generalising $\tau_2$ with respect to $M$. These constraints can only be solved once the set $\text{freevars}(\tau_2) - M$ ($\tau_2$’s generalisable variables) is stable, i.e., disjoint from type variables in other constraints. We omit details of our straightforward constraint solving algorithm for brevity.

Formalisation details. We first verify Top declarative inference rules, proving soundness with respect to PureLang’s type system (§ 3.5). We implement a concrete inference algorithm in a state-exception monad to enable generation of fresh unification variables. We prove that if the algorithm succeeds, its output is sound with respect to the declarative rules. Composing these proofs with type soundness shows that successful inference in a well-formed namespace of data types (ns) guarantees safe semantics:

$$\vdash \text{nsOK?} \land \text{infer}_\text{ns} ce = \text{OK} \Rightarrow \text{safe_itree}[\text{exp_of ce}]$$

Note that we do not expect to prove completeness of type inference. This is because PureLang aims to be a verified Haskell-like language, and in general Haskell type inference is not complete due to
its polymorphic recursion and rich type system. Testing shows that type inference accepts all of the well-typed programs we have written so far (§ 7).

4.4 Demand analysis

Laziness can be a powerful tool, allowing users to create unbounded structures, e.g., an infinite list of the factorials (fig. 1, pg. 3). However, delaying the evaluation of expressions (thunking) requires storing values on the heap. This can have a detrimental side-effect: the factorial function below has a space complexity of \( \Theta(n) \) if compiled naïvely:

\[
\text{letrec } \text{fact} = \lambda x. \lambda \text{acc}. \text{if } \text{var } x = 0 \text{ then } \text{var } \text{acc} \text{ else } (\text{var } x \cdot (\text{var } x - 1) \cdot (\text{var } x * \text{var } \text{acc})) \text{ in } e
\]

The accumulator \( \text{acc} \) is evaluated only on the very last recursive call of \( \text{fact} \), requiring storage of a new thunk on the heap at each iteration. If instead we use the \( \text{seq} \) operator to force the evaluation of \( \text{acc} \) every time we enter the \( \text{fact} \) function, overall space complexity is reduced to \( O(1) \).

\[
\text{letrec } \text{fact} = \lambda x. \lambda \text{acc}. \text{seq} (\text{var } \text{acc}) (\text{if } \text{var } x = 0 \text{ then } \ldots \ldots \text{ else } \ldots \ldots ) \text{ in } e
\]

The prefixing of \( \text{seq} \) (\( \text{var } \text{acc} \)) does not change semantics here, because in both branches of the \( \text{if} \)-statement the value of \( \text{acc} \) is unconditionally required. However, in general it is non-trivial to detect such requirements automatically. Therefore, the goal of our demand analysis is to force the evaluation of as many variables as possible without affecting semantics, so permitting later optimisations to remove as much unnecessary laziness as possible. Optimising compilers for lazy languages with any realism must implement demand analysis to avoid the bottleneck of heap usage, a common issue for functional programming languages. But we cannot be too greedy, or we may change program semantics, e.g., by transforming a terminating program into a diverging one.

An expression \( e \) \textit{demands} a variable \( x \) if we can prefix \( e \) by forcing \( x \) without changing its semantics, according to our equational theory (§ 3.4). By contrast, in GHC, demands are defined as an implication: if evaluating \( x \) gives a type error, then so should evaluating \( e \) [Sergey et al. 2014].

\[
e \text{ demands } x \overset{\text{def}}{=} e \overset{\equiv}{=} (\text{seq} (\text{var } x) \ e)
\]

\[
e \text{ demands}_{\text{GHC}} x \overset{\text{def}}{=} (\text{eval}_{\text{GHC}} (\text{var } x) = \text{error} \Rightarrow \text{eval}_{\text{GHC}} e = \text{error})
\]

A sanity check shows our definition is not weaker than GHC’s, i.e., \( e \text{ demands}_{\text{GHC}} x \Rightarrow e \text{ demands } x \). Its construction will also simplify soundness proofs for forcing variables using \( \text{seq} \). We derive some clean rules for this definition in the upper part of fig. 6. The context \( C \) allows us to use e.g., demands extracted from \( e_1 \) and \( e_2 \) in the expression \( \text{let } x = e_1 \text{ in } e_2 \).

To implement a non-trivial demand analysis, we need several other seemingly intuitive rules. However, these are not sound with respect to our equivalence relation (\( \equiv, \text{ § 3.4} \)) because PureLANG uses two inequivalent semantic values which correspond to a bottom element: silent divergence.
and runtime type errors. For example, a desirable rule and its counterexample are shown in the lower part of fig. 6. Instead, we follow the approach of Sergey et al. [2014] by verifying our demand analysis with respect to an equational theory which conflates silent divergence and runtime type errors. In particular, we refine \(\cong\) (§ 3.4) to produce a new relation, \(\approx\). Using \(\approx\), we can rederive all rules that held for \(\cong\), as well as the desirable rules that did not hold for \(\cong\). However, we must ensure that demand analysis does not transform a diverging program into one which produces type errors. Note that the converse (turning an ill-typed program into one that diverges) is not possible, as ill-typed programs are rejected before demand analysis. We therefore prove that demand analysis preserves well-typing of its input, and so its output cannot produce type errors either.

The definition above is simplistic; to capture more complex patterns we need new properties. In particular, function demands and demands when applied allow us to derive the rules in fig. 7.

**Function demands.** We wish to capture the demand on \(y\) in the expression \((\lambda x. \text{var } x) \cdot \text{var } y\), so we define function demands: function \(e\) demands its \(n\)th argument when sufficiently applied.

\[
e \text{demands}_f (n, m) \overset{\text{def}}{=} \forall x \ e_m. \ e_n \text{ demands } x \Rightarrow (e \cdot e_m) \text{ demands } x
\]

**Demands when applied.** Partially applied functions arise commonly in functional programming. For these, we must be able to calculate information about the potential demands that occur when the function is fully applied. For example, this can be useful in the following code:

\[
\text{let } f = (\lambda x_1. \lambda x_2. \text{var } x_1 = \text{var } x_2) \text{ in } (\text{var } f) \cdot (\text{var } y_1) \cdot (\text{var } y_2)
\]

We formalise this notion quite directly below, i.e., recording that an expression \(e\) demands \(x\) when applied to \(n\) arguments. From this definition, it is clear that \(e \text{ demands}_{wa} (x, 0)\) iff \(e\) demands \(x\).

\[
e \text{ demands}_{wa} (x, n) \overset{\text{def}}{=} \forall e_n. \ (e \cdot e_n) \text{ demands } x
\]

**Analysis of recursive functions.** Critically, we have also formalised an analysis for closed recursive functions. This is necessary for cases such as the factorial example at the beginning of this section. Assuming closed expressions and distinctness of bound variables, we prove the following theorem, where reformulate binds \(e'\) prefixes all recursive calls in \(e'\) to functions in binds with a forcing of the
demands in \( \text{binds} \), and \( \text{mark-demanded} ds e_f \) prefixes arguments \( ds \) with \( \text{seq} \) in function body \( e_f \):

\[
\left( \forall f' \ ds \ xs \ e' \ d \ (f', ds, \lambda xs. e') \in \text{binds} \land d \in ds \Rightarrow \text{reformulate binds } e' \text{ demands } d \right)
\]

\[
\Rightarrow \text{letrec } \{ f = e_f \mid (f, ds, e_f) \in \text{binds} \} e \approx
\]

\[
\text{letrec } \{ f = \text{mark-demanded } ds e_f \mid (f, ds, e_f) \in \text{binds} \} e
\]

Our implementation is naïve: it inserts \( \text{seq} \) too eagerly. Future work will make it more strategic, relying on our verification methodology (§ 5.1) to minimise the proof effort incurred when introducing e.g., heuristics for \( \text{seq} \)-insertion. We will optimise away more artefacts in later intermediate languages (§ 5.2), and find further demands by analysing pattern matching and constructors.

5 COMPILER BACK END

Figure 8 illustrates the structure of the PureCake compiler. This section describes the back end, i.e., the parts under the dotted line. We first explain our general approach to compiler proofs, before examining each intermediate language and the verification of its passes in turn.

5.1 Method: verify compiler relations, not functions

We separate verification of compiler passes from their implementations to keep the PureCake compiler extensible: verification of new passes should be minimally impacted by the design choices of previous ones. Therefore, instead of verifying semantics-preservation of concrete compiler functions, we verify syntactic relations encapsulating each pass. It then suffices to define concrete functions and prove they inhabit the relations. This two-phase approach has several benefits:

- Relations can easily impose syntactic restrictions on input code by narrowing their domains. By contrast, similarly restricting total functions requires cumbersome invariants which must be carried between proofs. Now, each relation can be verified orthogonally instead.
- Unlike functions, relations can remain high-level, avoiding concrete details such as free/bound variables, inventing fresh names, etc.
- Complicated passes can be composed of several simpler relations while requiring only a single implementation function, reducing proof complexity without sacrificing performance.
- Compiler functions can be optimised without redoing their core verification.

This can be viewed as a simple separation of concerns: relation verification focuses on code transformations, and function verification focuses on bookkeeping between compiler passes (that is, ensuring each function produces code satisfying the restrictions of its relation).
<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler implementation</th>
<th>Comments on verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete syntax</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PureLang</strong> (§ 4.2)</td>
<td>lexing, parsing, desugaring</td>
<td>can reject input; unverified</td>
</tr>
<tr>
<td>ce from fig. 2</td>
<td>split letrecs; simplify</td>
<td>preserves ≡ (§ 3.4)</td>
</tr>
<tr>
<td>pure call-by-name (subst. semantics)</td>
<td>type inference</td>
<td>sound: rejects ill-typed programs</td>
</tr>
<tr>
<td></td>
<td>simplify</td>
<td>preserves ≡ (§ 3.4)</td>
</tr>
<tr>
<td></td>
<td>demand analysis</td>
<td>preserves ≈ (§ 4.4) and well-typing</td>
</tr>
<tr>
<td></td>
<td>annotate with seqs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>translate into call-by-value; introduce delay/force; avoid delay (force (var_))</td>
<td>proof split into five relations; implementation stays within their composition</td>
</tr>
<tr>
<td></td>
<td>lift (\lambda)-abstractions out of let/letrecs</td>
<td>implementation stays within transitive closure of semantics-preserving syntactic relations</td>
</tr>
<tr>
<td></td>
<td>simplify force expressions</td>
<td>proof composed of three relations:</td>
</tr>
<tr>
<td></td>
<td>reformulate to simplify compilation to StateLang</td>
<td>1. implement IO monad statefully</td>
</tr>
<tr>
<td></td>
<td>translate into call-by-value; introduce delay/force; avoid delay (force (var_))</td>
<td>2. implement delay/force statefully</td>
</tr>
<tr>
<td></td>
<td>push _:: unit inwards</td>
<td>3. tidy the result</td>
</tr>
<tr>
<td></td>
<td>make every (\lambda)-abstraction bind a variable</td>
<td>implementation stays within transitive closure of semantics-preserving syntactic relation</td>
</tr>
<tr>
<td></td>
<td>translate to CakeML; attach helper functions</td>
<td></td>
</tr>
<tr>
<td><strong>ThunkLang</strong> (§ 5.2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pure call-by-value (subst. semantics)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EnvLang</strong> (§ 5.3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pure call-by-value (env. semantics)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>StateLang</strong> (§ 5.4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>impure call-by-value (env. semantics)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8. High-level summary of the compiler’s intermediate languages and compilation passes. Back end proofs adopt a recurring approach: syntactic relations characterise code transformations, the relations are proven to preserve semantics, and the compiler is verified to stay within the relation (§ 5.1).

### 5.2 ThunkLang

ThunkLang is the first intermediate language of the PureCake compiler. It resembles PureLang closely, and similarly has two ASTs (ce for compilation, and e for semantics, where exp_of expands the former to the latter). However, ThunkLang is call-by-value and introduces new primitives for handling thunks: delay and force. These new constructs have the following semantics:

- **delay e** delays evaluation of e by embedding it into a thunk value;
- **force e** evaluates e to a thunk value, before forcing evaluation of the expression within.

Note that ThunkLang remains pure and stateless: each time a thunk is forced, the expression within is re-evaluated. We implement sharing of thunk evaluations later, in StateLang (§ 5.4).

Compilation from PureLang to ThunkLang is verified by composing five semantics-preserving, syntactic relations. By dividing one big leap in implementation into five smaller steps in proof, we keep verification tractable without sacrificing performance.

The first relation (\(\rightarrow_{\text{thunk}}\)) characterises a naïve, direct translation. We show some of its rules in fig. 9; note that in ThkSeq, we simply assert the existence of a sufficiently fresh name (fresh). Though this relation expresses one possible semantics-preserving translation, it does not produce high quality code. For example, it permits generation of code containing many occurrences of delay (force (var x)). From just the few rules above, this inefficient pattern can arise whenever a variable is passed as a function argument, or let-bound. We avoid this bad code pattern by a simple
mechanism: we use a smart constructor \( \text{mk\_delay} \) instead of producing \text{delay} operations directly.

\[
\text{mk\_delay}
\begin{array}{l}
\text{ce} \\
\end{array}
\stackrel{\text{def}}{=}
\begin{cases}
\text{var } x & \text{if } \text{ce} = \text{force} (\text{var } x), \\
\text{delay } \text{ce} & \text{otherwise}.
\end{cases}
\]

Usage of \text{mk\_delay} is justified by the unhunk compiler relation, which permits conversion of any \text{delay} (\text{force} (\text{var } x)) to \text{var} x. We use three other syntactic relations to justify removal of various other bad code patterns, particularly around case-statements.

For each of the five syntactic relations \( \text{−R−} \) (\( R \in \{ \text{thunk}, \text{unthunk}, \ldots \} \)), we prove that any transformation mapping \( \epsilon \) to \( \epsilon' \) and satisfying \( \epsilon \text{ R } \epsilon' \) must give rise to equal observational semantics:

\[
\vdash \epsilon \text{ R } \epsilon' \land \text{safe\_i\text{tree}} [\epsilon]_{\text{source}} \land \text{closed } \epsilon \Rightarrow [\epsilon]_{\text{source}} = [\epsilon']_{\text{target}}
\]

As is common, this correctness theorem assumes that the source expression (\( \epsilon \)) never fails (safe\_i\text{tree}) and is closed, i.e., it is a whole program. Both are guaranteed by type inference (§ 4.3). We prove these theorems in three stages: one simulation proof per layer of our three-layered semantics (§ 3.3). The uppermost simulation proof produces an equality between ITrees: \([\cdot]_{\text{pure}} = [\cdot]_{\text{thunk}} \) when \( R = \text{thunk} \), otherwise \([\cdot]_{\text{thunk}} \neq [\cdot]_{\text{thunk}} \) for the other four relations.

We prove that compilation from PureLang to ThunkLang (pure\_to\_thunk) stays within the composition of our five syntactic relations (first \( \text{−thunk} \), then three others, and finally unhunk). We appeal to the correctness theorems for all five relations to derive correctness for pure\_to\_thunk:

\[
\vdash \text{safe\_i\text{tree}} [\text{exp\_of } \text{ce}]_{\text{pure}} \land \text{closed } (\text{exp\_of } \text{ce}) \Rightarrow [\text{exp\_of } \text{ce}]_{\text{pure}} = [\text{exp\_of } (\text{pure\_to\_thunk } \text{ce})]_{\text{thunk}}
\]

We intend to perform several further ThunkLang-to-ThunkLang optimisations; at the time of writing only two are implemented. The first lifts delayed \( \lambda \)-abstractions bound by a letrec\-statement into their own variable bindings; lookups of the delayed \( \lambda \)-abstraction under a \text{force} operation can be replaced with direct lookups of the \( \lambda \)-abstraction itself, speeding up calls to known functions. The second reduces repeated forcings of variables with a form of common sub-expression elimination: \text{force} (\text{var } x) is transformed into \text{var } y, hoisting \text{let } y = \text{force} (\text{var } x) \text{ in } \ldots \text{ up in the syntax tree.}

### 5.3 EnvLang

EnvLang is the intermediate language which follows ThunkLang, and is only a minor stepping stone towards its own successor, StateLang (§ 5.4). It closely mirrors ThunkLang, except its semantics relies on environments rather than substitution. Its \text{ce} data type also specifies top-level constructors for each monadic operation (\text{return}, \text{bind}, \text{raise}, \text{action}, etc.) for ease of compilation to StateLang.
Fig. 10. Extracts of the compilation (⌊−⌋) from EnvLang to StateLang. Here, $\lambda_\text{-} ce$ is a $\lambda$-abstraction which does not bind an argument.

5.4 StateLang

EnvLang is compiled into StateLang, a language which differs significantly from its predecessors. In StateLang, we introduce stateful (as the name suggests) and I/O primitives, and compile away thunk operations delay and force.

We perform two major steps in the first pass of compilation to StateLang:

- Monadic operations are compiled to suspended computations: functions that accept a "trigger" unit input before performing their effectful operations. Stateful (i.e., exception-handling, arrays, I/O) monadic operations are realised as stateful primitives, a form of monadic reflection [Filinski 1994, 2010].
- Thunk operations delay and force are compiled to stateful operations which share values, so that repeated forcing of a thunk does not incur duplicate evaluations.

The compiler implements both transformations simultaneously. However, we use two distinct syntactic relations to keep proofs tractable: one for a naïve translation into StateLang, and another which compiles away thunk operations. Therefore, the semantics of StateLang must support the thunk values (and corresponding force/delay operations) of EnvLang. Note also that a top-level PureLang program is monadic; it is therefore compiled to a suspended computation in StateLang, and must await unit input to trigger the evaluation of its monadic effects in the correct order.

Figure 10 illustrates some of the implementation of this initial pass (denoted [−]). In particular, monadic operations return, raise, and bind are compiled to suspended computations; correct compilation of bind relies on the right-to-left evaluation order of StateLang and CakeML. Monadic raise is also compiled to primitive raiseprim. Thunks become arrays of length two: the first element is a flag indicating whether the value has already been forced, and the second is either a suspended computation (if the flag is false) or the final value (if the flag is true). Therefore, delay compiles to an array allocation (alloc) with the flag set to false. To compile force ce, we first compile ce and read the flag (x[0]) in the resulting thunk-array: if true, we simply return the already-forced result (x[1]); otherwise we force the suspended computation by applying it to unit, update (−[−] := −) the thunk-array (setting the flag to true and storing the final value w), and finally return the value w. Note here that the update to a thunk-array is not type-preserving, i.e., it is a strong update.

The key correctness proof here is that of the removal of thunk operations. Unusually, we perform this low-level simulation proof in two directions, i.e., both a forward and a backward
simulation—usually, it suffices to do either, but the other is not required. This is due to a mismatch in step-counting: in forward simulation, we prove that for every \( n \) steps of the source program, the target program takes at least \( n \) sufficiently similar steps (vice versa for backward simulation). In other words, to rely on only one direction of simulation, compilation must monotonically increase or monotonically decrease the number of steps taken by a program. However, this does not hold for the compilation of thunks to stateful operations (fig. 10) which “remember” the results of previous computations: compiled delay operations require more steps; and compiled force operations can effectively skip any finite number of steps incurred in EnvLang by simply “remembering” a previously-forced result rather than recalculating it.

Following this initial pass, we implement one further optimisation within StateLang. Figure 10 shows that the result of compilation is a soup containing \( \lambda \)-abstractions that ignore their argument (\( \lambda \_ \cdot \cdot \)) and applications to a unit (\( \cdot \cdot \) unit). We clean this up by pushing applications to unit in through let-/letrec-/case-statements and simplifying \( \lambda \_ \cdot \cdot \) unit to ce wherever possible.

A note on semantics. As in all predecessor languages, exp_of : ce \( \rightarrow \) e maps compiler expressions to semantic expressions in StateLang. However, the semantics of StateLang is otherwise quite different: a small-step CESK machine [Felleisen and Friedman 1987] produces the expected ITree. There is also no need for stateful interpretation of monadic operations (§ 3.3), as StateLang implements its effectful operations as primitives (i.e., they are handled by the CESK machine).

We note one subtlety: when forcing a thunk value (i.e., using force to compute the delayed expression contained within), all stateful operations are temporarily forbidden by StateLang semantics. As force is inherited directly from the pure EnvLang, it must also be pure in StateLang. In particular, repeated forcings of the same thunk must produce the same value: it does not make sense to statefully share a previously computed value if this value can change. StateLang semantics is therefore parametrised by a flag which indicates whether we are mid-evaluation of a thunk. Forcing a thunk temporarily sets this flag (forbidding stateful operations), only clearing it once the evaluation is complete. In the meantime, any stateful operations cause runtime type errors.

Compiling StateLang to CakeML. StateLang was designed to match CakeML closely, simplifying its compilation. However, CakeML does not support \( \lambda \)-abstractions which do not bind arguments, so a StateLang-to-StateLang pass gives a fresh bound variable name to each of these. We realise PureLang primitive operations as CakeML primitives, noting that PureLang’s FFI calls operate on strings and must be implemented as CakeML FFI calls operating on low-level byte arrays. PureLang data type declarations extracted from parsing (§ 4.1) must be compiled into CakeML data type declarations too. We elide these technicalities as they are of limited interest.

6 TARGETING CAKEML

Targeting CakeML leverages its mature optimising compiler and end-to-end correctness guarantees. However, CakeML’s results are with respect to its oracle semantics, which produces linear I/O traces by using an oracle function to model the surrounding execution environment. This is incompatible with PureLang’s more general ITrees, which model all possible environment responses in their branching structure. We must therefore equate these two semantic styles.

Existing definitions and results in CakeML. CakeML uses a functional big-step style [Owens et al. 2016] semantics: a clocked (or fuelled) recursive interpreter. This style streamlines compiler proofs, particularly as the functional style leverages HOL4’s powerful equational rewriter. Semantic results are pairs of an outcome and an I/O trace; as in eq. (2) (pg. 6), classical quantification over the clock distinguishes diverging and terminating computations. The I/O trace is a growing log of the program’s I/O events: each entry contains an output from the program and the response
received from the environment. An oracle parametrises the semantics, and is used to determine the environment’s response on any given output. Note that the I/O trace may be infinite for diverging programs: it is calculated as the limit of all finite I/O traces.

CakeML’s core correctness results refer to its functional big-step semantics; however relational big-step and CESK machine [Felleisen and Friedman 1987] semantics have also been specified. The former is proved equivalent to the functional big-step style, but the latter is not specified for top-level declarations and its equivalence proofs omit I/O traces for diverging outcomes.

CakeML’s compiler correctness theorem states: compilation of a program which does not cause a runtime type error produces machine code with identical semantics up to out-of-memory errors. Machine code semantics is also specified in a functional big-step style per-architecture (Fox et al. [2017] give a full account). The “lack of runtime type error” assumption is common for optimising compilers: ill-formed programs can be optimised arbitrarily.

New definitions and results. To support our work on PureCake, we augment CakeML’s CESK semantics with top-level declarations, and strengthen equivalence results to equate I/O traces for diverging outcomes. We also define a new ITree-producing semantics for CakeML in a CESK style. Though we cannot directly equate linear oracle semantics with branching ITree semantics, we can derive a linear trace \( tr \) from an ITree \( tree \) by traversing it using an oracle \( \Delta \):

\[
\text{tree} \xrightarrow{\Delta} \text{tr}.
\]

To traverse a Vis node, we obtain the environment’s response by appealing to the oracle, informally:

\[
\Delta(e) = r \land k(r) \xrightarrow{\Delta} (e, r) \xrightarrow{\text{Vis} k} \text{tr}.
\]

For any oracle, the trace derived from the ITree semantics is identical to the one derived from the CESK semantics. Combining the various equivalence results gives:

\[
\text{trace derived from ITree semantics} = \text{CESK semantics} = \text{relational big-step semantics} = \text{functional big-step semantics}.
\]

We now prove a novel version of CakeML’s compiler correctness theorem phrased only in terms of ITrees. We define an ITree-producing semantics of machine code, proving trace-equivalence with the existing functional big-step semantics. Then the composition of compiler correctness and eq. (3) states for a given oracle and program: if the trace derived from the source semantics does not include a type error, the trace derived from the machine semantics must be either identical, or some prefix terminated by an out-of-memory error. We lift this single-trace result to a relation on ITrees extensionally (an ITree is characterised exactly by its derivable traces), producing theorem 1.

**Theorem 1.** ITree-based CakeML compiler correctness.

\[
\begin{align*}
\text{target_configs_ok} & \land \text{config machine} \land \text{safe_itree} \left[ prog \right]_s \land \\
\text{compile}_{\text{config prog}} & = \text{Some code} \land \text{code_in_memory config code machine} \\
\Rightarrow & \left[ \text{machine} \right]_M \text{prunes} \left[ prog \right]_s.
\end{align*}
\]

Given well-formed compiler/machine configurations (target_configs_ok) and a source semantics with no derivable type errors (safe_itree), the compiled program can be installed in memory (code_in_memory) with a machine semantics \( \left[ - \right]_M \) which prunes the source semantics. Pruning is straightforwardly defined as ITree equality up to out-of-memory errors and ill-formed FFI responses.

**PureCake compiler correctness.** Theorem 2 establishes correctness for the PureCake compiler. If the compiler converts an input string \( str \) to output CakeML AST \( ast_s \), then the compiler frontend has both successfully parsed \( str \) to a PURELANG compiler expression \( ce \) and a well-formed namespace of data types \( ns \) and type-checked \( ce \) to deduce that it is safe, i.e., free from runtime errors. The observable semantics of \( ce \) and \( ast_s \) are then related by itree_rel, which equates the PURELANG and
CakeML ITTree-based semantics modulo their slightly differing types. In particular, itree_rel relates CakeML’s low-level byte array I/O with PureLang’s string I/O.

Theorem 3 reformulates compiler correctness to highlight that apart from its frontend, the compiler is total: all programs which parse and type-check successfully will also compile successfully. Testing shows that the frontend accepts all of the well-formed programs we have written so far (§ 7).

Theorem 4 composes theorems 1 and 2 to give end-to-end guarantees from ce to machine code.

7 EVALUATION

We have written non-trivial programs to demonstrate expressivity of PureLang and usability of the PureCake compiler, and implemented various library functions from Haskell’s Prelude. So far, we have encountered no issues: PureCake accepts all well-formed programs we have written. QuviQ have further used PureLang to reimplement a virtual machine for smart contracts from the Haskell-based Cardano blockchain platform. In future work, they will automatically generate test cases to probe for discrepancies in the behaviour of binaries compiled by PureCake and GHC.

The PureCake development totals over 100 kLoC (measured using wc -l) in HOL4. For comparison, CakeML’s source semantics and compiler are 350 kLoC; though, CakeML’s development is much more mature and PureCake’s contains several unincorporated features at the time of writing.

7.1 Performance metrics

We measure performance of PureCake optimisations in the style of an ablation study: removing individual optimisations highlights their contributions to the efficiency of generated code. Isolating optimisations is tricky due to PureCake’s verification and multiple intermediate languages: some passes cannot be disabled without breaking proofs, and others are necessary to transform between languages. For example, compilation from PureLang to ThunkLang (§ 5.2) carries out several optimisations which cannot be separated. Therefore, we consider the following, isolatable passes:

- pure, binding group analysis and the associated cleanup within PureLang (§ 4.2);
- demands, demand analysis within PureLang (§ 4.4);
- thunk, the mk_delay smart constructor and optimisations within ThunkLang (§ 5.2); and
- state, pushing in applications to unit within StateLang (§ 5.4).

Experimental setup. We use five benchmark programs, each taking natural number input $n$ and outputting: primes, the $n$th prime; collatz, the longest Collatz sequence of numbers less than $n$; life,
the $n$th iteration of Conway’s Game of Life from a particular initial state; queens, the number of
solutions to the $n$-queens problem; and qsort, imperative quicksort of an array with length $n$. For
each program, we measure execution time and total heap allocated by the CakeML runtime (as
reported by CakeML’s debug output) using an Intel® Xeon® E-2186G and 64 GB RAM.

Analysis of results. Figure 11 shows our results. Optimisations reducing unnecessary thunk
allocation and forcing in ThunkLang improve time and space efficiency considerably. Pushing in
unit in StateLang improves efficiency for the monad-heavy qsort benchmark in particular. Binding
group analysis in PureLang has little effect, but is necessary for other passes to operate.
However, demand analysis in PureLang can cause slight regressions: it is overeager (§ 4.4),
aggressively inserting seq to produce many force operations in ThunkLang. Some of these are
removed by optimisations, but a number are not caught. In future work, we will strategically insert
fewer seq operations (with minimal proof burden due to our methodology, § 5.1), and optimise away
more force operations in ThunkLang with additional passes. Note that though demand analysis can
cause increased allocations, it produces better liveness properties: data is live for shorter durations,
enabling effective garbage collection and a lowered heap footprint. In particular, we have written
simple programs which quickly exhaust heap space unless transformed by demand analysis.

8 RELATED WORK

Verified compilation of Haskell-like languages. We touch on some work on verified variants of
Haskell by the CoreSpec project. Formal specifications of the syntax, semantics, and typing rules
of GHC’s Core have been used to propose language extensions such as dependent types [Weirich
et al. 2017]. The hs-to-coq [Breitner et al. 2018] tool translates Haskell code to Gallina, Coq’s
specification language, permitting reasoning about real-world Haskell library code. The tool itself
is unverified, but effectively internalises Haskell’s equational reasoning within Coq. In future
work, the project aims to derive an executable Coq model of Core automatically from GHC’s
implementation, permitting integration of Coq-verified optimisations in GHC as plugins. The
CoreSpec project focuses on faithfulness to GHC, where we consider end-to-end correctness.
Stelle and Stefanovic [2018] produce the first verified compiler for a minimal, lazy $\lambda$-calculus with an explicitly call-by-need semantics. Compilation to a high-level instruction machine preserves call-by-need value sharing; conversely reasoning about source-level programs is challenging and non-termination is not considered. PureLang uses a call-by-name semantics to enable straightforward equational reasoning while supporting a featureful language.

McCreight et al. [2010] compile a textual version of GHC’s Core to CompCert’s Cminor via Dminor: a strict, first-order, purely functional language with monadic effects, whose minimal type system provides memory safety in cooperation with runtime checks. Much of the pipeline from Dminor to Cminor is verified, including allocation of thunks; however, compilation of force is not.

GHC’s arity analysis pass [Breitner 2015] $\eta$-expands functions to avoid excessive thunk allocations. It is proved correct with respect to a simplified Core language, indirectly relying on Launchbury’s natural semantics for lazy languages [Launchbury 1993] (HOLCF [Müller et al. 1999] provides necessary domain-theoretic constructs). A call-by-need semantics is necessary to prove performance preservation, i.e., that $\eta$-expansion will not produce repeated computation.

Optimising compilation for lazy languages. Many decades of research have culminated in GHC, providing clear inspiration for future versions of PureCake. At a very high level, techniques such as closure conversion [Peyton Jones 1992] and selective lambda lifting [Graf and Peyton Jones 2019] reduce local definitions to sets of recursive equations, which can be evaluated using graph reduction techniques [Johnsson 1984] (in particular, the spineless tagless G-machine [Peyton Jones and Salkild 1989]). Meanwhile, strictness analyses reduce unnecessary thunk allocations and associated bookkeeping [Peyton Jones and Partain 1993; Wadler and Hughes 1987], and deforestation techniques [Wadler 1990] reduce allocation of intermediate data structures.

Reasoning about lazy languages. Much work has focused on the cost of lazy evaluation, which is complicated by stateful value-sharing. Moran and Sands [1999] created a framework to verify the correctness and cost-improvement of compiler transformations: i.e., preserving semantics without increasing evaluation cost. Recently, the clairvoyant call-by-value semantics [Hackett and Hutton 2019] has enabled local, modular reasoning for cost and improvement analyses [Li et al. 2021].

Schmidt-Schauß et al. [2015] prove equivalence of several notions of contextual equivalence in a Core-like call-by-need calculus. Proof is via fully abstract translation to a call-by-name calculus, in which they too employ Howe’s method.

Verified compilation to CakeML. The mature CakeML ecosystem has become a useful common back end for verified compilation, e.g., by Kalas and Isabelle/HOL.

Kalas [Pohjola et al. 2022] verifiably compiles a choreographic language, expressing global specifications of deadlock-free communicating systems. Compilation produces a program per endpoint such that simultaneous execution of all programs implements the global specification.

Hupel and Nipkow [2018] reify Isabelle/HOL terms in-prover before verifiably compiling them to CakeML, so removing the formalisation gap of extracting programs verified within Isabelle/HOL (c.f. Myreen and Owens [2014] for similar motivation in HOL).

Usage of ITrees. ITrees are inspired by previous work on monads and algebraic effects/handlers: they generalise the inductive I/O & action trees [Hancock and Setzer 2000; Swamy et al. 2020] and general/program monads [Letan and Régis-Gianas 2020; McBride 2015], build on the modularity of the “freer” monad [Kiselyov and Ishii 2015], and apply a resumption monad transformer [Piróg and Gibbons 2014] to the delay monad [Capretta 2005] and its general recursion. Each ITree encapsulates a (potentially infinite) series of uninterpreted events and continuations. Genericity over the type of events permits compositionality of: specification of semantics via the ITree monad; construction of
interpreters from event handlers; and equational reasoning. It further enables tailored extraction of executable ITrees: users can flexibly target language primitives for efficient testing.

ITrees can be considered a Coq-compatible version of the prior constructions above. Our usage is similarly motivated: we need to model uninterpreted effects, non-termination, and general recursion in HOL4’s simple type theory. ITrees are conveniently expressible in HOL4 with minimal modification. We noted (§ 3.3) that we use ITrees as a “convenient semantic domain only”, i.e., we encode the observable behaviour of PureLang programs in the branching structure of ITrees. We do not specify a compositional semantics using the ITree monad, or construct interpreters from event handlers (even to the lesser extent permitted by our modifications). To reason about preservation of semantics in compiler correctness proofs, we simply equate ITrees using strong bisimulation.

Future work might explore a denotational semantics for PureLang using the ITree monad.

As in § 6, ITrees are known to admit simple connections to trace-based semantics [Xia et al. 2020, §7]. Koh et al. [2019] axiomatise system calls in the Verified Software Toolchain’s [Appel 2014] separation logic for CompCert-flavoured C by using ITrees to specify their permitted external interactions. Mansky et al. [2020] build on this by proving the ITree-based specifications sound with respect to oracle-based CertiKOS [Gu et al. 2016] specifications, which consider only linear traces. Like in § 6, their proofs rely on traversal of ITrees to derive traces. However, they consider only one direction, showing that each ITree specifying the permitted interactions of a CompCert system call encompasses all possible traces of the underlying CertiKOS specification. In particular, the ITree can contain traces which are not derivable in the CertiKOS specification.

9 CONCLUSION AND FUTURE WORK

We have presented PureCake, an end-to-end verified compiler for PureLang, a featureful, Haskell-like language. Our correctness results lift the achievements of CompCert and CakeML to the non-strict, purely functional paradigm. PureLang’s equational theory permits straightforward reasoning about its programs; its compiler front end implements novel formalisations of indentation-sensitive parsing and constraint-based type inference; its compiler back end produces realistic code in the presence of non-strict semantics, and targets the mature CakeML ecosystem. To the best of our knowledge, ours is the first such language to provide these features.

Our work is only a first version of PureCake, and we have identified several immediate avenues for fruitful improvements throughout the compilation pipeline: enriching PureLang’s syntax, particularly the expressiveness of case-expressions; iterating on our proof-of-concept type inference, taking inspiration from Helium to support more of Haskell 98; improving our implementation of demand analysis; and further optimisations within our compiler back end. Further afield, we envision a verified REPL for PureCake inspired by CakeML [Sewell et al. 2022].

ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their constructive feedback and helpful shepherding. We thank Maximilian Algehed and Ulf Norell of QuviQ for their real-world usage of PureCake (§ 7) and for providing a thorough experience report. Kanabar is supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS); Abrahamsson by the Swedish Foundation for Strategic Research; and Myreen by the Swedish Research Council (grant no. 2021-05165).

ARTIFACT AVAILABILITY

An artifact supporting the results presented in this paper is openly available in Zenodo [Kanabar et al. 2023]. It contains a README.md file which describes how to verify our claims and build on our work. The latest development version of PureCake is available from our GitHub repository.
REFERENCES
Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10801). Springer. https://doi.org/10.1007/978-3-319-89884-1_35
Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In Symposium on Haskell. ACM. https://doi.org/10.1145/2804302.2804319
Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In Certified Programs and Proofs (CPP). ACM. https://doi.org/10.1145/3293880.3294106
Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In Principles of Programming Languages (POPL). ACM. https://doi.org/10.1145/2535838.2535841


Received 2022-11-10; accepted 2023-03-31