CHALMERS
UNIVERSITY OF TECHNOLOGY

PureCake: A Verified Compiler for a Lazy Functional Language

Downloaded from: https://research.chalmers.se, 2024-03-20 08:30 UTC

Citation for the original published paper (version of record):

Kanabar, H., Vivien, S., Abrahamsson, O. et al (2023). PureCake: A Verified Compiler for a Lazy Functional Language. Proceedings of the ACM on Programming Languages, 7.
http://dx.doi.org/10.1145/3591259

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology.
It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004.
research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

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

ACM Reference Format:
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. https://doi.org/10.1145/3591259

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.

Authors' addresses: Hrutvik Kanabar, University of Kent, United Kingdom, hk324@kent.ac.uk; Samuel Vivien, École Normale Supérieure PSL, France and Chalmers University of Technology, Sweden, samuel.vivien@ens.psl.eu; Oskar Abrahamsson, Chalmers University of Technology, Sweden, oskar8192@gmail.com; Magnus O. Myreen, Chalmers University of Technology, Sweden, myreen@chalmers.se; Michael Norrish, Australian National University, Australia, michael.norrish@anu.edu.au; Johannes Åman Pohjola, University of New South Wales, Australia, j.amanpohjola@unsw.edu.au; Riccardo Zanetti, Chalmers University of Technology, Sweden, znt.riccardo@gmail.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART145
https://doi.org/10.1145/3591259

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.

PureLang and its Metatheory (§ 3). Our source language (PureLang) implements standard functional constructs: first-class functions, recursion, algebraic data types, and pattern matching (§ 3.1). 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.

---

[1] A recently-fixed bug sent type-checking into an infinite loop.

```haskell
numbers :: [Integer]
numbers =
  let num n = n : num (n + 1)
  in num 0

factA :: Integer -> Integer -> Integer
factA a n =
  if n < 2 then a
  else factA (a * n) (n - 1)

factorials :: [Integer]
factorials = map (factA 1) numbers
```

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

Fig. 1. A small PureLang program printing a user-specified prefix of the factorial sequence. We have removed boilerplate definitions for brevity.

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.

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 `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: ThunkLang, EnvLang, and StateLang (§§ 5.2 to 5.4 respectively). Like PureLang, each has two ASTs: compiler and semantic expressions. Compilation to the call-by-value ThunkLang introduces thunks, and optimisation passes minimise their usage in key locations. In EnvLang, semantic definitions begin using environments rather than substitutions. 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.

$$
\begin{array}{l}
op ::= \\
\quad \mid \ \mathbf{cons}\ cname \\
\quad \mid \ \mathbf{tuple} \\
\quad \mid \ \mathbf{prim}\ primop \\
\quad \mid \ \mathbf{monadic}\ mop
\end{array}
$$

$$
\begin{array}{l}
ce ::= \\
\quad \mid \ \mathbf{var}\ x \\
\quad \mid \ op[\ \overline{ce_n}\ ] \\
\quad \mid \ \lambda\, \overline{x_n}\,.\ ce \\
\quad \mid \ ce \cdot \overline{ce_n} \\
\quad \mid \ \mathbf{let}\ x = ce_1\ \mathbf{in}\ ce_2 \\
\quad \mid \ \mathbf{letrec}\ \overline{x_n = ce_n}\ \mathbf{in}\ ce \\
\quad \mid \ \mathbf{seq}\ ce_1\ ce_2 \\
\quad \mid \ \mathbf{case}\ x = ce\ \mathbf{of}\ \overline{cname_n[\ \overline{x_{n\,m}}\ ] \longrightarrow ce_n}
\end{array}
$$

$$
\begin{array}{l}
e ::= \\
\quad \mid \ \mathbf{var}\ x \\
\quad \mid \ op[\ \overline{e_n}\ ] \\
\quad \mid \ \lambda x.\ e \\
\quad \mid \ e_1 \cdot e_2 \\
\quad \mid \ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
\quad \mid \ \mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ e \\
\quad \mid \ \mathbf{seq}\ e_1\ e_2 \\
\quad \mid \ \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
\quad \mid \ \mathbf{eq}_?\ cname\ arity\ e \\
\quad \mid \ \mathbf{proj}_n\ cname\ e
\end{array}
$$

Fig. 2. PureLang operations *op*, compiler expressions *ce*, and semantic expressions *e*.

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 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 functional languages are supported: first-class functions (`map` on line 12), general recursion (`factA` on lines 6-9), algebraic data types and pattern matching (`[]`/`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 `IO` 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 we formalise the operations **tuple** and **monadic** *mop* as **cons** *cname* using reserved *cname*s, but 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**-, $\mathbf{eq}_?$-, and **proj**-statements (eq. (1), below). To desugar $\mathbf{case}\ x = ce\ \mathbf{of}\ \overline{row_n}$ we assign exp_of *ce* to *x* and test it against each row of the pattern match ($row = cname[\ \overline{y_n}\ ] \longrightarrow ce'$) with **if**/$\mathbf{eq}_?$. 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.

$$\begin{array}{rcl}
\text{exp\_of}\,(\textbf{case}\ x = ce\ \textbf{of}\ \overline{row_n}) & \overset{\text{def}}{=} & \textbf{let}\ x = \text{exp\_of}\ ce\ \textbf{in}\ \text{expand}_x\ [\ \overline{row_n}\ ] \\
\text{expand}_x\ [cname[\ \overline{y_n}\ ] \longrightarrow ce',\ \overline{row_m}\ ] & \overset{\text{def}}{=} & \textbf{if}\ (\textbf{eq}_?\ cname\ n\ (\textbf{var}\ x))\ \textbf{then} \\
 & & \quad \textbf{let}\ \overline{y_n = \textbf{proj}_n\ cname\ (\textbf{var}\ x)}\ \textbf{in}\ (\text{exp\_of}\ ce') \\
\text{expand}_x\ [\,]\ \overset{\text{def}}{=}\ \textbf{fail} & & \textbf{else}\ \text{expand}_x\ [\ \overline{row_m}\ ]
\end{array} \tag{1}$$

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:

$$\text{itree}\ E\ R ::= \text{Ret}\ (r : R) \mid \text{Tau}\ (t : \text{itree}\ E\ R) \mid \text{Vis}\ (A : \text{Type})\ (e : E\ A)\ (k : A \longrightarrow \text{itree}\ E\ R)$$

That is, an ITree with event type E and return type R is either: $\text{Ret}\ r$, an immediate halt to produce r; $\text{Tau}\ t$, a silent step which carries on as t; or $\text{Vis}\ A\ e\ k$, an output e which expects a response $a : A$, before continuing as $k\ a$. 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.

$$\text{itree}\ E\ A\ R ::= \text{Ret}\ (r : R) \mid \text{Div} \mid \text{Vis}\ (e : E)\ (k : A \longrightarrow \text{itree}\ E\ A\ R)$$

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

$$\begin{array}{l} wh ::= \\ \quad \mid\ \textbf{constructor}\ cname[\ \overline{e_n}\ ] \\ \quad \mid\ \textbf{tuple}\ [\ \overline{e_n}\ ] \\ \quad \mid\ \textbf{monadic}\ mop[\ \overline{e_n}\ ] \\ \quad \mid\ \textbf{lambda}\ x\ e \\ \quad \mid\ \textbf{literal}\ lit \\ \quad \mid\ \textbf{error} \\ \quad \mid\ \textbf{diverge} \end{array} \qquad \begin{array}{l} E ::= \\ \quad \mid\ \textbf{ffi}\ (ch, s) \\ A ::= \\ \quad \mid\ \textbf{ok}\ s \\ \quad \mid\ \textbf{fail}_{\text{ffi}} \\ \quad \mid\ \textbf{diverge}_{\text{ffi}} \end{array} \qquad \begin{array}{l} R ::= \\ \quad \mid\ \textbf{termination} \\ \quad \mid\ \textbf{error} \\ \quad \mid\ \textbf{fail}_{\text{ffi}} \\ \quad \mid\ \textbf{diverge}_{\text{ffi}} \end{array}$$

Fig. 3. Machinery for PureLang semantics: weak-head normal forms wh and instantiations of parameters E, A, and R for the itree type.

FFI channel name ch and argument s (both of type 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:

$$1.\ \mathsf{eval}^{n}_{\mathsf{wh}} : (n : num) \longrightarrow e \longrightarrow wh \qquad 2.\ \mathsf{eval}_{\mathsf{wh}} : e \longrightarrow wh$$

$$3.\ \llparenthesis -, -, - \rrparenthesis : wh \longrightarrow \kappa \longrightarrow \sigma \longrightarrow \mathsf{itree}\ E\ A\ R \quad \text{for stacks } \kappa \text{ and mutable stores } \sigma.$$

First, pure evaluation produces weak-head normal forms in a functional big-step [Owens et al. 2016] style ($\mathsf{eval}^{n}_{\mathsf{wh}}\ e = wh$), *i.e.*, a recursive, fuelled, call-by-name interpreter for semantic expressions. The weak-head normal form **diverge** indicates running out of fuel. Note that monadic operations **monadic** $mop[\ \overline{e_n}\ ]$ are both weak-head normal forms and expressions (e).

Second, we lift to non-fuelled evaluation ($\mathsf{eval}_{\mathsf{wh}}\ e = wh$) by classically quantifying over clock n, and derive clean semantic rules, *e.g.*, for $\textbf{eq}_?$ and **proj** (eq. (1), pg. 5):

$$\mathsf{eval}_{\mathsf{wh}}\ e \overset{\text{def}}{=} \begin{cases} wh & \text{if } \exists n.\ \mathsf{eval}^{n}_{\mathsf{wh}}\ e = wh\ \wedge\ wh \neq \textbf{diverge}, \\ \textbf{diverge} & \text{otherwise, } i.e.,\ \forall n.\ \mathsf{eval}^{n}_{\mathsf{wh}}\ e = \textbf{diverge}. \end{cases} \tag{2}$$

$$\frac{\mathsf{eval}_{\mathsf{wh}}\ e = \textbf{constructor}\ cname[\ \overline{e_n}\ ]}{\mathsf{eval}_{\mathsf{wh}}\ (\textbf{eq}_?\ cname\ n\ e) = \textbf{true}} \qquad \frac{n < m \qquad \mathsf{eval}_{\mathsf{wh}}\ e_n = wh \qquad \mathsf{eval}_{\mathsf{wh}}\ e = \textbf{constructor}\ cname[\ \overline{e_m}\ ]}{\mathsf{eval}_{\mathsf{wh}}\ (\textbf{proj}_n\ cname\ e) = wh}$$

Third and last, we statefully interpret monadic operations **monadic** $mop[\ \overline{e_n}\ ]$ using a stack machine with mutable state. Machine states consist of a weak-head normal form wh, a stack of continuations κ, and mutable state σ: $\langle wh,\ \kappa,\ \sigma\rangle$. We define this stack machine as a coinductive function $\llparenthesis -, -, - \rrparenthesis$ 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 **bind**-statement continues as its left-hand expression, pushing a **bind**-continuation onto the stack; conversely a **return** pops a **bind**-continuation off the stack. Raising an exception pops successive continuations off the stack until a **handle**-continuation is reached. Array operations interact with mutable state, *e.g.*, **len** queries the length of an array in the state. An **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

---

[2] Unlike Haskell's `String`, PureLang's string is *not* a list of characters—rather, an efficient representation using packed bytes (like Haskell's `Text`.)

$$
\begin{aligned}
\llparenthesis\, \mathbf{diverge},\ \kappa,\ \sigma \,\rrparenthesis &= \mathsf{Div} \qquad\qquad \llparenthesis\, \mathbf{error},\ \kappa,\ \sigma \,\rrparenthesis = \mathsf{Ret}\ \mathbf{error} \\
\llparenthesis\, \mathbf{bind}\, e_1\, e_2,\ \kappa,\ \sigma \,\rrparenthesis &= \llparenthesis\, \mathsf{eval_{wh}}\ e_1,\ \mathbf{bind} \bullet e_2 :: \kappa,\ \sigma \,\rrparenthesis \\
\llparenthesis\, \mathbf{return}\, e,\ \varepsilon,\ \sigma \,\rrparenthesis &= \mathsf{Ret}\ \mathbf{termination} \\
\llparenthesis\, \mathbf{return}\, e_1,\ \mathbf{bind} \bullet e_2 :: \kappa,\ \sigma \,\rrparenthesis &= \llparenthesis\, \mathsf{eval_{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma \,\rrparenthesis \\
\llparenthesis\, \mathbf{raise}\, e_1,\ \mathit{frame} :: \ldots :: \mathbf{handle} \bullet e_2 :: \kappa,\ \sigma \,\rrparenthesis &= \llparenthesis\, \mathsf{eval_{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma \,\rrparenthesis \\
\mathsf{eval_{wh}}\ e = \mathbf{literal}\,(\mathbf{loc}\, l) \Rightarrow \llparenthesis\, \mathbf{len}\, e,\ \kappa,\ \sigma \,\rrparenthesis &= \llparenthesis\, \mathbf{return}\,(\mathbf{int}\, |\sigma(l)|),\ \kappa,\ \sigma \,\rrparenthesis \\
\llparenthesis\, \mathbf{action}\,(\mathbf{msg}\, \mathit{ch}\, s),\ \kappa,\ \sigma \,\rrparenthesis &= \mathsf{Vis}\ (\mathit{ch}, s)\ (\lambda a.\ \ldots)
\end{aligned}
$$

$$\textit{where } \mathbf{bind}\, e_1\, e_2 \overset{\text{def}}{=} \mathbf{monadic\,bind}[e_1\ e_2]\textit{, similarly for other monadic operations above.}$$

Fig. 4. Selected derived rules for $\llparenthesis -, -, - \rrparenthesis$, PureLang's stateful interpreter for monadic operations.

definition of the Vis construction due to its verbose checking of FFI errors; suffice it to say that a well-formed environment response of **ok** s' generates the ITree $\llparenthesis\, \mathbf{return}\,(\mathbf{str}\, s'),\ \kappa,\ \sigma \,\rrparenthesis$.

We can then define the semantics of expressions straightforwardly using the empty continuation stack and initial empty state: $[\![\, e \,]\!] \overset{\text{def}}{=} \llparenthesis\, \mathsf{eval_{wh}}\ e,\ \varepsilon,\ \varnothing \,\rrparenthesis$.

Note that the equation for **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):

$$e_1\ \mathcal{R}\ e_2 \Rightarrow \mathsf{eval_{wh}}\ e_1 = \mathbf{lambda}\, x_1\, e_1' \Rightarrow \exists x_2\ e_2'.\ \mathsf{eval_{wh}}\ e_2 = \mathbf{lambda}\, x_2\, e_2' \ \wedge\ \forall e.\ e_1'[{}^{e}/_{x_1}]\ \mathcal{R}\ e_2'[{}^{e}/_{x_2}]$$

Applicative bisimulations are symmetric applicative simulations, and applicative bisimilarity ($e_1 \simeq e_2$) 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*, $\Gamma \vdash e_1 \lesssim e_2$ and $\Gamma \vdash e_1 \simeq e_2$ for free variables Γ). 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 $\Gamma \vdash e_1 \lesssim^H e_2$ iff $\Gamma \vdash e_1 \lesssim e_2$. *Expression equivalence* ($e_1 \cong e_2$) is straightforwardly defined in terms of open bisimilarity. We turn our attention to uses of expression equivalence.

---

[3] See https://mail.haskell.org/pipermail/haskell/2009-March/021064.html.

We define α-equivalence using perm_{xy}, which swaps all instances of variables x and y in an expression (whether bound or free). In particular, α-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.*:

$$y \notin \mathsf{freevars}\ e \Longrightarrow (\lambda x.\ e)\ \mathcal{R}_\alpha\ (\lambda y.\ \mathsf{perm}_{xy}\ e)$$

We show that α-equivalence is contained within expression equivalence. We define capture-avoiding substitution ($e\langle {}^{e'}\!/_{x}\rangle$) as a freshening of bound variables followed by ordinary substitution. By noting that freshening is a special case of α-conversion, we prove the following standard β-equivalence:

$$(\lambda x.\ e) \cdot e' \ \cong\ e\langle {}^{e'}\!/_{x}\rangle$$

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:

$$e_1 \sim e_2 \text{ iff } e_1 \cong e_2 \qquad\qquad e_1 \sim e_2 \overset{\text{def}}{=} \forall C.\ [\![\, C[e_1]\,]\!] = [\![\, C[e_2]\,]\!]$$

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. 5). A successful pattern match substitutes a bare **proj**-statement into the continuation expression, *e.g.*:

$$(\mathsf{eval_{wh}} \circ \mathsf{exp\_of})\Big(\textbf{case}\ x = \textbf{cons}\ \mathit{cname}[\mathit{ce}]\ \textbf{of}\ \mathit{cname}[y] \longrightarrow \mathit{ce}'\Big) =$$

$$\mathsf{eval_{wh}}\Big((\mathsf{exp\_of}\ \mathit{ce}')\big[\textbf{cons}\ \mathit{cname}[\mathsf{exp\_of}\ \mathit{ce}]/x\big]\big[\textbf{proj}_0\ \mathit{cname}\ (\textbf{cons}\ \mathit{cname}[\mathsf{exp\_of}\ \mathit{ce}])/y\big]\Big)$$

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:

$$\mathsf{exp\_of}\,(\textbf{safeproj}_n^m\ \mathit{cname}\ \mathit{tce}) \overset{\text{def}}{=} \textbf{if}\ (\textbf{eq}_?\ \mathit{cname}\ m\,(\mathsf{exp\_of}\ \mathit{tce}))\ \textbf{then}\ \textbf{proj}_n\ \mathit{cname}\ (\mathsf{exp\_of}\ \mathit{tce})\ \textbf{else}\ \bot$$

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 (tcexp_of):

$$\vdash \mathsf{wf}_?\ \mathit{ce} \Longrightarrow \mathsf{exp\_of}\ \mathit{ce}\ \cong\ \mathsf{exp\_of}\,(\mathsf{tcexp\_of}\ \mathit{ce})$$

The precondition $\mathsf{wf}_?$ 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 `let {`$def_1; \ldots; def_n$`} in 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 $N \longrightarrow X_1 X_2 \ldots X_n$ 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]: $\{=, >, \geq, \circledast\}$, where the first and second arguments are the indentations of an X_i and the non-terminal N respectively ($\circledast$ is the universal relation: $\forall n\, m.\, n \circledast m$). For example, our PEG rule for type signatures is: $|\mathsf{Decl}| \longrightarrow |\mathsf{Ident}|^{=}\ \text{'::'}^{>}\ \mathsf{Ty}^{>}$, 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 ($\mathbb{N}$), or nowhere ($\varnothing$). 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 $x \leq y \wedge y \leq x$. 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 $\overline{x_n = e_n}$ into $\overline{y_m = e_{1\,m}}$ and $\overline{w_j = e_{2\,j}}$ ensures all $\overline{e_{1\,m}}$ do not depend on any $\overline{w_j}$. 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**:

$$\mathsf{valid\_split}\ \overline{x_n = e_n}\ \overline{y_m = e_{1\,m}}\ \overline{w_j = e_{2\,j}} \Longrightarrow$$
$$\textbf{letrec}\,\overline{x_n = e_n}\ \textbf{in}\ e \;\cong\; \textbf{letrec}\,\overline{y_m = e_{1\,m}}\ \textbf{in}\ (\textbf{letrec}\,\overline{w_j = e_{2\,j}}\ \textbf{in}\ e)$$

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 ($\sigma ::= \forall\,\overline{\alpha}\,.\ \tau$), 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 Γ to produce $\forall\,\overline{\alpha_n}\,.\ \tau_1$, 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 τ_1 with respect to Γ. Therefore, constraint syntax must be expressive enough to "remember" the monomorphic type variables found in Γ. 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 $M \vdash ce : \tau \implies A\ ;\ C$, which reads "for monomorphic type variables M, ce has type τ 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: τ_1 must specialise the scheme obtained by generalising τ_2 with respect to M. These constraints can only be solved once the set $\mathsf{freevars}(\tau_2) - M$ (τ_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.

$$\frac{\Gamma \vdash ce_1 : \tau_1 \qquad \overline{\alpha_n} \notin \Gamma \qquad \Gamma, x : \forall \overline{\alpha_n}.\, \tau_1 \vdash ce_2 : \tau_2}{\Gamma \vdash \textbf{let}\, x = ce_1 \;\textbf{in}\; ce_2 : \tau_2}\;\textsc{HMLet} \qquad \frac{}{M \vdash \textbf{var}\, x : \alpha \;\Rightarrow\; [x : \alpha] \;;\; \varnothing}\;\textsc{TopVar}$$

$$\frac{\overline{\alpha_n}, M \vdash ce : \tau' \;\Rightarrow\; A \;;\; C}{M \vdash (\lambda\, \overline{x_n}.\, ce) : (\overline{\alpha_n} \to \tau') \;\Rightarrow\; A \setminus \overline{x_n} \;;\; C \cup \bigcup_n \{\tau \equiv \alpha_n \mid x_n : \tau \in A\}}\;\textsc{TopLam}$$

$$\frac{M \vdash ce_1 : \tau_1 \;\Rightarrow\; A_1 \;;\; C_1 \qquad M \vdash ce_2 : \tau_2 \;\Rightarrow\; A_2 \;;\; C_2}{M \vdash (\textbf{let}\, x = ce_1 \;\textbf{in}\; ce_2) : \tau_2 \;\Rightarrow\; A_1 \cup A_2 \setminus x \;;\; C_1 \cup C_2 \cup \{\tau \leq_M \tau_1 \mid x : \tau \in A_2\}}\;\textsc{TopLet}$$

Fig. 5. Selected Hindley-Milner typing rules and Top inference rules.

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:

$$\vdash \mathsf{nsOK?}\; ns \wedge \mathsf{infer}_{ns}\; ce = \mathsf{OK} \Rightarrow \mathsf{safe\_itree}[\![\, \mathsf{exp\_of}\; ce \,]\!]$$

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 (*thunk*ing) 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:

$$\textbf{letrec}\, \mathit{fact} = \lambda x.\, \lambda \mathit{acc}.\; \textbf{if var}\, x = 0\; \textbf{then var}\, \mathit{acc}\; \textbf{else}\; (\textbf{var}\, \mathit{fact}) \cdot (\textbf{var}\, x - 1) \cdot (\textbf{var}\, x * \textbf{var}\, \mathit{acc})\;\; \textbf{in}\;\; e$$

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

$$\textbf{letrec}\, \mathit{fact} = \lambda x.\, \lambda \mathit{acc}.\; \textbf{seq}\; (\textbf{var}\, \mathit{acc})\; (\textbf{if var}\, x = 0\; \textbf{then}\; \ldots\; \textbf{else}\; \ldots)\;\; \textbf{in}\;\; e$$

The prefixing of **seq** (**var** *acc*) does not change semantics here, because in both branches of the **if**-statement the value of *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* *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

$$\frac{}{C \vdash (\textbf{var}\,x)\ \text{demands}\ x} \qquad \frac{C \vdash e_2\ \text{demands}\ x \qquad x \neq y}{C \vdash (\textbf{let}\ y = e_1\ \textbf{in}\ e_2)\ \text{demands}\ x}$$

$$\frac{\begin{array}{c} C \vdash e_1\ \text{demands}\ x \\ C \vdash e_2\ \text{demands}\ y \end{array}}{C \vdash (\textbf{let}\ y = e_1\ \textbf{in}\ e_2)\ \text{demands}\ x} \qquad \frac{C \vdash e_1\ \text{demands}\ x}{C \vdash (\textbf{seq}\ e_1\ e_2)\ \text{demands}\ x}$$

$$\frac{C \vdash e_2\ \text{demands}\ x}{C \vdash (\textbf{seq}\ e_1\ e_2)\ \text{demands}\ x} \qquad \begin{array}{c} [\![\ \textbf{let}\ x = \bot\ \textbf{in}\ \textbf{seq}\ \textbf{fail}\ (\textbf{var}\ x)\ ]\!] = \text{Ret}\ \textbf{error} \\ [\![\ \textbf{let}\ x = \bot\ \textbf{in}\ \textbf{seq}\ (\textbf{var}\ x)\ (\textbf{seq}\ \textbf{fail}\ (\textbf{var}\ x))\ ]\!] = \text{Div} \end{array}$$

Fig. 6. Selected demand analysis rules. Rules above the line are sound with respect to ≅ (§ 3.4); the rule below requires ≈ due to the counterexample shown.

an implication: if evaluating x gives a type error, then so should evaluating e [Sergey et al. 2014].

$$e\ \text{demands}\ x \overset{\text{def}}{=} e \cong (\textbf{seq}\ (\textbf{var}\ x)\ e)$$

$$e\ \text{demands}_{\text{GHC}}\ x \overset{\text{def}}{=} \left(\text{eval}_{\text{GHC}}\ (\textbf{var}\ x) = \textbf{error} \Rightarrow \text{eval}_{\text{GHC}}\ e = \textbf{error}\right)$$

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 **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 $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$.

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 (≅, § 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 ≅ (§ 3.4) to produce a new relation, ≈. Using ≈, we can rederive all rules that held for ≅, as well as the desirable rules that did not hold for ≅. 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 $(\lambda x.\ \textbf{var}\ x) \cdot \textbf{var}\ y$, so we define *function demands*: function e demands its nth argument when sufficiently applied.

$$e\ \text{demands}_{\text{f}}\ (n, m) \overset{\text{def}}{=} \forall x\ \overline{e_m}.\ e_n\ \text{demands}\ x \Rightarrow (e \cdot \overline{e_m})\ \text{demands}\ x$$

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:

$$(\textbf{let}\ f = (\lambda x_1.\ \lambda x_2.\ \textbf{var}\ x_1 = \textbf{var}\ x_2)\ \textbf{in}\ (\textbf{var}\ f) \cdot (\textbf{var}\ y_1)) \cdot (\textbf{var}\ y_2)$$

$$\frac{C \vdash e_1 \ \mathsf{demands_{wa}}\ (x, n+1)}{C \vdash (e_1 \cdot e_2)\ \mathsf{demands_{wa}}\ (x, n)}$$

$$\frac{C \vdash e\ \mathsf{demands_{wa}}\ (x, n)}{C \vdash (\lambda w.\ e)\ \mathsf{demands_{wa}}\ (x, n+1)}$$

$$\frac{C \vdash e\ \mathsf{demands_{f}}\ (n, m)}{C \vdash (\lambda w.\ e)\ \mathsf{demands_{f}}\ (n+1, m+1)}$$

$$\frac{C \vdash e\ \mathsf{demands_{wa}}\ (x, m)}{C \vdash (\lambda x.\ e)\ \mathsf{demands_{f}}\ (0, m+1)}$$

$$\frac{C \vdash e_1\ \mathsf{demands_{f}}\ (n+1, m+1)}{C \vdash (e_1 \cdot e_2)\ \mathsf{demands_{f}}\ (n, m)}$$

$$\frac{\begin{array}{c} C \vdash e_1\ \mathsf{demands_{f}}\ (0, n+1) \\ C \vdash e_2\ \mathsf{demands}\ x \end{array}}{C \vdash (e_1 \cdot e_2)\ \mathsf{demands_{wa}}\ (x, n)}$$

Fig. 7. Selected rules for $-\mathsf{demands_f}-$ and $-\mathsf{demands_{wa}}-$.

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 $e\ \mathsf{demands_{wa}}\ (x, 0)$ iff $e\ \mathsf{demands}\ x$.

$$e\ \mathsf{demands_{wa}}\ (x, n) \ \overset{\text{def}}{=}\ \forall\, \overline{e_n}.\ (e \cdot \overline{e_n})\ \mathsf{demands}\ x$$

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 *binds*, and mark_demanded *ds* e_f prefixes arguments *ds* with **seq** in function body e_f:

$$\Big(\forall f'\ ds\ xs\ e'\ d.\ (f',\ ds,\ \lambda xs.\ e') \in binds\ \land\ d \in ds \Rightarrow (\mathsf{reformulate}\ binds\ e')\ \mathsf{demands}\ d\Big)$$
$$\Rightarrow \mathbf{letrec}\ \left\{ f = e_f \,\middle|\, (f,\ ds,\ e_f) \in binds \right\}\ e\ \approx$$
$$\mathbf{letrec}\ \left\{ f = \mathsf{mark\_demanded}\ ds\ e_f \,\middle|\, (f,\ ds,\ e_f) \in binds \right\}\ e$$

Our implementation is naïve: it inserts **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 **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.*

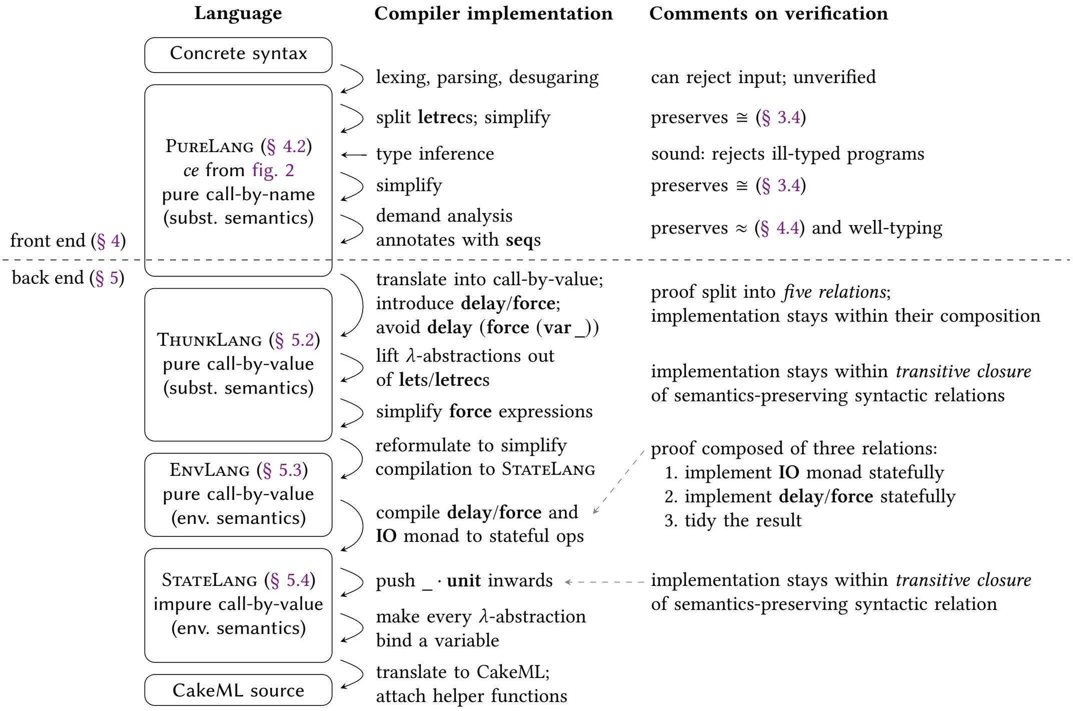
Language
Compiler implementation
Comments on verification
Concrete syntax
lexing, parsing, desugaring
can reject input; unverified
split letrecs; simplify
preserves ≅ (§ 3.4)
PureLang (§ 4.2)
ce from fig. 2
pure call-by-name
(subst. semantics)
type inference
sound: rejects ill-typed programs
simplify
preserves ≅ (§ 3.4)
demand analysis
annotates with seqs
preserves ≈ (§ 4.4) and well-typing
front end (§ 4)
back end (§ 5)
translate into call-by-value;
introduce delay/force;
avoid delay (force (var _))
proof split into five relations;
implementation stays within their composition
ThunkLang (§ 5.2)
pure call-by-value
(subst. semantics)
lift λ-abstractions out
of lets/letrecs
implementation stays within transitive closure
of semantics-preserving syntactic relations
simplify force expressions
reformulate to simplify
compilation to StateLang
proof composed of three relations:
1. implement IO monad statefully
2. implement delay/force statefully
3. tidy the result
EnvLang (§ 5.3)
pure call-by-value
(env. semantics)
compile delay/force and
IO monad to stateful ops
StateLang (§ 5.4)
impure call-by-value
(env. semantics)
push _ · unit inwards
implementation stays within transitive closure
of semantics-preserving syntactic relation
make every λ-abstraction
bind a variable
CakeML source
translate to CakeML;
attach helper functions

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).

- 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).

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.

$$\frac{}{\mathbf{var}\,x \xrightarrow{\mathsf{thunk}} \mathbf{force}\,(\mathbf{var}\,x)}\ \textsc{ThkVar} \qquad \frac{e_1 \xrightarrow{\mathsf{thunk}} e_1' \quad e_2 \xrightarrow{\mathsf{thunk}} e_2'}{e_1 \cdot e_2 \xrightarrow{\mathsf{thunk}} e_1' \cdot \mathbf{delay}\; e_2'}\ \textsc{ThkApp}$$

$$\frac{e_1 \xrightarrow{\mathsf{thunk}} e_1' \quad e_2 \xrightarrow{\mathsf{thunk}} e_2'}{\mathbf{let}\,x = e_1\ \mathbf{in}\ e_2 \xrightarrow{\mathsf{thunk}} \mathbf{let}\,x = \mathbf{delay}\; e_1'\ \mathbf{in}\ e_2'}\ \textsc{ThkLet}$$

$$\frac{e_1 \xrightarrow{\mathsf{thunk}} e_1' \quad e_2 \xrightarrow{\mathsf{thunk}} e_2' \quad \mathit{fresh} \notin \mathsf{freevars}\; e_2}{\mathbf{seq}\; e_1\; e_2 \xrightarrow{\mathsf{thunk}} \mathbf{let}\,\mathit{fresh} = e_1'\ \mathbf{in}\ e_2'}\ \textsc{ThkSeq}$$

Fig. 9. Selected rules from $- \xrightarrow{\mathsf{thunk}} -$, the first proof relation between PureLang and ThunkLang.

The first relation ($- \xrightarrow{\mathsf{thunk}} -$) 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 (mk_delay) instead of producing **delay** operations directly.

$$\mathsf{mk\_delay}\ \mathit{ce} \overset{\text{def}}{=} \begin{cases} \mathbf{var}\,x & \text{if } \mathit{ce} = \mathbf{force}\,(\mathbf{var}\,x), \\ \mathbf{delay}\,\mathit{ce} & \text{otherwise.} \end{cases}$$

Usage of mk_delay is justified by the unthunk compiler relation, which permits conversion of any **delay** (**force** (**var** x)) to **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 $-\mathcal{R}-$ ($\mathcal{R} \in \{\xrightarrow{\mathsf{thunk}}, \mathsf{unthunk}, \ldots\}$), we prove that any transformation mapping e to e' and satisfying $e\ \mathcal{R}\ e'$ must give rise to *equal* observational semantics:

$$\vdash e\ \mathcal{R}\ e' \wedge \mathsf{safe\_itree}\ [\![\, e \,]\!]_{\mathsf{source}} \wedge \mathsf{closed}\ e \Rightarrow [\![\, e \,]\!]_{\mathsf{source}} = [\![\, e' \,]\!]_{\mathsf{target}}$$

As is common, this correctness theorem assumes that the source expression (e) never fails (safe_itree) 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: $[\![-]\!]_{\mathsf{pure}} = [\![-]\!]_{\mathsf{thunk}}$ when $\mathcal{R} = \xrightarrow{\mathsf{thunk}}$, otherwise $[\![-]\!]_{\mathsf{thunk}} = [\![-]\!]_{\mathsf{thunk}}$ 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 $- \xrightarrow{\mathsf{thunk}} -$, then three others, and finally unthunk). We appeal to the correctness theorems for all five relations to derive correctness for pure_to_thunk:

$$\begin{aligned} &\vdash \mathsf{safe\_itree}\ [\![\, \mathsf{exp\_of}\ \mathit{ce} \,]\!]_{\mathsf{pure}} \wedge \mathsf{closed}\ (\mathsf{exp\_of}\ \mathit{ce}) \Rightarrow \\ &\qquad\qquad [\![\, \mathsf{exp\_of}\ \mathit{ce} \,]\!]_{\mathsf{pure}} = [\![\, \mathsf{exp\_of}\ (\mathsf{pure\_to\_thunk}\ \mathit{ce}) \,]\!]_{\mathsf{thunk}} \end{aligned}$$

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

$$\begin{aligned}
\lfloor \textbf{return}\ ce \rfloor &\overset{\text{def}}{=} \textbf{let}\ x = \lfloor ce \rfloor\ \textbf{in}\ \lambda\_.\ \textbf{var}\ x \\
\lfloor \textbf{raise}\ ce \rfloor &\overset{\text{def}}{=} \textbf{let}\ x = \lfloor ce \rfloor\ \textbf{in}\ \lambda\_.\ \textbf{raise}_{\text{prim}}(\textbf{var}\ x) \\
\lfloor \textbf{bind}\ ce_1\ ce_2 \rfloor &\overset{\text{def}}{=} \lambda\_.\ \lfloor ce_2 \rfloor \cdot (\lfloor ce_1 \rfloor \cdot \textbf{unit}) \cdot \textbf{unit}
\end{aligned}$$

$$\begin{aligned}
\lfloor \textbf{delay}\ ce \rfloor &\overset{\text{def}}{=} \textbf{alloc}\ [\textbf{false}, \lambda\_.\ \lfloor ce \rfloor] \\
\lfloor \textbf{force}\ ce \rfloor &\overset{\text{def}}{=} \textbf{let}\ x = \lfloor ce \rfloor\ ;\ x_0 = x[0]\ ;\ x_1 = x[1]\ \textbf{in} \\
&\qquad\qquad \textbf{if}\ \textbf{var}\ x_0\ \textbf{then}\ \textbf{var}\ x_1\ \textbf{else} \\
&\qquad\qquad\quad \textbf{let}\ w = (\textbf{var}\ x_1) \cdot \textbf{unit}\ \textbf{in} \\
&\qquad\qquad\qquad x[0] := \textbf{true}\ ;\ x[1] := \textbf{var}\ w\ ;\ \textbf{var}\ w
\end{aligned}$$

Fig. 10. Extracts of the compilation ($\lfloor - \rfloor$) from EnvLang to StateLang. Here, $\lambda\_.\ ce$ is a λ-abstraction which does not bind an argument.

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 *ce* data type also specifies top-level constructors for each monadic operation (**return**, **bind**, **raise**, **action**, *etc.*) for ease of compilation to StateLang.

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 $\lfloor - \rfloor$). 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 $\textbf{raise}_{\text{prim}}$. 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 λ-abstractions that ignore their argument ($\lambda\_.\ -$) and applications to a unit ($- \cdot$ **unit**). We clean this up by pushing applications to **unit** in through **let**-/**letrec**-/**case**-statements and simplifying ($\lambda\_.\ ce$) $\cdot$ **unit** to ce wherever possible.

A Note on Semantics. As in all predecessor languages, $\mathrm{exp\_of} : ce \longrightarrow e$ 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 λ-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 (Δ): $\mathit{tree} \overset{\Delta}{\rightsquigarrow} \mathit{tr}$. To traverse a Vis node, we obtain the environment's response by appealing to the oracle, informally:

$$\Delta(e) = r \;\wedge\; k(r) \overset{\Delta}{\rightsquigarrow} \mathit{tr} \;\Rightarrow\; \mathsf{Vis}\; e\; k \overset{\Delta}{\rightsquigarrow} (e, r) :: \mathit{tr}$$

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:

$$\begin{array}{c}\text{trace derived from}\\ \text{ITree semantics}\end{array} = \begin{array}{c}\text{CESK}\\ \text{semantics}\end{array} = \begin{array}{c}\text{relational big-step}\\ \text{semantics}\end{array} = \begin{array}{c}\text{functional big-step}\\ \text{semantics}\end{array} \tag{3}$$

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.

$$\begin{array}{l}\vdash \mathsf{target\_configs\_ok}\; \mathit{config}\; \mathit{machine} \;\wedge\; \mathsf{safe\_itree}\; [\![\, \mathit{prog} \,]\!]_{\text{🍰}} \;\wedge\\ \quad \mathsf{compile}_{\text{🍰}}\; \mathit{config}\; \mathit{prog} \;=\; \mathsf{Some}\; \mathit{code} \;\wedge\; \mathsf{code\_in\_memory}\; \mathit{config}\; \mathit{code}\; \mathit{machine}\\ \quad \Rightarrow\; [\![\, \mathit{machine} \,]\!]_{\mathrm{M}}\; \mathsf{prunes}\; [\![\, \mathit{prog} \,]\!]_{\text{🍰}}\end{array}$$

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 ($[\![-]\!]_{\mathrm{M}}$) 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 $\mathit{ast}_{\text{🍰}}$, 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 $\mathit{ast}_{\text{🍰}}$ are then related by itree_rel, which equates the PureLang and CakeML ITree-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.

Theorem 2. Compiler correctness.

$$\begin{array}{l}
\vdash \mathsf{compiler}\ \mathit{str}\ =\ \mathsf{Some}\ \mathit{ast}_{\text{🍰}}\ \Rightarrow \\
\quad \exists\, \mathit{ce}\ \mathit{ns}.\ \ \mathsf{frontend}\ \mathit{str}\ =\ \mathsf{Some}\ (\mathit{ce},\ \mathit{ns})\ \wedge \\
\qquad \mathsf{safe\_itree}\ \ [\![\, \mathsf{exp\_of}\ \mathit{ce}\,]\!]_{\mathrm{pure}}\ \wedge \\
\qquad \mathsf{itree\_rel}\ \ [\![\, \mathsf{exp\_of}\ \mathit{ce}\,]\!]_{\mathrm{pure}}\ \ [\![\, \mathit{ast}_{\text{🍰}}\,]\!]_{\text{🍰}}
\end{array}$$

Theorem 3. Alternative compiler correctness.

$$\begin{array}{l}
\vdash \mathsf{frontend}\ \mathit{str}\ =\ \mathsf{Some}\ (\mathit{ce},\ \mathit{ns})\ \Rightarrow \\
\quad \mathsf{safe\_itree}\ \ [\![\, \mathsf{exp\_of}\ \mathit{ce}\,]\!]_{\mathrm{pure}}\ \wedge \\
\quad \exists\, \mathit{ast}_{\text{🍰}}.\ \ \mathsf{compiler}\ \mathit{str}\ =\ \mathsf{Some}\ \mathit{ast}_{\text{🍰}}\ \wedge \\
\qquad \mathsf{itree\_rel}\ \ [\![\, \mathsf{exp\_of}\ \mathit{ce}\,]\!]_{\mathrm{pure}}\ \ [\![\, \mathit{ast}_{\text{🍰}}\,]\!]_{\text{🍰}}
\end{array}$$

Theorem 4. End-to-end correctness.

$$\begin{array}{l}
\vdash \mathsf{compiler}\ \mathit{str}\ =\ \mathsf{Some}\ \mathit{ast}_{\text{🍰}}\ \wedge\ \mathsf{compile}_{\text{🍰}}\ \mathit{config}\ \mathit{ast}_{\text{🍰}}\ =\ \mathsf{Some}\ \mathit{code}\ \wedge \\
\quad \mathsf{target\_configs\_ok}\ \mathit{config}\ \mathit{machine}\ \wedge\ \mathsf{code\_in\_memory}\ \mathit{config}\ \mathit{code}\ \mathit{machine} \\
\quad \Rightarrow \exists\, \mathit{ce}\ \mathit{ns}.\ \ \mathsf{frontend}\ \mathit{str}\ =\ \mathsf{Some}\ (\mathit{ce},\ \mathit{ns})\ \wedge\ [\![\, \mathit{machine}\,]\!]_{\mathrm{M}}\ \mathsf{prunes}\ \ [\![\, \mathsf{exp\_of}\ \mathit{ce}\,]\!]
\end{array}$$

A Verified Compiler Binary. We build on CakeML's verified bootstrapping [Myreen 2021b], which relies on proof-producing synthesis of CakeML AST [Myreen and Owens 2014]. Given computable HOL4 functions (*e.g.*, the PureCake compiler), this synthesises CakeML code which is proven to implement the input HOL4 faithfully. Composing synthesis with in-logic evaluation of the CakeML compiler produces a binary that is faithful to the verified PureCake compiler definition.

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);

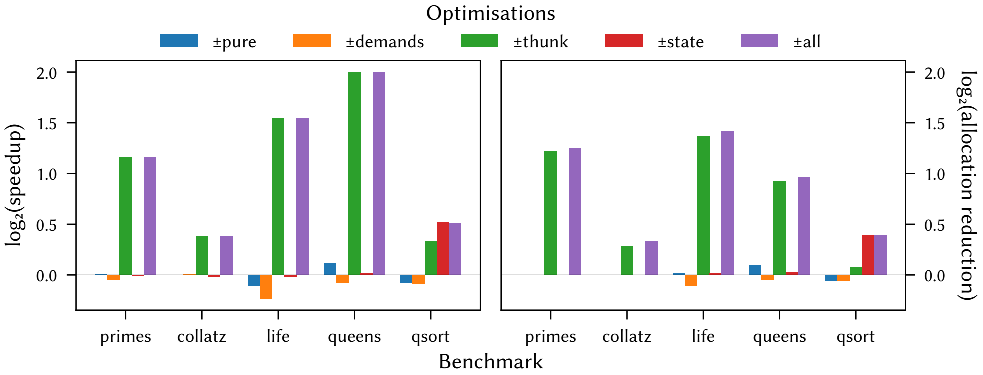

Fig. 11. Graphs showing the performance impact of optimisations—the base-2 logarithm of a ratio of measurements (execution time or heap allocations in bytes) with/without the optimisation: $\log_2 \left(m_{\text{without}}/m_{\text{with}}\right)$. The ±all bar shows impact with/without all optimisations considered. Negligible error bars are omitted.

- 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 nth prime; collatz, the longest Collatz sequence of numbers less than n; life, the nth 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 λ-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] η-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 η-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 HOL4).

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

Samson Abramsky. 1990. The Lazy λ-Calculus. In *Research Topics in Functional Programming*. Addison Wesley.

Michael D. Adams. 2013. Principled parsing for indentation-sensitive languages: revisiting Landin's offside rule. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/2429069.2429129

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

Joachim Breitner. 2015. Formally proving a compiler transformation safe. In *Symposium on Haskell*. ACM. https://doi.org/10.1145/2804302.2804312

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). *Proc. ACM Program. Lang.* 2, ICFP (2018). https://doi.org/10.1145/3236784

Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Matthias Felleisen and Daniel P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Principles of Programming Languages (POPL)*. ACM Press. https://doi.org/10.1145/41625.41654

Andrzej Filinski. 1994. Representing Monads. In *Principles of Programming Languages (POPL)*. ACM Press. https://doi.org/10.1145/174675.178047

Andrzej Filinski. 2010. Monads in action. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/1706299.1706354

Simon Foster, Chung-Kil Hur, and Jim Woodcock. 2021. Formally Verified Simulations of State-Rich Processes Using Interaction Trees in Isabelle/HOL. In *Conference on Concurrency Theory (CONCUR) (LIPIcs, Vol. 203)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CONCUR.2021.20

Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified compilation of CakeML to multiple machine-code targets. In *Certified Programs and Proofs (CPP)*. ACM. https://doi.org/10.1145/3018610.3018621

Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. *CoRR* abs/1910.11717 (2019). arXiv:1910.11717

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association. https://doi.org/10.5555/3026877.3026928

Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* 3, ICFP (2019). https://doi.org/10.1145/3341718

Peter G. Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic (CSL)*, Vol. 1862. Springer. https://doi.org/10.1007/3-540-44622-2_21

Bastiaan Heeren. 2005. *Top Quality Type Error Messages*. Ph. D. Dissertation. Utrecht University, Netherlands. http://dspace.library.uu.nl/handle/1874/7297

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In *Workshop on Haskell*. ACM. https://doi.org/10.1145/871895.871902

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969). https://doi.org/10.2307/1995158

Douglas J. Howe. 1996. Proving Congruence of Bisimulation in Functional Programming Languages. *Inf. Comput.* 124, 2 (1996). https://doi.org/10.1006/inco.1996.0008

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

Thomas Johnsson. 1984. Efficient compilation of lazy evaluation. In *Symposium on Compiler Construction*. ACM. https://doi.org/10.1145/502874.502880

Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. *Artifact for "PureCake: A Verified Compiler for a Lazy Functional Language"*. https://doi.org/10.5281/zenodo.7782305

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

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Principles of Programming Languages (POPL)*. ACM Press. https://doi.org/10.1145/158511.158618

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a Folklore Let-Polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (1998). https://doi.org/10.1145/291891.291892

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). https://doi.org/10.1145/1538788.1538814

Thomas Letan and Yann Régis-Gianas. 2020. FreeSpec: specifying, verifying, and executing impure computations in Coq. In *Certified Programs and Proofs (CPP)*. ACM. https://doi.org/10.1145/3372885.3373812

Yao Li, Li-yao Xia, and Stephanie Weirich. 2021. Reasoning about the garden of forking paths. *Proc. ACM Program. Lang.* 5, ICFP (2021). https://doi.org/10.1145/3473585

William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 12075)*. Springer. https://doi.org/10.1007/978-3-030-44914-8_16

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, Vol. 9129)*. Springer. https://doi.org/10.1007/978-3-319-19797-5_13

Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. 2010. A certified framework for compiling and executing garbage-collected languages. In *International Conference on Functional programming (ICFP)*. ACM. https://doi.org/10.1145/1863543.1863584

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978). https://doi.org/10.1016/0022-0000(78)90014-4

Andrew Moran and David Sands. 1999. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/292540.292547

Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. 1999. HOLCF=HOL+LCF. *J. Funct. Program.* 9, 2 (1999). https://doi.org/10.1017/s095679689900341x

Magnus O. Myreen. 2021a. The CakeML Project's Quest for Ever Stronger Correctness Theorems (Invited Paper). In *Interactive Theorem Proving (ITP) (LIPIcs, Vol. 193)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2021.1

Magnus O. Myreen. 2021b. A minimalistic verified bootstrapped compiler (proof pearl). In *Certified Programs and Proofs (CPP)*. ACM. https://doi.org/10.1145/3437992.3439915

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 (2014). https://doi.org/10.1017/S0956796813000282

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999).

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9632)*. Springer. https://doi.org/10.1007/978-3-662-49498-1_23

Simon Peyton Jones and Will Partain. 1993. Measuring the effectiveness of a simple strictness analyser. In *Glasgow Workshop on Functional Programming (Workshops in Computing)*. Springer. https://doi.org/10.1007/978-1-4471-3236-3_17

Simon L. Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *J. Funct. Program.* 2, 2 (1992). https://doi.org/10.1017/S0956796800000319

Simon L. Peyton Jones and John Launchbury. 1991. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 523)*. Springer. https://doi.org/10.1007/3540543961_30

Simon L. Peyton Jones and Jon Salkild. 1989. The Spineless Tagless G-Machine. In *Functional Programming Languages and Computer Architecture (FPCA)*. ACM. https://doi.org/10.1145/99370.99385

Maciej Piróg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. In *Mathematical Foundations of Programming Semantics (MFPS) (Electronic Notes in Theoretical Computer Science, Vol. 308)*. Elsevier. https://doi.org/10.1016/j.entcs.2014.10.015

Andrew M. Pitts. 2012. Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*. Cambridge tracts in theoretical computer science, Vol. 52. Cambridge University Press.

Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In *Interactive Theorem Proving (ITP) (LIPIcs, Vol. 237)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2022.27

Damien Pous. 2016. Coinduction All the Way Up. In *Logic in Computer Science (LICS)*. ACM. https://doi.org/10.1145/2933575.2934564

Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. 2015. Simulation in the Call-by-Need Lambda-Calculus with Letrec, Case, Constructors, and Seq. *Log. Methods Comput. Sci.* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:7)2015

Ilya Sergey, Simon Peyton Jones, and Dimitrios Vytiniotis. 2014. Theory and practice of demand analysis in Haskell. (June 2014). https://core.ac.uk/display/357603019 (Unpublished draft).

Thomas Sewell, Magnus O. Myreen, Yong Kiam Tan, Ramana Kumar, Alexander Mihajlovic, Oskar Abrahamsson, and Scott Owens. 2022. Cakes that Bake Cakes: Dynamic computation in CakeML. In *Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3591266

George Stelle and Darko Stefanovic. 2018. Verifiably Lazy: Verified Compilation of Call-by-Need. In *Implementation and Application of Functional Languages (IFL)*. ACM. https://doi.org/10.1145/3310232.3310236

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020). https://doi.org/10.1145/3409003

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990). https://doi.org/10.1016/0304-3975(90)90147-A

Philip Wadler and R. J. M. Hughes. 1987. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture (FPCA) (Lecture Notes in Computer Science, Vol. 274)*. Springer. https://doi.org/10.1007/3-540-18317-5_21

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP (2017). https://doi.org/10.1145/3110275

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020). https://doi.org/10.1145/3371119

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