Published at PLDI 2023

PureCake

A Verified Compiler for a Lazy Functional Language

Monday 19th June — PLDI 2023

Hrutvik Kanabar *University of Kent*
Samuel Vivien *Chalmers University of Technology, École Normale Supérieure PSL*
Oskar Abrahamsson *Chalmers University of Technology*
Magnus O. Myreen *Chalmers University of Technology*
Michael Norrish *Australian National University*
Johannes Åman Pohjola *University of New South Wales*
Riccardo Zanetti *Chalmers University of Technology*

Language guarantees

Implementing MyCriticalSoftware

C	ML	
?	type safety	type safety
	memory safety	memory safety
		purity vs. I/O
		ref. transparency
		laziness
		free theorems

Compiler guarantees

Compiling MyCriticalSoftware

C

CompCert

`0xa1b2c3...`

✓

ML

CakeML

`0xbcd456...`

✓

PureCake

`0xf9e8d7...`

✓

Why verified compilation?

How do we know what our programs will do?

understand source semantics + **trust compiler to preserve it**

We may trust GHC — but does its source semantics match our understanding?

Verified compilation gives us **a verified link** between:

- formally-specified source semantics
- semantics-preserving compilation

Our work

The **PureCake** project:

a HOL4-**verified compiler** for

a **lazy**, **purely functional** language which

is inspired by **Haskell** and

targets **CakeML**

CakeML = a verified implementation of a subset of ML [POPL14]

Key contributions

Highlights

- sound equational reasoning
- parsing expression grammar (PEG) for Haskell-like syntax
- two-phase constraint-based type inference*
- demand analysis*
- optimisations for non-strict idioms
- monadic reflection* (monadic → imperative)
- CakeML as a back end for end-to-end verified compilation

***Not mechanically verified before**

This talk

High level, whistle-stop tour!

For more details:

- Read our paper: cakeml.org/pldi23-purecake.pdf
- Visit our GitHub: github.com/cakeml/pure
- Talk to us!

Roadmap

Introduction

Source language

Compiler front end

Compiler back end

Connection with CakeML

A realistic functional language

PureLang has standard functional idioms ...

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

general recursion

```
map :: (a -> b) -> [a] -> [b]
map f l = case l of
          [] -> []
          h:t -> f h : map f t
```

algebraic data types + pattern-matching

```
factorials :: [Integer]
factorials = map (fact 1) (numbers 0)
```

higher-order functions

A realistic subset of Haskell

… and Haskell extras

```
numbers :: Integer -> [Integer]
numbers n = n : numbers (n + 1)


main :: IO ()
main = do
  n <- readInt -- read from stdin
  let facts = take n factorials
  app (\i -> print $ toString i) facts
```

laziness → infinite data

pure by default, monads for:

- sequencing
- stateful computations
- I/O

Single `IO` monad for arrays, exceptions, and I/O (via FFI calls)

Also: indentation-sensitivity, `do` notation, mutual recursion, …

Formal syntax

A tale of two ASTs… separate implementation and verification

$$ce \xrightarrow{\text{desugar}} e$$

compiler *expressions*

- higher-level
- used in implementation
- includes `case`

semantic *expressions*

- ground truth for semantics
- constructor operations: test name/arity equality & argument projection

Semantics — ITrees

Need to model non-termination and I/O

HOL4-expressible coinductive *interaction trees [Xia et. al., POPL20]*:

$$\begin{array}{rcll} \mathsf{itree}\ E\ A\ R & ::= & \mathsf{Ret}\ R & \textit{termination} \\ & \mid & \mathsf{Vis}\ E\ (A \to \mathsf{itree}\ E\ A\ R) & \textit{I/O via FFI channel} \\ & \mid & \mathsf{Div} & \textit{silent divergence} \end{array}$$

vs.

No Tau nodes? Div!?

rely on non-constructivity of HOL4's logic

strong bisimulation coincides with equality

Semantics — definitions

Operational semantics in layers:

1. Weak-head evaluation: call-by-name, functional big-step

$$\mathrm{eval}^{n}_{\mathrm{wh}}\ e = wh$$

2. Lift to unclocked evaluation

$$\mathrm{eval}_{\mathrm{wh}}\ e \overset{\mathrm{def}}{=} \begin{cases} wh & \text{for } \textbf{some } n,\ \mathrm{eval}^{n}_{\mathrm{wh}}\ e = wh \neq \mathrm{Timeout} \\ \mathrm{Timeout} & \text{for } \textbf{all } n,\ \mathrm{eval}^{n}_{\mathrm{wh}}\ e = \mathrm{Timeout} \end{cases}$$

3. Stateful interpretation of `IO` operations

$$(\!|\, -,\ -,\ - \,|\!) : wh \to \kappa \to \sigma \to \mathrm{itree}\ E\ A\ R$$

Finally, $[\![\, e \,]\!] \overset{\mathrm{def}}{=} (\!|\, \mathrm{eval}_{\mathrm{wh}}\ e,\ \varepsilon,\ \varnothing \,|\!)$

Mechanised equational reasoning

Mechanise untyped applicative bisimulation, $\cong$ *[Abramsky, 1990]*

Proved *congruent* via Howe's method *[Howe, 1996]*
i.e. bisimilar sub-expressions $\implies$ *bisimilarity*

Definitions:

- α-equivalence: $e_1 =_\alpha e_2 \overset{\text{def}}{=} \text{perm\_vars}\ e_1\ e_2$
- β-equivalence: $(\lambda x.\, e_1) \cdot e_2 =_\beta (\text{freshen}_{e_2}\ e_1)[{}^{e_2}/_{x}]$
- A standard contextual equivalence: $e_1 \sim e_2$
  (equality under all closing contexts)

Derived results:

$$e_1 \cong e_2 \iff e_1 \sim e_2 \qquad \frac{e_1 =_\alpha e_2}{e_1 \cong e_2} \qquad \frac{e_1 =_\beta e_2}{e_1 \cong e_2}$$

Type system

Standard Hindley-Milner rules… with an unusual soundness proof

Problem:

non-strict semantics + exhaustive `case` splits mean that
“preservation” (subject reduction) does not hold

Solution:

- Define an alternative syntax for typing
- Prove subject reduction by construction
- Use equational theory to bridge the gap to original syntax

Roadmap

Introduction

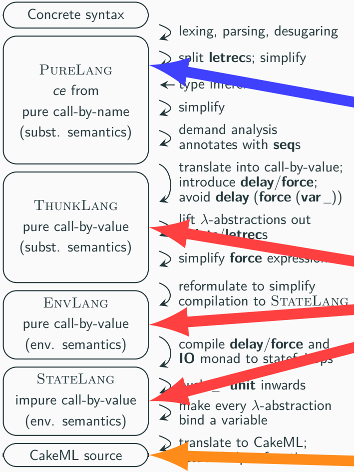
Source language

Compiler front end

Compiler back end

Connection with CakeML

Compiler structure



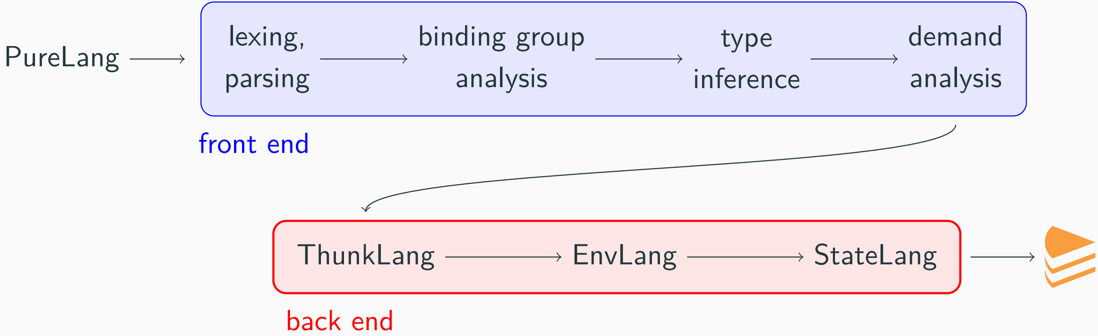
Upcoming slides examine the compiler top to bottom

Frontend accepts PureLang

Three intermediate languages, each with a specific focus

Targets CakeML

Compiler structure



PureLang → lexing, parsing → binding group analysis → type inference → demand analysis

front end

ThunkLang → EnvLang → StateLang →

back end

Parsing

Indentation-sensitive parsing expression grammar (PEG):

PEG + *[Adams POPL13]*

$$N \to X_1^{\mathcal{R}_1}\, X_2^{\mathcal{R}_2}\, \dots\, X_n^{\mathcal{R}_n}$$

where $\mathcal{R} \in \{ =,\ >,\ \geq,\ \mathcal{U} \}$

- Symbolic sets of possible indentations for each non-terminal
- Verified to terminate on all inputs

Binding group analysis

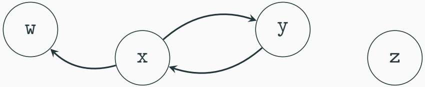
Parsing

```
z = 42
y = x + 1
x = w + y
w = 0
main
```

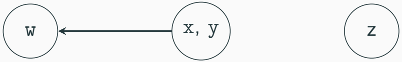
⟶

```
let rec z = 42
        y = x + 1
        x = w + y
        w = 0
    in main
```

Analyse dependencies



Pseudo-topological sort



Transform code + tidy

```
let w = 0 in
let rec x = w + y ; y = x + 1
    in main
```

Verified entirely within equational theory

Sound constraint-based type inference

Two-phases: generate *all* constraints ⟶ solve constraints

Subset of Helium's $\textsc{Top}$ framework *[Heeren et. al., Haskell 2003]*

- Open to high-quality error messages
- Path to various Haskell 98 features

Soundness theorem:

$$\begin{aligned} &\text{infer } \mathit{ce} \ \ \textbf{succeeds} \\ &\qquad \Longrightarrow\ \mathit{ce} \vdash_{\textsc{Top}} \text{cs} \quad \textit{and} \quad \text{cs } \textbf{solveable} \\ &\qquad \Longrightarrow\ \Gamma \vdash \mathit{ce} : \tau \end{aligned}$$

Demand analysis

Avoid excessive thunks — `acc` heap-allocated each recursive call!

```
fact acc n =
     if ... then acc else fact (acc * n) (n - 1)
```

- e **demands** $\overline{x_n} \overset{\text{def}}{=} e \cong (x_1$ `` `seq` `` $\ldots$ x_n `` `seq` `` $e)$
- Implement/verify* analysis: e **demands** (analyse e)
- Prefix code with `seq`, including in recursive functions

*Verify with an alternative equational theory, $\approx$ *[Sergey et. al., 2014]*:

stuck $\approx$ diverged $\approx \bot$ **but** well-typed $\Longrightarrow$ $\approx$, $\cong$ coincide

`seq`-prefixing preserves typing

Roadmap

Introduction

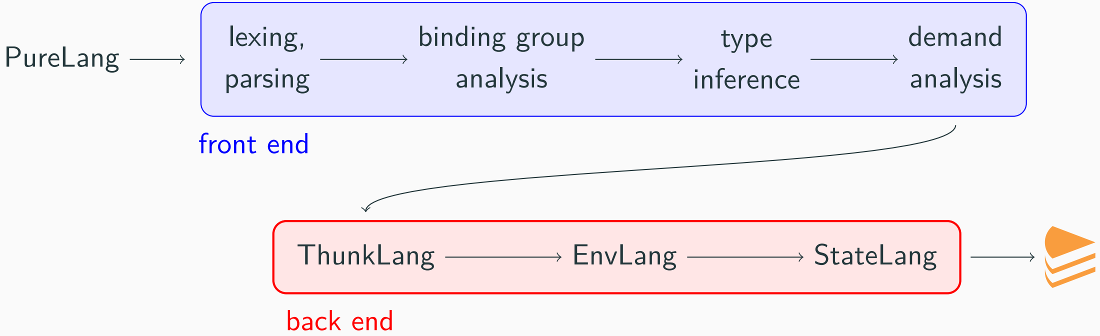
Source language

Compiler front end

Compiler back end

Connection with CakeML

Compiler structure



PureLang → lexing, parsing → binding group analysis → type inference → demand analysis

front end

ThunkLang → EnvLang → StateLang →

back end

Methodology — implementation vs. verification

Prior work: (such as CakeML)

- Define implementation function: $\text{transform} : e \to e$
- Verify: $\text{wf } e \implies [\![\, \text{transform } e \,]\!] = [\![\, e \,]\!]$

This work:

$$e \;\mathcal{R}\; e'$$

syntactic relations

$$\text{compile } ce = ce'$$

code transformation

- for **verification**
- an implementation *envelope*

- for **implementation**
- must fit in relation envelope

Methodology — workflow

1. **Define** and **verify** $\mathcal{R}$: $\quad e \; \mathcal{R} \; e' \implies [\![e]\!] = [\![e']\!]$

   Three simulation proofs: one per layer of the semantics
2. **Define** $\text{compile} : \mathit{ce} \to \mathit{ce}$
3. **Verify** $\text{wf } \mathit{ce} \implies (\text{desugar } \mathit{ce}) \;\; \mathcal{R} \;\; (\text{desugar}\,(\text{compile } \mathit{ce}))$
4. **Compose** theorems:

   $$\text{wf } \mathit{ce} \implies [\![\text{desugar } \mathit{ce}]\!] = [\![\text{desugar}\,(\text{compile } \mathit{ce})]\!]$$
5. **Integrate** into compiler, discharge wf *ce*

Separation of concerns for modularity and ease-of-verification

ThunkLang

Call-by-value semantics

Syntax: $e \ ::= \ \ldots \mid \textbf{delay}\ e \mid \textbf{force}\ e$ introduce *thunk*s

Semantics:

$$\text{eval}\ (\textbf{delay}\ e) = \textbf{thunk}\ e \qquad \frac{\begin{array}{c}\text{eval}\ e = \textbf{thunk}\ e' \\ \text{eval}\ e' = v\end{array}}{\text{eval}\ (\textbf{force}\ e) = v}$$

NB thunks are pure, value-sharing comes later

Optimisation: reduce **delay** (**force** x); two forms of restricted CSE

Verification: *seven* syntactic relations in total

EnvLang

Environment-based semantics + minor reformulations

Syntax:	essentially unchanged
Semantics:	~~substitutions~~ **environments**
Verification:	focuses on the change in semantic style

StateLang

IO monad compiled to **effectful primitives, thunks shared statefully**

Syntax: $e \ ::= \ \ldots \mid \texttt{malloc}\ n\ e \mid \ldots$

remove **delay**/**force**, **return**/**bind**/...

Semantics: *stateful* CESK machine

Compilation:

$$\textbf{alloc}\ n\ e \longmapsto \lambda\_.\ \texttt{malloc}\ n\ e$$

$$\textbf{force}\ e \longmapsto \begin{array}{l} \textbf{let}\ x = e'\ \textbf{in} \\ \textbf{if}\ x[0]\ \textbf{then}\ x[1] \\ \textbf{else}\ \ldots\ x[0] := \textbf{true};\ x[1] := v\ \ldots \end{array}$$

$[\textbf{true},\ v]$ *or* $[\textbf{false},\ \lambda\_.\ \ldots]$

Optimisation: simplify $\lambda\_.\ e$ and **unit**

Roadmap

Introduction

Source language

Compiler front end

Compiler back end

Connection with CakeML

Oracles vs. ITrees

Reconciling differing semantic styles

$$o_1 \xrightarrow{\Delta(o_1)} o_2 \xrightarrow{\Delta(o_2)} \dots$$

linear oracles: $\text{semantics}_{\Delta}\ e = tr$

$$\begin{tikzcd}
 & \text{Vis } o_1\ k_1\ \dots \\
\text{Vis } o\ k \arrow[ur, "k(a_1)"] \arrow[r] & \text{Vis } o_2\ k_2\ \dots \\
 & \vdots
\end{tikzcd}$$

branching ITrees: $[\![\, e \,]\!] = \text{Vis } \dots$

- Verified ITree semantics: $[\![\, e \,]\!]_{\text{🍰}} \overset{\Delta}{\rightsquigarrow} tr \iff \text{semantics}_{\Delta}\ e = tr$
- New compiler correctness:

$$\frac{\begin{array}{c}\text{cakeml } e = \text{Some } code \\ code\ \textbf{in memory of}\ machine\end{array}}{[\![\, machine \,]\!]_{\mathrm{M}}\ \text{prunes}\ [\![\, e \,]\!]_{\text{🍰}}}$$

Compiler correctness

$$\frac{\text{purecake } \mathit{str} = \text{Some } \mathit{ast}_{🍰}}{\begin{array}{c}\textbf{exists } \mathit{ce} \textbf{ such that} \\ \text{frontend } \mathit{str} = \text{Some } (\mathit{ce}, \_) \\ \mathit{ce} \textbf{ is type safe} \\ [\![\, \text{desugar } \mathit{ce} \,]\!]_{\text{pure}} \simeq [\![\, \mathit{ast}_{🍰} \,]\!]_{🍰}\end{array}}$$

End-to-end correctness

$$\frac{\begin{array}{c}\text{purecake } \mathit{str} = \text{Some } \mathit{ast}_{\text{🍰}} \\[1em] \text{cakeml } \mathit{ast}_{\text{🍰}} = \text{Some } \mathit{code} \\[1em] \mathit{code}\ \textbf{in memory of}\ \mathit{machine}\end{array}}{\begin{array}{c}\textbf{exists}\ \mathit{ce}\ \textbf{such that} \\[1em] \text{frontend } \mathit{str} = \text{Some } (\mathit{ce},\ \_) \\[1em] [\![\, \mathit{machine} \,]\!]_{\text{M}}\ \text{ prunes }\ [\![\, \text{desugar } \mathit{ce} \,]\!]_{\text{pure}}\end{array}}$$

A verified binary

Verified bootstrapping using *proof-producing synthesis [ICFP12]*

$$\text{HOL4 functions} \xrightarrow{\text{synthesise}} \mathit{AST}_{🍰} + \mathit{equivalence\ proof}$$

The PureCake compiler is a HOL4 function...

$$\text{purecake} \xrightarrow{\text{synthesise}} \mathit{purecake\_AST}_{🍰} + \mathit{equivalence\ proof}$$

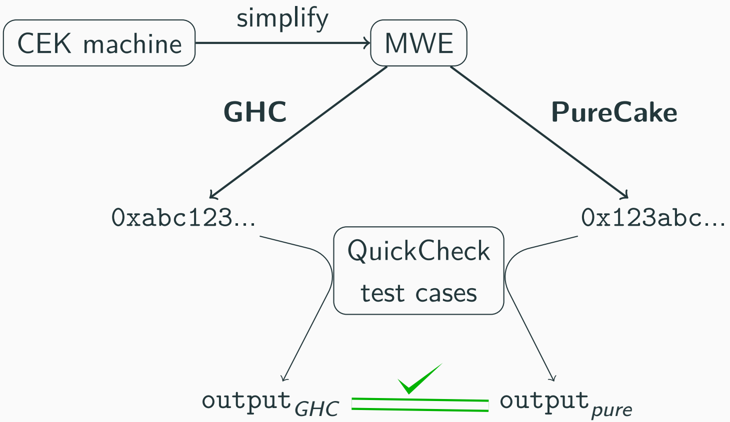
... and the CakeML compiler can be evaluated in-logic:

$$\vdash \text{cakeml}\ (\mathit{purecake\_AST}_{🍰}) = \texttt{0xabc123...}$$

Equivalence proofs transport verification to the binary

Real-world usage of PureCake

Testing on the Cardano block chain platform by QuviQ



Summary

PureCake

cakeml.org/purecake

a **verified compiler** for a Haskell-like language

- sound equational reasoning
- Haskell-like syntax
- two-phase constraint-based sound type inference
- verified demand analysis
- optimisations to handle non-strict code realistically
- end-to-end guarantees by targeting CakeML
- feasible to use on real code

Questions?

Backup slides

Future work

Only a first version! Many possible extensions, for example:

- Increasing source **expressivity** (e.g. for `case`)
- More Haskell 98 types, e.g. **typeclasses**
- More effective **demand analysis**
- Back end **optimisations**

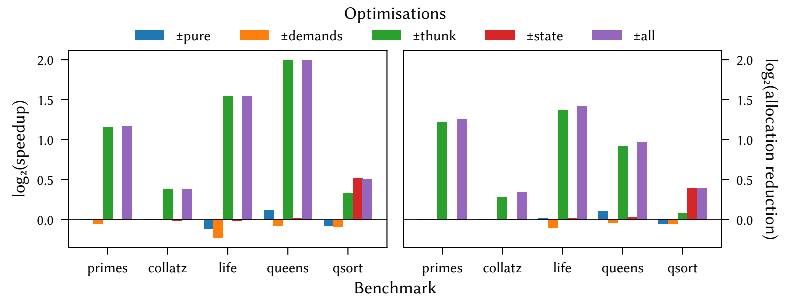
A **verified REPL** for PureCake *[Sewell et. al., PLDI23]*

Evaluation — setup

Measure **execution time** and **memory allocations**

- Turn off individual optimisations to highlight their effect
  - pure: binding group analysis
  - demands: demand analysis
  - thunk: some **force** (**delay** *e*) reduction and CSE in ThunkLang
  - state: $\lambda\_.\ e/$**unit** optimisations in StateLang
- Five benchmarks, each accepting integer *n* input
  - primes: *n*th prime calculation
  - collatz: longest Collatz sequence for a number less than *n*
  - life: Conway's Game of Life for *n* generations
  - queens: solutions for the *n*-queens problem
  - qsort: imperative quicksort for an array of length *n*

Evaluation — results



Results

- ThunkLang optimisations provide significant benefit
- StateLang optimisations improve monadic code particularly
- Binding group analysis negligible
- Regressions for demand analysis: `seq`-insertion is too eager!

Alternative compiler correctness

$$\frac{\text{frontend}\ \mathit{str} = \text{Some}\ (\mathit{ce},\ \_)}{\begin{array}{c} \mathit{ce}\ \textbf{is type safe} \\ \textbf{exists}\ \mathit{ast}_{\text{🍰}}\ \textbf{such that} \\ \text{purecake}\ \mathit{str} = \text{Some}\ \mathit{ast}_{\text{🍰}} \\ [\![\, \text{desugar}\ \mathit{ce} \,]\!]_{\text{pure}} \approx [\![\, \mathit{ast}_{\text{🍰}} \,]\!]_{\text{🍰}} \end{array}}$$