How to prove that you need Cake?

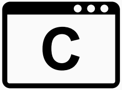
Based on PureCake A Verified Compiler for a Lazy Functional Language

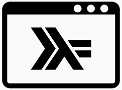
17th Novembre 2023 — Cambium Seminar

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*
Kacper Korban *Chalmers University of Technology*
Gordon Sau *University of New South Wales*

Implementing MyCriticalSoftware

Implementing MyCriticalSoftware





Implementing MyCriticalSoftware

?

Language guarantees

Implementing MyCriticalSoftware

C

?

ML

type safety
memory safety

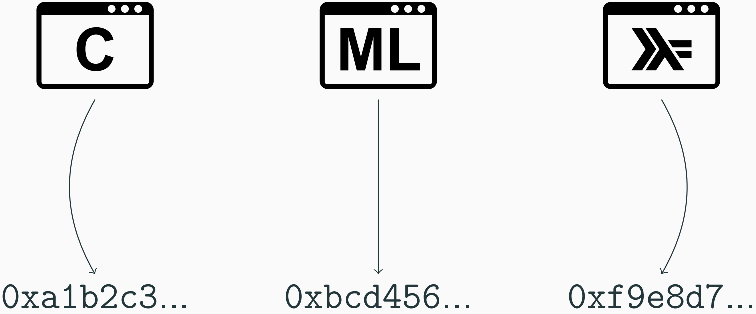
Implementing MyCriticalSoftware

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

Compiling MyCriticalSoftware

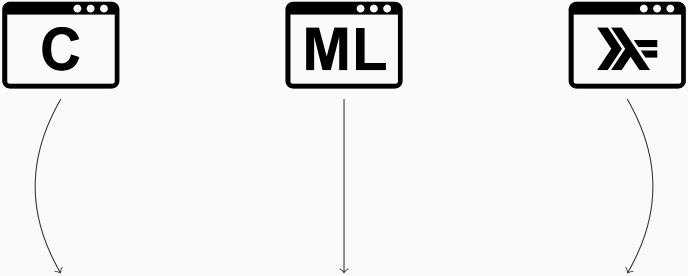
Compiler guarantees

Compiling MyCriticalSoftware



Compiler guarantees

Compiling MyCriticalSoftware

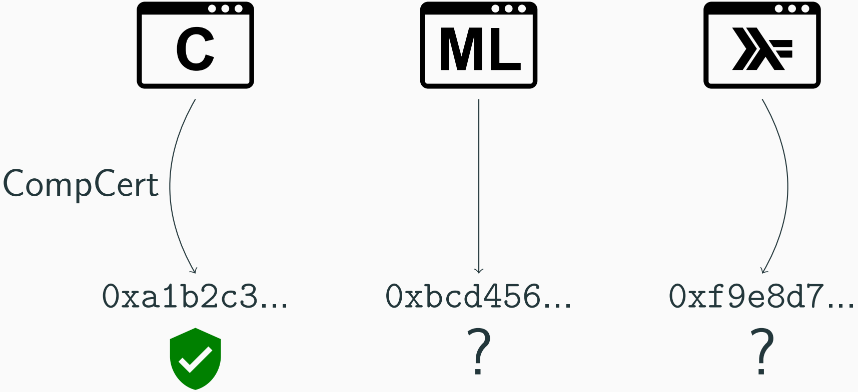


`0xa1b2c3...`   `0xbcd456...`   `0xf9e8d7...`

?   ?   ?

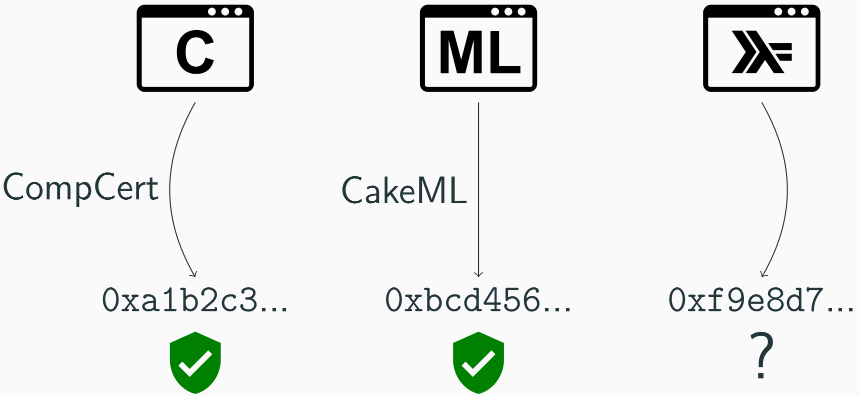
Compiler guarantees

Compiling MyCriticalSoftware



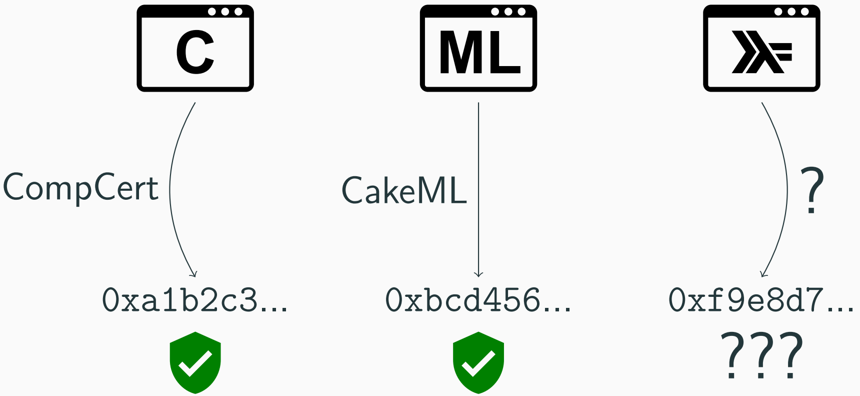
Compiler guarantees

Compiling MyCriticalSoftware



Compiler guarantees

Compiling MyCriticalSoftware



Our work

The **PureCake** project:

Our work

The **PureCake** project:

a HOL4-**verified compiler** for

Our work

The **PureCake** project:

a HOL4-**verified compiler** for

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

Our work

The **PureCake** project:

a HOL4-**verified compiler** for

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

is inspired by **Haskell** and

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

Key contributions

Highlights

- sound equational reasoning

Key contributions

Highlights

- sound equational reasoning
- parsing expression grammar (PEG) for Haskell-like syntax
- two-phase constraint-based type inference
- demand analysis

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 $\rightarrow$ imperative)
- CakeML as a back end for end-to-end verified compilation

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 $\rightarrow$ imperative)
- CakeML as a back end for end-to-end verified compilation

***Not mechanically verified before**

This talk

Global overview + demands analysis

This talk

Global overview + demands analysis

For more details:

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

Roadmap

Introduction

Source language

Compiler front end

Compiler back end

Connection with CakeML

A realistic functional language

PureLang has standard functional idioms ...

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

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

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

A realistic subset of Haskell

… and Haskell extras

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

laziness $\rightarrow$ infinite data

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

A realistic subset of Haskell

… and Haskell extras

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

laziness → infinite data

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

pure by default, monads for:

- sequencing
- stateful computations
- I/O

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

A realistic subset of Haskell

… and Haskell extras

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

laziness → infinite data

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

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

Formal syntax

A tale of two ASTs... separate implementation and verification

ce	e
compiler *expressions*	***semantic*** *expressions*

Formal syntax

A tale of two ASTs... separate implementation and verification

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

compiler *expressions* ***semantic*** *expressions*

Formal syntax

A tale of two ASTs... separate implementation and verification

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

compiler *expressions*

semantic *expressions*

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

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 — definitions

Operational semantics in layers:

Semantics — definitions

Operational semantics in layers:

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

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

Semantics — definitions

Operational semantics in layers:

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

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

2. Lift to unclocked evaluation

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

Semantics — definitions

Operational semantics in layers:

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

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

2. Lift to unclocked evaluation

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

3. Stateful interpretation of monadic operations

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

Semantics — definitions

Operational semantics in layers:

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

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

2. Lift to unclocked evaluation

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

3. Stateful interpretation of monadic operations

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

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

Mechanised equational reasoning

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

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*

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:

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: $e_1 =_\alpha e_2 \stackrel{\text{def}}{=}$ perm_vars e_1 e_2

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: $e_1 =_\alpha e_2 \stackrel{\text{def}}{=} \text{perm\_vars}\ e_1\ e_2$
- β-equivalence: $(\lambda x.\, e_1) \cdot e_2 =_\beta (\text{freshen}_{e_2}\ e_1)[{}^{e_2}/_x]$

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: $e_1 =_\alpha e_2 \stackrel{\text{def}}{=} \text{perm\_vars}\ e_1\ e_2$
- 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)

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: $e_1 =_\alpha e_2 \stackrel{\text{def}}{=} \text{perm\_vars}\ e_1\ e_2$
- 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:

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: $e_1 =_\alpha e_2 \stackrel{\text{def}}{=} \text{perm\_vars}\ e_1\ e_2$
- 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:

$$e_1 \cong e_2 \iff e_1 \sim e_2$$

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: $e_1 =_\alpha e_2 \stackrel{\text{def}}{=} \text{perm\_vars } e_1 \ e_2$
- 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

Read the paper for more details

Roadmap

Introduction

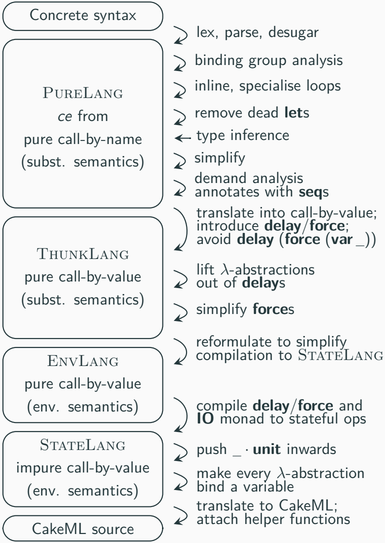
Source language

Compiler front end

Compiler back end

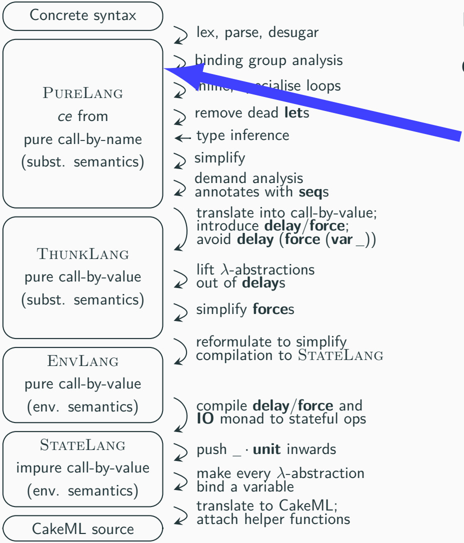
Connection with CakeML

Compiler structure



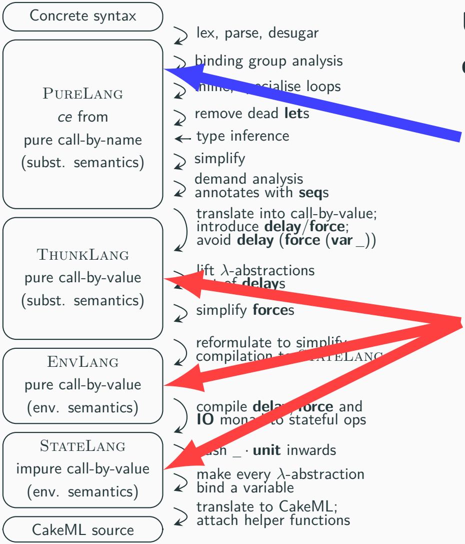
Upcoming slides examine the compiler top to bottom

Compiler structure



Upcoming slides examine the compiler top to bottom

Frontend accepts PureLang

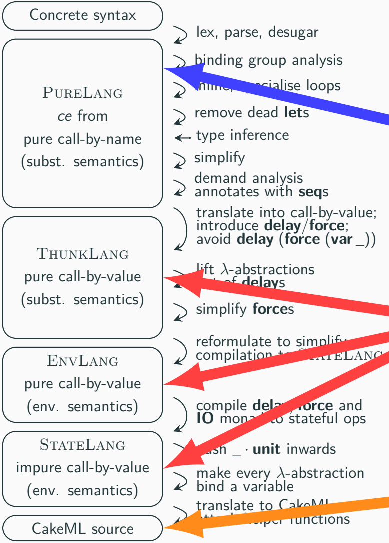


Upcoming slides examine the compiler top to bottom

Frontend accepts PureLang

Three intermediate languages, each with a specific focus

Compiler structure



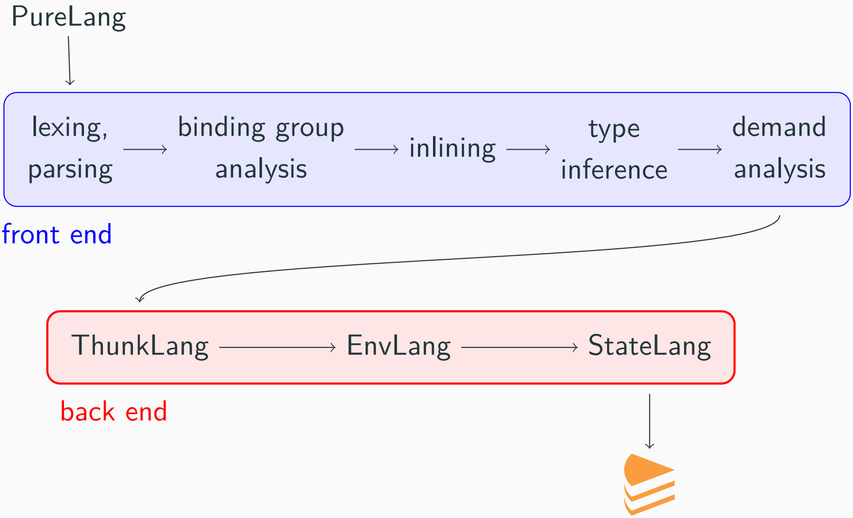
Upcoming slides examine the compiler top to bottom

Frontend accepts PureLang

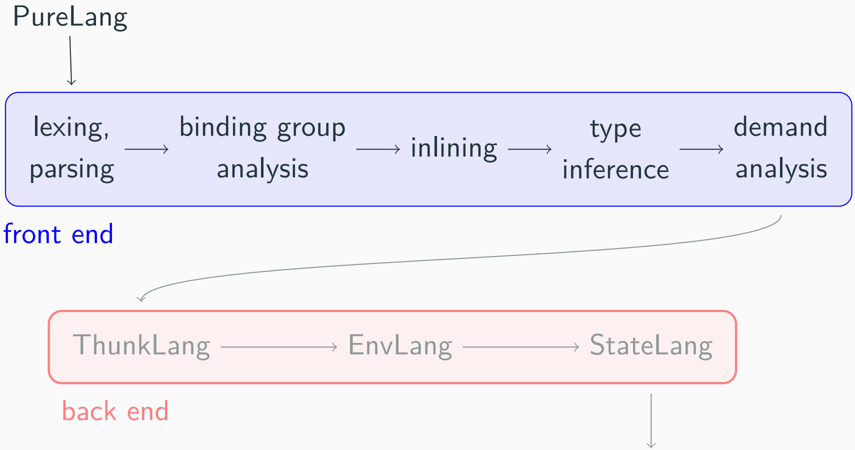
Three intermediate languages, each with a specific focus

Targets CakeML

Compiler structure



Compiler structure



Parsing

Indentation-sensitive parsing expression grammar (PEG)

Parsing

Indentation-sensitive parsing expression grammar (PEG)

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

Binding group analysis

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

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
```

Binding group analysis

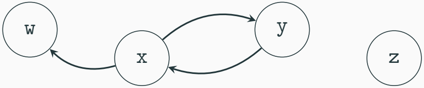
Parsing

```
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



Binding group analysis

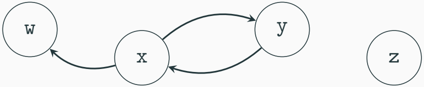
Parsing

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

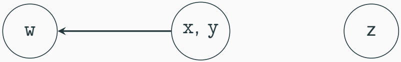
⟶

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

Analyse dependencies



Pseudo-topological sort



Binding group analysis

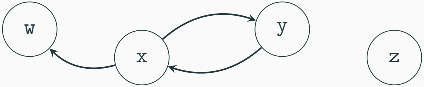
Parsing

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

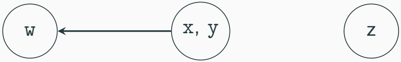
⟶

```
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
```

Binding group analysis

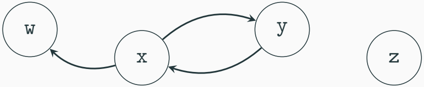
Parsing

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

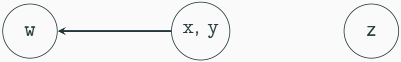
⟶

```
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

Methodology — implementation vs. verification

Methodology — implementation vs. verification

Prior work: (such as CakeML)

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

Methodology — implementation vs. verification

Prior work: (such as CakeML)

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

This work:

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

syntactic relations

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

code transformation

Methodology — implementation vs. verification

Prior work: (such as CakeML)

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

This work:

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

syntactic relations

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

code transformation

- express core transformations
- easy to underspecify and make domain assumptions

Methodology — implementation vs. verification

Prior work: (such as CakeML)

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

This work:

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

syntactic relations

- express core transformations
- easy to underspecify and make domain assumptions

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

code transformation

- must fit in relation envelope: $ce \ \mathcal{R}' \ (\text{compile } ce)$ for **all** ce
- must satisfy bookkeeping

Methodology — workflow

Methodology — workflow

1. **Define** and **verify** $\mathcal{R}$: $e \mathrel{\mathcal{R}} e' \implies [\![e]\!] = [\![e']\!]$
   Three simulation proofs: one per layer of the semantics

Methodology — workflow

1. **Define** and **verify** $\mathcal{R}$: $e \; \mathcal{R} \; e' \implies [\![\, e \,]\!] = [\![\, e' \,]\!]$
   Three simulation proofs: one per layer of the semantics
2. **Define** compile : $ce \to ce$

Methodology — workflow

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

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

Methodology — workflow

1. **Define** and **verify** $\mathcal{R}$: $e \mathrel{\mathcal{R}} e' \implies [\![e]\!] = [\![e']\!]$
   Three simulation proofs: one per layer of the semantics
2. **Define** compile : $ce \to ce$
3. **Verify** wf $ce \implies (\text{desugar } ce) \ \mathcal{R} \ (\text{desugar} (\text{compile } ce))$
4. **Compose** theorems:
   $$\text{wf } ce \implies [\![\text{desugar } ce]\!] = [\![\text{desugar} (\text{compile } ce)]\!]$$

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** compile : $ce \rightarrow ce$
3. **Verify** wf $ce \implies$ (desugar ce) $\mathcal{R}$ (desugar (compile ce))
4. **Compose** theorems:
   wf $ce \implies [\![\, \text{desugar } ce \,]\!] = [\![\, \text{desugar (compile } ce) \,]\!]$
5. **Integrate** into compiler, discharge wf ce

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** compile : $ce \to ce$
3. **Verify** wf $ce \implies (\text{desugar}\ ce)\ \ \mathcal{R}\ \ (\text{desugar}\,(\text{compile}\ ce))$
4. **Compose** theorems:
   $$\text{wf}\ ce \implies [\![\, \text{desugar}\ ce \,]\!] = [\![\, \text{desugar}\,(\text{compile}\ ce) \,]\!]$$
5. **Integrate** into compiler, discharge wf ce

Separation of concerns for modularity and ease-of-verification

Demand analysis

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

```
fact acc n =          if n = 0
                      then acc
                      else fact (acc * n) (n - 1)
```

Demand analysis

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

```
fact acc n = seq acc $ if n = 0
                       then acc
                       else fact (acc * n) (n - 1)
```

Demand analysis

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

```
fact acc n = seq acc $ if n = 0
                        then acc
                        else fact (acc * n) (n - 1)
```

- e **demands** $\overline{x_n}$ $\stackrel{\text{def}}{=}$ $e \cong (x_1$ `` `seq` `` $\ldots$ x_n `` `seq` `` $e)$

Demand analysis

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

```
fact acc n = seq acc $ if n = 0
                       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)

Demand analysis

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

```
fact acc n = seq acc $ if n = 0
                       then acc
                       else fact (acc * n) (n - 1)
```

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

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                               then acc
                               else fact (acc * n) (n - 1)
```

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                            then acc
                            else fact (acc * n) (n - 1)
```

- x **demands** $v \implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                               then acc
                               else fact (acc * n) (n - 1)
```

- x **demands** $v \implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v

```
if (Seq v x) then y else z ≅ Seq v (if x then y else z)
```

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                                 then acc
                                 else fact (acc * n) (n - 1)
```

- x **demands** $v \implies (\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z)$ **demands** v
- y **demands** $v \quad \wedge \quad z$ **demands** v
  $\implies (\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z)$ **demands** v

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                               then acc
                               else fact (acc * n) (n - 1)
```

- x **demands** $v \implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v
- y **demands** $v \ \wedge \ z$ **demands** v
  $\implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v

```
if x then (Seq v y) else (Seq v z)
      ≅ Seq v (if x then y else z)
```

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                               then acc
                               else fact (acc * n) (n - 1)
```

- x **demands** $v \implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v
- y **demands** $v \;\wedge\; z$ **demands** v
  $\implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v

```
if x then (Seq v y) else (Seq v z)
        ≅ Seq v (if x then y else z)
```

We have two distinct values for $\bot$: `Err` and `Div`

Demand analysis – definition

```
fact acc n = seq n $ seq acc $ if n = 0
                               then acc
                               else fact (acc * n) (n - 1)
```

- x **demands** $v \implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v
- y **demands** $v \;\wedge\; z$ **demands** v
  $\implies (\texttt{if } x \texttt{ then } y \texttt{ else } z)$ **demands** v
- Fixpoint analysis for recursive functions

Roadmap

Introduction

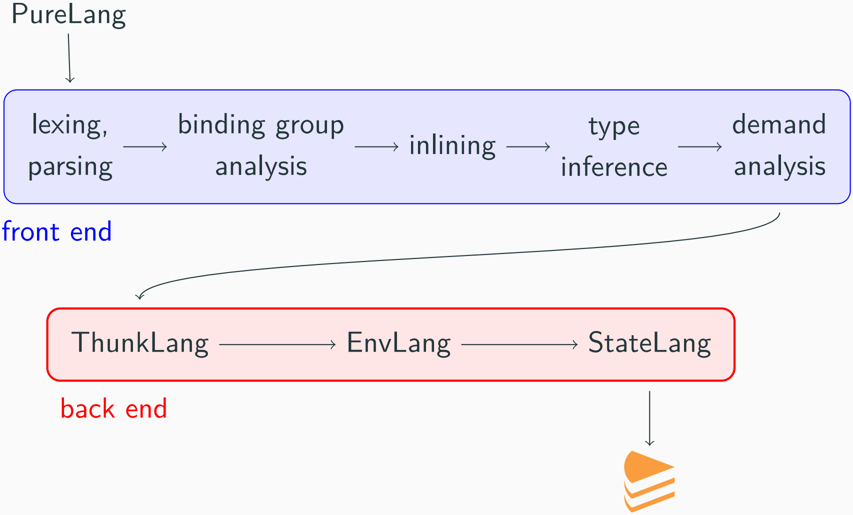
Source language

Compiler front end

Compiler back end

Connection with CakeML

Compiler structure



Compiler structure

PureLang

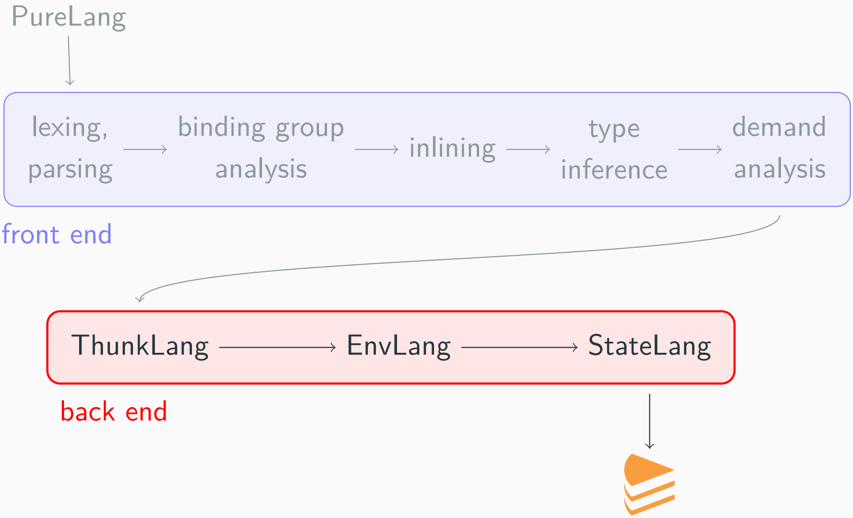
lexing, parsing → binding group analysis → inlining → type inference → demand analysis

front end

ThunkLang → EnvLang → StateLang

back end

Compiler structure



ThunkLang

Call-by-value semantics

ThunkLang

Call-by-value semantics

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

ThunkLang

Call-by-value semantics

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

Semantics:

$$\text{eval}\ (\textbf{delay}\ e) = \textbf{thunk}\ e$$

$$\frac{\text{eval}\ e = \textbf{thunk}\ e' \qquad \text{eval}\ e' = v}{\text{eval}\ (\textbf{force}\ e) = v}$$

ThunkLang

Call-by-value semantics

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

Semantics:

$$\text{eval } (\textbf{delay } e) = \textbf{thunk } e \qquad \frac{\text{eval } e = \textbf{thunk } e' \quad \text{eval } e' = v}{\text{eval } (\textbf{force } e) = v}$$

NB thunks are pure, value-sharing comes later

ThunkLang

Call-by-value semantics

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

Semantics:

$$\text{eval}\ (\textbf{delay}\ e) = \textbf{thunk}\ e \qquad \frac{\text{eval}\ e = \textbf{thunk}\ e' \quad \text{eval}\ e' = v}{\text{eval}\ (\textbf{force}\ e) = v}$$

NB thunks are pure, value-sharing comes later

Optimisation: reduce **force** (**delay** e); two forms of restricted CSE; lift lambdas out of **delay**

ThunkLang

Call-by-value semantics

Syntax: $e \ ::= \ \dots \mid \textbf{delay}\ e \mid \textbf{force}\ e$ introduce *thunks*

Semantics:

$$\text{eval}\ (\textbf{delay}\ e) = \textbf{thunk}\ e \qquad \frac{\text{eval}\ e = \textbf{thunk}\ e' \quad \text{eval}\ e' = v}{\text{eval}\ (\textbf{force}\ e) = v}$$

NB thunks are pure, value-sharing comes later

Optimisation: reduce **force** (**delay** e); two forms of restricted CSE; lift lambdas out of **delay**

Verification: *seven* syntactic relations in total

Pure to Thunk

```
fact acc n = seq n $ seq acc $ if n = 0
                               then acc
                               else fact (acc * n) (n - 1)
```

Pure to Thunk

```
fact acc n = seq n $ seq acc $ if n = 0
                              then acc
                              else fact (acc * n) (n - 1)

fact = Delay (\acc  n. Force n $ Force acc $
      if Force n = 0
          then Force acc
          else (Force fact) (Delay (Force acc * Force n))
                            (Delay (Force n - 1)))
```

EnvLang

Environment-based semantics + minor reformulations

EnvLang

Environment-based semantics + minor reformulations

Syntax: essentially unchanged

EnvLang

Environment-based semantics + minor reformulations

Syntax: essentially unchanged

Semantics: ~~substitutions~~ **environments**

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 **stateful and I/O primitives, thunks shared statefully**

StateLang

IO monad compiled to **stateful and I/O primitives, thunks shared statefully**

Syntax: $e \ ::= \ \ldots \mid \textbf{alloc}\ e \mid \ \ldots$ remove **return**/**bind**/...

StateLang

IO monad compiled to **stateful and I/O primitives, thunks shared statefully**

Syntax: $e ::= \ \dots \mid \textbf{alloc}\ e \mid \dots$ remove **return**/**bind**/...

Semantics: *stateful* CESK machine

StateLang

IO monad compiled to **stateful and I/O primitives, thunks shared statefully**

Syntax: $e \ ::= \ \ldots \mid \textbf{alloc}\ e \mid \ \ldots$ remove **return**/**bind**/...

Semantics: *stateful* CESK machine

Compilation: $\textbf{return}\ e \ \longmapsto\ \textbf{let}\ x = e'\ \textbf{in}\ \lambda\_.\ x$

StateLang

IO monad compiled to **stateful and I/O primitives, thunks shared statefully**

Syntax: $e \ ::= \ \ldots \mid \textbf{alloc}\ e \mid \ldots$ remove **return**/**bind**/...

Semantics: *stateful* CESK machine

Compilation:

$$\textbf{return}\ e \longmapsto \textbf{let}\ x = e'\ \textbf{in}\ \lambda\_.\ x$$

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

[**true**, v] or [**false**, $\lambda\_.\ e$]

StateLang

IO monad compiled to **stateful and I/O primitives, thunks shared statefully**

Syntax: $e \ ::= \ \ldots \mid \textbf{alloc}\ e \mid \ldots$ remove **return**/**bind**/...

Semantics: *stateful* CESK machine

Compilation:

$\textbf{return}\ e \longmapsto \textbf{let}\ x = e' \ \textbf{in}\ \lambda\_.\ x$

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

[**true**, v] or [**false**, $\lambda\_.\ e$]

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

Oracles vs. ITrees

Reconciling differing semantic styles

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

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

Oracles vs. ITrees

Reconciling differing semantic styles

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

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

$$\text{Vis}\ o\ k \begin{cases} \xrightarrow{k(a_1)} \text{Vis}\ o_1\ k_1\ \dots \\ \longrightarrow \text{Vis}\ o_2\ k_2\ \dots \\ \vdots \end{cases}$$

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

Oracles vs. ITrees

Reconciling differing semantic styles

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

$$\text{Vis } o\ k \begin{cases} \xrightarrow{k(a_1)} \text{Vis } o_1\ k_1\ \ldots \\ \longrightarrow \text{Vis } o_2\ k_2\ \ldots \\ \vdots \end{cases}$$

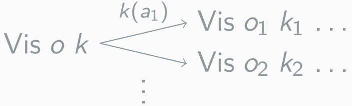
linear oracles: $\text{semantics}_\Delta\ e = tr$ **branching** ITrees: $[\![\, e \,]\!] = \text{Vis}\ \ldots$

- Verified ITree semantics: $[\![\, e \,]\!]$

Oracles vs. ITrees

Reconciling differing semantic styles

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



linear oracles: $\text{semantics}_{\Delta}\ e = tr$ **branching** ITrees: $[\![e]\!] = \text{Vis} \ \ldots$

- Verified ITree semantics: $[\![e]\!] \overset{\Delta}{\rightsquigarrow} tr$

Oracles vs. ITrees

Reconciling differing semantic styles

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

$$\text{Vis } o\ k \begin{cases} \xrightarrow{k(a_1)} \text{Vis } o_1\ k_1\ \ldots \\ \longrightarrow \text{Vis } o_2\ k_2\ \ldots \\ \quad \vdots \end{cases}$$

linear oracles: $\text{semantics}_\Delta\ e = tr$ **branching** ITrees: $[\![\, e \,]\!] = \text{Vis}\ \ldots$

- Verified ITree semantics: $[\![\, e \,]\!] \overset{\Delta}{\leadsto} tr \;\Leftrightarrow\; \text{semantics}_\Delta\ e = tr$

Reconciling differing semantic styles

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

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

$$\text{Vis}\ o\ k \xrightarrow{k(a_1)} \text{Vis}\ o_1\ k_1\ \ldots \qquad \text{Vis}\ o\ k \longrightarrow \text{Vis}\ o_2\ k_2\ \ldots \qquad \vdots$$

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

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

Oracles vs. ITrees

Reconciling differing semantic styles

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

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

$$\text{Vis}\ o\ k \begin{cases} \xrightarrow{k(a_1)} \text{Vis}\ o_1\ k_1\ \dots \\ \longrightarrow \text{Vis}\ o_2\ k_2\ \dots \\ \vdots \end{cases}$$

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

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

$$\frac{\text{cakeml}\ e = \text{Some}\ code}{}$$

Oracles vs. ITrees

Reconciling differing semantic styles

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

$$\text{Vis } o\ k \begin{cases} \xrightarrow{k(a_1)} \text{Vis } o_1\ k_1\ \ldots \\ \rightarrow \text{Vis } o_2\ k_2\ \ldots \\ \vdots \end{cases}$$

linear oracles: $\text{semantics}_\Delta\ e = tr$ **branching** ITrees: $[\![\, e \,]\!] = \text{Vis}\ \ldots$

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

$$\frac{\text{cakeml } e = \text{Some } \textit{code} \qquad \textit{code} \textbf{ in memory of } \textit{machine}}{}$$

Oracles vs. ITrees

Reconciling differing semantic styles

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

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

$$\text{Vis}\ o\ k \xrightarrow{k(a_1)} \text{Vis}\ o_1\ k_1\ \dots \qquad \text{Vis}\ o\ k \rightarrow \text{Vis}\ o_2\ k_2\ \dots \qquad \vdots$$

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

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

$$\frac{\text{cakeml}\ e = \text{Some}\ \mathit{code} \qquad \mathit{code}\ \textbf{in memory of}\ \mathit{machine}}{[\![\, \mathit{machine} \,]\!]_{\text{M}}\ \text{prunes}\ [\![\, e \,]\!]}$$

Compiler correctness

Compiler correctness

$$\frac{\text{purecake } \mathit{str} = \text{Some } \mathit{ast}}{}$$

Compiler correctness

$$\frac{\text{purecake } \mathit{str} = \text{Some } \mathit{ast}}{\textbf{exists } \mathit{ce} \textbf{ such that}}$$

Compiler correctness

$$\frac{\text{purecake } \mathit{str} = \text{Some } \mathit{ast}}{\textbf{exists } \mathit{ce} \textbf{ such that}}$$

$$\text{frontend } \mathit{str} = \text{Some } (\mathit{ce},\ \_)$$

Compiler correctness

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

Compiler correctness

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

End-to-end correctness

End-to-end correctness

$$\text{purecake } \mathit{str} = \text{Some } \mathit{ast}$$

End-to-end correctness

$$\text{purecake } \mathit{str} = \text{Some } \mathit{ast}_{🍰}$$

$$\text{cakeml } \mathit{ast}_{🍰} = \text{Some } \mathit{code}$$

End-to-end correctness

$$\text{purecake } \mathit{str} = \text{Some } \mathit{ast}$$

$$\text{cakeml } \mathit{ast} = \text{Some } \mathit{code}$$

$$\mathit{code} \textbf{ in memory of } \mathit{machine}$$

End-to-end correctness

purecake *str* = Some *ast*

cakeml *ast* = Some *code*

code **in memory of** *machine*

---

exists *ce* **such that**

$$\frac{\begin{array}{c}\text{purecake } \mathit{str} = \text{Some } \mathit{ast} \\ \text{cakeml } \mathit{ast} = \text{Some } \mathit{code} \\ \mathit{code} \textbf{ in memory of } \mathit{machine}\end{array}}{\begin{array}{c}\textbf{exists } \mathit{ce} \textbf{ such that} \\ \text{frontend } \mathit{str} = \text{Some } (\mathit{ce}, \_)\end{array}}$$

End-to-end correctness

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

PureCake

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

PureCake

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

- sound equational reasoning

Summary

PureCake

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

- sound equational reasoning
- Haskell-like syntax

Summary

PureCake

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

- sound equational reasoning
- Haskell-like syntax
- two-phase constraint-based sound type inference

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

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

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

Summary

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

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**

Evaluation — setup

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

- Turn off individual optimisations to highlight their effect

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

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: nth 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

Alternative compiler correctness

$$\frac{\text{frontend } str = \text{Some } (ce, \_)}{}$$

Alternative compiler correctness

$$\frac{\text{frontend } \mathit{str} = \text{Some } (\mathit{ce},\ \_)}{\mathit{ce} \textbf{ is type safe}}$$

Alternative compiler correctness

$$\frac{\text{frontend } \mathit{str} = \text{Some } (\mathit{ce}, \_)}{}$$

ce **is type safe**

exists *ast* **such that**

Alternative compiler correctness

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

Alternative compiler correctness

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