Viva voce introductory presentation

Verified compilation of a purely functional language
to a realistic machine semantics

Hrutvik Kanabar, University of Kent

Wednesday 13th September

Roadmap

Introduction

A purely functional language

- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- Semantics preservation
- Compiler correctness

Contributions

Two contributions in general purpose, end-to-end verified compilation

PureCake

an end-to-end verified compiler for a purely functional, Haskell-like language

Realistic machine semantics

compiler correctness theorems backed by an official instruction set specification

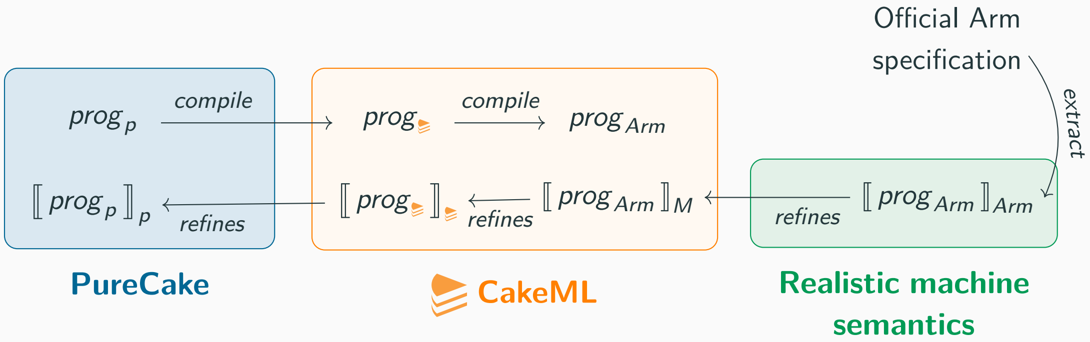
Connected by CakeML

(an end-to-end verified implementation of a subset of ML)

Built using HOL4

Building on CakeML

From a purely functional language to a realistic machine semantics



Roadmap

Introduction

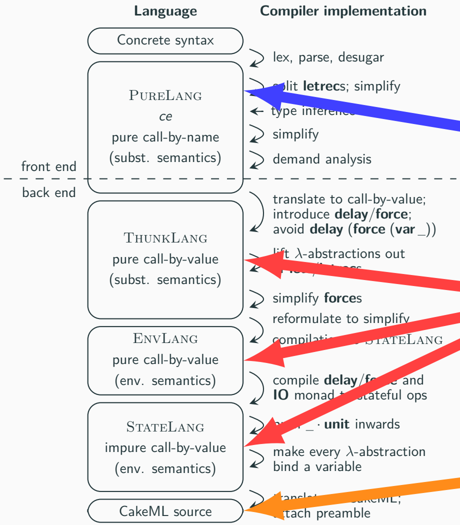
A purely functional language

- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- Semantics preservation
- Compiler correctness

Structure of PureCake



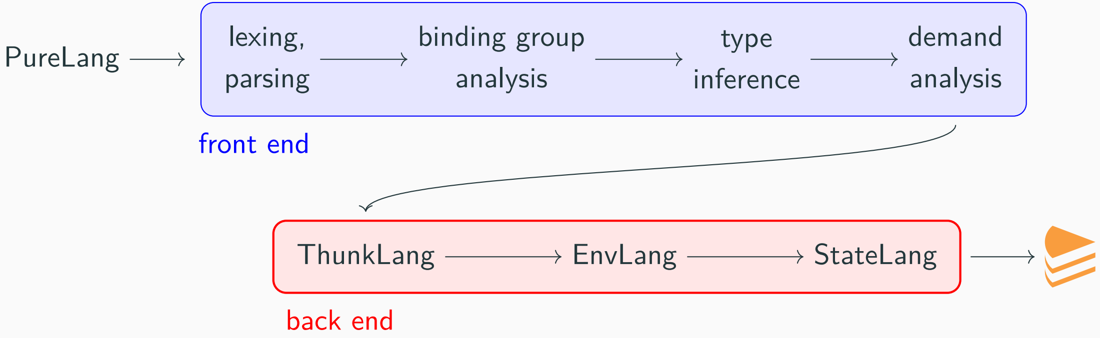
Next slide: simplified diagram

Frontend accepts PureLang

Three intermediate languages, each with a specific focus

Targets CakeML

Simplified structure of PureCake



Roadmap

Features inspired by Haskell

```haskell
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

semantic *expressions*

- lower level
- used for verification: ground truth for semantics

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 using classical quantification

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

3. Stateful interpretation of `IO` operations

$$(\!|\, wh,\ \kappa,\ \sigma \,|\!) : (\varepsilon,\ \alpha,\ \rho)\ \mathrm{itree}$$

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

Mechanised equational reasoning

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

Proved *congruent* via Howe's method *[Howe, 1996]*, i.e.

$$\textit{bisimilar sub-expressions} \implies \textit{bisimilarity}$$

Definitions:

α-equivalence	β-equivalence	contextual equivalence
$e_1 =_\alpha e_2$	$(\lambda x.\, e_1) \cdot e_2 =_\beta e_1'[{}^{e_2}/_x]$	$e_1 \sim e_2$

Derived results:

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

Type system

Standard Hindley-Milner rules... with an unusual soundness proof

"preservation" (subject reduction) does not hold

Proof strategy:

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

Roadmap

Introduction

A purely functional language

- Source language
- **Compiler front end**
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- Semantics preservation
- Compiler correctness

Indentation-sensitive parsing expression grammar (PEG):

PEG + *[Adams POPL13]*

$$N \rightarrow X_1^{\mathcal{R}_1}\, X_2^{\mathcal{R}_2}\, \ldots\, X_n^{\mathcal{R}_n}$$

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

Parsing algorithm verified to terminate on all inputs

Binding group analysis

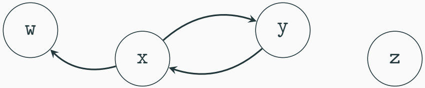
Parsing

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

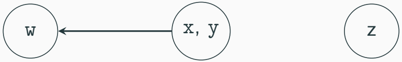
⟶

```
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 $\longrightarrow$ 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:

infer *ce* **succeeds**

$\implies$ $ce \vdash_{\textsc{Top}}$ cs *and* cs **solveable**

$\implies$ $\Gamma \vdash ce : \tau$

Demand analysis

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

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

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

stuck $\approx$ diverged $\approx \perp$ **but** well-typed $\implies \approx, \cong$ coincide
`seq`-prefixing preserves typing

Roadmap

Introduction

A purely functional language

- Source language
- Compiler front end
- **Compiler back end**
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- Semantics preservation
- Compiler correctness

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

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

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

code transformation

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

Methodology — workflow

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

Intermediate languages

ThunkLang

introduce pure thunks

- compile to call-by-value
- remove harmful code patterns
- optimisation around thunks

EnvLang

introduce environments

- semantics uses environments
- prove correctness of reformulation

StateLang

introduce state

- CESK semantics
- compile `IO` operations
- share thunk values statefully
- optimise artefacts

Roadmap

Introduction

A purely functional language
- Source language
- Compiler front end
- Compiler back end
- **Connection with CakeML**

A realistic machine semantics
- Arm ISA specifications
- Semantics preservation
- Compiler correctness

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:

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

End-to-end correctness

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

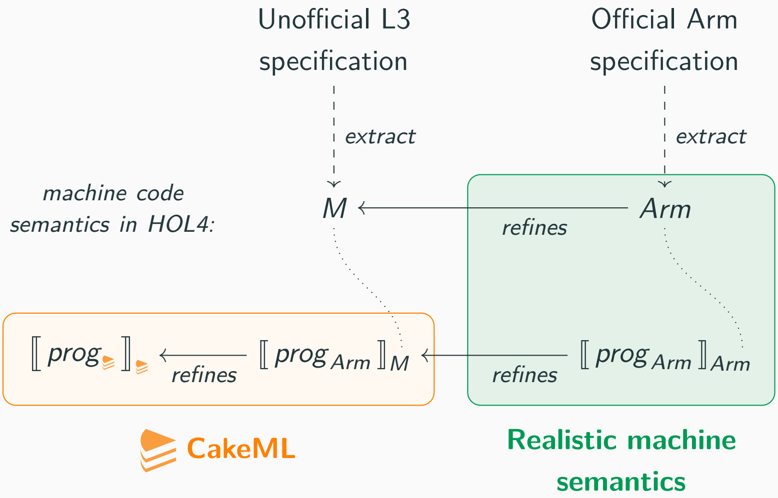
Roadmap

Introduction

A purely functional language
- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics
- Arm ISA specifications
- Semantics preservation
- Compiler correctness

Realistic machine semantics for CakeML



Roadmap

Introduction

A purely functional language

- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- **Arm ISA specifications**
- Semantics preservation
- Compiler correctness

ISA specification languages

DSLs to specify the ISA abstraction

i.e. an envelope of permitted processor implementations

- Machine-readable: parsing, type-checking, simulation, modelling, verification, . . .
- First order, imperative languages with common features:
  - static typing + type inference
  - strong bit vector support
  - scattered functions

DSLs for Arm specifications

ASL

for Arm documentation

- originally Arm-internal pseudocode
- source of (near-)truth
- **official** public releases

Sail

one size fits all

- developed in academia
- ASL front end, many back ends
- well-exercised

L3

state of art for ITP

- developed in academia
- HOL4 and Isabelle/HOL back ends
- designed for ITP + well-exercised

Extracting an official specification to HOL4

$$\text{ASL} \xrightarrow[\texttt{asl\_to\_sail}]{} \text{Sail} \xrightarrow[\texttt{sail -lem}]{} \text{Lem} \xrightarrow[\texttt{lem -hol}]{} \text{HOL4}$$

Result: Armv8.6 in HOL4

imperative declarations		monadic definitions
bitvector built-ins		derived operations
primitive operations	*become*	implemented operations
liquid dependent types		simple polymorphic types
scattered functions		monolithic definitions

Roadmap

Introduction

A purely functional language

- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- **Semantics preservation**
- Compiler correctness

Modifying the specification

Adapting to proofs of semantics preservation

Monad

avoid set-based non-determinism in favour of Hilbert choice

Address translation

stub out as an identity, update physical addresses (52-bit $\mapsto$ 64-bit)

modify hand-written libraries

rely on Sail

Trust???

Specification size

Character counts:

53 k	4,200 k
Armv8 in L3	Armv8 in ASL
70 k	12,200 k
Armv8 in HOL4 via L3	Armv8 in HOL4 via Sail

Specification complexity

Long extraction pathway produces non-idiomatic specification

$$\text{ASL} \xrightarrow[\texttt{asl\_to\_sail}]{} \text{Sail} \xrightarrow[\texttt{sail -lem}]{} \text{Lem} \xrightarrow[\texttt{lem -hol}]{} \text{HOL4}$$

Difficult to inspect or interact with:

monadic bloat	large monolithic definitions
non-idiomatic operations	poor in-logic evaluation
lack of instruction AST	translation artefacts

Taming the specification

Interactively abstract to theorem-prover-friendly L3

Per *opcode*, prove a simulation:

$$
\begin{array}{ccc}
\mathit{l3} & \xrightarrow{\text{Run (Decode } \mathit{opcode}\text{)}} & \mathit{l3}' \\
\text{state\_rel} \ \Big\| & & \Big\| \ \text{state\_rel} \\
\mathit{asl} & \xrightarrow[\text{ExecA64 } \mathit{opcode}]{} & \mathit{asl}'
\end{array}
$$

with some fixed system register bits in asl, asl'

Once and for all proof, not tied to CakeML

Instructions verified

Instruction class	Assembly shorthands
move wide operations	MOVK, MOVN, MOVZ
bit field moves	BFM, SBFM, UBFM
logical operations*†	AND[S], BIC[S], EON, EOR, ORN, ORR
addition/subtraction*†	ADD[S], SUB[S]
addition/subtraction with carry	ADC[S], SBC[S]
division	SDIV, UDIV
multiply with addition/subtraction	MADD, MSUB
multiply high	SMULH, UMULH
conditional compare*	CCMN, CCMP
conditional select	CSEL, CSINC, CSINV, CSNEG
branch immediate (call/jump)	B, BL
conditional branches	B.COND
branch register (jump)	BR
register extract	EXTR
address calculation	ADR, ADRP
byte/register loads/stores*‡	LD[U]R, LD[U]R[S]B, ST[U]R, ST[U]RB

*For immediate operands. †For shifted register operands.

‡Scaled 12-bit unsigned immediate offset and unscaled 9-bit signed immediate offset addressing modes.

Roadmap

Introduction

A purely functional language

- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- Semantics preservation
- Compiler correctness

Context: CakeML's support for multiple targets

Design choices: *target-agnostic* and *realistic*

- LabLang/Asm: generic assembly-like intermediate language
- Target-agnostic semantics which models execution environment
  - Instruction execution steps: next $trgt_n = trgt_{n+1}$
  - *Interference* between successive instructions: $\iota_1, \iota_2, \ldots$
  - Projection π of processor state always preserved

i.e. $trgt_0 \xrightarrow{\text{next}} trgt_1 \xrightarrow{\iota_1} trgt'_1 \longrightarrow \ldots$ where $\pi\ trgt'_1 = \pi\ trgt_1$

Compiler correctness factored to a core proof obligation

$$
\begin{array}{ccccccccccccc}
 & asm_1 & & & & \xrightarrow{\hspace{4em}\text{next\_asm } instr\hspace{4em}} & & & & & asm_2 & \\
\text{target\_state\_rel} & \| & & & & & & & & & \| & \text{target\_state\_rel} \\
 & trgt_0 & \xrightarrow{\text{next}} & trgt_1 & \xrightarrow{\iota_1} & trgt'_1 & \xrightarrow{\text{next}} & \cdots & \xrightarrow{\text{next}} & trgt_n & \xrightarrow{\iota_n} & trgt'_n
\end{array}
$$

Proof strategy

- Replay existing proof for L3 specification
- Step through ASL states alongside by applying simulation result
- Carefully manage interference to preserve invariants

Roadmap

Introduction

A purely functional language

- Source language
- Compiler front end
- Compiler back end
- Connection with CakeML

A realistic machine semantics

- Arm ISA specifications
- Semantics preservation
- Compiler correctness