Thesis for the degree of Doctor of Philosophy in Computer Science

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

Hrutvik Kanabar

School of Computing, University of Kent

Canterbury, United Kingdom

August 2023

Abstract

Formal verification of a compiler offers the ultimate understanding of the behaviour of compiled code: a mathematical proof relates the semantics of each output program to that of its corresponding input. Users can rely on the same formally-specified understanding of source-level behaviour as the compiler, so any reasoning about source code applies equally to the machine code which is actually executed. Critically, these guarantees demand faith only in a minimal trusted computing base (TCB). To date, only two general-purpose, end-to-end verified compilers exist: CompCert and CakeML, which compile a C-like and an ML-like language respectively.

In this dissertation, I advance the state of the art in general-purpose, end-to-end compiler verification in two ways. First, I present PureCake, the first such verified compiler for a purely functional, Haskell-like language. Second, I derive the first compiler correctness theorem backed by a realistic machine semantics, that is, an official specification for the Armv8 instruction set architecture.

Both advancements build on CakeML. PureCake extends CakeML's guarantees outwards, using it as an unmodified building block to demonstrate that we can reuse verified compilers as we do unverified ones. The key difference is that reuse of a verified compiler must consider not only its external implementation interface, but also its proof interface: its top-level theorems and TCB. Conversely, a realistic machine semantics for Armv8 strengthens the root of CakeML's trust, reducing its TCB. Now, both CakeML and the hardware it targets share a common understanding of Armv8 behaviour which is derived from the same official sources.

Composing these two advancements fulfils the title of this dissertation: PureCake has an end-to-end correctness theorem which spans from a purely functional, Haskell-like language to a realistic, official machine semantics.

Contents

Chapter 1

Introduction

Why verify a compiler? Transforming human-readable source code to a machine-readable binary is complex, and *miscompilations* can introduce unexpected behaviour—bugs. A complete mathematical proof of compiler correctness guarantees that the behaviour of source and binary are identical, so any bugs encountered when executing the binary must also be found in the source. Such a proof is too costly for everyday software in which source-level bugs may be common, making the rate of miscompilation negligible. However, hard-won source-level assurances about safety-critical software must not be thrown away by use of a buggy compiler. For high-assurance software, the cost is worth it: all software compiled by a verified compiler benefits from its guarantees, and any source-level software verification is transported down to the code executed by hardware.

But how do we trust a proof of compiler correctness? It is no less complex than the compiler itself, so we distil it down to its *trusted computing base* (TCB): some parts of the proof such that trust in those parts means we can trust the whole. Indeed, unverified compilers are well-known to contain bugs throughout, but bugs discovered in verified compilers have been confined to their TCBs [Yang et al. 2011]. The goal then, is to minimise the TCB and so minimise the need for trust.

Unfortunately, realistic software is so complex that we cannot trust human-written proofs about it. Instead, we can use a interactive theorem prover, a program which aids human-directed reasoning by carefully checking proofs. If we trust the theorem prover, we can trust the proofs it checks. So, our goal should be to verify as much as possible within a trusted theorem prover. In other words, we should pursue *end-to-end* verification: a single theorem which spans the entire software system and discharges any assumptions on the interactions between its various components.

What is the right end-to-end result for a compiler? It must equate the behaviour of input source code with that of an output binary. But the behaviour of a binary is tied to the hardware on which it runs, and modern mainstream hardware is highly

complex. Verified compilers which target mainstream hardware must use simplified specifications of unknown fidelity, sacrificing trust in their results. There are only two such *general-purpose*, end-to-end verified compilers to date: CompCert and CakeML, which accept a C-like and an ML-like language respectively.

1.1 Contributions

I present two advances in general-purpose, end-to-end compiler verification.

- *The first general-purpose, end-to-end verified compiler for a purely functional, call-by-name language.* Known as PureCake, its input language is modelled on a subset of Haskell, while its compilation targets CakeML to produce correct machine code for several mainstream architectures. Composition with unmodified CakeML considers only its externally-presented interfaces: its compiler correctness theorem and TCB. PureCake therefore inherits this TCB at a fraction of the verification effort, and demonstrates reusability of CakeML.
- *The first formal connection between an authoritative mainstream instruction set specification and a compiler correctness theorem.* I use an official Armv8 instruction set specification to maximise trust in the compilation of CakeML (and therefore PureCake) to Armv8 hardware, reducing its TCB by removing a prior unvalidated Armv8 specification. To do so, I introduce a technique for taming industrial-strength instruction set specifications for repeated use in verification: once and for all in-prover abstraction.

All proofs are machine-checked by the HOL4 interactive theorem prover.

1.1.1 Publications

All contributions have been published in peer-reviewed conferences: each of the two parts of this dissertation corresponds to a paper. Both papers are the result of collaboration with the listed authors, but this dissertation is my own work. Below, I will indicate material based on work completed by my collaborators, but which I present here for completeness. All work is open-source, and the links below provide a detailed commit history of contributions.

Part I

> H. Kanabar, S. Vivien, O. Abrahamsson, M. O. Myreen, M. Norrish, J. Åman Pohjola, and R. Zanetti. PureCake: A verified compiler for a lazy functional language. In *Programming Language Design and Implementation (PLDI)*. ACM, 2023. DOI 10.1145/3591259

This part and its associated paper describe the PureCake[1] project, an ongoing collaboration under the CakeML[2] umbrella. I wrote most of the paper, and I have worked on almost all parts of the project in some capacity. I am grateful to Magnus Myreen for originating and overseeing the project. Material described in §§ 3.1 and 3.4 was completed almost exclusively by others. My collaborators were responsible for most of the material in §§ 4.2 and 4.3, notably excepting compilation of monadic operations; the semantics and compilation relation of § 4.4 were initially my work but evolved over the project. All of the material in §§ 2.1 to 2.4 was a collective effort between a subset of authors, of which I was a primary contributor.

Open-source. The PureCake repository can be found at `https://github.com/CakeML/pure`. Significant contributions to the HOL4 and CakeML repositories are listed below.

- Material in § 5.2, integrated into CakeML:
  `https://github.com/CakeML/cakeml/pull/834`,
  `https://github.com/CakeML/cakeml/pull/866`,
  `https://github.com/CakeML/cakeml/pull/870`.
- Material in § 3.2, integrated into HOL4:
  `https://github.com/HOL-Theorem-Prover/HOL/pull/889`.

Part II

> H. Kanabar, A. C. J. Fox, and M. O. Myreen. Taming an authoritative Armv8 ISA specification: L3 validation and CakeML compiler verification. In *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI `10.4230/LIPIcs.ITP.2022.20`

I carried out almost all proofs, and wrote the entire paper. I am grateful to Anthony Fox and Magnus Myreen for originating the project while I was an intern at Arm Limited, and for advising me through its execution and write-up.

Open-source. All material described in this part has been integrated into the HOL4 and CakeML repositories. The majority of the contributions are listed below.

- Material in §§ 7.3 and 8.1:
  `https://github.com/HOL-Theorem-Prover/armv8.6-asl-snapshot`.
- Material in § 8: `https://github.com/HOL-Theorem-Prover/HOL/pull/981`.
- Material in § 9: `https://github.com/CakeML/cakeml/pull/858`.

[1] `https://cakeml.org/purecake`
[2] `https://cakeml.org`

1.2 Background

The contributions above build directly on the verified compiler CakeML, which relies on the interactive theorem prover HOL4. Here, I describe some key parts of these projects which are necessary to understand the remainder of this dissertation. Readers closely familiar with HOL4 and CakeML can safely skip this section.

First, I describe HOL4 and its in-logic evaluation capabilities (§ 1.2.1), which we will later rely on to handle complex industrial instruction set specifications in-prover (§ 8.4), to bootstrap the CakeML compiler (§ 1.2.2), and to produce a verified PureCake binary (§ 5.3.1). Second, I showcase CakeML's style of specifying language semantics, its top-level compiler correctness theorem, and its bootstrapping to produce a verified binary (§ 1.2.2). Naturally CakeML's source semantics plays a part in its compiler correctness theorem, which will be extended by PureCake (§ 5.3) and instantiated with an official semantics for Armv8 (§ 9.2). However, this style is so convenient in interactive proof that we will also use it for PureCake's language semantics (§ 2.3). Finally, CakeML's verified bootstrapping provides a blueprint to verifiably compile PureCake (§ 5.3.1).

Aside from these descriptions, I defer discussion of CakeML's architecture-specific compiler correctness proofs to § 9.1, where it becomes most relevant.

1.2.1 HOL4

HOL4[3] is an interactive theorem prover: a program which helps a human to prove theorems in some logic. Interactive theorem provers implement a *logic* (sometimes called an object language) and expose a *meta-language*. The meta-language is effectively a user interface to the logic: a way to programmatically construct and manipulate logical terms and theorems. Critically, any useful theorem prover must permit construction only of valid theorems. In other words, all proofs are systematically *machine-checked* by the prover. Machine-checked proofs provide the only practical way to derive trustworthy theorems about realistic, complex software.

HOL4's logic is a "higher-order logic" (HOL), specifically Church's simple theory of types [Church 1940]. In what is known as the "LCF tradition/style", HOL4 is implemented using a strongly typed *meta-language*, in this case Standard ML.[4] Its LCF-style *kernel* then defines a data type of theorems alongside primitive inference rules which can construct the theorem data type. Outside of the kernel, this data type is held abstract: the meta-language's strong typing therefore ensures theorems are constructed only by the primitive inference rules. This minimises its TCB, and therefore the TCBs of any HOL4-verified projects: all complicated proof rules and automation are implemented

[3] `https://hol-theorem-prover.org/`

[4] The ML language family derives from the **m**eta-**l**anguage of Edinburgh/LCF [Gordon et al. 1979].

outside of HOL4's kernel, and require no additional trust in the validity of its theorems. HOL4 can further generate artifacts which permit independent checking of its usage of the primitive inference rules.

HOL4 is a mature prover, but this dissertation needs only basic features of its logic and one feature implemented using its meta-language: in-logic evaluation, discussed below. Unless otherwise stated, all formulae in this dissertation map fairly directly to terms of HOL4's logic. Logical operators are standard ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Longleftrightarrow$); true/false are denoted T/F. Binders ($\forall$, $\exists$, λ) extend as far to the right as possible. I distinguish *variables* and constants by font; constants are fully-defined logical terms and variables can be instantiated to any logical term. Rules consist of premises above a conclusion, separated by a horizontal line; these may characterise an inductive relation or simplify presentation of an implication depending on context. Every term has a type, which can be denoted *term* : *type*. Type variables use Greek letters; HOL4 data types are presented using standard Backus-Naur form (::=); polymorphic types are prefix except for function types (*i.e.*, α list, β option, $\gamma \rightarrow \delta$). Theorems are denoted by a turnstile ($\vdash$); any free variables in a theorem can be considered universally quantified.

All theorems in this dissertation are proven using HOL4, and I will omit details of most proofs unless I consider them to be of particular interest. Further information on HOL4 can be found in prior work [Slind and Norrish 2008].

Non-constructivity

HOL4's logic is classical: the law of excluded middle holds. For any proposition P, either it or its negation must be true: $\vdash \forall P.\ P \vee \neg P$. This permits non-constructive proofs. Another source of non-constructivity is a version of Hilbert's choice operator, $\varepsilon : (\alpha \rightarrow \mathsf{bool}) \rightarrow \alpha$, which we will use in § 8.1. This is usually written as a binder, *i.e.*, $\varepsilon x.\ P\ x$ denotes $\varepsilon\ (\lambda x.\ P\ x)$. The term $\varepsilon(x : \alpha).\ P\ x$ is an element x of the type α satisfying $P\ x$ if one exists, and some arbitrary element of type α otherwise. The latter is possible because all HOL4 types are inhabited. In other words, the choice operator has the following characterising theorem:

$$\vdash P\,(\varepsilon x.\ P\ x) \Longleftrightarrow \exists x.\ P\ x$$

In-logic evaluation

The goal of in-logic evaluation is to "compute" terms in HOL4's logic. That is, the automated reduction of a term to a suitable normal form using a call-by-value evaluation strategy. This reduction must be verified: it must produce a theorem equating the original and reduced forms.

HOL4 has no primitive notion of in-logic evaluation, unlike the type theories implemented by theorem provers such as Coq, Agda, and Lean. Instead, in-logic evaluation is provided by a meta-language tool which I will refer to as `EVAL` [Barras 2000]. Given a HOL4 term, it will attempt to reduce that term bottom-up using β-conversion and characteristic equations of any constants. For example:

$$\texttt{EVAL}\ (\mathsf{length}\ [1,\ 2,\ 3])\quad \textit{produces the theorem}\quad \vdash \mathsf{length}\ [1,\ 2,\ 3] = 3$$

That is, application of `EVAL` to a term which denotes the length of the list [1, 2, 3] produces a theorem equating the term to 3 by using the following equations for length:

$$\vdash \mathsf{length}\ [] = 0 \qquad \vdash \mathsf{length}\ (x :: xs) = 1 + \mathsf{length}\ xs$$

`EVAL` can be seen as a highly automated and efficient form of rewriting. It is also highly customisable, allowing users to: choose exactly which equations it uses; control its strictness; and provide custom Standard ML functions to manipulate specified term patterns. These custom functions are used to handle *e.g.* arithmetic division efficiently.

1.2.2 CakeML

CakeML[5] is an end-to-end verified compiler which accepts a language resembling a subset of Standard ML and targets several mainstream architectures [Fox et al. 2017; Kumar et al. 2014]. In other words, its source language syntax, semantics, and compiler algorithm are formally specified in the logic of HOL4. The compiler algorithm is verified to preserve semantics: the behaviour of a compiled target program is the same as that of the input source program.

This is a simplified picture of a significant, mature ecosystem [Myreen 2021b]. For example: CakeML's compiler back end is rich in optimisations and generates code which performs comparably to code produced by industrial-strength compilers [Tan et al. 2019]; its source language permits reasoning based on separation logic [Guéneau et al. 2017]; and a verified read-evaluate-print loop (REPL) permits proof of soundness of an implementation of HOL Light [Abrahamsson et al. 2022; Sewell et al. 2022]. However, PureCake uses CakeML as a black box compiler, and our work on official instruction set specifications only reduces CakeML's TCB. So we need not concern ourselves with details *within* the CakeML project: we can focus only on its externally-presented interfaces, and trust that it optimises input code effectively. A verified compiler exposes two such interfaces: its implementation entry points (how do we invoke the compiler?) and its proof footprint (what does the compiler guarantee, what is its TCB?) In other words, we

[5]Note that "CakeML" can refer to one of three things depending on context: the overall project/ecosystem, a source language, and the verified compiler for that language.

need only consider: CakeML's source language semantics; and its compiler correctness results (both the general version discussed below and architecture-specific ones in § 9.1). I will briefly describe these below, with the addition of verified bootstrapping: a process by which we can obtain a binary which correctly implements a verified compiler.

Functional big-step semantics

CakeML specifies semantics for all of its languages (source, intermediate, and target) in the functional big-step style [Owens et al. 2016]. That is, each language is equipped with a recursive interpreter (evaluate) which is *clocked* (or *fuelled*) to ensure totality: parametrised by a natural number which is decremented on recursive calls that do not reduce the input expression size. When the clock runs out, evaluate times out.

A simple example below shows the action of $\mathsf{evaluate}_\lambda$, a functional big-step interpreter for a toy untyped λ-calculus, on function application $(e_1 \cdot e_2)$. I show $\mathsf{evaluate}_\lambda$ rather than the evaluate function for either CakeML's or PureCake's source language for simplicity. Superscripts j, k, and $l - 1$ denote the input clock parameter. Here, Haskell-like do-notation embeds $\mathsf{evaluate}_\lambda$ within an exception monad which supports two kinds of exception: Timeout to denote running out of fuel, and Error to denote a *runtime type error* (when a program gets "stuck" or "goes wrong", *e.g.*, by attempting to evaluate 1 + true).

```
1  evaluate^j_λ η (e1 · e2) =def do
2    (v, k) ← evaluate^j_λ η e2;
3    (closure_η' x. e, l) ← evaluate^k_λ η e1;
4    if l = 0 then (Timeout, l) else evaluate^(l−1)_λ η'[x ↦ v] e
```

The design choices here follow those made by CakeML: semantics is untyped (or extrinsic, Curry-style); evaluation is call-by-value; environments η provide values for free variables (instead of substitutions); the clock j is "threaded through" the semantics (*i.e.*, evaluation of e_2 may consume some of clock j to produce clock $k \leq j$, reducing the fuel available for evaluation of e_1). From here on, all semantics will be untyped, but in § 2.3 the functional big-step style is also applied to a call-by-name, substitution-based semantics with a different clock-passing strategy.

Overall, this example evaluates argument e_2 to a value v (line 2) and e_1 to a function value $\mathsf{closure}_{\eta'}\ x.\ e$ (line 3), before evaluating function body e in an environment which extends the closure environment η' by binding parameter name x to v (line 4). Timeouts and runtime type errors are propagated by standard monadic operations; failed pattern matches (*e.g.*, assignment to $\mathsf{closure}_{\eta'}\ x.\ e$ on line 3) produce Error. The final recursive call decrements the clock to ensure totality (line 4), all other recursive calls are structurally recursive. Running out of clock produces Timeout (line 4).

To derive a top-level semantics, we classically quantify over $\mathsf{evaluate}_\lambda$'s clock:

$$\mathsf{semantics}_\lambda\ e \stackrel{\text{def}}{=} \begin{cases} \mathsf{Terminate}\ res & \exists j.\ \mathsf{evaluate}^j_\lambda\ \varnothing\ e = (res,\ \_)\ \wedge\ res \neq \mathsf{Timeout} \\ \mathsf{Diverge} & \forall j.\ \mathsf{evaluate}^j_\lambda\ \varnothing\ e = (\mathsf{Timeout},\ \_) \end{cases}$$

Here, $\varnothing$ denotes the empty environment. An expression terminates if $\mathsf{evaluate}_\lambda$ produces a value or runtime type error for some clock. Otherwise, all clocks result in Timeout, so overall the expression diverges. This definition relies on *monotonicity* of $\mathsf{evaluate}_\lambda$: if $\mathsf{evaluate}^j_\lambda$ produces a value or runtime type error, then $\mathsf{evaluate}^{j'}_\lambda$ must produce the same result for all $j' > j$.

The approach generalises straightforwardly to handle non-determinism: both $\mathsf{evaluate}_\lambda$ and $\mathsf{semantics}_\lambda$ are parametrised by an oracle Δ which supplies outcomes for non-deterministic choices. For example, for I/O the oracle models the surrounding execution environment by supplying user/environment responses to any program output. Consider a toy construct input, which prints a string (*prompt*) to ask for user input and returns the user's response (*response*).

$$\begin{array}{l} \mathsf{evaluate}^j_\lambda\ \Delta\ \eta\ (\mathsf{input}\ e) \stackrel{\text{def}}{=} \mathsf{do} \\ \quad (\mathsf{String}\ prompt,\ k,\ \Delta',\ io) \leftarrow \mathsf{evaluate}^j_\lambda\ \Delta\ \eta\ e; \\ \quad \mathsf{let}\ (\Delta'',\ response) = \Delta'\ (\mathsf{Input}\ prompt); \\ \quad \big(\mathsf{String}\ response,\ k,\ \Delta'',\ io +\!\!+ [(\mathsf{Input}\ prompt,\ response)]\big) \end{array}$$

The input expression e must evaluate to the *prompt* string, and invoking the oracle Δ on *prompt* gives the user *response*, which is then returned. There are some subtleties hidden here. Oracles evolve through successive invocations so that future outcomes may depend on previous ones; they are threaded through $\mathsf{evaluate}_\lambda$ alongside the clock. To capture all *observable* program behaviour, $\mathsf{evaluate}_\lambda$ now maintains an I/O trace: a growing log of the program's interactions with the surrounding execution environment (*i.e.*, the oracle), where each interaction contains an observable program output paired with the environment's response (dictated by the oracle). The constructor Input belongs to a sum type which has one constructor per type of non-determinism modelled by the oracle. CakeML semantics considers only I/O, so we reuse the Input constructor in the I/O trace. All CakeML I/O is via its foreign function interface (FFI), which I will describe further in § 5.1; in summary, observable program outputs are FFI calls (function name and argument), and environment responses are return values of FFI functions. Note that oracles model the environment extensionally, saying nothing about what the Input constructor actually means: it is simply recorded as an *uninterpreted effect* in the I/O trace. Here we have assumed that the environment always responds successfully, but CakeML's I/O oracle also models two environment failure modes: when the foreign

function encounters an error, or fails to respond at all (*i.e.*, diverges).

Last, in CakeML the clock, oracle, I/O trace, and a mutable store are bundled into a single record, but I have presented them separately here to remain explicit.

Top-level semantics must now consider both termination behaviour and I/O traces. The I/O trace for a diverging program is the (potentially infinite) upper bound of all finite I/O traces derivable using $\mathsf{evaluate}_\lambda$. To understand all possible program behaviours, we must consider all possible oracles (*i.e.*, all possible execution environments).

$$\mathsf{semantics}_\lambda\ \Delta\ e \stackrel{\text{def}}{=} \begin{cases} \mathsf{Terminate}\ \ res\ \ io & \exists j.\ \mathsf{evaluate}^j_\lambda\ \Delta\ \varnothing\ e = (res,\ \_,\ \_,\ io) \wedge \\ & \quad res \neq \mathsf{Timeout} \\ \mathsf{Diverge}\ io & \left(\forall j.\ \mathsf{evaluate}^j_\lambda\ \Delta\ \varnothing\ e = (\mathsf{Timeout},\ \ldots)\right) \wedge \\ & io = \bigsqcup_j \left\{ io' \;\middle|\; \mathsf{evaluate}^j_\lambda\ \Delta\ \varnothing\ e = (\ldots,\ io') \right\} \end{cases}$$

Owens et al. [2016] describe the full advantages of the functional big-step style in detail. In brief: it avoids duplication in semantics by specifying termination, errors, and divergence all at once; it is straightforwardly total and deterministic (for a given oracle, provided monotonicity holds); its functional nature simplifies proofs by rewriting; and it produces highly readable semantics.

CakeML compiler correctness

I focus on CakeML's back end compiler correctness theorem rather than its end-to-end correctness theorem; it omits parsing and type inference, which are not relevant for this dissertation. The back end theorem is phrased in terms of two functional big-step specifications: CakeML source semantics ($\mathsf{semantics}_{\gg}$) and a generic machine semantics ($\mathsf{semantics}_{\mathrm{M}}$) which can be instantiated to any one of the targets supported by CakeML. For now, we will consider only the shape of this theorem, deferring a more in-depth description to § 9.1.

Theorem 1.1. CakeML compiler correctness.

$$\begin{aligned} \vdash\ & \mathsf{target\_configs\_ok}\ config\ machine\ \wedge\ \mathsf{semantics}_{\gg}\ \Delta\ prog \neq \mathsf{Terminate\ Error}\ \_\ \wedge \\ & \mathsf{compile}_{\gg}\ config\ prog\ =\ \mathsf{Some}\ code\ \wedge\ \mathsf{code\_in\_memory}\ config\ code\ machine \\ & \Rightarrow\ \mathsf{semantics}_{\mathrm{M}}\ \Delta\ machine \in \mathsf{extend\_with\_oom}\ (\mathsf{semantics}_{\gg}\ \Delta\ prog) \end{aligned}$$

Given some well-formed configurations ($\mathsf{target\_configs\_ok}$) and a program (*prog*) whose source semantics produces no runtime type errors, the compiled *code* can be installed in the memory ($\mathsf{code\_in\_memory}$) of a machine with a semantics which is either identical to that of the source, or runs out of memory. I will define $\mathsf{target\_configs\_ok}$ and

its input configurations in § 9.1. The "lack of runtime type error" assumption is common for optimising compilers: ill-formed programs can be optimised arbitrarily.

Below, extend_with_oom specifies permitted out-of-memory errors, which can be produced by machine semantics (which has finite memory) but not by CakeML's source semantics. In particular, extend_with_oom requires machines which run out of memory to produce a prefix ($\preccurlyeq$) of the observable I/O trace of the source.

$$\mathsf{extend\_with\_oom}\ (\mathsf{Terminate}\ r\ io) \stackrel{\mathsf{def}}{=} \{\mathsf{Terminate}\ r\ io\} \cup \{\mathsf{OOM}\ io' \mid io' \preccurlyeq io\}$$

$$\mathsf{extend\_with\_oom}\ (\mathsf{Diverge}\ io) \stackrel{\mathsf{def}}{=} \{\mathsf{Diverge}\ io\} \cup \{\mathsf{OOM}\ io' \mid io' \preccurlyeq io\}$$

Verified bootstrapping

CakeML uses verified bootstrapping (or proof-grounded bootstrapping) to produce a binary which is proven to implement its compiler faithfully [Myreen 2021a]. The technique relies on both *proof-producing synthesis* of CakeML AST and HOL4's in-logic evaluation (§ 1.2.1).

Proof-producing synthesis is a proof automation tool written in Standard ML, *i.e.*, living in HOL4's meta-language rather than its logic. Given computable HOL4 functions, it synthesises CakeML code alongside a HOL4 proof that the code implements the input HOL4 faithfully [Myreen and Owens 2014]. Overall, this transports verification of functions in HOL4's logic to verified code. Here, "computable" functions are those written in the subset of HOL4 which corresponds to an ML-like programming language. The notion of "faithful implementation" is expressed using *refinement invariants*: relations that equate the behaviour of CakeML AST (according to the CakeML semantics) with HOL4 terms. Precise details of the synthesis are not necessary for this dissertation, but Myreen and Owens [2014] explain the process in full. In brief, synthesis traverses the input HOL4 term syntax bottom-up, generating CakeML AST and associated refinement invariant proofs per subexpression. *Refinement combinators* combine refinement invariants to compose per-subexpression proofs. Recursive functions require some care, as their proofs must be inductive. Synthesis can also generate imperative and exception-handling CakeML code from HOL4 functions which are written in a state/exception monad [Abrahamsson et al. 2020]. Note that as a tool external to HOL4's logic, synthesis can fail to produce any CakeML AST—however, it cannot produce invalid theorems, and so cannot produce an invalid AST.

The CakeML compiler is such a computable function ($\mathsf{compile}_{\gg}$). Therefore, proof-producing synthesis can generate corresponding CakeML AST ($ast_{\gg}$) which faithfully implements the compiler. Next, we can use in-logic evaluation to compile the compiler (*i.e.*, bootstrap it): we apply the HOL4-specified compiler to the synthesised AST and evaluate it to produce binary code (*code*). We can express this workflow informally:

1. Write and verify the CakeML compiler in HOL4:
   $\mathsf{compile}_{\gg}$ + theorem 1.1
2. Synthesise CakeML AST from the compiler:
   $\mathsf{compile}_{\gg} \xrightarrow{\text{synthesise}} ast_{\gg}$ + $\left(\vdash ast_{\gg} \text{ implements } \mathsf{compile}_{\gg}\right)$
3. Evaluate compilation of the CakeML AST in-logic:
   `EVAL` $(\mathsf{compile}_{\gg}\ ast_{\gg})$ *produces the theorem* $\vdash \mathsf{compile}_{\gg}\ ast_{\gg} = \mathsf{Some}\ code$

Critically, each stage produces a theorem within HOL4. Composing all theorems in-prover produces a top-level theorem which asserts correctness of the binary code.

1.3 Related work: verified compilation

In this section, I review prior work in verified compilation. The work I describe here is quite general; I will discuss work more specific to PureCake and realistic machine semantics in later sections (§§ 6.2 and 10.2).

1.3.1 CompCert

CompCert [Leroy 2009] first showed that end-to-end verification was feasible for general-purpose optimising compilers. It correctly compiles a subset of C99 (known as CompCert C) to Arm, RISC-V, x86, and PowerPC machine code. Other than CakeML, it remains the only general-purpose, end-to-end verified compiler, and is verified using the Coq interactive theorem prover. Performance of CompCert-generated code is comparable to that of the GNU Compiler Collection using moderate optimisation settings (`-O1`).

Like CakeML's observable behaviour (§ 1.2.2), CompCert C's observable behaviour encapsulates both termination behaviour and an I/O trace, which consists of calls to C standard library functions (akin to CakeML's FFI) and reads/writes to global volatiles, which can model memory-mapped I/O in C. However, CompCert must account for C-like non-determinism (*e.g.*, due to evaluation order) and undefined behaviour. Its correctness theorem therefore asserts that generated machine code implements an *improvement* of *one* of the source code behaviours: the compiler can choose between non-deterministic behaviours and optimise expressions with undefined behaviour arbitrarily, even turning them into well-defined expressions. CakeML compiles a deterministic, total (*i.e.*, no undefined behaviour) source language and does not consider ill-defined expressions. CompCert also formalises a C-like memory model at a suitable level of abstraction to permit fine-grained reasoning and permit reuse between its various languages [Besson et al. 2015; Leroy and Blazy 2008].

CompCert's compiler uses several intermediate languages and passes. Splitting up compilation simplifies verification, and CakeML mimics this design choice: each intermediate language is tailored to its optimisations, and tractable per-pass proofs are composed to produce the overall correctness result. In CompCert, this result spans from CompCert C AST to assembly AST and further includes parsing [Jourdan et al. 2012]. However, the conversion from C concrete syntax to CompCert C AST aside from parsing (*i.e.*, preprocessing, type-checking, elaboration) and the generation of binary code from assembly AST are unverified. CakeML requires no preprocessing and its type-checking and assembling are verified, so its results span from concrete syntax to binary code. CompCert does provide the Valex tool, which can post-hoc check linked binaries against the assembly AST produced by the compiler. PureCake's guarantees resemble those of CompCert in that they begin at a source AST not concrete syntax; however, like those of CakeML they bottom out in binary code (§ 5.3).

Extensions of CompCert. A significant ecosystem of work builds on CompCert; I briefly describe some of it.

The Verified Software Toolchain (VST, Appel [2011]) formalises a program logic (a concurrent separation logic) known as Verifiable C over Clight, the first intermediate language of CompCert. Clight is a slightly simplified CompCert C (*e.g.*, it forbids side-effects within expressions) and so is often used when extending CompCert. Verifiable C is proved sound with respect to Clight semantics, so CompCert compilation transports source-level VST verification to assembly AST. Compatibility with VST necessitates a small-step semantics for CompCert, which previously used a big-step semantics for ease of compiler verification.

CertiCoq [Anand et al. 2017] and Œuf [Mullen et al. 2018] are verified compilers from the logic of Coq, Gallina, to intermediate languages in CompCert: Clight and Cminor respectively. Their motivation mirrors that of CakeML's proof-producing synthesis (§ 1.2.2), but the translations span a larger gap (Gallina to C-like language vs. HOL4 to ML-like language). They could enable verified bootstrapping of CompCert, removing its current reliance on unverified extraction of Coq to OCaml followed by unverified compilation of the resulting OCaml.

CompCert has further been extended with correct compilation of concurrent behaviours [Jiang et al. 2019; Sevcík et al. 2013]. Below, I also refer to work on separate compilation in the context of CompCert (§ 1.3.2).

1.3.2 Compositional compiler correctness

CakeML is a *whole-program* compiler: it accepts only entire programs, rather than individual modules which can be compiled and later linked. CompCert too began as a

whole-program compiler, but several extensions offer varying degrees of compositional compilation [Gu et al. 2015; Kang et al. 2016; Koenig and Shao 2021; Song et al. 2020; Stewart et al. 2015; Wang et al. 2019]. That is, correctness relates the behaviour of compiled-then-linked modules to the behaviour of their linked sources. Note that the notion of linking used in correctness statements may not correspond to any linker used in practice: a verified linker is required to avoid increasing the TCB. These extensions vary significantly in their flexibility: some can only link modules which are all compiled by CompCert, others permit interoperation with hand-written target code or even any code in a language satisfying CompCert's memory model. One extension even supports verified assembling for x86 to the standard ELF binary format [Wang et al. 2020].

Indeed, compositional compiler correctness is an imprecise term, and it is not clear that there exists a "best" version of it. Patterson and Ahmed [2019] provide a thorough overview (including the extensions to CompCert above) and propose a framework for categorising and understanding variations; I defer a full discussion to their work but briefly outline some key points.

A recurring theme is the use of *cross-language relations* to prove correctness, often step-indexed, Kripke logical relations [Ahmed 2004; Ahmed et al. 2009], in contrast to the lightweight simulations often used by whole-program compilers. These cross-language relations pair source expressions with their correctly compiled target expressions and are compositional by construction. For example, Hur and Dreyer [2011] establish a relation between an ML-like language and idealised assembly, permitting correct single-pass compilation which can link with hand-written assembly. Transitivity of these relations is necessary to compose verification of successive passes and generalise to multi-pass compilers, but it is non-trivial in the presence of stateful features. Neis et al. [2015] define transitive-by-construction parametric inter-language simulations (known as PILS) to verify the multi-pass Pilsner compiler.

A key limitation of the cross-language relations approach is that it permits linking only with target code that has behaviour equivalent to *some* source code. Multi-language approaches offer a solution [Perconti and Ahmed 2014]: instead of a single relation which spans source and target, these use a single multi-language which embeds both languages with suitable mediating interfaces, expressing interoperation "for free". The statement of compiler correctness then becomes a contextual equivalence within the multi-language which equates source and compiled target expressions. The approach of Compositional CompCert [Stewart et al. 2015] is related: it permits linking between any languages satisfying CompCert's memory model (such as CompCert's source, intermediate, and target languages) by using the memory model to derive an "interaction semantics" for each language. Each interaction semantics inhabits the same semantic domain, allowing a common notion of contextual equivalence across all languages: where multi-languages

syntactically embed all linkable languages, interaction semantics consolidates all of their semantics instead. Other than Compositional CompCert, compositional compiler correctness is currently a theoretical exploration.

1.3.3 Alternatives to ahead-of-time compilation

CompCert and CakeML are ahead-of-time compilers: source programs are converted into target programs *before* execution. Just-in-time (JIT) compilers offer an alternative which compiles to target code on-the-fly. Verifying JIT compilation requires reconciling execution of static and dynamically generated code which can also modify itself (*e.g.*, to update code pointers to generated code on-the-fly).

Myreen and Davis [2011] build a verified runtime environment for the Milawa theorem prover, relying on JIT compilation of stack-based bytecode to x86 [Myreen 2010]. Both runtime and JIT rely on a precursor of CakeML's proof-producing synthesis to produce verified x86 machine code [Myreen et al. 2009]. This first generates unverified x86 code, which is then proved correct after decompilation to sound Hoare triple specifications [Myreen and Gordon 2007; Myreen et al. 2008], a form of machine code verification.

In the CompCert ecosystem, JitK [Wang et al. 2014] verifiably implements two interpreters from the Linux kernel by compilation via Cminor. These interpreters permit integration of userspace policies written in their respective languages, BPF and INET.DIAG, into trusted kernel code for monitoring and network filtering. JitK further implements a high-level policy language which it correctly compiles to BPF. CoreJIT [Barrière et al. 2021] reuses various CompCert back end components to verify a JIT compiler which speculatively optimises code and can dynamically roll back these optimisations if its assumptions fail to hold. FM-JIT [Barrière et al. 2023] uses the CompCert back end to generate native code for a modern JIT architecture which interleaves execution of an interpreter and dynamically compiled native code. However, certain JIT operations (*e.g.*, installing bytes in memory, calling native functions, and replacing stack frames during handoffs between interpretation and native code execution) are not expressible as shallow embeddings in Coq's pure and terminating logic. Instead, they are considered unimplemented primitives and assumed to adhere to their specifications, permitting verification with respect to an abstract memory model. Code extraction to OCaml realises the primitives; if the extracted implementations adhere to the assumed specifications then the JIT performs correctly.

1.3.4 Non-functional verification

CakeML and CompCert guarantee functional correctness: the observable input-output behaviour of source code is preserved by compilation, so functional software verification applies to target code too. Non-functional verification considers other observable behaviour, such as execution time, memory usage, confidentiality, and so on.

Verification of secure applications must consider interoperation with untrusted components, reasoning about *e.g.*, confidentiality of private data or lack of information leaks via timing side-channels. *Secure compilation* must guarantee more than functional correctness then; Patrignani et al. [2019] provide a detailed overview. For confidentiality, full abstraction is a common criterion: contextual equivalence is preserved during compilation, so programs which are indistinguishable by any source context remain indistinguishable under any target context, despite the greater distinguishing power of low-level target contexts. In other words, no target-level vulnerabilities are introduced by compilation. As alluded to in § 1.3.2, contextual equivalence is integral in compositional compilation too, as compilation of individual components must be correct regardless of the contexts in which they are eventually used. In the context of CompCert, Barthe et al. [2020] show that source programs which execute in constant time (and therefore do not leak information via timing side-channels) produce constant-time target code too.

Verifying the time and space usage of programs written in high-level languages is non-trivial: the actual costs only become apparent once programs are compiled to lower-level languages. Several projects find ways to express these low-level costs in higher-level languages, so that high-level verification can reason about the resource consumption of compiled code. The CerCo project [Amadio et al. 2013] has a compiler which annotates source programs with the final time and space costs of the compiled code. Quantitative CompCert [Carbonneaux et al. 2014] augments the I/O trace of CompCert semantics with function call and return events; each trace therefore encodes a symbolic representation of stack usage, which is preserved or reduced by compilation. The compiler then calculates the actual stack frame size of each function which is used in the symbolic representation. CompCertS [Besson et al. 2017] build on a version of CompCert C with finite memory to estimate memory usage directly at the source, verifying that compilation must preserve or reduce the inferred upper bounds. Stack-Aware CompCert [Wang et al. 2019] considers a bounded stack, and verifies preservation of stack usage through compilation. Gómez-Londoño et al. [2020] instrument the semantics of a heapless intermediate language of the CakeML compiler to reason about upper bounds on stack and live heap usage in the presence of garbage collection.

In § 6.2.1, I will discuss prior work on verifying that optimisations for Haskell-like languages do not worsen complexity, and so do not compromise lazy evaluation.

Part I

A verified compiler for a purely functional language

Outlook

High-level languages often claim to provide strong guarantees for the programmer. For example: memory managed languages ensure memory safety; 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 high-level languages may use static checks to 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.

Compilation of Haskell-like languages in particular presents unique challenges. These languages implement *lazy* or *call-by-need* 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*. Compiling these languages with some semblance of realism requires efficient lazy evaluation which does not compromise equational reasoning, as well as optimisations which reduce unnecessary laziness and generate idiomatic imperative code.

In this part, I describe PureCake, the most realistic certified compiler for a Haskell-like language to date. PureCake correctly compiles its source language, PureLang, to CakeML's source language, leveraging CakeML's mature verified compiler. First, I define PureLang and derive its metatheory (§ 2), before describing the front and back ends of the PureCake compiler (§§ 3 and 4). Next, I show how PureCake composes with CakeML to produce its end-to-end guarantees (§ 5). Finally, I discuss PureCake's contributions and real-world usability, including its performance in comparison to the Glasgow Haskell Compiler (§ 6).

Chapter 2

PureLang and its metatheory

In this chapter, I describe PureCake's source language, PureLang, and its associated metatheory. PureLang's features encompass both standard functional constructs and those typically associated with Haskell (§ 2.1). Its formal syntax is specified in HOL4 using two ASTs (§ 2.2), and its untyped operational semantics defined in stages (§ 2.3) using a variant of interaction trees [Xia et al. 2020]. We have mechanised an equational theory over this semantics, proved congruent via Howe's method (Howe [1996]), and comparing favourably with α-, β-, and contextual equivalences (§ 2.4). PureLang's typing rules are standard Hindley-Milner, but require an unusual proof of type soundness (§ 2.5).

2.1 Features

The features of PureLang are carefully chosen to imitate a subset of Haskell faithfully while permitting tractable verification. They are most clearly showcased by example: figure 2.1 shows a short PureLang program which accepts integer n on the command line and prints the first n numbers of the factorial sequence. PureLang syntax is indentation-sensitive (§ 3.1) 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 (`fact` on lines 6-9), algebraic data types and pattern matching (`[]`/`h:t` on lines 16-17).

Other than its syntax, PureLang borrows several other features from Haskell. Top-level definitions 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. As in this short example, a complete PureLang program contains a definition of `main :: IO ()`.

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

fact :: Integer -> Integer -> Integer
fact acc x =
  if x < 2 then acc
  else fact (acc * x) (x - 1)

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

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
```

Figure 2.1. A small PureLang program printing a user-specified prefix of the factorial sequence. Boilerplate definitions are omitted.

Aside from the standard `return`/`bind` (which are masked by `do`-notation), the `IO` monad provides the ability to create/update/query mutable arrays, raise/handle exceptions, and perform I/O using a foreign function interface (FFI). Valid exceptions are defined as a single, extensible sum type in an ML-like style. Users invoke FFI functions by applying `Action` to a predefined FFI channel and string input; a string is returned. For example, `read_arg1` and `print` are defined as follows:

```
read_arg1 :: IO String
read_arg1 = Action (#(cline_arg) " ")
```

```
print :: String -> IO ()
print s = do
  Action (#(stdout) (s ++ "\n "))
  return ()
```

PureLang is inspired only by a *subset* of Haskell. Two notably missing features are type classes and partiality (*e.g.*, non-exhaustive pattern matches, Haskell's `undefined`). I discuss expressivity of PureLang further in § 6.1.1.

2.2 Formal syntax

Formally, we specify PureLang syntax in HOL4 using two AST data types: high-level *compiler* expressions are used in implementation, and are considered syntactic sugar for simpler *semantic* expressions, which provide ground truth for semantics. This separates concerns: compilation is conveniently expressed over high-level expressions, but semantics is straightforwardly specified using simpler primitives. For example: semantic expressions use a minimal set of variable-binding forms to simplify specification of and reasoning about semantics (*e.g.*, compiler expressions include **case**-statements but semantic expressions omit them); compiler expressions use variadic λ-abstractions/-applications to simplify the implementations of future optimisations which must be arity-aware (*e.g.*, arity analysis, inlining). Each intermediate language in

primop ::=	Primitive operations
| **message** *ch*	message construction
| **add**	integer addition
| ...	*etc.*
op ::=	Operations
| **cons** *cname*	data constructors/tuples
| **prim** *primop*	primitive operations
| **monadic** *mop*	monadic operations
lit ::=	Literals
| **int** z	integer
| **str** s	string
| **msg** *ch s*	message
| **loc** l	location
ce ::=	Compiler expressions
| **var** x	variable
| *lit*	literal
| $op[\,\overline{ce_n}\,]$	operator application
| $\lambda\,\overline{x_n}\,.\,ce$	variadic λ-abstraction
| $ce \cdot \overline{ce_n}$	variadic λ-application
| **let** $x = ce_1$ **in** ce_2	let-binding
| **letrec** $\overline{x_n = ce_n}$ **in** ce	recursive let-bindings
| **seq** $ce_1\, ce_2$	sequencing
| **case** $x = ce$ **of** $\overline{cname_n[\,\overline{x_{n\,m}}\,] \rightarrow ce_n}$	pattern match
e ::=	Semantic expressions
| **var** x	variable
| *lit*	literal
| $op[\,\overline{e_n}\,]$	operator application
| $\lambda x.\, e$	λ-abstraction
| $e_1 \cdot e_2$	λ-application
| **letrec** $\overline{x_n = e_n}$ **in** e	recursive let-bindings
| **seq** $e_1\, e_2$	sequencing
| **if** e **then** e_1 **else** e_2	if-statement
| **eq**$_?$ *cname arity e*	constructor name/arity test
| **proj**$_n$ *cname e*	constructor argument projection

Figure 2.2. PURELANG operations *op*, literals *lit*, compiler expressions *ce*, and semantic expressions *e*.

PureCake's compiler back end makes the same design choice, further enabling a modular strategy for verification of compiler optimisations (§ 4.1).

Figure 2.2 defines data types for PureLang operations *op*, literals *lit*, compiler expressions *ce*, and semantic expressions *e*. Operations and literals are shared by both compiler and semantic expressions. Primitive operations include integer/string operations, and creation of a message with input *s* for FFI channel *ch*: **prim** (**message** *ch*) [**str** *s*]. The monadic operations encompass: **return**, **bind**; exception-handling (**raise**, **handle**); mutable array operations (**alloc**, **len**, **deref**, **update**); and FFI interaction (**action**). Literals include integers, strings, messages, and locations. Users cannot directly write the last two, but they will be used to specify PureLang semantics (§ 2.3).

PureLang's compiler expressions are inspired by Core in the Glasgow Haskell Compiler (GHC) [Peyton Jones and Launchbury 1991]. Note that tuples are formalised using as **cons** *cname* with an empty *cname*. For simplicity, **monadic** *mop* are also formalised as **cons** *cname* using reserved *cname*s, but we present them separately here for clarity. Patterns in **case**-expressions consist of a constructor name (or tuple) applied to variables only. In other words, pattern matching is shallow, considering only constructor name and arity, and nested patterns are not supported. Currently this choice trades programmer convenience for more tractable verification; in future work PureCake will support nested patterns by flattening them early in its compilation pipeline (§ 6.3). The last branch of **case**-expression is optionally a catch-all (_ $\rightarrow$ *ce*', omitted from the formal syntax here for brevity), and each **case**-expression either must be exhaustive or contain such a catch-all. In other words, failed pattern matches are not permitted.

Semantic expressions are designed to be as expressive as compiler expressions, but permit simpler specification of semantics. They specify unary (instead of variadic) λ-abstraction/-application and primitive operations on constructors (instead of **case**): testing for name and arity, and indexing arguments. Though semantic expressions do not specify **let**-bindings, we will use **let** $x = e_1$ **in** e_2 as syntactic sugar for $(\lambda x.\ e_2) \cdot e_1$. I will discuss desugaring of compiler expressions to semantic expressions in § 2.3.4.

2.3 Operational semantics

The semantics of PureLang must cleanly model non-termination and I/O via FFI calls. Interaction trees [Xia et al. 2020] provide a suitable semantic domain (§ 2.3.1), but implementing them in HOL4 requires non-trivial changes (§ 2.3.2). We specify the operational semantics of PureLang semantic expressions in terms of these modified ITrees (§ 2.3.3), and the semantics of compiler expressions via desugaring (§ 2.3.4).

2.3.1 Interaction trees (ITrees)

ITrees are a coinductive data type 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. This permits straightforward specification of program behaviour such as non-termination and interaction with the surround environment.

In Coq, ITrees are the coinductive interpretation of the following grammar:

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

That is, an ITree with event type E and return type R is either: $\mathsf{Ret}\ r$, an immediate halt to produce r; $\mathsf{Tau}\ t$, a silent step which carries on as t; or $\mathsf{Vis}\ A\ e\ k$, an output e which expects a response $a : A$, before continuing as $k\ a$. Tau nodes are necessary to express silently diverging computations in Coq without violating its *guardedness condition*: co-recursive calls must be guarded by a constructor application. However, as Tau nodes contain no computational information, ITrees must be equated by weak bisimulation, *i.e.*, two ITrees which differ only by finite sequences of silent steps are considered equal.

2.3.2 Modified ITrees in HOL4

The definition above is not expressible in HOL4's simple type theory: E is a type-level function and Vis quantifies over the type A. We must require E to be simply typed, and avoid quantification over A except at the top-level. Foster et al. [2021] faced the same issue when defining ITrees in Isabelle/HOL. They chose to remove quantification of A entirely, instead restricting expressivity by requiring environment responses to share the same type as program outputs E.

As HOL4 has no guardedness condition when writing co-recursive functions, we can further remove Tau; ITrees can now be equated by strong bisimulation, which coincides with HOL4's built-in equality. This simplifies proofs of semantics preservation (*i.e.*, ITree equality) considerably in compiler verification. To ensure we can still express silent divergence, we add a nullary Div constructor, which we can straightforwardly produce using non-constructivity of HOL4's logic (as described in definition 2.2, pg. 27).

The final resulting coinductive grammar is below, replacing Coq's Roman postfix type variables for HOL4's Greek prefix ones.

$$(\varepsilon,\ \alpha,\ \rho)\ \mathsf{itree} ::= \mathsf{Ret}\ (r : \rho) \mid \mathsf{Div} \mid \mathsf{Vis}\ (e : \varepsilon)\ \big(k : \alpha \to (\varepsilon,\ \alpha,\ \rho)\ \mathsf{itree}\big)$$

Formalisation details. *In this paragraph, I describe work completed by Magnus Myreen.* HOL4 has no built-in support for defining codatatypes or co-recursive functions in the style of Coq or Agda. Instead, we define ITrees by carving out a subset of an existing type itree', known as the *representation type*:

$$(\varepsilon,\ \alpha,\ \rho)\ \mathsf{itree'} \stackrel{\mathsf{def}}{=} \alpha\ \mathsf{list} \to (\varepsilon,\ \rho)\ \mathsf{node} \qquad (\varepsilon,\ \rho)\ \mathsf{node} ::= \mathsf{Return}\ \rho \mid \mathsf{Stuck} \mid \mathsf{Event}\ \varepsilon$$

Each function $f : (\varepsilon,\ \alpha,\ \rho)$ itree' represents an ITree of type $(\varepsilon,\ \alpha,\ \rho)$ itree piecewise: when applied to a list $l : \alpha$ list, it returns a single node of the tree. In particular, we can provide successive environment responses from list l to Vis continuations in the ITree, producing the final node indicated by $f\ l : (\varepsilon,\ \rho)$ node.

To permit co-recursive definition of an itree, we derive an unfolding function: unfold iterates a function $f : \alpha \to (\varepsilon,\ \alpha,\ \rho)$ next on a seed value of type α. Each application of f produces a node in the ITree (denoted by a constructor of next), and optionally permits the iteration to continue. A descriptive equation of unfold is shown below.

Lemma 2.1. ITree unfolding.

$$\vdash \mathsf{unfold}\ f\ \mathit{seed} = \begin{cases} \mathsf{Ret}\ r & f\ \mathit{seed}\ = \mathsf{Ret'}\ r \\ \mathsf{Div} & f\ \mathit{seed}\ = \mathsf{Div'} \\ \mathsf{Vis}\ e\ \left(\lambda a.\ \mathsf{unfold}\ f\ (g\ a)\right) & f\ \mathit{seed}\ = \mathsf{Vis'}\ e\ g \end{cases}$$

where $(\varepsilon,\ \alpha,\ \rho)$ next ::= Ret' $(r : \rho)$ | Div' | Vis' $(e : \varepsilon)\ (g : \alpha \to \alpha)$

Expressivity and usage of HOL4-compatible 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 our limited modifications. However, the result is less expressive than the original: the (simple) types of program outputs E and environment

ε ::=	Observable events	ρ ::=	Return values
$\mid (ch, s)$	FFI call	$\mid$ **terminate**	successful termination
α ::=	Environment responses	$\mid$ **error**	runtime type error
$\mid$ **ok** s	successful FFI return	$\mid$ **fail**$_{\text{ffi}}$	FFI error
$\mid$ **fail**$_{\text{ffi}}$	FFI error	$\mid$ **diverge**$_{\text{ffi}}$	lack of FFI return
$\mid$ **diverge**$_{\text{ffi}}$	lack of FFI return		

Figure 2.3. Instantiations of parameters ε, α, and ρ for the itree type used in PureLang semantics.

responses A are fixed at the top-level. Fortunately, PureLang semantics requires only fixed types for both (§ 2.3.3). We use our modified 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.

2.3.3 Semantics of PureLang: semantic expressions

We define the semantics of PureLang semantic expressions operationally in three stages. I summarise these stages below by showing their top-level functions and associated types, which I describe further in following paragraphs.

1. Functional big-step (clocked) evaluation of a semantic expression e to produce a weak-head normal form wh:
   $\mathsf{eval}^{j}_{\text{wh}} : (j : \mathsf{num}) \to e \to wh$

2. Unclocked evaluation of an expression to a weak-head normal form:
   $\mathsf{eval}_{\text{wh}} : e \to wh$

3. Stateful interpretation of `IO` operations using stack κ and mutable store σ:
   $(\!|-,-,-|\!) : wh \to \kappa \to \sigma \to (\varepsilon,\ \alpha,\ \rho)\ \mathsf{itree}$

The final stage (stateful interpretation) produces an ITree of type $(\varepsilon,\ \alpha,\ \rho)$ itree for the instantiations of ε, α, and ρ defined in figure 2.3. In particular, the only externally observable events ε produced by PureLang programs are FFI calls: a pair of an FFI channel name ch and argument s, both of type string.[1] An environment response α to an FFI call is either: 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 C function

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

op_{wh} ::=	Weak-head operations
| $\mathbf{cons}_{\text{wh}}\ cname$	data constructor/tuple
| $\mathbf{monadic}_{\text{wh}}\ mop$	monadic operation
wh ::=	Weak-head forms
| $op_{\text{wh}}[\ \overline{e_n}\ ]$	weak-head operation
| $\mathbf{lambda}\ x\ e$	function
| $\mathbf{lit}\ lit$	literal value
| **error**	runtime type error
| **diverge**	timeout/divergence

Figure 2.4. Weak-head operations op_{wh} and normal forms wh.

name, and the various strings are proxies for the low-level byte arrays used to interface with the C functions (§ 5.1). The observable return values ρ of PureLang are therefore successful termination, FFI failure/divergence, or runtime type error (crash). The latter are necessary because the semantics of PureLang is untyped.

Below, I describe each of the three stages listed above.

1. Clocked evaluation. Pure, call-by-name evaluation defined in the functional big-step style attempts to produce weak-head normal forms, $\mathsf{eval}^{j}_{\text{wh}}\ e = wh$. Intuitively, weak-head normal forms are expressions with an irreducible outermost part, on which no top-level reduction can be performed. Weak-head normal forms wh for PureLang are shown in figure 2.4; **diverge** indicates a timeout (running out of fuel).

The clocked evaluator is most clearly showcased in the style of an exception monad in figure 2.5, where **error** and **diverge** are considered monadic exceptions and I have used Haskell-like do-notation to hide monadic operators bind, assert, and tick. All failed assertions and pattern matches produce runtime type errors. So too do unbound variables, so the semantics of PureLang effectively considers only closed expressions (those without free variables). Constructor/tuple/monadic operations and λ-abstractions immediately produce corresponding weak-head normal forms. Sequencing diverges or crashes whenever its first argument does. Both recursive bindings and λ-applications substitute ($e[^{e'}/_{x}]$) only closed terms to avoid variable capture; evaluation is call-by-name. An $\mathbf{eq}_?$-statement expects a constructor, whose name and arity it tests (**true**/**false** are shorthands for $\mathbf{cons}\ cname[\ ]$ where $cname$ = true/false respectively). Note that matching names with mismatching arities result in type errors: constructor names are expected to be unique with a well-defined arity. Projections also expect a constructor, but enforce matching constructor names and require sufficient arity to extract the indexed argument.

Literals directly produce **lit** forms (omitted here), message construction produces a message literal, and integer addition evaluates each operand to an integer and returns the sum. Evaluation of other primitive operations is defined similarly. However, PureLang is a pure language, with exceptions confined to the IO monad: we must give a reasonable

$$\mathsf{bind}\ x\ f \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \in \{\mathbf{error},\ \mathbf{diverge}\}, \\ f\ x & \text{otherwise.} \end{cases}$$

$$\mathsf{assert}\ P\ x \stackrel{\text{def}}{=} \mathsf{if}\ P\ \mathsf{then}\ x\ \mathsf{else}\ \mathbf{error}$$

$$\mathsf{tick}\ j\ f \stackrel{\text{def}}{=} \mathsf{if}\ j = 0\ \mathsf{then}\ \mathbf{diverge}\ \mathsf{else}\ f\ (j-1)$$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{var}\, x) \stackrel{\text{def}}{=} \mathbf{error}$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{monadic}\, mop[\,\overline{e_n}\,]) \stackrel{\text{def}}{=}$
  $\mathbf{monadic}_{\mathrm{wh}}\, mop[\,\overline{e_n}\,]$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{seq}\, e_1\, e_2) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathsf{eval}^{j}_{\mathrm{wh}}\ e_1;\ \mathsf{eval}^{j}_{\mathrm{wh}}\ e_2$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{letrec}\, \overline{x_n = e_n}\ \mathbf{in}\ e) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathsf{assert}\ (\mathsf{freevars}\ \overline{e_n} \subseteq \overline{x_n});$
  $j' \leftarrow \mathsf{tick}\ j;$
  $\mathsf{eval}^{j'}_{\mathrm{wh}}\left(e\left[\overline{\mathbf{letrec}\, \overline{x_n = e_n}\ \mathbf{in}\ e_n / x_n}\right]\right)$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{eq}_?\, cname\, n\, e) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathbf{cons}\, cname'[\,\overline{e_m}\,] \leftarrow \mathsf{eval}^{j}_{\mathrm{wh}}\ e;$
  $\mathsf{if}\ cname \neq cname'\ \mathsf{then}\ \mathbf{false};$
  $\mathsf{else}\ \mathsf{assert}\ (n = m);\ \mathbf{true}$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{prim}\ (\mathbf{message}\, ch)\, [e]) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathbf{lit}\ (\mathbf{str}\, s) \leftarrow \mathsf{eval}^{j}_{\mathrm{wh}}\ e;$
  $\mathbf{lit}\ (\mathbf{msg}\, ch\, s)$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{cons}\, cname[\,\overline{e_n}\,]) \stackrel{\text{def}}{=}$
  $\mathbf{cons}_{\mathrm{wh}}\, cname[\,\overline{e_n}\,]$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\lambda x.\ e) \stackrel{\text{def}}{=} \mathbf{lambda}\, x\, e$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (e_1 \cdot e_2) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathbf{lambda}\, x\, e \leftarrow \mathsf{eval}^{j}_{\mathrm{wh}}\ e_1;$
  $\mathsf{assert}\ (\mathsf{closed}\ e_2);$
  $j' \leftarrow \mathsf{tick}\ j;$
  $\mathsf{eval}^{j'}_{\mathrm{wh}}\ (e[e_2/x])$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{proj}_n\ cname\ e) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathbf{cons}\, cname'[\,\overline{e_m}\,] \leftarrow \mathsf{eval}^{j}_{\mathrm{wh}}\ e;$
  $\mathsf{assert}\ (cname = cname' \ \wedge\ n < m);$
  $j' \leftarrow \mathsf{tick}\ j;$
  $\mathsf{eval}^{j'}_{\mathrm{wh}}\ e_m$

$\mathsf{eval}^{j}_{\mathrm{wh}}\ (\mathbf{prim}\, \mathbf{add}[e_1\, e_2]) \stackrel{\text{def}}{=} \mathsf{do}$
  $\mathbf{lit}\ (\mathbf{int}\, z_1) \leftarrow \mathsf{eval}^{j}_{\mathrm{wh}}\ e_1;$
  $\mathbf{lit}\ (\mathbf{int}\, z_2) \leftarrow \mathsf{eval}^{j}_{\mathrm{wh}}\ e_2;$
  $\mathbf{lit}\ (\mathbf{int}\ (z_1 + z_2))$

Figure 2.5. Core clauses of clocked evaluation, $\mathsf{eval}^{j}_{\mathrm{wh}} : (j : \mathsf{num}) \to e \to wh$

semantics to out-of-bounds string indexing and division/modulo by zero. The semantics adjusts indices to remain in-bounds, and returns zero for division/modulo by zero. The latter is sometimes controversial, but matches the (total) definitions of various theorem provers and is generally considered conservative.

Clocked evaluation always terminates: on each recursive call, either the clock is constant and the expression size decreases, or the clock is decremented if the expression size does not decrease (**letrec**, λ-application, and **proj**).

2. Unclocked evaluation. We lift to unclocked evaluation, $\mathsf{eval}_{\mathrm{wh}}\, e = wh$, by classically quantifying over clock j: unclocked evaluation produces a weak-head form wh if there is some clock j for which clocked evaluation produces wh without timing out. Otherwise, clocked evaluation always times out, so unclocked evaluation should produce **diverge**.

Definition 2.2. Unclocked evaluation.

$$\mathsf{eval}_{\mathrm{wh}}\, e \stackrel{\text{def}}{=} \begin{cases} wh & \text{if } \exists j.\ \mathsf{eval}^{j}_{\mathrm{wh}}\, e = wh \ \wedge\ wh \neq \mathbf{diverge}, \\ \mathbf{diverge} & \text{otherwise, } \textit{i.e.}, \ \forall j.\ \mathsf{eval}^{j}_{\mathrm{wh}}\, e = \mathbf{diverge}. \end{cases}$$

There is a moral difference between **diverge** in the two evaluators: clocked **diverge** is really a timeout, where unclocked **diverge** is true (silent) divergence. This definition of $\mathsf{eval}_{\mathrm{wh}}$ is unwieldy, but we can derive clean characterising equations. These are nearly identical to those in figure 2.5 (pg. 26), but they do not mention superscripted clocks or tick. For example, equations for λ-application and **proj** follow:

$$\begin{array}{l} \vdash \mathsf{eval}_{\mathrm{wh}}\ (e_1 \cdot e_2) = \mathsf{do} \\ \quad \mathbf{lambda}\, x\, e \leftarrow \mathsf{eval}_{\mathrm{wh}}\, e_1; \\ \quad \mathsf{assert}\ (\mathsf{closed}\ e_2); \\ \quad \mathsf{eval}_{\mathrm{wh}}\ (e[{}^{e_2}/_{x}]) \end{array} \qquad \begin{array}{l} \vdash \mathsf{eval}_{\mathrm{wh}}\ \left(\mathbf{proj}_n\ cname\ e\right) = \mathsf{do} \\ \quad \mathbf{cons}\, cname'[\,\overline{e_m}\,] \leftarrow \mathsf{eval}_{\mathrm{wh}}\, e; \\ \quad \mathsf{assert}\ (cname = cname' \ \wedge\ n < m); \\ \quad \mathsf{eval}_{\mathrm{wh}}\ e_m \end{array}$$

3. Stateful interpretation of `IO` operations. We must model the sequencing, stateful updates, and I/O of monadic operations $\mathbf{monadic}\, mop[\,\overline{e_n}\,]$. We use a stack machine with mutable store. Its states $\langle wh,\ \kappa,\ \sigma\rangle$ consist of the weak-head normal form wh being interpreted, a stack of continuations κ, and mutable store σ. The continuation stack models monadic sequencing, and the mutable store models stateful operations. Machine transitions induce an ITree co-recursively, modelling I/O by emitting Vis nodes.

Once again, we use two layers to define the machine's semantics: clocked and unclocked. The clocked version, $\mathsf{nextNode}^{j}\ \langle wh,\ \kappa,\ \sigma\rangle$, iterates machine transitions, attempting to produce an element of the next type (lemma 2.1, pg. 23). It defaults to Div' when it times out. Classical quantification lifts to the unclocked version, $\mathsf{nextNode}\ \langle wh,\ \kappa,\ \sigma\rangle$;

now Div' denotes true (silent) divergence. Application of unfold (lemma 2.1, pg. 23) produces the final co-recursive semantics:

$$(\!|\, wh,\ \kappa,\ \sigma \,|\!) \stackrel{\text{def}}{=} \mathsf{unfold}\ \big(\mathsf{nextNode}\ \langle wh,\ \kappa,\ \sigma\rangle\big)$$

I omit formal descriptions of $\mathsf{nextNode}^j$ and nextNode, instead showing descriptive equations derivable for $(\!|-,-,-|\!)$ in figure 2.6. Diverging and crashing weak-head normal forms produce Div and Ret **error** nodes respectively. A **bind**-statement continues as its left-hand expression, pushing a **bind**-continuation onto the stack; conversely a **return** attempts to pop a **bind**-continuation off the stack, ignoring any intervening **handle**-continuations, and terminating if the stack is exhausted (ε). Here, the bullets ($\bullet$) denotes the "holes" formed in the continuations by removing their subexpressions for evaluation. Like **bind**, **handle** produces a **handle**-continuation frame and **raise** pops the stack in search of such a frame, terminating if none is found. Array operations interact with mutable store: **alloc** extends the store; **len** queries an array length; **deref** queries an array element; and **update** stores an array element. Each array operation implicitly **return**s or **raise**s a weak-head normal form for further processing; the **subscript** exception is **raise**d on an out-of-bounds access. Operation **alloc** demonstrates the need to include location literals in semantic expressions. An **action** produces a Vis node which *both* contains the output message of the program *and* accepts response r from the environment. This response is checked by checkFFI, which terminates on receiving an FFI error, and constructs the remaining ITree coinductively for a valid response. There is one subtlety: FFI response lengths are bounded by responseBound, and overlong responses are considered FFI failures. The need for this restriction is discussed in § 5.1.

Throughout figure 2.6, note the marked absence of Tau nodes during silent steps, those which make an internal transition of the form $(\!|\, wh,\ \kappa,\ \sigma \,|\!) = (\!|\, wh',\ \kappa',\ \sigma' \,|\!)$. In Coq, these equations would not hold: either a Tau node is required on the right-hand side (*i.e.*, $\ldots$ = Tau $\ldots$) or we must replace equality (strong bisimulation) with weak bisimulation.

Note too 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, so all of its stateful and I/O effects are modelled. Critically, this is the case even if e_1 is not used in e_2. PureLang I/O is therefore *not* lazy; lazy I/O is known to break referential transparency.[2]

Each PureLang program defines an entrypoint `main :: IO ()`, so the weak-head form of the entire program is a monadic operation. Therefore, the semantics of a whole program is specified using the empty continuation stack and empty initial state:

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

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

$$
\begin{aligned}
(\!|\, \mathbf{diverge},\ \kappa,\ \sigma \,|\!) &= \mathsf{Div} & (\!|\, \mathbf{error},\ \kappa,\ \sigma \,|\!) &= \mathsf{Ret}\ \mathbf{error} \\
(\!|\, \mathbf{bind}\, e_1\, e_2,\ \kappa,\ \sigma \,|\!) &= (\!|\, \mathsf{eval}_{\mathrm{wh}}\, e_1,\ \mathbf{bind} \bullet e_2 :: \kappa,\ \sigma \,|\!) \\
(\!|\, \mathbf{return}\, e_1,\ \mathbf{bind} \bullet e_2 :: \kappa,\ \sigma \,|\!) &= (\!|\, \mathsf{eval}_{\mathrm{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma \,|\!) \\
(\!|\, \mathbf{return}\, e_1,\ \mathbf{handle} \bullet e_2 :: \kappa,\ \sigma \,|\!) &= (\!|\, \mathbf{return}\, e_1,\ \kappa,\ \sigma \,|\!) \\
(\!|\, \mathbf{return}\, e,\ \varepsilon,\ \sigma \,|\!) &= \mathsf{Ret}\ \mathbf{terminate} \\
(\!|\, \mathbf{handle}\, e_1\, e_2,\ \kappa,\ \sigma \,|\!) &= (\!|\, \mathsf{eval}_{\mathrm{wh}}\, e_1,\ \mathbf{handle} \bullet e_2 :: \kappa,\ \sigma \,|\!) \\
(\!|\, \mathbf{raise}\, e_1,\ \mathit{frame} :: \ldots :: \mathbf{handle} \bullet e_2 :: \kappa,\ \sigma \,|\!) &= (\!|\, \mathsf{eval}_{\mathrm{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma \,|\!) \\
(\!|\, \mathbf{raise}\, e,\ \varepsilon,\ \sigma \,|\!) &= \mathsf{Ret}\ \mathbf{terminate}
\end{aligned}
$$

$$
\frac{l \notin \mathrm{domain}\ \sigma}{(\!|\, \mathbf{alloc}\ (\mathbf{int}\, z)\ e,\ \kappa,\ \sigma \,|\!) = (\!|\, \mathbf{return}\, (\mathbf{loc}\ l),\ \kappa,\ \sigma\big[l \mapsto [\underbrace{e,\ \ldots,\ e}_{\max\, z\ 0}]\big] \,|\!)}
$$

$$
\frac{\mathsf{eval}_{\mathrm{wh}}\, e = \mathbf{lit}\ (\mathbf{loc}\, l) \qquad l \in \mathrm{domain}\ \sigma}{(\!|\, \mathbf{len}\, e,\ \kappa,\ \sigma \,|\!) = (\!|\, \mathbf{return}\, \big(\mathbf{int}\ |\sigma(l)|\big),\ \kappa,\ \sigma \,|\!)}
$$

$$
\frac{\mathsf{eval}_{\mathrm{wh}}\, e = \mathbf{lit}\ (\mathbf{loc}\, l) \qquad l \in \mathrm{domain}\ \sigma \qquad 0 \le z < |\sigma(l)|}{(\!|\, \mathbf{deref}\, e\ (\mathbf{int}\, z)\,,\ \kappa,\ \sigma \,|\!) = (\!|\, \mathbf{return}\, \big(\sigma(l)[z]\big),\ \kappa\ , \sigma \,|\!)}
$$

$$
\frac{\mathsf{eval}_{\mathrm{wh}}\, e = \mathbf{lit}\ (\mathbf{loc}\, l) \qquad l \in \mathrm{domain}\ \sigma \qquad z < 0\ \vee\ z \ge |\sigma(l)|}{(\!|\, \mathbf{deref}\, e\ (\mathbf{int}\, z)\,,\ \kappa,\ \sigma \,|\!) = (\!|\, \mathbf{raise\ subscript},\ \kappa\ , \sigma \,|\!)}
$$

$$
\frac{\mathsf{eval}_{\mathrm{wh}}\, e = \mathbf{lit}\ (\mathbf{loc}\, l) \qquad l \in \mathrm{domain}\ \sigma \qquad 0 \le z < |\sigma(l)|}{(\!|\, \mathbf{update}\, e\ (\mathbf{int}\, z)\ e',\ \kappa,\ \sigma \,|\!) = (\!|\, \mathbf{return\ unit},\ \kappa\ , \sigma\big[l \mapsto [\underbrace{\ldots\ldots\ldots}_{\max\, (z-1)\ 0},\ e',\ \ldots]\big] \,|\!)}
$$

$$
\frac{\mathsf{eval}_{\mathrm{wh}}\, e = \mathbf{lit}\ (\mathbf{loc}\, l) \qquad l \in \mathrm{domain}\ \sigma \qquad z < 0\ \vee\ z \ge |\sigma(l)|}{(\!|\, \mathbf{update}\, e\ (\mathbf{int}\, z)\ e',\ \kappa,\ \sigma \,|\!) = (\!|\, \mathbf{raise\ subscript},\ \kappa\ , \sigma \,|\!)}
$$

$$
\frac{\mathsf{eval}_{\mathrm{wh}}\, e = \mathbf{lit}\ (\mathbf{msg}\, ch\, s)}{(\!|\, \mathbf{action}\, e,\ \kappa,\ \sigma \,|\!) = \mathsf{Vis}\ (ch, s)\ (\lambda r.\ \mathsf{checkFFI}\ \kappa\ \sigma\ r)}
$$

$$
\mathsf{checkFFI}\ \kappa\ \sigma\ r \stackrel{\mathrm{def}}{=} \begin{cases} \mathsf{Ret}\ r & r \in \{\mathbf{fail}_{\mathrm{ffi}},\ \mathbf{diverge}_{\mathrm{ffi}}\} \\ (\!|\, \mathbf{return}\ (\mathbf{str}\, s')\,,\ \kappa,\ \sigma \,|\!) & r = \mathbf{ok}\, s' \ \wedge\ |s'| \le \mathsf{responseBound} \\ \mathbf{fail}_{\mathrm{ffi}} & \text{otherwise} \end{cases}
$$

Figure 2.6. Derived rules for $(\!|-,-,-|\!)$, PURELANG's stateful interpreter for monadic operations. Here, **subscript** is shorthand for a nullary constructor and **bind** $e_1\, e_2$ is shorthand for **monadic**$_{\mathrm{wh}}$ **bind**$[e_1\, e_2]$ (similarly for other monadic operations).

2.3.4 Semantics of PureLang: compiler expressions

Semantics of compiler expressions is defined via desugaring (exp_of) to semantic expressions in definition 2.3: variadic λ-abstractions/-applications become nested unary ones; **let** becomes function application; and **case** becomes a mixture of **if**-, **eq**$_?$-, and **proj**-statements. All other cases are simple recursion. To desugar **case** $x = ce$ **of** $\overline{row_n}$ we assign exp_of ce to x and test it against each row of the pattern match ($row = cname[\,\overline{y_n}\,] \to ce'$) with **if**/**eq**$_?$. If a row matches both constructor name ($cname$) and arity (n), **proj** statements extract constructor arguments, which are assigned to the corresponding pattern variables ($\overline{y_n}$) before desugaring the continuation. Again, PureLang is pure and exceptions are confined to the `IO` monad: failed pattern matches produce runtime type errors (**fail**, which stands for any always-crashing program); PureLang enforces exhaustive pattern matches (potentially via a final optional catch-all *row*) to avoid these.

Definition 2.3. Desugaring of compiler expressions.

$$
\begin{aligned}
\mathsf{exp\_of}\ (\lambda\,\overline{x_n}\,.\,ce) &\stackrel{\text{def}}{=} \lambda x_1.\ \lambda x_2.\ \ldots\ \lambda x_n.\ \mathsf{exp\_of}\,ce \\
\mathsf{exp\_of}\ (ce \cdot \overline{ce_n}) &\stackrel{\text{def}}{=} \big(\ldots(\mathsf{exp\_of}\,ce \cdot \mathsf{exp\_of}\,ce_1) \cdot \ldots\big) \cdot \mathsf{exp\_of}\,ce_n \\
\mathsf{exp\_of}\ (\mathbf{let}\,x = ce_1\ \mathbf{in}\ ce_2) &\stackrel{\text{def}}{=} (\lambda x.\ \mathsf{exp\_of}\,ce_2) \cdot (\mathsf{exp\_of}\,ce_1) \\
\mathsf{exp\_of}\,(\mathbf{case}\,x = ce\,\mathbf{of}\,\overline{row_n}) &\stackrel{\text{def}}{=} \mathbf{let}\,x = \mathsf{exp\_of}\,ce\ \mathbf{in}\ \mathsf{expand}_x\,[\,\overline{row_n}\,] \\
\\
\mathsf{expand}_x\,[cname[\,\overline{y_n}\,] \to ce',\,\overline{row_m}\,] &\stackrel{\text{def}}{=} \mathbf{if}\ (\mathbf{eq}_?\ cname\ n\ (\mathbf{var}\,x))\ \mathbf{then} \\
&\qquad \mathbf{let}\,\overline{y_n = \mathbf{proj}_n\ cname\ (\mathbf{var}\,x)}\ \mathbf{in}\ (\mathsf{exp\_of}\,ce') \\
&\qquad \mathbf{else}\ \mathsf{expand}_x\,[\,\overline{row_m}\,] \\
\mathsf{expand}_x\,[\_ \to ce'] &\stackrel{\text{def}}{=} \mathsf{exp\_of}\,ce' \\
\mathsf{expand}_x\,[\,] &\stackrel{\text{def}}{=} \mathbf{fail}
\end{aligned}
$$

2.4 Equational reasoning

Haskell programmers rely on equational reasoning to understand code, "stepping through" program execution by unfolding function definitions and substituting equal terms. A verified implementation of a Haskell-like language allows us to formalise and mechanise this intuition, and verify it is preserved through compilation. In this section, I summarise the formalisation of PureLang's equational theory: its formulation and proof of congruence (§§ 2.4.1 and 2.4.2), and its interaction with other standard equivalences (§ 2.4.3). This formalisation follows a detailed account by Pitts [2012].

2.4.1 Defining an equivalence

We adopt untyped *applicative bisimilarity* [Abramsky 1990] as an equivalence relation on expressions. Intuitively, a relation satisfies *applicative simulation* if it is closed under weak-head reduction and usage of any resulting weak-head normal form.

Definition 2.4. Applicative simulation. A binary relation $\mathcal{R}$ on closed expressions is an applicative simulation if, for all closed expressions e_1 and e_2 such that $e_1 \mathrel{\mathcal{R}} e_2$:

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

$$\mathsf{eval}_{\mathrm{wh}}\, e_1 = op_{\mathrm{wh}}[\,\overline{e_{1\,n}}\,] \Rightarrow \\ \quad \exists \overline{e_{2\,n}}.\ \mathsf{eval}_{\mathrm{wh}}\, e_2 = op_{\mathrm{wh}}[\,\overline{e_{2\,n}}\,] \ \wedge\ \forall\, m < n.\ e_{1m} \mathrel{\mathcal{R}} e_{2m}$$

$$\mathsf{eval}_{\mathrm{wh}}\, e_1 = \mathbf{lit}\, lit \Rightarrow \mathsf{eval}_{\mathrm{wh}}\, e_2 = \mathbf{lit}\, lit$$

$$\mathsf{eval}_{\mathrm{wh}}\, e_1 = \mathbf{error} \Rightarrow \mathsf{eval}_{\mathrm{wh}}\, e_2 = \mathbf{error}$$

Note that no restrictions are placed on $\mathcal{R}$ when its left-hand argument diverges. *Applicative bisimulations* are symmetric applicative simulations. These recover divergence preservation by totality of $\mathsf{eval}_{\mathrm{wh}}$. *Applicative similarity* ($e_1 \lesssim e_2$) and *bisimilarity* ($e_1 \simeq e_2$) are defined as the greatest applicative simulation and bisimulation respectively. In particular, the definition of applicative (bi)simulation induces a monotone functor, for which we can derive a greatest fixed point. Applicative similarity is reflexive and transitive, and applicative bisimilarity is equivalent to two-way applicative similarity. It is therefore also symmetric, and so an equivalence.

2.4.2 Proof of congruence via Howe's method

Our relation should also be a congruence: expressions formed from bisimilar subexpressions should also be bisimilar. We define a congruence as a symmetric precongruence, where a precongruence is transitive and compatible. Compatible relations $\mathcal{R}$ satisfy at least the rules in figure 2.7. Howe's method [Howe 1996] is a well-studied technique for establishing congruence. Applicative (bi)similarity is extended to open terms using closing substitutions, *e.g.*, in the case of *open* bisimilarity:

$$\overline{x_n} \vdash e \simeq e' \stackrel{\text{def}}{=} \mathsf{freevars}\, e \cup \mathsf{freevars}\, e' \subseteq \overline{x_n} \ \wedge\ \forall \text{ closed } \overline{e_n}.\ e\left[\overline{{}^{e_n}/_{x_n}}\right] \simeq e'\left[\overline{{}^{e_n}/_{x_n}}\right]$$

We define Howe's construction in figure 2.8. Intuitively, an expression e with subexpressions e_i is Howe-related to e' if the e_i are Howe-related to some e_i' and replacing subexpressions e_i by e_i' produces an expression related to e'.

$$\frac{}{(\mathbf{var}\, x)\ \ \mathcal{R}\ \ (\mathbf{var}\, x)} \qquad \frac{\forall\, m < n.\ e_m\ \ \mathcal{R}\ \ e'_m}{(op[\,\overline{e_n}\,])\ \ \mathcal{R}\ \ \left(op[\,\overline{e'_n}\,]\right)} \qquad \frac{e\ \ \mathcal{R}\ \ e'}{(\lambda x.\, e)\ \ \mathcal{R}\ \ (\lambda x.\, e')}$$

$$\frac{e_1\ \ \mathcal{R}\ \ e'_1 \qquad e_2\ \ \mathcal{R}\ \ e'_2}{(e_1 \cdot e_2)\ \ \mathcal{R}\ \ \left(e'_1 \cdot e'_2\right)} \qquad \frac{e\ \ \mathcal{R}\ \ e' \qquad \forall\, m < n.\ e_m\ \ \mathcal{R}\ \ e'_m}{(\mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ e)\ \ \mathcal{R}\ \ \left(\mathbf{letrec}\ \overline{x_n = e'_n}\ \mathbf{in}\ e'\right)}$$

$$\frac{e_1\ \ \mathcal{R}\ \ e'_1 \qquad e_2\ \ \mathcal{R}\ \ e'_2}{(\mathbf{seq}\, e_1\, e_2)\ \ \mathcal{R}\ \ \left(\mathbf{seq}\, e'_1\, e'_2\right)} \qquad \frac{e\ \ \mathcal{R}\ \ e' \qquad e_1\ \ \mathcal{R}\ \ e'_1 \qquad e_2\ \ \mathcal{R}\ \ e'_2}{(\mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2)\ \ \mathcal{R}\ \ \left(\mathbf{if}\, e'\, \mathbf{then}\, e'_1\, \mathbf{else}\, e'_2\right)}$$

$$\frac{e\ \ \mathcal{R}\ \ e'}{(\mathbf{eq}_?\, nm\, ar\, e)\ \ \mathcal{R}\ \ (\mathbf{eq}_?\, nm\, ar\, e')} \qquad \frac{e\ \ \mathcal{R}\ \ e'}{\left(\mathbf{proj}_n\ nm\ e\right)\ \ \mathcal{R}\ \ \left(\mathbf{proj}_n\ nm\ e'\right)}$$

Figure 2.7. An inductive definition of compatibility for a relation $\mathcal{R}$.

$$\frac{\overline{x_n} \vdash (\mathbf{var}\, x)\ \ \mathcal{R}\ \ e}{\overline{x_n} \vdash (\mathbf{var}\, x)\ \ \mathcal{R}^H\ e} \qquad \frac{\forall\, m < j.\ \overline{x_n} \vdash e_m\ \ \mathcal{R}^H\ e'_m \qquad \overline{x_n} \vdash \left(op[\,\overline{e'_j}\,]\right)\ \ \mathcal{R}\ \ e}{\overline{x_n} \vdash \left(op[\,\overline{e_j}\,]\right)\ \ \mathcal{R}^H\ e}$$

$$\frac{\{x\} \cup \overline{x_n} \vdash e_1\ \ \mathcal{R}^H\ e'_1 \qquad \overline{x_n} \vdash \lambda x.\, e'_1\ \ \mathcal{R}\ \ e}{\overline{x_n} \vdash (\lambda x.\, e_1)\ \ \mathcal{R}^H\ e}$$

$$\frac{\overline{x_n} \vdash e_1\ \ \mathcal{R}^H\ e'_1 \qquad \overline{x_n} \vdash e_2\ \ \mathcal{R}^H\ e'_2 \qquad \overline{x_n} \vdash \left(e'_1 \cdot e'_2\right)\ \ \mathcal{R}\ \ e}{\overline{x_n} \vdash (e_1 \cdot e_2)\ \ \mathcal{R}^H\ e}$$

$$\frac{\begin{array}{c}\overline{x_j} \cup \overline{x_n} \vdash e\ \ \mathcal{R}^H\ e' \\ \forall\, m < j.\ \overline{x_j} \cup \overline{x_n} \vdash e_m\ \ \mathcal{R}^H\ e'_m \qquad \overline{x_n} \vdash \left(\mathbf{letrec}\ \overline{x_j = e'_j}\ \mathbf{in}\ e'\right)\ \ \mathcal{R}\ \ e_0\end{array}}{\overline{x_n} \vdash \left(\mathbf{letrec}\ \overline{x_j = e_j}\ \mathbf{in}\ e\right)\ \ \mathcal{R}^H\ e_0}$$

Figure 2.8. Howe's construction, an inductive relation $- \vdash - \mathcal{R}^H -$ defined in terms of a relation $- \vdash - \mathcal{R} -$. Simple rules concerning **seq**, **if**, $\mathbf{eq}_?$, and **proj** are omitted.

By construction, $\mathcal{R}^H$ is compatible if $\mathcal{R}$ is reflexive. If $\mathcal{R}$ is also transitive, then $\overline{x_n} \vdash e \ \mathcal{R} \ e'$ is contained within $\overline{x_n} \vdash e \ \mathcal{R}^H \ e'$. We show that $\varnothing \vdash - \lesssim^H -$ (Howe applicative similarity on closed terms) is an applicative simulation, and so contained within applicative similarity, the greatest applicative simulation. Therefore, $\lesssim$ and $\lesssim^H$ coincide for closed expressions. Both are closed under substitutions, so we also have coincidence for open expressions:

$$\vdash \left(\overline{x_n} \vdash e \lesssim e'\right) \iff \left(\overline{x_n} \vdash e \lesssim^H e'\right)$$

Because $\lesssim$ is reflexive and transitive, it is also compatible and a precongruence. Its symmetric version $\simeq$ is therefore a congruence. Now, *expression equivalence* is as follows:

$$e \cong e' \stackrel{\text{def}}{=} \mathsf{freevars}\, e \cup \mathsf{freevars}\, e' \vdash e \simeq e'$$

2.4.3 Interaction with other equivalences

We define other standard notions of equivalence and prove that expression equivalence interacts favourably with them. Later, we will use our equational theory to verify PureLang compiler passes: either by proving some syntactic relation is an applicative simulation after closing substitution (§ 3.2), or by appealing to equational reasoning (§ 3.4).

α-equivalence. *In this paragraph, I describe work completed by Johannes Åman Pohjola.* Intuitively, α-equivalent expressions differ only in their choices of bound variable names: formal parameters to functions can be renamed as long as their uses are also updated consistently. Pen-and-paper formalisations often regard α-equivalent expressions as identical, considering only α-equivalent *classes* of expressions. However, a mechanical formalisation of this intuition can be non-trivial [Aydemir et al. 2005].

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 ($- =_\alpha -$) is the transitive closure of a relation $-\mathcal{R}_\alpha-$, which can swap the variable bound at a single site while carefully avoiding free variables (its other cases are defined by simple recursion):

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

$$\frac{y \notin \mathsf{freevars}\ \overline{e_n} \cup \mathsf{freevars}\ \overline{e_m} \cup \mathsf{freevars}\ e}{(\textbf{letrec}\ \overline{x_n = e_n},\ x = e,\ \overline{x_m = e_m}\ \textbf{in}\ e') \ \mathcal{R}_\alpha \left(\textbf{letrec}\ \overline{x_n = \mathsf{perm}_{xy}\ e_n},\ y = \mathsf{perm}_{xy}\ e,\ \overline{x_m = \mathsf{perm}_{xy}\ e_m},\ \textbf{in}\ \mathsf{perm}_{xy}\ e'\right)}$$

We will show that α-equivalence is an applicative simulation after closing substitution, and so contained within expression equivalence. First, note that substitutions respect permutation and so closed substitutions preserve α-equivalence:

$$\vdash \mathsf{perm}_{xy}\ \left(e\left[\overline{e_n/x_n}\right]\right) = (\mathsf{perm}_{xy}\ e)\left[\overline{\mathsf{perm}_{xy}\ e_n/\mathsf{perm}_{xy}\ x_n}\right]$$
$$\vdash e =_\alpha e' \ \wedge\ (\forall\, m < n.\ \mathsf{closed}\ e_m \ \wedge\ e_m =_\alpha e'_m) \Rightarrow e\left[\overline{e_n/x_n}\right] =_\alpha e\left[\overline{e'_n/x_n}\right]$$

Then, α-equivalence is preserved by evaluation: α-equivalent terms result in α-equivalent weak-head normal forms. Applicative simulation follows from these.

$$\vdash e =_\alpha e' \Rightarrow (\mathsf{eval}_{\mathrm{wh}}\, e) =_\alpha (\mathsf{eval}_{\mathrm{wh}}\, e')$$

***β*-equivalence.** *In this paragraph, I describe work completed in part by Riccardo Zanetti.* We define capture-avoiding substitution ($e\langle\overline{e_n/x_n}\rangle$) as a freshening of bound variables followed by ordinary substitution:

$$e\langle\overline{e_n/x_n}\rangle \stackrel{\mathsf{def}}{=} (\mathsf{freshen}_{\overline{e_n}}\ e)[\overline{e_n/x_n}]$$

Here, $\mathsf{freshen}_{\overline{e_n}}\ e$ renames all bound variables in e that occur in the set $\bigcup_n \mathsf{freevars}\ e_n$ by using perm. Internally, it maintains a set of variable names to avoid: each bound variable is renamed to avoid these names, and any uses of the bound variable are appropriately permuted before recursing with an augmented set of avoided names.

Freshening is a special case of α-equivalence: $\mathsf{freshen}_{\overline{e_n}}\ e =_\alpha e$. We can therefore prove the following standard β-equivalences:

$$\vdash (\lambda x.\ e)\cdot e' \ \cong\ e\langle e'/x\rangle \qquad \vdash (\mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ e) \cong e\left\langle\overline{\mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ e_n/x_n}\right\rangle$$

Contextual equivalence. Expression equivalence also coincides with contextual equivalence ($e \sim e'$), which we define as equality of observable semantics (that is, ITree equality) under all closing contexts.

$$e_1 \sim e_2 \stackrel{\mathsf{def}}{=} \forall\ \mathsf{closing}\ C.\ [\![\, C[e_1] \,]\!] = [\![\, C[e_2] \,]\!] \qquad \vdash e_1 \sim e_2 \iff e_1 \cong e_2$$

$$\begin{aligned} C ::= {} & \bullet \mid op[\overline{e_n},\ C,\ \overline{e_m}] \mid \lambda x.\ C \mid C\cdot e_2 \mid e_1\cdot C \mid \mathbf{letrec}\ \overline{x_n = e_n},\ x = C,\ \overline{x_m = e_m}\ \mathbf{in}\ e \mid \\ & \mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ C \mid \mathbf{seq}\, C\, e_2 \mid \mathbf{seq}\, e_1\, C \mid \mathbf{if}\, C\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid \mathbf{if}\, e\, \mathbf{then}\, C\, \mathbf{else}\, e_2 \mid \\ & \mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, C \mid \mathbf{eq}_?\, nm\, ar\, C \mid \mathbf{proj}\, n\, nm\, C \end{aligned}$$

Here, contexts C are expressions with a single hole ($\bullet$), and context application $C[e]$ replaces that hole with an expression e, potentially capturing free variables in e. A

closing context for an expression e is one for which $C[e]$ is closed.

The right-to-left direction follows from congruence of expression equivalence. We take the contrapositive to prove the other direction, *i.e.*, given two inequivalent expressions, we construct a context which distinguishes them:

$$\vdash e_1 \not\cong e_2 \Rightarrow \exists \,\mathsf{closing}\, C.\ [\![C[e_1]]\!] \neq [\![C[e_2]]\!]$$

To construct a context, we must first sufficiently "apply" the weak-head normal forms produced by evaluating e_1 and e_2: the assumed inequivalence may arise after several steps of applicative simulation. At this point, nearly all inequivalences are easily distinguished. To distinguish messages, we pass them to **action** to produce differing Vis nodes. To distinguish inequivalent store locations l and l', we rely on our mechanisation of store locations as natural numbers which are sequentially allocated. Without loss of generality take $l < l'$ and construct a context which allocates $l + 1$ arrays: in this context, a lookup of location l is valid but l' is out-of-bounds.

Equivalences and monads. Our formalisation of expression equivalence concerns only pure evaluation, not stateful interpretation of monadic operations. Monad laws such as **bind** (**return** e) $e' = e' \cdot e$ trivially do not hold for expression equivalence, as the action of $\mathsf{eval}_{\mathrm{wh}}$ produces different weak-head forms on each side in general.

However, this means that monad laws are not valid with respect to contextual equivalence either. Consider the following expressions:

$$\textbf{bind}\ (\textbf{return}\ \textbf{true})\ (\lambda x.\ \textbf{return}\ \textbf{unit}) \qquad \textbf{bind}\ (\textbf{return}\ \textbf{false})\ (\lambda x.\ \textbf{return}\ \textbf{unit})$$

We may expect contextual equivalence: the differing booleans are ignored to produce a unit, so these terms should be extensionally indistinguishable. However, we formalise **monadic** *mop* as **cons** *cname* with reserved *cname*s in PureLang: we can therefore project out the inequivalent booleans with a distinguishing context, $\textbf{proj}_0\ \mathit{bind}\ \bullet$. Other monad laws do not hold because we can construct similarly distinguishing contexts.

If we forbid projection out of monadic operations, such monad laws hold for contextual equivalence but not expression equivalence. In other words, the two no longer coincide: all expression equivalences remain contextual equivalences, but not *vice versa*.

Even without this modification, we can still verify optimisations which manipulate monadic operations (*e.g.*, re-association of **bind**) using simulation proofs. This is because contextual equivalence is not necessary to prove semantics preservation for whole programs: it requires equivalent semantics under *all* closing contexts, not just the ones found in the program. For example, the programs above are contextually *in*equivalent but have identical semantics.

2.5 Type system

PureLang has a standard Hindley-Milner type system [Hindley 1969; Milner 1978], permitted by its lack of *e.g.* type classes. However, we require an unusual proof of type soundness due to tensions between non-strict semantics defined via desugaring and exhaustive pattern matches. In this section, I briefly review the type system and describe its non-standard soundness proof.

Typing judgements are defined over compiler expressions: $\Gamma;\ \Sigma\ \vdash_{cns} ce : \tau$. A standard typing environment Γ gives a type scheme ($\sigma ::= \forall\, \overline{\alpha}\,.\,\tau$) for each free variable in *ce*, store typing Σ gives types for locations l in the mutable store, and the judgement is parametrised over a constructor environment *cns* which both provides type schemes for data/exception constructors and specifies exhaustive sets of constructors for each type (ensuring non-exhaustive pattern matches without a catch-all row are considered ill-typed). In the style of the ML language family, exceptions are monomorphic and belong to an extensible sum type.

Type soundness must be phrased with respect to semantic expressions (§§ 2.2 and 2.3). We consider a semantic expression *e* well-typed if there is some well-typed compiler expression *ce* which desugars to it (top); we then lift typing judgements to weak-head normal forms and aim to prove type soundness for unclocked evaluation (bottom, here $\not\vdash$ denotes an unproven theorem statement).

$$\Gamma;\ \Sigma\ \vdash_{cns} e : \tau \ \stackrel{\text{def}}{=}\ \exists ce.\ \mathsf{exp\_of}\, ce = e\ \wedge\ \Gamma;\ \Sigma\ \vdash_{cns} ce : \tau$$

$$\not\vdash\ \mathsf{cnsOK_?}\ cns\ \wedge\ \Gamma;\ \Sigma\ \vdash_{cns} e : \tau\ \Rightarrow\ \Gamma;\ \Sigma\ \vdash_{cns} \mathsf{eval_{wh}}\, e : \tau$$

That is, in a well-formed constructor environment ($\mathsf{cnsOK_?}$), well-typed semantic expressions should evaluate to well-typed weak-head normal forms. Here, well-formed constructor environments are those which contain unique constructor names whose type schemes are closed and do not refer to undefined types. This statement encapsulates standard progress and preservation: evaluation must produce a weak-head normal form, which cannot be the ill-typed **error**. Note that **diverge** is always well-typed.

However, this proof is doomed due to **case**-statements, which desugar into nested **if**-/**eq**$_?$-/**proj**-statements (definition 2.3, pg. 30). A successful pattern match substitutes a bare **proj**-statement into the continuation expression, *e.g.*:

$$\begin{aligned}
\vdash\ \mathsf{eval_{wh}}\ (\mathsf{exp_of}\, ce) = \mathbf{cons}\, nm[e] \Rightarrow\ & \\
(\mathsf{eval_{wh}} \circ \mathsf{exp_of})\Big(\mathbf{case}\, x = ce\ \mathbf{of}\, nm[y] \to ce'\Big) = & \\
\mathsf{eval_{wh}}\Big(\ (\mathsf{exp_of}\, ce')\ \big[\mathbf{cons}\, nm[e]/x\big]\big[\mathbf{proj}_0\ nm\ \ (\mathbf{cons}\, nm[e])\ /y\big]\Big) &
\end{aligned}$$

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 (the result of desugaring an exhaustive pattern match). A bare **proj**-statement must be ill-typed in general, because when not guarded by sufficient $\mathbf{eq}_?$-statements it can easily produce a type error (§ 2.3.3). This reduction therefore violates type preservation: its left-hand side is well-typed, but its right-hand side may be ill-typed due to the substitution of variables for ill-typed **proj** statements. Any attempted inductive proof of the above type soundness statement will be unable to fulfil the preconditions of the inductive hypothesis for **case**-statements.

We must prove type soundness in a non-standard way. We define a syntax of "typing expressions" *tce*, adding a single **safeproj**-statement to compiler expressions *ce*, which desugars as follows:

$$\begin{aligned}\mathsf{exp\_of}\ \left(\mathbf{safeproj}_n^m\ cname\ tce\right) \stackrel{\text{def}}{=} \ & \mathbf{if}\ \mathbf{eq}_?\ cname\ m\ (\mathsf{exp\_of}\ tce)\\ & \mathbf{then}\ \mathbf{proj}_n\ cname\ (\mathsf{exp\_of}\ tce)\\ & \mathbf{else}\ \bot\end{aligned}$$

That is, it checks if its input constructor matches in name and arity before projecting out arguments. Use of the always-diverging $\bot$ ensures that a bare **safeproj** never produces a type error, so we can give it a typing rule:

$$\frac{\Gamma;\ \Sigma\ \vdash_{cns} tce : \mathsf{Type}\ id\ \overline{\tau_j} \qquad \mathsf{lookup}_{cns}\ cname = \forall\overline{\alpha_j}.\ \tau_1 \to \ldots \to \tau_n \to \ldots \to \tau_m \to \mathsf{Type}\ id\ \overline{\alpha_j}}{\Gamma;\ \Sigma\ \vdash_{cns} \mathbf{safeproj}_n^m\ cname\ tce : \tau_n[\overline{\tau_j}/\overline{\alpha_j}]}$$

Typing expression *tce* must be well-typed as a constructor of type identifier *id* applied to j type arguments $\overline{\tau_j}$. The *cname* specified by **safeproj** should be in the constructor environment with a type scheme which accepts m expression arguments to produce the same type identifier *id* applied to the generalised type variables $\overline{\alpha_j}$. Then, application of $\mathbf{safeproj}_n^m$ selects the nth expression argument, whose type is given by instantiating $\overline{\alpha_j}$ with the appropriate $\overline{\tau_j}$ in the type of the nth argument of *cname*'s type scheme.

We desugar *tce*-flavoured **case** to **safeproj**s instead of **proj**s in definition 2.3 (pg. 30) and lift the remaining typing rules up to typing expressions. Now we can derive type preservation by construction:

$$\vdash \mathsf{cnsOK}_?\ cns\ \wedge\ \Gamma;\ \Sigma\ \vdash_{cns} tce : \tau\ \Rightarrow\ \Gamma;\ \Sigma\ \vdash_{cns} \mathsf{eval}_{\mathrm{wh}}\ (\mathsf{exp\_of}\ tce) : \tau$$

We lift typing judgements to states of the stack machine which interprets monadic operations (§ 2.3.3), and so show that $\mathsf{nextNode}_?$ and nextNode are type-preserving. This

is mostly mechanical; one interesting detail is the typing of continuation stacks. When a stack consists of a **bind**-continuation followed by a **handle**-continuation, **bind** $\bullet$ e :: **handle** $\bullet$ e' :: κ, we must ensure the types returned by e and e' are equally compatible with the rest of the stack, κ. This is because the "current" monadic operation may return normally (passing to the **bind**-continuation) or raise an exception (passing directly to the **handle**-continuation).

Type-preservation of nextNode directly implies a coinductive safety property on ITrees: safe_itree denotes a type-safe ITree, which cannot produce a type error no matter the environment response(s) supplied to it.

Definition 2.5. Type-safety for ITrees.

$$\frac{r \neq \textbf{error}}{\mathsf{safe\_itree}\ (\mathsf{Ret}\ r)} \qquad \frac{}{\mathsf{safe\_itree}\ \mathsf{Div}} \qquad \frac{\forall r.\ \mathsf{safe\_itree}\ (k\ r)}{\mathsf{safe\_itree}\ (\mathsf{Vis}\ e\ k)}$$

However, we must transport safety from typing expressions to compiler expressions. Using our equational theory (§ 2.4), we show that any compiler expression *ce* is equivalent to its injection into typing expressions (tcexp_of) after desugaring:

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

We know that equivalence implies equal ITrees, and so equal safety properties. This equivalence holds because desugaring both exp_of *ce* and tcexp_of *ce* produces nearly identical semantics expressions, except **proj**s on the left become **safeproj**s on the right. Each such **proj**/**safeproj** is guarded by **eq**$_?$ (definition 2.3, pg. 30), and in this context they have equal semantics. The precondition wf$_?$ asserts that expressions are syntactically well-formed (*e.g.*, no empty **case**-statements), and is guaranteed by well-typing of tcexp_of *ce*.

Finally, type soundness states that the injection of a compiler expression into a well-typed typing expression induces a safe semantics.

Theorem 2.6. Type soundness.

$$\vdash \mathsf{cnsOK}_?\ cns\ \wedge\ \varnothing;\ \varnothing\ \vdash_{cns} \mathsf{tcexp\_of}\ ce : \mathbf{IO}\ \tau \Longrightarrow \mathsf{safe\_itree}\ [\![\ \mathsf{exp\_of}\ ce\ ]\!]$$

Empty type environment and store typing imply that *ce* is closed and does not mention any store location l (though it may allocate fresh ones). The type of tcexp_of *ce* must be monadic: recall that the entry point to a PureLang program is a definition `main :: IO ()`. Note that the indirection through typing expressions *tce* is just a proof technique; they do not appear in the compiler implementation.

Chapter 3

Compiler front end

In this chapter, I describe the front end of the PureCake compiler. Its indentation-sensitive parsing expression grammar (§ 3.1) is inspired by Adams [2013] and the CakeML parser. Binding group analysis sorts top-level functions to minimise mutual recursion (§ 3.2). Verified type inference relies on a non-standard, two-phase algorithm: first all typing constraints are generated, then later solved (§ 3.3). Demand analysis (§ 3.4) uses `seq` to enforce eager evaluation of expressions whose values are required.

3.1 Parsing expression grammar (PEG) parsing

In this section, I describe work completed by Michael Norrish.

Parsing converts textual source files into PureLang compiler expressions *ce* (figure 2.2, pg. 20). To handle Haskell-like indentation-sensitivity, we marry CakeML's parser with prior work on indentation-sensitive context-free grammars [Adams 2013] to produce a novel indentation-sensitive parsing expression grammar.

Prior work: CakeML's parser

CakeML's parser is based on a formalisation of parsing expression grammars in Coq [Koprowski and Binsztok 2011]. PEGs closely resemble context-free grammars (CFGs), but are unambiguous by construction. The right-hand sides of their production rules consist of *parsing expressions*, which enforce ordered choice and greedy repetition (*i.e.*, longest repetition). Negation[1] effectively permits unlimited look-ahead and backtracking: a negation-guarded expression is tested without consuming input. This lack of ambiguity makes PEGs inherently executable, and we can define the semantics of a given PEG as the execution of a deterministic state transition system on an input string: either it

[1]Conjunction is often considered, but can be implemented in terms of negation.

terminates to accept/reject the string, or it diverges due to left-recursion. Koprowski and Binsztok [2011] further verify a simple syntactic well-formedness check to guarantee totality: well-formed PEGs have no left-recursion and so their semantics must terminate on all input strings.

CakeML ports this work to HOL4, implementing a stack-based tail-recursive parser which precisely executes PEG semantics for well-formed PEGs. CakeML's PEG is proved well-formed, and verified sound and complete with respect to CakeML's official CFG: successful PEG parses have corresponding CFG parse trees, and each CFG parse tree can be found by PEG parsing. Together with totality of well-formed PEGs, this implies that CakeML's CFG is unambiguous and is implemented by its PEG parsing.

Prior work: indentation-sensitive context-free-grammars

Adams [2013] augments traditional CFG productions ($N \rightarrow X_1 \ldots X_n$) with *indentation relations*: each symbol X_i on the right-hand side is annotated to describe how its indentation relates to that of the non-terminal N.

The indentation of a terminal is simply its column number; the indentation of a non-terminal does not necessarily have a physical meaning, but often indentation-sensitive CFGs are written so that it refers to the leftmost column in which any token of the fully-parsed non-terminal appears. For practical languages, it often suffices to consider only the four relations below.

$$\text{for production } N \rightarrow X_1^{\mathcal{R}_1}\ X_2^{\mathcal{R}_2}\ \ldots\ \ X_n^{\mathcal{R}_n}$$
$$\forall i.\ \mathcal{R}_i \in \{=,\ >,\ \geq,\ \circledast\}\ \wedge\ (\text{indentation } X_i)\ \mathcal{R}_i\ (\text{indentation } N)$$
$$\text{where } =,\ >,\ \geq \text{ are standard and } \forall\, x\ y.\ x \circledast y \text{ (\textit{i.e.}, a universal relation)}$$

This work: indentation-sensitive parsing expression grammars

We augment CakeML's parsing expressions with the indentation relations above. The resulting PEG semantics tracks an *indentation predicate*, the possible indentations of the current non-terminal. This is represented symbolically as one of four forms: a closed interval $[i \ldots j]$, a lower-bounded set $[i \ldots]$, anywhere ($\mathbb{N}$), or nowhere ($\varnothing$). The latter effectively indicates an indentation error.

Intuitively, indentation predicates are used to define PEG semantics as follows. Let N be the current non-terminal with production $N \rightarrow \ldots\ X_i^{\mathcal{R}_i}\ \ldots$, and P_N be the current indentation predicate. When starting the execution of the production for X_i, we initialise its predicate P_{X_i} by considering the possible indentations permitted by the combination of relation $\mathcal{R}_i$ and predicate P_N (*i.e.*, the parent production's predicate). When ending the execution of the production for X_i, we update P_N by composing it with the resulting

P_{X_i} and check it for validity (*i.e.*, ensure the composition does not return $\varnothing$).

As in CakeML, we implement a tail-recursive parser which executes the indentation-sensitive semantics. We define a well-formed PEG for PureLang, ensuring our parser terminates on all inputs.

Abstract syntax to compiler expressions. After indentation-sensitive parsing, transforming AST to compiler expressions *ce* is mostly standard. The definitions of each PureLang program (one of which must be `main`) are combined into a single **letrec**: **letrec** def_1; ... ; def_n **in** `main`. Data type declarations (using `data`, only at the top-level) are passed separately to type inference (§ 3.3); the compiler only needs them in the last stage of compilation to CakeML (§ 5).

Future directions. Unlike CakeML, PureLang has no CFG to verify our PEG against. Testing shows that we accept many well-formed Haskell-like programs (§ 6.1), but specifying a CFG would provide stronger guarantees. Traditional PEGs should be equivalent to indentation-sensitive ones with always-universal (⊛) indentations, but we have not verified this.

3.2 Binding group analysis

Binding group analysis transforms a PureLang program from a single **letrec**-statement to nested **let**-/**letrec**-statements, partitioning bindings into minimally mutually recursive groups without changing behaviour. This simplifies both type inference (§ 3.3) and future optimisation passes.

To preserve semantics, bindings must be nested so that the variables on which each definition depends are either: defined higher up in the nesting; or defined at the same time in mutual recursion. In other words, no definition should depend on variables which are defined deeper in the nested structure. For example, in **letrec** ... $x = e_x$; ... ; $y = e_y$ ... **in** ..., the binding $y = e_y$ can only be moved above $x = e_x$ if e_y does not depend on x. At best, the two can appear in mutual recursion (as they are currently).

Immediate dependencies ($-$ uses $-$) are apparent from free variables, and transitive closure ($-$ uses$^+$ $-$) gives all dependencies. Dependencies induce a partial order:

$$x \text{ uses } y \stackrel{\text{def}}{=} y \in \text{freevars } e_x \qquad x \leq y \stackrel{\text{def}}{=} y \text{ uses}^+ x \qquad x = y \stackrel{\text{def}}{=} x \leq y \wedge y \leq x$$

Binding group analysis is almost akin to a standard topological graph sort, where definitions are nodes in the graph and the partial order provides directed edges. If this graph is acyclic, topological sort produces a sequence of appropriately ordered individual bindings. However, mutual recursion produces cycles in the graph; so we

employ a *pseudo*-topological sort to produce a sequence of binding *groups*. Bindings appear in groups which are no earlier than those of their dependencies, and appearance in the same group represents mutual recursion.

The overall pass can be expressed as the following pseudocode:

$$\begin{aligned}\mathsf{bindingGroups}\ (\mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ e) \stackrel{\text{def}}{=}\ & \mathsf{let}\ \mathcal{G} = \bigcup_m \{x_m \to x_j \mid x_j \in \mathsf{freevars}\ e_m\}\ \mathsf{in} \\ & \mathsf{let}\ \mathcal{G}^+ = \mathsf{transitiveClosure}\ \mathcal{G}\ \mathsf{in} \\ & \mathsf{let}\ \mathit{bindings} = \mathsf{pseudoTopological}\ \mathcal{G}^+\ \mathsf{in} \\ & \mathsf{nest}\ \mathit{bindings}\ e\end{aligned}$$

$$\begin{aligned}\mathsf{pseudoTopological}\ (\{x\} \cup \mathcal{G}) \stackrel{\text{def}}{=}\ & \mathsf{let}\ (lt, eq, gt) = \mathsf{partition}\ x\ \mathcal{G}\ \mathsf{in} \\ & \mathsf{pseudoTopological}\ lt +\!\!+ [eq] +\!\!+ \mathsf{pseudoTopological}\ gt\end{aligned}$$

$$\mathsf{nest}\ [\overline{x_n = e_n},\ \ldots,\ \overline{x_m = e_m}]\ e \stackrel{\text{def}}{=} \mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ \ldots\ \mathbf{in}\ \mathbf{letrec}\ \overline{x_m = e_m}\ \mathbf{in}\ e$$

A graph of immediate dependencies ($\mathcal{G}$) is derived from free variables, and then transitively closed. The resulting graph ($\mathcal{G}^+$) is three-way partitioned with respect to one of its bindings x to produce subgraphs of those bindings which are less, equal, and greater than x according to the dependency partial order. Recursion on lesser/greater bindings sorts them into binding groups, and in the final sequence the equal bindings form a single binding group. Nested **letrec**-statements are produced directly from this binding group sequence.

The initial binding group analysis occurs before type inference (§ 3.3), but we perform some post-processing after type inference: we delete unused bindings, and convert non-recursive, singleton **letrec**-bindings to **let**. We perform these steps together *after* type inference to ensure that dead code is still type-checked. The following rules illustrate this clean-up:

$$\frac{\overline{x_n} \notin \mathsf{freevars}\ e}{\mathbf{letrec}\ \overline{x_n = e_n}\ \mathbf{in}\ e \xrightarrow{\text{clean}} e} \qquad \frac{x \notin \mathsf{freevars}\ e}{\mathbf{letrec}\ x = e\ \mathbf{in}\ e' \xrightarrow{\text{clean}} \mathbf{let}\ x = e\ \mathbf{in}\ e'}$$

Implementation details. Before binding group analysis we remove shadowed bindings: if x is defined twice within a single **letrec**, the first definition is not accessible and can be safely deleted. Our implementation of pseudoTopological is actually tail-recursive, but presented otherwise here for simplicity. Graphs are represented using HOL4's functional data structure for gappy arrays: a dependency graph is an array of arrays of natural numbers, *i.e.*, each binding is assigned an index, at which an array of its dependencies are stored. Therefore, we carry mappings to convert between bindings and natural numbers.

Breadth-first search gives transitive closures of graphs, by mapping a fold operation over the tree-like structure of the functional array.

Verification

We prove this pass sound entirely within our equational theory (§ 2.4).

First, we verify the pseudo-topological sort in stages. It is straightforward to show that transitiveClosure correctly computes transitive closures, and that partition produces correct partitions. Correctness of pseudoTopological follows from correctness of partition:

$$\vdash \mathsf{pseudoTopological}\ \mathcal{G} = bs +\!\!+ [y_1, \ldots, y_n] +\!\!+ \ldots \Rightarrow$$
$$(\forall x.\ x \in bs \iff x <_{\mathcal{G}} y_n) \wedge (\forall y_m\ y_l.\ y_m =_{\mathcal{G}} y_l)$$

That is, bindings sequenced before the $\overline{y_n}$ must be lesser according to the partial order represented by the graph $\mathcal{G}$; all the $\overline{y_n}$ must be equal.

To prove expression equivalence, we define a valid split of recursive bindings $\overline{x_n = e_n}$ into $\overline{y_m = e_{1\,m}}$ and $\overline{w_j = e_{2\,j}}$ such that all $\overline{e_{1\,m}}$ do not depend on any $\overline{w_j}$. We construct a syntactic relation which can validly split **letrec**s and otherwise recurses through PureLang syntax; we show that this is an applicative simulation for closed expressions and lift this to expression equivalence by closing substitution.

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

All that remains is to show that pseudo-topological sort produces nested valid splits, which is straightforward given its correctness proof.

3.3 Constraint-based type inference

In this section, I briefly review Top, a Hindley-Milner constraint-based type inference framework, and verify PureCake's proof-of-concept implementation of Top to demonstrate its applicability to verified compilation. Top is used in the Helium teaching compiler [Heeren 2005; Heeren et al. 2003]: Helium's open-source, near-complete implementation of Haskell 98 provides a roadmap to port its features to PureCake in future work. However, such constraint-based type inference has never been mechanically verified.

Hindley-Milner type systems are popular due to their decidable type inference, but well-studied algorithms $\mathcal{W}$, $\mathcal{J}$, and $\mathcal{M}$ [Lee and Yi 1998] are notorious for impenetrable type errors. Their fixed-order interleaving of AST traversal and type unification lead

to known weaknesses: type errors are often detected far from the code which causes them and typically towards the end of a program; error reporting is restricted by the order in which types are unified. Therefore, error messages become difficult to understand without significant experience or knowledge of the inference algorithm. $\mathcal{M}$ improves on $\mathcal{W}$ with better localisation of type errors by recording the expected type of the current expression according to its context, but is still subject to ordering biases. Constraint-based type inference has been proposed as a generalisation of these algorithms [Heeren et al. 2022] which enables clear, precise error messages. Top avoids inherent biases in standard algorithms by operating in two phases: a single AST traversal collects typing constraints, which are only solved post-traversal. This permits flexible application of solving strategies and heuristics; algorithms $\mathcal{W}$ and $\mathcal{M}$ are equivalent to predefined solving strategies within Top. HM(X) [Odersky et al. 1999] is another example of constraint-based inference, generic over a signature X to permit proof of soundness for a variety of ML-like type systems at once. This generality is unnecessary for PureCake, so we focus on the Haskell-tailored Top and its well-tested implementation in Helium.

A full discussion of the benefits of Top and comparison with related approaches can be found in prior work [Heeren 2005; Heeren et al. 2022]. In our setting, separating implementation of constraint generation and solving also modularises verification: solving more efficiently or with a different strategy does not require significant re-verification of constraint generation (and *vice versa*). In the remainder of this section, I provide an intuitive overview of Top's constraints, and summarise our verification. I assume familiarity with Hindley-Milner type systems: unification, parametric polymorphism, type schemes, and type scheme specialisation.

Hindley-Milner type systems introduce parametric polymorphism with rule HMLet below, which becomes unsound without the restriction that free variables in the typing context cannot be generalised: these are *monomorphic*, representing a constant but unknown type, as opposed to *polymorphic* variables which can represent any type. Traditional one-pass algorithms first infer the type of ce_1 fully, generalise it with respect to monomorphic type variables, and then use the result to infer the type of ce_2.

$$\frac{\Gamma \vdash_{\text{HM}} ce_1 : \tau_1 \qquad \overline{\alpha_n} \notin \Gamma \qquad \Gamma,\ x : \forall\, \overline{\alpha_n}.\, \tau_1 \vdash_{\text{HM}} ce_2 : \tau_2}{\Gamma \vdash_{\text{HM}} \textbf{let}\, x = ce_1\ \textbf{in}\ ce_2 : \tau_2}\ \text{HMLet}$$

Two-phase inference must generate constraints for ce_2 *without* type information for ce_1. Later, during solving, it must soundly generalise τ_1 with respect to the relevant monomorphic type variables. Both Top and HM(X) generate expressive constraints that "remember" monomorphic type variables for solving.

The constraint generation judgement below reads "for monomorphic type variables M, ce has type τ in constructor environment cns subject to assumptions A and constraints C". Here, M is a set of monomorphic type variables. Assumptions A map term variables to sets of types: we will see that the types in $A(x)$ are those assigned to x when traversing ce, and we write $x : \tau \in A$ to mean $\tau \in A(x)$. Constraint sets C contain unification and *implicit instance* constraints. Each implicit instance constraint carries a set of monomorphic type variables; we will soon see that this is how TOP soundly "remembers" monomorphic type variables for later use in solving.

$$\begin{aligned} \text{Constraint generation:} &\quad M \vdash ce : \tau \Rightarrow_{cns} A \,;\, C \\ \text{Assumptions:} &\quad A : x \to \tau \ \mathsf{set} \\ \text{Constraints:} &\quad c ::= \tau_1 \equiv \tau_2 \text{ (unification)} \mid \tau_1 \preceq_M \tau_2 \text{ (implicit instance)} \end{aligned}$$

Figure 3.1 shows selected constraint generation rules. Constraint generation is bottom-up except for management of monomorphic variables M. A free term variable $\mathbf{var}\, x$ is assigned a fresh type variable, which is recorded in the assumptions A (TOPVAR). This assumption will bubble up to the binding which introduces x: as each use of x must share the same type, we remove assumptions on x and unify all the usage types (*e.g.*, TOPLAM). Note that assumptions are created only by TOPVAR, so each $x : \tau \in A$ represents a single usage of x in ce and τ is always a fresh type variable. Rule TOPLAM augments monomorphic type variables top-down: fresh types $\overline{\alpha_n}$ are introduced for the λ-abstracted $\overline{x_n}$, which are monomorphic within the subexpression ce.

The rule TOPLET records the "current" monomorphic type variables in implicit instance constraints. Intuitively, each usage of x in ce_2 produces an assumption $x : \tau$ in A_2, and each such τ must soundly specialise the scheme obtained by generalising τ_1, the type of ce_1. The semantics of implicit instance constraints records this precisely: each τ solves to some τ' which specialises the scheme obtained by sound generalisation of τ_1' (the solved version of τ_1) with respect to the free type variables in M' (the solved version of M). This imposes a restriction on constraint solving: implicit instance constraints $\tau_1 \preceq_M \tau_2$ can only be solved once τ_2's generalisable variables ($\mathsf{freevars}(\tau_2) - M$) are stable (*i.e.*, cannot be changed by solving any further constraints). This will depend on the variables free in other constraints. This restriction is inherent to Hindley-Milner type systems; TOP permits maximum possible flexibility in solving order.

The remaining rules follow from the intuition captured in these three; a λ-application $ce_1 \cdot ce_2$ also produces a unification constraint $\tau_1 \equiv \tau_2 \to \alpha$ (where α is fresh): the type of ce_1 must be a function whose argument type matches the type of ce_2.

A simple solving strategy produces a substitution consistent with constraint semantics (when successful). Each constraint is solved in turn, and the remainder updated by the

$$\frac{}{M \vdash \mathbf{var}\, x : \alpha \;\Rightarrow_{cns}\; [x : \alpha]\;;\; \varnothing}\;\textsc{TopVar}$$

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

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

Figure 3.1. Selected Top constraint generation rules.

resulting substitution. Unification constraints $\tau_1 \equiv \tau_2$ are solved by unifying τ_1 and τ_2. Implicit instance constraints are solved once τ_2's generalisable variables are stable; then these variables are freshened to produce τ_2' which is unified with τ_1.

3.3.1 Implementation and verification details

We specify Top's constraint generation rules and prove their soundness with respect to PureLang's type system (§ 2.5). Existence of a type substitution s consistent with the generated constraints (*i.e.*, one which solves them) implies well-typing in a context which generalises the assumed types of free term variables. Informally:

Lemma 3.1. Soundness of constraint generation.

$$\begin{aligned}\vdash\; & M \vdash ce : \tau \;\Rightarrow_{cns}\; A\;;\; C \;\wedge\; s \;\mathsf{solves}\; C \;\wedge\; \mathsf{cnsOK?}\; cns \;\wedge \\ & \big(\forall\, x : \tau \in A.\; s\Gamma(x) \;\mathsf{generalises}\; s\tau\big) \;\wedge\; \mathsf{freetyvars}\,(sM) = \mathsf{freetyvars}\,(s\Gamma) \\ & \quad\Rightarrow\; \Gamma;\; \varnothing \;\vdash_{cns} \mathsf{tcexp\_of}\, ce : \tau\end{aligned}$$

The set of solved monomorphic variables must equal the set of free type variables in the typing context to ensure that generalisation of free type variables agrees in both systems. In reality, we prove a more verbose theorem which reconciles nameless de Bruijn indices (used to formalise typing rules naturally) and unification variables (used to formalise inference). In particular, our theorem is stated in terms of substitutions which close types with respect to unification variables, leaving only de Bruijn indices. Its proof handles the permeative bookkeeping associated with these closing substitutions, and the interactions between substitutions of unification variables and de Bruijn indices. Some of this bookkeeping is required in CakeML too, but the explicit equality above between free unification variables in M and de Bruijn indices in Γ is particularly problematic.

Constraint generation is implemented within a state-exception monad which provides fresh unification variables, and we prove that successful algorithm outputs are sound

with respect to TOP rules. To provide unification, we instantiate HOL4's reusable triangular unification [Kumar and Norrish 2010] to PURELANG types. A top-level function infer implements a simple syntactic check which establishes $\mathsf{cnsOK}_?$, before composing constraint generation and solving. We prove it sound by first verifying that successful solving produces consistent substitutions, and then composing with lemma 3.1.

$$\vdash \mathsf{solve}\ C = \mathsf{OK}\ s \Longrightarrow s\ \mathsf{solves}\ C$$

$$\vdash \mathsf{infer}_{cns}\ ce = \mathsf{OK} \Longrightarrow \exists \tau.\ \emptyset;\ \emptyset\ \vdash_{cns} \mathsf{tcexp\_of}\ ce : \mathbf{IO}\ \tau$$

In other words, all programs accepted by infer must be well-typed and so type-safe by type soundness (theorem 2.6, pg. 38). All ill-typed programs are therefore rejected. Note that we do not prove completeness of type inference, nor do we intend to. PURELANG aims to be a 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 (§ 6.1).

3.3.2 Future directions

Our implementation of TOP is only a proof-of-concept, demonstrating its applicability to mechanised compiler verification. We have not yet implemented enforced type signatures, and both generation and solving are simplistic, lacking the orderable constraint trees that Helium uses to store its constraints or the type graphs that permit completely unbiased solving and precise error messages. Modular verification of generation and solving will allow us to incorporate these gradually.

To simplify proofs, we could use only named representations of type variables in typing rules (*i.e.*, converting de Bruijn indices into unification variables). Currently types are first-order only (quantification only at the top-level in type schemes), so variable capture is minimised. However, this could hinder future efforts to add higher-order types with local type quantification. Further afield, we intend to support type classes and modules (§ 6.3).

3.4 Demand analysis

In this section, I describe work completed mostly by Samuel Vivien.

Compilation of lazy code to a strict language produces *thunk*s: suspended computations on the heap. However, excessive laziness can lead to well-known bottlenecks on space usage. Consider the factorial function from figure 2.1 (pg. 19), expressed below as a

pseudo-semantic expression:

$$\textbf{letrec}\ \mathit{fact}\ \mathit{acc}\ x = \textbf{if}\ \textbf{var}\, x\ <\ 2\ \textbf{then}\ \textbf{var}\, \mathit{acc}\ \textbf{else}\ (\textbf{var}\, \mathit{fact}) \cdot (\textbf{var}\, \mathit{acc} \times \textbf{var}\, x) \cdot (\textbf{var}\, x - 1)$$

This has space complexity of $\Theta(n)$: each recursive call allocates a suspended computation of $\textbf{var}\, x \times \textbf{var}\, \mathit{acc}$ as a thunk, which is not evaluated until inspected during the final recursive call. However, *acc* can be evaluated eagerly without changing program behaviour, reducing space complexity to $O(1)$. The goal of demand analysis is to detect this and annotate code using **seq** (*i.e.*, $\textbf{letrec}\ \mathit{fact}\ x\ \mathit{acc} = \textbf{seq}\ (\textbf{var}\, \mathit{acc})\ \ldots$), allowing future optimisations to minimise such unnecessary laziness.

Our strategy is as follows. First, we formalise a semantic notion of *demands*: expression e demands variables $\overline{x_n}$ if eager evaluation of the variables does not change e's semantics. Second, we derive *propagation rules* which describe sound ways to propagate demands through PureLang expressions. Third, we implement an analysis function which traverses expressions bottom-up: it gathers and propagates demands according to the propagation rules, and transforms code by inserting **seq** annotations for demanded variables.

More precisely, we define demands as an equivalence between an expression e and its prefixing by the eager evaluation of variables $\overline{x_n}$ using **seq**:

$$e\ \mathsf{demands}\ \overline{x_n} \stackrel{\text{def}}{=} e \approx \mathsf{seqs}\ \overline{\textbf{var}\, x_n}\ e \qquad \mathsf{seqs}\ \overline{e_n}\ e \stackrel{\text{def}}{=} \textbf{seq}\ e_1\ \left(\textbf{seq}\ e_2\ \ldots (\textbf{seq}\ e_n\ e)\ldots\right)$$

This definition justifies sound insertion of **seq** annotations by construction. Following prior work [Sergey et al. 2014], it is phrased using $\approx$, a weaker variant of our equational theory ($\cong$, § 2.4) which does not distinguish between runtime type errors and silent divergence. In particular, for $\approx$ the final line in definition 2.4 (pg. 31) is removed.

Figure 3.2 shows some non-trivial propagation rules we have derived[2] from this definition: **seq** propagates all demands from its subexpressions; $\textbf{let}\, y = e_1\ \textbf{in}\ e_2$ inherits demands from e_2 except for the bound variable y, and from e_1 only if e_2 demands y; $\textbf{if}\, e\, \textbf{then}\, e_1\, \textbf{else}\, e_2$ inherits the demands of e and any demands shared by e_1 and e_2. The propagation rule for **seq** demonstrates the need for $\approx$ in the definition of demands: using $\cong$ instead prevents propagation of demands from e_2 due to the counterexample below (here, $\bot$/**fail** are any always-diverging/-crashing programs respectively).

$$[\![\ \textbf{let}\, x = \bot\ \textbf{in}\ \textbf{seq}\ \textbf{fail}\ (\textbf{var}\, x)\ ]\!] = \mathsf{Ret}\ \textbf{error}$$
$$[\![\ \textbf{let}\, x = \bot\ \textbf{in}\ \textbf{seq}\ (\textbf{var}\, x)\ (\textbf{seq}\ \textbf{fail}\ (\textbf{var}\, x))\ ]\!] = \mathsf{Div}$$

This propagation is essential: otherwise, each time we annotate some e_2 with some $\textbf{seq}\ e_1$, we would have to discard the demands of e_2.

[2]Note that there are infinitely many derivable rules; in practice, we rely on a finite number.

$$\frac{e_1 \text{ demands } \overline{x_n} \qquad e_2 \text{ demands } \overline{y_n}}{(\mathbf{seq}\, e_1\, e_2) \text{ demands } (\overline{x_n} \cup \overline{y_n})} \qquad \frac{e_2 \text{ demands } \overline{x_n}}{(\mathbf{let}\, y = e_1 \ \mathbf{in}\ e_2) \text{ demands } (\overline{x_n} \setminus y)}$$

$$\frac{e_1 \text{ demands } \overline{x_n} \qquad e_2 \text{ demands } y}{(\mathbf{let}\, y = e_1 \ \mathbf{in}\ e_2) \text{ demands } \overline{x_n}} \qquad \frac{e \text{ demands } \overline{w_n} \qquad e_1 \text{ demands } \overline{x_n} \qquad e_2 \text{ demands } \overline{y_n}}{(\mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) \text{ demands } \left(\overline{w_n} \cup (\overline{x_n} \cap \overline{y_n})\right)}$$

Figure 3.2. Derived propagation rules for demands.

To prove that demand analysis preserves semantics, we must further demonstrate that it does not convert between diverging and crashing programs, as $\approx$ (and so demands) cannot distinguish these. Fortunately, demand analysis receives only well-typed programs from type inference (§ 3.3), which cannot crash by type soundness (theorem 2.6, pg. 38). A simple syntactic proof shows that annotating with **seq** preserves well-typing:

$$\vdash\ x \in \text{freevars } ce\ \wedge\ \Gamma;\ \Sigma\ \vdash_{cns} ce : \tau\ \Rightarrow\ \Gamma;\ \Sigma\ \vdash_{cns} \mathbf{seq}\ (\mathbf{var}\, x)\ ce : \tau$$

Therefore, demand analysis preserves well-typing too. This is sufficient to recover semantics preservation, despite use of the weaker $\approx$-equivalence.

Further proof rules

To propagate more demands for idiomatic functional code, we define *function demands* (demandsF) and *application demands* (demandsA):

$$e\ \text{demandsF}^n_m \stackrel{\text{def}}{=} \forall\, \overline{e_m}\ \overline{x_j}\,.\ e_n \text{ demands } \overline{x_j}\ \Rightarrow\ (e \cdot \overline{e_m}) \text{ demands } \overline{x_j}$$
$$e\ \text{demandsA}_m\ \overline{x_j} \stackrel{\text{def}}{=} \forall\, \overline{e_m}.\ (e \cdot \overline{e_m}) \text{ demands } \overline{x_j}$$

The former indicates that the application of e to m arguments will demand the nth argument, the latter that the application will demand $\overline{x_j}$. Using the derived rules below, we can propagate demands from bodies of functions to their call-sites: λ-abstractions can produce function demands, and λ-applications can produce application demands.

$$\frac{e\ \text{demandsA}_m\ x}{(\lambda x.\ e)\ \text{demandsF}^0_{m+1}}\ \textsc{demandsF} \qquad \frac{e_1\ \text{demandsF}^0_{m+1} \qquad e_2 \text{ demands } \overline{x_n}}{(e_1 \cdot e_2)\ \text{demandsA}_m\ \overline{x_n}}\ \textsc{demandsA}$$

$$e \text{ demands } \overline{x_n} \iff e\ \text{demandsA}_0\ \overline{x_n} \qquad \Leftrightarrow\text{-}\textsc{demandsA}$$

That is, if e demands x when applied to m arguments, then $\lambda x.\ e$ demands its first argument when applied to $m+1$ arguments (DEMANDSF). Conversely, if e_1 demands its first

argument when applied to $m+1$ arguments, its λ-application to argument e_2 inherits demands from e_2 when applied to m arguments (DEMANDSA). Finally, we can convert between regular demands and application demands ($\Leftrightarrow$-DEMANDSA). For example, we can recover the demand of y by the simple expression $(\lambda x.\ \mathbf{var}\, x) \cdot \mathbf{var}\, y$ as follows:

$$
\begin{aligned}
(\mathbf{var}\, x)\ \mathsf{demands}\, x &\Rightarrow (\mathbf{var}\, x)\ \mathsf{demandsA}_0\ x && (\Leftrightarrow\text{-DEMANDSA}) \\
&\Rightarrow (\lambda x.\ \mathbf{var}\, x)\ \mathsf{demandsF}^0_1 && (\text{DEMANDSF}) \\
&\Rightarrow ((\lambda x.\ \mathbf{var}\, x) \cdot \mathbf{var}\, y)\ \mathsf{demandsA}_0\ y && (\text{DEMANDSA}) \\
&\Rightarrow ((\lambda x.\ \mathbf{var}\, x) \cdot \mathbf{var}\, y)\ \mathsf{demands}\, y && (\Leftrightarrow\text{-DEMANDSA})
\end{aligned}
$$

To handle such expressions, our analysis function gathers and propagates demands, function demands, and application demands separately.

Recursive functions. We have derived propagation rules for recursive functions which are closed, allowing demand analysis to annotate functions such as *fact* at the top of this section. Considering the singleton binding $\mathbf{letrec} f = \lambda\, \overline{x_n}\, .\ ce\ \mathbf{in}\ ce'$ for simplicity, our analysis begins by assuming that all of the arguments $\overline{x_n}$ are demanded in body *ce*. We then iterate over *ce* a fixed number of times, discounting false demands until we reach a stable set of true demands $\overline{x_j}$ (or run out of iterations and give up). We can transform the recursive binding to evaluate its true demands eagerly ($f = \lambda \overline{x_n}.\ \mathsf{seqs}\ \overline{x_j}\ \ldots$), and transform calls in ce': each call $f \cdot \overline{\mathbf{var}\, y_n}$ becomes $\mathsf{seqs}\ \overline{\mathbf{var}\, y_j}\ (f \cdot \overline{\mathbf{var}\, y_n})$. Note that this effectively requires A-normal form uses of f: all function arguments must be variables [Sabry and Felleisen 1992].

Future directions. Our current implementation is naïve, inserting **seq** annotations wherever the propagation rules permit. However, the separation between implementation of the analysis and the propagation rules which justify it mirrors the approach described later in § 4.1.[3] This allows us to reduce aggressiveness using heuristics without incurring significant proof overhead. For now, we have disabled **seq**-insertion at call sites to work around poor performance. In future work, we intend to analyse pattern matches and constructors to propagate their demands, and optimise away inserted **seq**s.

[3]Strictly, the approach in § 4.1 involves *syntactic* relations, but demands are defined *semantically*. However, the key resemblance lies in the separation between implementation and verification.

Chapter 4

Compiler back end

In this chapter, I describe the PureCake compiler back end and its three intermediate languages: ThunkLang, EnvLang, and StateLang.

As in other verified compilers, use of several intermediate languages permits tractable verification by isolating significant code transformations. Each intermediate language of the PureCake compiler is designed with such a code transformation in mind. At a high level, 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 monadic operations as stateful primitives and shares thunk values statefully. This last transformation neatly introduces efficient lazy evaluation.

Figure 4.1 illustrates the structure of the PureCake compiler. We have already seen the compiler front end, those parts above the dotted line (§ 3). After a review of our general approach to compiler proofs (§ 4.1), I describe each intermediate language of the back end and the verification of its passes in turn (those parts below the dotted line, §§ 4.2 to 4.4 respectively).

4.1 Method: verify compiler *relations*, not *functions*

PureCake verifies its back end with a different methodology to that of CakeML. This section first describes the approach, and then justifies it.

Approach

In CakeML, verification of back end compiler passes is tied to their implementation: concrete compiler *functions* are directly proven semantics-preserving [Tan et al. 2019, §2.3]. Informally, any original expression e that satisfies some preconditions has equivalent

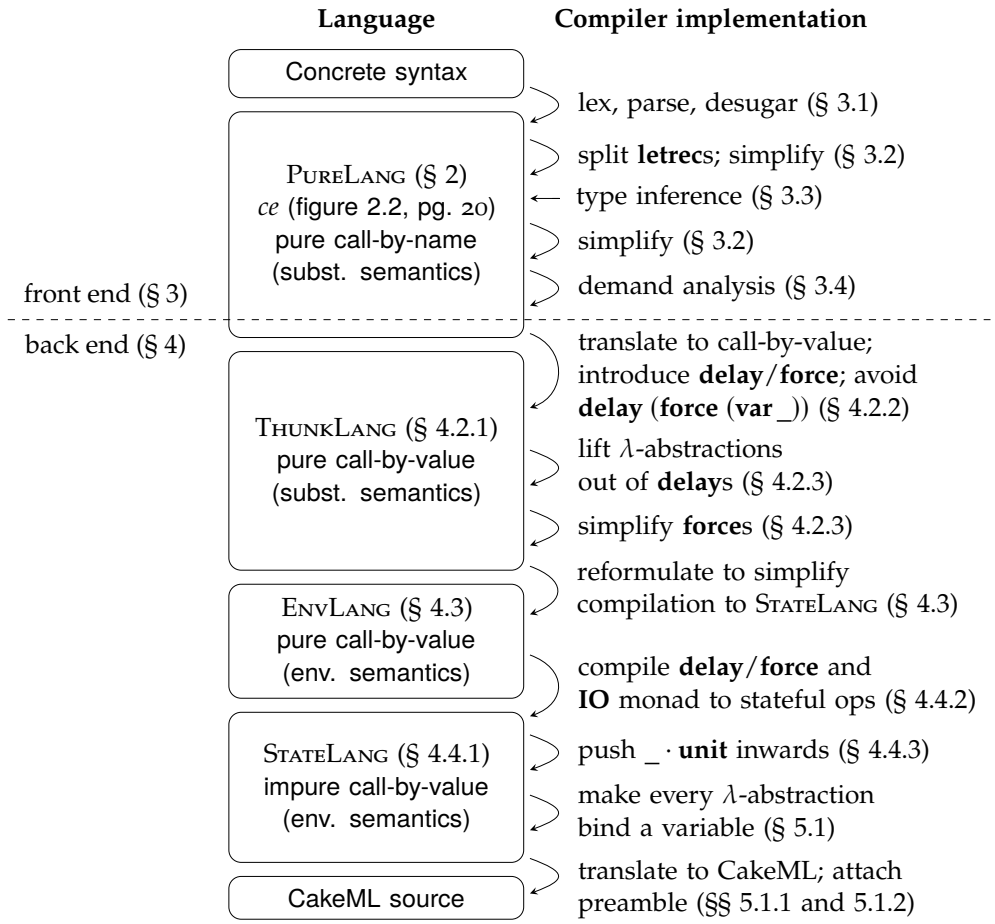


Figure 4.1. High-level summary of the compiler's intermediate languages and compilation passes.

semantics to the transformed expression, which satisfies some postconditions:

$$\mathsf{preconditions}\ e \implies [\![\ e\ ]\!] = [\![\ \mathsf{transform}\ e\ ]\!] \wedge \mathsf{postconditions}\ (\mathsf{transform}\ e) \tag{4.1}$$

In PureCake, we separate verification of back end compiler passes from their implementations. We first define syntactic *relations* $-\mathcal{R}-$ over semantic expressions, which characterise the code transformations of each pass. We verify that each such relation is semantics-preserving; in other words, any transformation mapping e to e' and satisfying $e\ \mathcal{R}\ e'$ must give rise to equal observational semantics:

$$e\ \mathcal{R}\ e' \wedge \mathsf{safe\_itree}\ [\![\ e\ ]\!]_{\text{source}} \wedge \mathsf{closed}\ e \implies [\![\ e\ ]\!]_{\text{source}} = [\![\ e'\ ]\!]_{\text{target}} \tag{4.2}$$

Intuitively, each such relation specifies an envelope of possible semantics-preserving implementations. As is common, this correctness theorem assumes that the source expression e never fails (safe_itree, definition 2.5, pg. 38) and is closed, *i.e.*, it is a whole program. We prove these theorems in three stages: one simulation proof per layer of our three-layered semantics (§ 2.3). The uppermost simulation proof produces the equality between ITrees: $[\![-]\!]_{\text{source}} = [\![-]\!]_{\text{target}}$. The subscripts "source" and "target" are each one of "pure", "thunk", "env", or "state": when compiling between intermediate languages the two differ, for intra-language compilation they are the same.

The lowermost simulation concerns the functional big-step style evaluation, and this is usually the most involved. In general, we prove *either* a forward *or* a backward simulation. A forward simulation is shown below: if evaluation of expression e in the source language successfully produces a result r within n steps, then any related expression e' must produce a result r' after evaluation for at least n steps (m s.t. $m \geq n$) in the target language. The results must continue to be related by some $\mathcal{R}_{\text{res}}$, which effectively lifts $\mathcal{R}$ to the relevant result type (weak-head normal forms in the case of PureLang), ensuring that timeouts and runtime type errors match on both sides.

$$e\ \mathcal{R}\ e'\ \wedge\ \mathsf{eval}^{n}_{\text{source}}\ e = r \neq \mathbf{error} \implies \exists\, m \geq n,\ r'.\ \mathsf{eval}^{m}_{\text{target}}\ e' = r'\ \wedge\ r\ \mathcal{R}_{\text{res}}\ r' \tag{4.3}$$

Intuitively, for each step the source takes, the target can take one or more steps to re-establish the relation between source and target. This "one or more" requirement is necessary to prove divergence-preservation: when the source times out for all n, the target must also time out for all m. No such step-counting is needed to prove preservation of terminating behaviours. Backward simulation is similar, but with the "source" and "target" subscripts reversed (though, the requirement for no type errors remains on the source or is omitted). Therefore, each compiler pass must monotonically increase (for forward simulation) or monotonically decrease (for backward simulation) the number

of steps taken by the semantics, and $\mathcal{R}$ must be such that stateful interpretation of $\mathcal{R}_{\text{res}}$-related results produces equivalent ITrees.

It then suffices to define concrete compiler functions over compiler expressions and prove that they inhabit the relevant relation envelopes. Again, these proofs can assume syntactic well-formedness preconditions and must establish new well-formedness postconditions for future passes:

$$\mathsf{wf}_?\ ce\ \Rightarrow\ (\mathsf{exp\_of}\ ce)\ \mathcal{R}\ \big(\mathsf{exp\_of}\ (\mathsf{compile}\ ce)\big)\ \wedge\ \mathsf{wf}'_?\ (\mathsf{compile}\ ce) \tag{4.4}$$

Straightforward composition of equations 4.2 and 4.4 provides the final correctness result below, which fits the form of equation 4.1.

$$\begin{aligned}&\mathsf{wf}_?\ ce\ \wedge\ \mathsf{safe\_itree}\ [\![\ \mathsf{exp\_of}\ ce\ ]\!]_{\text{source}}\ \wedge\ \mathsf{closed}\ ce\ \Rightarrow\\ &\quad [\![\ \mathsf{exp\_of}\ ce\ ]\!]_{\text{source}} = [\![\ \mathsf{exp\_of}\ (\mathsf{compile}\ ce)\ ]\!]_{\text{target}}\ \wedge\\ &\quad \mathsf{wf}'_?\ (\mathsf{compile}\ ce)\ \wedge\ \mathsf{closed}\ (\mathsf{compile}\ ce)\end{aligned} \tag{4.5}$$

When integrating the compiler function into the overall compilation pipeline, we must discharge the precondition for each pass using the postcondition of the previous one. The initial precondition is guaranteed by type inference (§ 3.3). Syntactic well-formedness conditions ($\mathsf{wf}_?$ and closed) are tedious but straightforward to prove, and safe_itree is a corollary of semantics-preservation from type-safe code.

Advantages

Though it seems long-winded, this two-phase approach has several benefits.

- Relations can impose syntactic restrictions on input code by narrowing their domains. By contrast, restricting total functions requires carrying invariants between proofs. Now, each relation can be verified orthogonally, and adding a new pass to the compiler is minimally impacted by the design choices of existing ones.
- Relations can remain high-level, avoiding concrete details such as calculating free/bound variables, strategies for inventing fresh names, *etc.* Where such a detail is required, it can simply be assumed to exist.
- Relations can be prototyped rapidly, due to the points above and their usage of simpler semantic expressions. This provides early feedback on the key correctness concerns of a code transformation before implementation details are considered.
- Complicated passes can be implemented as a single function but verified by composing several simpler relations, reducing proof complexity without sacrificing performance.

- Compiler functions can be modified without redoing their core verification. Syntactic relations capture *possible* compiler implementations: we are free to change implementation details and internal heuristics as long as we remain within the envelope of the syntactic relation(s). Then, we need only re-derive equation 4.4—not equation 4.2. The former syntactic proof is simpler than the latter semantic one.

In summary, this technique separates concerns: relation verification focuses on code transformations, and function verification focuses on implementation details and bookkeeping between compiler passes.

The remainder of this chapter describes each intermediate language in turn—their syntax, semantics, and optimisations. Here, the methodology of syntactic relations has an additional benefit: optimisations are most clearly presented by showing the syntactic relations used to verify them. I focus on these core code transformations instead of the mostly standard details concerning relation verification, compiler function implementation/verification, and integration into the compilation pipeline. I omit trivial rules defined by simple recursion over expression syntax, *e.g.* for λ-application, rules of the form: $e_1 \mathcal{R}\, e_1' \;\wedge\; e_2 \mathcal{R}\, e_2' \;\Rightarrow\; (e_1 \cdot e_2) \mathcal{R}\, (e_1' \cdot e_2')$. These simply permit the application of any non-trivial rules deep within expressions.

4.2 ThunkLang

ThunkLang is the first intermediate language of the PureCake compiler. It resembles PureLang closely, but is call-by-value and introduces new primitives for handling *thunks*, *i.e.*, delayed computations (§ 4.2.1). These primitives are necessary to model PureLang's call-by-name evaluation after compilation into ThunkLang (§ 4.2.2). After the initial compilation from PureLang, we perform two intra-language optimisation passes within ThunkLang (§ 4.2.3).

4.2.1 Syntax and semantics

Like PureLang, ThunkLang has two ASTs (*ce* for compilation, and *e* for semantics, where exp_of expands the former to the latter). It semantics is also specified in three layers (§ 2.3.3), but is call-by-value: weak-head forms *wh* become traditional values *v*, and $\mathsf{eval}^{j}_{\mathrm{wh}}\ e / \mathsf{eval}_{\mathrm{wh}}\ e$ become $\mathsf{eval}^{j}\ e / \mathsf{eval}\ e$ respectively. Values *v* and expressions *e* are mutually recursive data types: **lambda** *x e* is a function value, and **value** *v* wraps a value into an expression, so that the semantics of ThunkLang can specify substitution of values into expressions. I omit the first layer of ThunkLang semantics (clocked evaluation, $\mathsf{eval}^{j}\ e$), instead showing characterising equations for the second layer (unclocked evaluation, $\mathsf{eval}\ e$). Below are two derived equations which exemplify the change to call-by-value:

constructors are evaluated *deeply*, *i.e.*, their arguments are also inspected and evaluated; function application evaluates its argument before substituting in the resulting value. Here, mapM maps its function argument over its list argument, returning **error** if the resulting list contains an **error** or **diverge** if the resulting list contains **diverge** (in that order), and otherwise returning the resulting list.

$$\begin{array}{l}\vdash \mathsf{eval}\ (\mathbf{cons}\ cname[\,\overline{e_m}\,]) = \mathsf{do} \\ \quad \overline{v_m} \leftarrow \mathsf{mapM}\ \mathsf{eval}\ \overline{e_m} \\ \quad \mathbf{cons}_v\ cname[\,\overline{v_m}\,]\end{array} \qquad \begin{array}{l}\vdash \mathsf{eval}\ (e_1 \cdot e_2) = \mathsf{do} \\ \quad v \leftarrow \mathsf{eval}\ e_2; \\ \quad \mathbf{lambda}\ x\ e \leftarrow \mathsf{eval}\ e_1; \\ \quad \mathsf{eval}\ (e[{}^{\mathbf{value}\ v}/_x])\end{array}$$

Monadic operations are still evaluated *shallowly* to align with PureLang, and evaluating a top-level program still produces a monadic value. However, in ThunkLang stateful interpretation becomes call-by-value too: during interpretation, arguments to monadic operations are eagerly evaluated before considering sequencing and effects, and arrays store values rather than expressions.

ThunkLang also introduces thunk-handling primitives **delay** and **force**. In particular, **delay** delays computation of an expression by embedding it into a thunk; its counterpart **force** forces the evaluation of a previously-delayed expression.

$$\vdash \mathsf{eval}\ (\mathbf{delay}\ e) = \mathbf{thunk}\ e \qquad \begin{array}{l}\vdash \mathsf{eval}\ (\mathbf{force}\ e) = \mathsf{do} \\ \quad \mathbf{thunk}\ e' \leftarrow \mathsf{eval}\ e;\ \mathsf{eval}\ e'\end{array}$$

ThunkLang's evaluation is pure and stateless: each time a thunk is forced, the expression within is re-evaluated. Therefore, these thunks are not the lazy thunks of Haskell. We implement Haskell-flavoured sharing of thunk evaluations later, in StateLang (§ 4.4).

There are two more minor changes in ThunkLang: **case**-statements in compiler expressions pattern match on a variable, rather than a compiler expression; and **let**-statements are lifted to top-level semantic expressions (in contrast to PureLang's view of **let** as syntactic sugar). We can summarise the key syntax changes as follows:

$$\begin{array}{l}e ::= \\ \quad \mid \ldots \\ \quad \mid \mathbf{value}\ v \\ \quad \mid \mathbf{delay}\ e \\ \quad \mid \mathbf{force}\ e \\ \quad \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\end{array} \qquad \begin{array}{l}v ::= \\ \quad \mid \mathbf{thunk}\ e \\ \quad \mid \mathbf{cons}_v\ cname[\,\overline{v_n}\,] \\ \quad \mid \mathbf{monadic}_{\mathrm{wh}}\ mop[\,\overline{e_n}\,] \\ \quad \mid \mathbf{lambda}\ x\ e \\ \quad \mid \mathbf{lit}\ lit \\ \quad \mid \mathbf{error} \\ \quad \mid \mathbf{diverge}\end{array} \qquad \begin{array}{l}ce ::= \\ \quad \mid \ldots \\ \quad \mid \mathbf{delay}\ e \\ \quad \mid \mathbf{force}\ e \\ \quad \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{row_n}\end{array}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2'}{e_1 \cdot e_2 \xrightarrow{\text{thunk}} e_1' \cdot \mathbf{delay}\, e_2'}\ \textsc{ThunkApp} \qquad \frac{}{\mathbf{var}\, x \xrightarrow{\text{thunk}} \mathbf{force}\ (\mathbf{var}\, x)}\ \textsc{ThunkVar}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2'}{(\lambda x.\ e_2) \cdot e_1 \xrightarrow{\text{thunk}} \mathbf{let}\, x = \mathbf{delay}\, e_1'\ \mathbf{in}\ e_2'}\ \textsc{ThunkLet}$$

$$\frac{\forall n.\ e_n \xrightarrow{\text{thunk}} e_n' \qquad e \xrightarrow{\text{thunk}} e'}{\mathbf{letrec}\, \overline{x_n = e_n}\ \mathbf{in}\ e \xrightarrow{\text{thunk}} \mathbf{letrec}\, \overline{x_n = \mathbf{delay}\, e_n'}\ \mathbf{in}\ e'}\ \textsc{ThunkLetrec}$$

$$\frac{\forall n.\ e_n \xrightarrow{\text{thunk}} e_n'}{\mathbf{cons}\, cname[\, \overline{e_n}\,] \xrightarrow{\text{thunk}} \mathbf{cons}\, cname[\, \overline{\mathbf{delay}\, e_n'}\,]}\ \textsc{ThunkCons}$$

$$\frac{e \xrightarrow{\text{thunk}} e'}{\mathbf{proj}_n\ cname\ e \xrightarrow{\text{thunk}} \mathbf{force}\ \left(\mathbf{proj}_n\ cname\ e'\right)}\ \textsc{ThunkProj}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2' \qquad fresh \notin \mathsf{freevars}\, e_2}{\mathbf{seq}\, e_1\, e_2 \xrightarrow{\text{thunk}} \mathbf{let}\, fresh = e_1'\ \mathbf{in}\ e_2'}\ \textsc{ThunkSeq}$$

Figure 4.2. Core rules from $- \xrightarrow{\text{thunk}} -$, the first syntactic relation between PureLang and ThunkLang.

4.2.2 Compilation from PureLang

To model the weak-head, call-by-name PureLang in the call-by-value ThunkLang, we **delay** computations to prevent their evaluation until dictated by the PureLang semantics. We must then **force** each such delayed computation before we use it.

The syntactic relation $- \xrightarrow{\text{thunk}} -$ characterises this direct embedding. Its core rules are shown in figure 4.2. ThunkApp models call-by-name evaluation by delaying arguments at application sites (and ThunkLet is a special case of this rule which produces a **let**-statement). ThunkLetrec delays **letrec**-bound variables to bring them in line with λ-/**let**-bound variables; all variables are now delayed at their binding sites, so ThunkVar forces all usages of variables. ThunkCons models weak-head constructor forms: delaying all arguments to constructors ensures they evaluate immediately to a thunk without inspecting the expression within. ThunkProj ensures that any such delayed argument is forced once projected. ThunkSeq compiles **seq** to a **let**-statement which binds a fresh variable, relying on call-by-value evaluation order. Note how this relation simply asserts the existence of a sufficiently fresh name *fresh*.

Rules for monadic operations are shown in figure 4.3, and fall out naturally from two invariants: all arguments to **return** or **raise** are delayed to avoid premature evaluation

$$\frac{e \xrightarrow{\text{thunk}} e'}{\mathbf{raise}\, e \xrightarrow{\text{thunk}} \mathbf{raise}\ (\mathbf{delay}\, e')}\ \textsc{ThunkRaise}$$

$$\frac{e \xrightarrow{\text{thunk}} e'}{\mathbf{len}\, e \xrightarrow{\text{thunk}} \mathbf{bind}\ (\mathbf{len}\, e')\ (\lambda x.\ \mathbf{return}\ (\mathbf{delay}\ (\mathbf{var}\, x)))}\ \textsc{ThunkLen}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2'}{\mathbf{alloc}\, e_1\, e_2 \xrightarrow{\text{thunk}} \mathbf{bind}\ \left(\mathbf{alloc}\, e_1'\ \left(\mathbf{delay}\, e_2'\right)\right)\ (\lambda x.\ \mathbf{return}\ (\mathbf{delay}\ (\mathbf{var}\, x)))}\ \textsc{ThunkAlloc}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2'}{\mathbf{deref}\, e_1\, e_2 \xrightarrow{\text{thunk}} \mathbf{handle}\ \left(\mathbf{deref}\, e_1'\, e_2'\right)\ (\lambda x.\ \mathbf{raise}\ (\mathbf{delay}\ (\mathbf{var}\, x)))}\ \textsc{ThunkDeref}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2' \qquad e_3 \xrightarrow{\text{thunk}} e_3'}{\begin{array}{l}\mathbf{update}\, e_1\, e_2\, e_3 \xrightarrow{\text{thunk}}\ \mathbf{bind}\ \left(\mathbf{handle}\ \left(\mathbf{update}\, e_1'\, e_2'\ \left(\mathbf{delay}\, e_3'\right)\right)\right. \\ \qquad\qquad\qquad\qquad\qquad\qquad \left.(\lambda x.\ \mathbf{raise}\ (\mathbf{delay}\ (\mathbf{var}\, x)))\right) \\ \qquad\qquad (\lambda x.\ \mathbf{return}\ (\mathbf{delay}\ (\mathbf{var}\, x)))\end{array}}\ \textsc{ThunkUpdate}$$

Figure 4.3. Monadic rules from $- \xrightarrow{\text{thunk}} -$, the first syntactic relation between PureLang and ThunkLang.

during stateful interpretation; and the value-based arrays of ThunkLang must store thunks to model the expression-based arrays of PureLang. ThunkRaise delays arguments to **raise** (**return** is similar). ThunkLen delays the result of a length query; this result is implicitly a monadic **return** of an integer (as in figure 2.6 on pg. 29, similarly **action** implicitly returns a string FFI response). ThunkAlloc also wraps an implicit monadic **return**, and further delays the expression that will be stored in the freshly-allocated array. Dereferencing can throw a subscript error on an out-of-bounds array access; ThunkDeref ensures this implicit **raise** is delayed. Note that though **deref** implicitly **return**s a value from an array, there is no need to **delay** it because arrays store only thunks (*i.e.*, it is already delayed due to the second invariant above). Last, ThunkUpdate delays both a successful result and any thrown exception, and further ensures arrays are updated only with delayed values.

We straightforwardly verify that $\xrightarrow{\text{thunk}}$ expresses semantics-preserving transformations. Unfortunately, any implementation within its envelope produces poor quality code. In particular, occurrences of the inefficient pattern **delay** (**force** ...) arise whenever a variable or projection is passed as a function argument or **let**-bound. Worse still, the desugaring of **case**-expressions produces many such **let**-bound projections (definition 2.3, pg. 30). We must remove this pattern wherever possible, focusing particularly on **case**-expressions to permit their straightforward compilation into ThunkLang. Four syntactic

$$\frac{\forall n.\ e_n \xrightarrow{\text{liftLet}} e'_n \qquad \forall n.\ x \notin \mathsf{freevars}\, e_n}{\begin{array}{l}\mathbf{if}\ (\mathbf{eq}_?\ \mathit{cname}\, n\, e_1)\ \mathbf{then}\, e_2\ \mathbf{else}\, e_3 \\ \qquad \xrightarrow{\text{liftLet}} \mathbf{let}\, x = e'_1\ \mathbf{in}\ \left(\mathbf{if}\ \left(\mathbf{eq}_?\ \mathit{cname}\, n\, e'_1\right)\ \mathbf{then}\, e'_2\ \mathbf{else}\, e'_3\right)\end{array}}$$

$$\frac{C \xrightarrow{\text{letForce}} C' \qquad x \neq y \qquad x,\ y \text{ not shadowed in } C[-]}{\mathbf{let}\, x = \mathbf{force}\ (\mathbf{var}\, y)\ \mathbf{in}\ C[\mathbf{force}\ (\mathbf{var}\, y)] \xrightarrow{\text{letForce}} \mathbf{let}\, x = \mathbf{force}\ (\mathbf{var}\, y)\ \mathbf{in}\ C'[\mathbf{var}\, x]}$$

$$\frac{e \xrightarrow{\text{caseProj}} e'}{\begin{array}{l}\mathbf{let}\, x = \mathbf{delay}\ \left(\mathbf{force}\ \left(\mathbf{proj}_n\ \mathit{cname}\ (\mathbf{var}\, y)\right)\right)\ \mathbf{in}\ e \\ \qquad \xrightarrow{\text{caseProj}} \mathbf{let}\, x = \mathbf{proj}_n\ \mathit{cname}\ (\mathbf{var}\, y)\ \mathbf{in}\ e'\end{array}}$$

$$\frac{}{\mathbf{delay}\ (\mathbf{force}\ (\mathbf{var}\, x)) \xrightarrow{\text{unthunk}} \mathbf{var}\, x}$$

Figure 4.4. Core rules for **let**-lifting, **let**-forcing, **case**-projection, and unthunking. All other rules are defined by simple recursion.

relations justify this removal: **let**-lifting, **let**-forcing, **case**-projection, and unthunking.

The rules in figure 4.4 showcase their critical transformations; otherwise they are defined by simple recursion over semantics expressions. Using **let**-lifting ($- \xrightarrow{\text{liftLet}} -$), we bind expression e_1 to x when e_1 is immediately tested using $\mathbf{if}/\mathbf{eq}_?$. Testing with $\mathbf{if}/\mathbf{eq}_?$ immediately evaluates e_1 and x is sufficiently fresh, so the eager **let**-binding cannot change semantics. We expect e_1 to be of the form $\mathbf{force}\ (\mathbf{var}\, y)$, produced by the action of $\xrightarrow{\text{thunk}}$ on the output of expand, which helps to desugar **case**-expressions (definition 2.3, pg. 30). Now, **let**-forcing ($- \xrightarrow{\text{letForce}} -$) uses the newly-introduced binding ($x = \mathbf{force}\ (\mathbf{var}\, y)$) to reduce unnecessary forcing in subexpressions, by converting $\mathbf{force}\ (\mathbf{var}\, y)$ to $\mathbf{var}\, x$ as long as x and y have not been shadowed by some intervening binding. I have abused notation here, using contexts $C[-]$ to illustrate a subexpression which contains an instance of $\mathbf{force}\ (\mathbf{var}\, y)$; the premise $C \xrightarrow{\text{letForce}} C'$ simply indicates that the rest of the subexpression may be transformed by $\xrightarrow{\text{letForce}}$ too. The final two relations directly remove $\mathbf{delay}\ (\mathbf{force}\ \ldots)$ patterns in certain locations: **case**-projection ($- \xrightarrow{\text{caseProj}} -$) around projections, where it arises due to the combination of ThunkLet and ThunkProj (figure 4.2, pg. 57); and unthunking ($- \xrightarrow{\text{unthunk}} -$) around variables.

Implementation

There is only one overall PureLang-to-ThunkLang compiler pass (pure_to_thunk), whose action lies in the envelope produced by composing the five syntactic relations we have seen ($\xrightarrow{\text{thunk}}$, $\xrightarrow{\text{liftLet}}$, $\xrightarrow{\text{letForce}}$, $\xrightarrow{\text{caseProj}}$, and $\xrightarrow{\text{unthunk}}$). Breaking down a single compiler pass into smaller verification steps keeps proofs tractable without sacrificing performance.

The actual implementation of the pass is relatively straightforward, guided entirely by the syntactic relations. Only **case**-expressions require further attention: PureLang versions are compiled to their counterparts in ThunkLang with the insertion of two additional **let**-bindings. The first represents the binding $x = ce$ in **case** $x = ce$ **of** $\overline{row_n}$, which is not present in ThunkLang's syntax for **case**. The second is the **let**-binding introduced by the combination of **let**-lifting and **let**-forcing. We can see that unthunking is implemented using a simple mechanism: instead of producing **delay** operations directly, the compiler uses a smart constructor, mk_delay.

$$\begin{aligned}
&\mathsf{pure_to_thunk}\ (\mathbf{case}\, x = ce\, \mathbf{of}\, cname[\,\overline{x_n}\,] \to ce_1 \mid \overline{row_m}) \stackrel{\text{def}}{=} \\
&\quad \mathbf{let}\, x = \mathsf{mk_delay}\ (\mathsf{pure_to_thunk}\ ce)\ \mathbf{in} \\
&\quad \mathbf{let}\, fresh = \mathbf{force}\ (\mathbf{var}\, x)\ \mathbf{in} \\
&\quad\quad \mathbf{case}\, fresh\, \mathbf{of}\, cname[\,\overline{x_n}\,] \to \mathsf{pure_to_thunk}\ ce_1 \mid \overline{row'_m}
\end{aligned}$$

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

Verification details

In this section, I describe work completed by Oskar Abrahamsson.

For each relation above, we must derive either a forward or backward simulation result akin to equation 4.3 (pg. 53), proved by induction using the principle generated when defining functional big-step evaluation. In either direction, the existentially-quantified step count m makes the proof awkward and tedious: each inductive hypothesis introduces a fresh step count m_i, and we must consolidate these by appealing to determinacy and monotonicity of our clocked semantics. This causes significant repetition amongst the verification of each syntactic relation.

If both source and target use the *same number of steps*, we can remove the existential quantification on the step count:

$$e\ \mathcal{R}\ e' \ \wedge\ \mathsf{eval}^n_{\text{source}}\ e = r \neq \mathbf{error} \Rightarrow \exists\, r'.\ \mathsf{eval}^n_{\text{target}}\ e' = r' \ \wedge\ r\ \mathcal{R}_{\text{res}}\ r'$$

Therefore, we modify each syntactic relation to preserve step counts by inserting dummy "ticks" which incur a single step on evaluation, but do not modify semantics. In particular, a tick wraps an expression e with an empty recursive binding, **letrec** _ **in** e. This form should not appear in compiled code, so we verify another syntactic relation $\xrightarrow{\text{untick}}$ to justify removal of ticks. Now, we incur the awkwardness of existentially quantified step counts exactly once, and simplify all other proofs. Each compiler function which was previously

verified with respect to some relation $\mathcal{R}$ is now verified with respect to $\mathcal{R}$ post-composed with $\xrightarrow{\text{untick}}$. Note that CakeML also uses dummy ticks to simplify verification; however, its implementation actually inserts them into code before removing them again in a later pass [Owens et al. 2017]. In PureCake, dummy ticks appear in proofs only.

There is one further subtlety. Recall that the $\xrightarrow{\text{unthunk}}$ relation transforms expression of the form **delay** (**force** (**var** x)) into **var** x. The former evaluates to a **thunk** immediately, but when later forced, incurs an additional step compared to forcing the right-hand side. To preserve step counts, we introduce "delayed ticks": ticks which incur their additional steps only when forced. In particular, we extend ThunkLang semantics expressions with **tick** e, which has a semantics that evaluates e to some v and produces a value **ticked** v. When forced, **ticked** v consumes a step before forcing v.

4.2.3 Intra-language optimisations

We perform two further ThunkLang-to-ThunkLang optimisations.

The first lifts delayed λ-abstractions which are bound by a **let**-/**letrec**-statement into their own variable bindings. Further lookups of the *delayed* λ-abstraction which occur under a **force** operation can be replaced with a direct lookup of the *non-delayed* λ-abstraction itself, reducing the overhead of calling known **let**-bound functions. The rule below illustrates the syntactic relation in the **let** case:

$$\frac{e \xrightarrow{\text{liftLam}} e' \qquad C \xrightarrow{\text{liftLam}} C' \qquad \mathit{fresh} \text{ is fresh} \qquad x \text{ not shadowed in } C[-]}{\begin{array}{l}\textbf{let}\, x = \textbf{delay}\ (\lambda y.\ e)\ \textbf{in}\ C[\textbf{force}\ (\textbf{var}\, x)] \\ \qquad \xrightarrow{\text{liftLam}} \textbf{let}\, \mathit{fresh} = (\lambda y.\ e')\ \textbf{in let}\, x = \textbf{delay}\ (\textbf{var}\, \mathit{fresh})\ \textbf{in}\ C'[\textbf{var}\, \mathit{fresh}]\end{array}}$$

The second pass is another instance of **let**-forcing, seen as a limited form of common subexpression elimination. We have already eliminated the common subexpression **force** (**var** y) around **case**-statements; now we apply the same $\xrightarrow{\text{letForce}}$ rule wherever we see expressions of the form **let** x = **force** (**var** y) **in** e. This is a limited form of value-sharing: once **var** y has been forced by the eager **let**-binding, its result can be reused in the continuation expression e.

4.3 EnvLang

EnvLang is the intermediate language which follows ThunkLang, and is only a minor stepping stone towards its own successor, StateLang (§ 4.4). Its syntax closely mirrors ThunkLang, except its compiler expressions also specify top-level constructors for each monadic operation (**bind**, **raise**, **action**, *etc.*). However, its semantics is expressed using environments rather than substitutions: evaluation of an expression is parametrised

by an environment η which provides the values of its free variables, and semantic expressions omit the **value** form. The following equations characterise the differences: variable evaluation requires a lookup in the environment; **delay**s and λ-abstractions capture their environments to produce environment-equipped thunks and closures; **force** evaluates a thunk using its captured environment; and function application extends a closure environment with a new binding for the passed argument.

$$\begin{aligned}
&\vdash \mathsf{eval}_\eta\ (\mathbf{var}\, x) = \\
&\quad \mathsf{if}\ x \in \mathrm{domain}(\eta) \\
&\quad \mathsf{then}\ \eta(x)\ \mathsf{else}\ \mathbf{error} \\
&\vdash \mathsf{eval}_\eta\ (\mathbf{force}\, e) = \mathsf{do} \\
&\quad \mathbf{thunk}\, \eta'\, e' \leftarrow \mathsf{eval}_\eta\ e; \\
&\quad \mathsf{eval}_{\eta'}\ e'
\end{aligned}
\qquad
\begin{aligned}
&\vdash \mathsf{eval}_\eta\ (\mathbf{delay}\, e) = \mathbf{thunk}\, \eta\, e \\
&\vdash \mathsf{eval}_\eta\ (\lambda x.\ e) = \mathbf{closure}\, \eta\, x\, e \\
&\vdash \mathsf{eval}_\eta\ (e_1 \cdot e_2) = \mathsf{do} \\
&\quad v \leftarrow \mathsf{eval}_\eta\ e_2; \\
&\quad \mathbf{closure}\, \eta'\, x\, e \leftarrow \mathsf{eval}_\eta\ e_1; \\
&\quad \mathsf{eval}_{\eta'[x \mapsto v]}\ e
\end{aligned}$$

Compilation from ThunkLang to EnvLang is almost an identity mapping, except monadic operations are lifted to their top-level constructors. The core verification for this pass concerns the transition from substitutions to environments. Our syntactic relation, $e \xrightarrow{\text{env}} e' \dashv \eta$, is parametrised by an environment which provides values for the free variables in e'. The characterising rules follow (all others are simple recursion):

$$\frac{\eta(x) = v' \qquad v \xrightarrow{\text{env}} v'}{\mathbf{value}\, v \xrightarrow{\text{env}} \mathbf{var}\, x \dashv \eta} \qquad \frac{x \notin \mathrm{domain}(\eta)}{\mathbf{var}\, x \xrightarrow{\text{env}} \mathbf{var}\, x \dashv \eta}$$

$$\frac{e \xrightarrow{\text{env}} e' \dashv \eta - \{x\}}{\lambda x.\ e \xrightarrow{\text{env}} \lambda x.\ e' \dashv \eta} \qquad \frac{e \xrightarrow{\text{env}} e' \dashv \eta}{\mathbf{thunk}\, e \xrightarrow{\text{env}} \mathbf{thunk}\, \eta\, e'}$$

A **value** v corresponds to any variable x for which we can find v' in the environment such that v and v' are suitably related, where I abuse notation to lift $\xrightarrow{\text{env}}$ to values (note the absence of an environment here, as values are expected to have no free variables); equal variables are related as long as the environment does not mention them; λ-abstractions are related if their bodies are related in the environment which does not know anything about the value passed in as their bound variable; and thunk values are related if their subexpressions are related in the environment captured by the EnvLang thunk. The lowermost simulation proof for this relation is phrased as follows:

$$\vdash \left(e \xrightarrow{\text{env}} e' \dashv \eta\right) \Rightarrow \left(\mathsf{eval}\, e \xrightarrow{\text{env}} \mathsf{eval}_\eta\ e'\right)$$

In other words, relatedness with respect to a particular environment is preserved by evaluation in that environment.

4.4 StateLang

EnvLang is compiled into StateLang, the final intermediate language of the PureCake compiler. StateLang differs somewhat from its predecessors: designed to match CakeML closely, its syntax introduces stateful and I/O primitives while its semantics is expressed as a CESK machine (§ 4.4.1). Compilation implements both monadic operations and the thunk operations **delay**/**force** in terms of more primitive constructs (§ 4.4.2); thunk values are shared statefully to introduce lazy evaluation. A final intra-language optimisation pass removes compilation artefacts (§ 4.4.3).

4.4.1 Syntax and semantics

As in all predecessor languages, exp_of : $ce \rightarrow e$ maps compiler expressions to semantic expressions in StateLang. However, its syntax removes monadic operations and introduces effectful primitives, and we specify its semantics using a CESK machine.

More concretely, StateLang removes monadic operations **return** and **bind**, and expresses others as primitive operations, denoted with the subscript "prim" to avoid ambiguity. Without monadic operations, there is no need for a third semantics layer of stateful interpretation (§ 2.3), so we specify StateLang semantics in a single layer as a CESK machine [Felleisen and Friedman 1987] which produces the expected ITree directly. We choose to use a CESK machine for two reasons: it naturally handles mutable store and the stack-like monadic control flow we wish to model; and CakeML expresses a version of its semantics as a CESK machine, which we augment to provide a suitable target semantics (§ 5). The values of StateLang no longer require the **error**/**diverge** forms used to specify the functional big-step semantics of previous languages. Overall, the key syntax changes are as follows:

$$\begin{aligned} op \;::=\; & \mathbf{cons}\, cname \mid \mathbf{prim}\, primop \mid \mathbf{raise}_{\text{prim}} \mid \mathbf{handle}_{\text{prim}} \mid \mathbf{action}_{\text{prim}} \mid \\ & \mathbf{alloc}_{\text{prim}} \mid \mathbf{len}_{\text{prim}} \mid \mathbf{deref}_{\text{prim}} \mid \mathbf{update}_{\text{prim}} \end{aligned}$$

$$v \;::=\; \mathbf{thunk}\, \eta\, e \mid \mathbf{cons}_v\, cname[\,\overline{v_n}\,] \mid \mathbf{closure}\, \eta\, x\, e \mid \mathbf{lit}\, lit$$

Our CESK (**C**ontrol **E**nvironment **S**tate **K**ontinuation) machine state is a four-tuple, consisting of the "current" expression or value being processed, an environment providing values for variables in scope, a mutable store, and a continuation stack. Evaluating an expression pushes frames onto the continuation stack in a right-to-left, call-by-value evaluation order until a value can be produced. This value is "returned", popping frames off the continuation stack appropriately. Effectful primitives query and update the mutable store. I omit full details as they are essentially standard, except for production of an ITree which is described further in § 5.2.1.

There is one subtlety, due to inheritance of thunk operations **delay**/**force** from EnvLang. Though we compile these operations away in StateLang, they persist in semantic expressions for ease-of-verification (§ 4.4.2). However, **force** is a *pure* operation in EnvLang, and must remain pure in StateLang if we want to share thunk values statefully. In particular, repeated forcings of the same thunk must produce the same value: it is unsound to share a previously computed value statefully if a different value can be obtained by another forcing. StateLang semantics therefore forbids all stateful operations while forcing a thunk value (*i.e.*, using **force** to compute the delayed expression contained within). To achieve this, we make the CESK machine's mutable store optional: when entering a **force**, we remove the mutable store and save it to the continuation stack; when exiting a **force** we revert the mutable store by restoring it from the continuation stack. While the store is removed, any stateful operations cause runtime type errors. We prove that StateLang code compiled from the pure fragment of EnvLang does not inspect or modify the store, and so cannot introduce any of these runtime type errors.

4.4.2 Compilation into StateLang

StateLang must model the sequencing of IO operations: the notion that monadic code expresses a computation as data, which only becomes effectful when it is "run". Therefore, we compile monadic operations to suspended computations which perform their operations only when triggered. Effectful (*i.e.*, exception-handling, arrays, I/O) monadic operations will use effectful primitives once triggered. The implementation of monadic operations as imperative code is sometimes referred to as monadic reflection [Filinski 1994, 2010]. We also leverage StateLang's primitives to compile thunk operations **delay**/**force** to stateful operations which share values, so that forcing a previously-computed thunk does not recompute it. This critical transformation introduces lazy evaluation quite late in compilation at minimal verification cost.

Monadic operations. The syntactic relation $- \xrightarrow{\text{reflect}} -$ defined in figure 4.5 encapsulates the compilation of monadic operations. Each monadic operation produces an expression of the form $\lambda\_.\ e$ (a λ-abstraction which does not bind an argument), representing a suspended computation which can be triggered by application to **unit**. As a top-level PureLang program is monadic, the overall action of $\xrightarrow{\text{reflect}}$ produces such a suspended computation which we must wrap in an application to a unit ($- \cdot \textbf{unit}$). This will trigger the evaluation of the program's monadic effects in the correct order.

Monadic **return**s are simply suspended. We rely on StateLang's right-to-left evaluation order for **bind**: e'_1 is triggered first, then passed to the continuation which is itself triggered. Though **message** primitives are not monadic, they hold FFI communication inputs (channel and message content) for use with **action**; we compile them to suspended

$$\frac{e \xrightarrow{\text{reflect}} e'}{\mathbf{return}\, e \xrightarrow{\text{reflect}} \lambda\_.\ e'} \qquad \frac{e_1 \xrightarrow{\text{reflect}} e_1' \qquad e_2 \xrightarrow{\text{reflect}} e_2'}{\mathbf{bind}\, e_1\, e_2 \xrightarrow{\text{reflect}} \lambda\_.\ e_2' \cdot (e_1' \cdot \mathbf{unit}) \cdot \mathbf{unit}}$$

$$\frac{e \xrightarrow{\text{reflect}} e'}{\mathbf{prim}\ (\mathbf{message}\, ch)\ e \xrightarrow{\text{reflect}} \mathbf{let}\, x = e'\ \mathbf{in}\ \lambda\_.\ \mathbf{action}_{\text{prim}}\ ch\ (\mathbf{var}\, x)}$$

$$\frac{e \xrightarrow{\text{reflect}} e'}{\mathbf{action}\, e \xrightarrow{\text{reflect}} \lambda\_.\ (e' \cdot \mathbf{unit})}$$

$$\frac{e_1 \xrightarrow{\text{reflect}} e_1' \qquad e_2 \xrightarrow{\text{reflect}} e_2'}{\mathbf{handle}\, e_1\, e_2 \xrightarrow{\text{reflect}} \lambda\_.\ \left(\mathbf{handle}_{\text{prim}}\ \left(\mathbf{let}\, x = e_1' \cdot \mathbf{unit}\ \mathbf{in}\ \lambda\_.\ \mathbf{var}\, x\right)\ e_2'\right) \cdot \mathbf{unit}}$$

$$\frac{\forall\, n.\ e_n \xrightarrow{\text{reflect}} e_n' \qquad mop \notin \{\mathbf{return},\ \mathbf{bind},\ \mathbf{handle},\ \mathbf{action}\}}{mop[\overline{e_n}] \xrightarrow{\text{reflect}} \lambda\_.\ mop_{\text{prim}}[\overline{e_n'}]}$$

Figure 4.5. Key rules from $- \xrightarrow{\text{reflect}} -$, concerning compilation of monadic operations.

$$\frac{e \xrightarrow{\text{dethunk}} e'}{\mathbf{delay}\, e \xrightarrow{\text{dethunk}} \mathbf{alloc}_{\text{prim}}\ [\mathbf{false},\ \lambda\_.\ e']}$$

$$\frac{e \xrightarrow{\text{dethunk}} e'}{\begin{array}{l}\mathbf{force}\, e \xrightarrow{\text{dethunk}} \mathbf{let}\, x = e'\ ;\ x_0 = x[0]\ ;\ x_1 = x[1]\ \mathbf{in} \\ \quad \mathbf{if}\ \mathbf{var}\, x_0\ \mathbf{then}\ \mathbf{var}\, x_1\ \mathbf{else} \\ \quad\quad \mathbf{let}\, w = \mathbf{var}\, x_1 \cdot \mathbf{unit}\ \mathbf{in} \\ \quad\quad\quad x[0] := \mathbf{true}\ ;\ x[1] := \mathbf{var}\, w\ ;\ \mathbf{var}\, w\end{array}}$$

Figure 4.6. Key rules from $- \xrightarrow{\text{dethunk}} -$, concerning compilation of **delay** and **force**. We use the following shorthands: $x[z] := e$ is short for $\mathbf{alloc}_{\text{prim}}\ (\mathbf{var}\, x)\ (\mathbf{int}\, z)\ e$, $x[z]$ is short for $\mathbf{deref}_{\text{prim}}\ (\mathbf{var}\, x)\ (\mathbf{int}\, z)$, and $e_1;\ e_2$ is short for $\mathbf{let}\ \_ = e_1\ \mathbf{in}\ e_2$ (a **let**-binding that does not bind a name).

FFI communication primitives, which are triggered by the compiled version of **action** itself. We might expect **handle** to mirror **bind**, but there is a key difference: a **handle** continuation is optional, executed only when an exception is raised by triggering e'_1. If no exception is produced, we must re-wrap the already-triggered $e'_1 \cdot \mathbf{unit}$ in a suspended computation so that the surrounding context can trigger it as expected. All other monadic operations (**raise** and the array operations) are straightforwardly compiled to suspended primitive operations.

Thunk operations. The syntactic relation $- \xrightarrow{\text{dethunk}} -$ defined in figure 4.6 encapsulates the compilation of thunk operations **delay** and **force** into stateful primitives. Thunks are now represented by *thunk-arrays*, arrays of length two: the first element is a flag indicating whether the thunk has previously 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** is represented as a primitive array allocation ($\mathbf{alloc}_{\text{prim}}$) of a thunk-array with the flag set to **false**. Compilation of **force** e expects e to produce a thunk-array. It reads the thunk-array's flag ($x[0]$) and branches: if **true**, the contained value ($x[1]$) has been computed previously and so is simply returned; otherwise, the suspended computation is forced by application to **unit** to produce a final value w, the thunk-array is updated with a **true** flag, and the final value is returned. The thunk-array index and update operations are implemented with *unchecked* array accesses for a small performance boost: the index is not bounds-checked before accessing the thunk-array.

Implementation and verification details

The compiler implements both transformations simultaneously, and use of two syntactic relations keeps proofs tractable. However, use of two relations requires StateLang's semantic expressions and semantics to support the thunk values (and **delay**/**force**) operations of EnvLang, where otherwise they could be removed.

Unusually, we perform the lowest-level simulation proof for $\xrightarrow{\text{dethunk}}$ in two directions, *i.e.*, both a forward and a backward simulation. This is due to a mismatch in step-counting: compilation of thunks to stateful operations neither monotonically increases nor monotonically decreases the number of steps taken by the semantics (§ 4.1). Compiled **delay** operations require more steps to set up the thunk-array; and compiled **force** operations can effectively skip a finite number of steps incurred in EnvLang by simply "remembering" a previously-forced result rather than recalculating it.

The key invariant in these simulation proofs is $\mathcal{R}_\sigma$, a relation between mutable stores. Before $\xrightarrow{\text{dethunk}}$, stores contain only regular arrays; afterwards, each location in the store maps *either* to a regular array *or* to a thunk-array allocated by a compiled **delay**. Therefore, $\mathcal{R}_\sigma$ is parametrised by a mapping ρ, which takes each location l' in the thunk-array store

σ' to *either* a location l in the original store σ (if l' contains a regular array) *or* to the thunk value represented by the thunk-array at l'.

The formal definition below captures this intuition, relying on auxiliary relations $\mathcal{R}_v$, $\mathcal{R}_\eta$, and $\mathcal{R}_{\text{thunk}}$. The first two lift $\xrightarrow{\text{dethunk}}$ to values and environments, and are also parametrised by ρ to handle store locations which are embedded in values. The latter relates pure thunk values to shared thunk-arrays, and is defined by the two rules below: a thunk-array is either unforced, containing a closure that suspends a computation which is suitably related to the pure thunk; or previously-forced, containing a value that is related to the result of forcing ($\Downarrow$) the pure thunk.

$$\rho \vdash \sigma \; \mathcal{R}_\sigma \; \sigma' \stackrel{\text{def}}{=} \forall\, l' \in \text{domain } \sigma'.\; \begin{cases} \rho \vdash (\mathbf{loc}\, l) \;\; \mathcal{R}_v \;\; (\mathbf{loc}\, l') & \rho\; l' = \mathbf{loc}\, l \\ \rho \vdash (\mathbf{thunk}\, \eta\, e) \;\; \mathcal{R}_{\text{thunk}} \;\; \big(\sigma'(l')\big) & \rho\; l' = \mathbf{thunk}\, \eta\, e \end{cases}$$

$$\frac{\rho \vdash \eta \; \mathcal{R}_\eta \; \eta' \qquad e \xrightarrow{\text{dethunk}} e'}{\rho \vdash (\mathbf{thunk}\, \eta\, e) \;\; \mathcal{R}_{\text{thunk}} \;\; [\mathbf{false},\; \mathbf{closure}\, \eta'\, \_\, e']} \qquad \frac{e \Downarrow_\eta v \qquad \rho \vdash v \; \mathcal{R}_v \; v'}{\rho \vdash (\mathbf{thunk}\, \eta\, e) \;\; \mathcal{R}_{\text{thunk}} \;\; [\mathbf{true},\; v']}$$

4.4.3 Intra-language optimisation

We perform one further StateLang-to-StateLang optimisation, which handles the soup of λ-abstractions that ignore their argument ($\lambda\_.\; -$) and applications to a unit ($- \cdot \mathbf{unit}$) produced by $\xrightarrow{\text{reflect}}$. We push applications to **unit** in through **let**-/**letrec**-/**if**-statements and β-contract ($\lambda\_.\; e) \cdot \mathbf{unit}$ to e. The relation $\xrightarrow{\text{appUnit}}$ below characterises this transformation; once again I elide details concerning dummy ticks inserted for ease-of-verification (§ 4.2.2).

$$\frac{e \xrightarrow{\text{appUnit}} e'}{(\lambda\_.\; e) \cdot \mathbf{unit} \xrightarrow{\text{appUnit}} e'}$$

$$\frac{e_1 \xrightarrow{\text{appUnit}} e_1' \qquad e_1 \xrightarrow{\text{appUnit}} e_2'}{(\mathbf{let}\, x = e_1\; \mathbf{in}\; e_2) \cdot \mathbf{unit} \xrightarrow{\text{appUnit}} \mathbf{let}\, x = e_1'\; \mathbf{in}\; \big(e_2' \cdot \mathbf{unit}\big)}$$

$$\frac{\forall n.\; e_n \xrightarrow{\text{appUnit}} e_n' \qquad e \xrightarrow{\text{appUnit}} e'}{(\mathbf{letrec}\, \overline{x_n = e_n}\; \mathbf{in}\; e) \cdot \mathbf{unit} \xrightarrow{\text{appUnit}} \mathbf{letrec}\, \overline{x_n = e_n'}\; \mathbf{in}\; (e' \cdot \mathbf{unit})}$$

$$\frac{e \xrightarrow{\text{appUnit}} e' \qquad e_1 \xrightarrow{\text{appUnit}} e_1' \qquad e_1 \xrightarrow{\text{appUnit}} e_2'}{(\mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) \cdot \mathbf{unit} \xrightarrow{\text{appUnit}} \mathbf{if}\, e\, \mathbf{then}\; (e_1 \cdot \mathbf{unit})\; \mathbf{else}\; (e_2 \cdot \mathbf{unit})}$$

Chapter 5

Connecting with CakeML

Targeting CakeML leverages its mature optimising compiler and end-to-end correctness guarantees. However, there are significant challenges in reconciling the PureCake and CakeML worlds to obtain a final end-to-end correctness theorem which spans from PureCake to machine code. In this chapter, I highlight these challenges and detail their solutions, before presenting PureCake's top-level correctness results.

First, I describe compilation of StateLang to CakeML (§ 5.1), focusing particularly on implementation of primitives. Next, I reconcile the trace-producing, oracle-based semantics of CakeML with the ITree-producing semantics of the PureCake project (§ 5.2). Last, I present PureCake's compiler correctness and end-to-end correctness theorems, and produce a verified PureCake binary (§ 5.3).

5.1 Compiling StateLang to CakeML

StateLang was designed with its compilation to CakeML in mind, so there are no irreconcilable differences between the two languages. However, the semantics of CakeML necessitates careful compilation of data types (§ 5.1.1), and we must implement built-in operations in CakeML (§ 5.1.2). Most notably, we must use CakeML's FFI (which uses byte arrays) to express the string-based FFI of PureLang.

Preliminaries: variable names

CakeML does not support λ-abstractions and **let**-bindings that do not bind arguments, so a StateLang-to-StateLang pass gives a fresh bound variable name to each of these. The simple relation $\xrightarrow{\text{name}}$ demonstrates this change:

$$\frac{e \xrightarrow{\text{name}} e' \qquad \mathit{fresh} \notin \mathsf{freevars}\, e}{\lambda\_.\ e \xrightarrow{\text{name}} \lambda \mathit{fresh}.\ e'} \qquad \frac{\forall n.\ e_n \xrightarrow{\text{name}} e'_n \qquad \mathit{fresh} \notin \mathsf{freevars}\ e_1\ \cup\ \mathsf{freevars}\ e_2}{\textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 \xrightarrow{\text{name}} \textbf{let}\, \mathit{fresh} = e'_1\ \textbf{in}\ e'_2}$$

5.1.1 Data types

CakeML's semantics is parametrised by an environment of data types and exceptions, recording the types and arities of all constructors currently in scope: when an ill-typed constructor or pattern match is encountered, the semantics produces a runtime type error. This enables optimisations in CakeML that modify its rich pattern matches: verification can assume a lack of runtime type errors and also learn that constructors and pattern matches are well-typed. Therefore, compilation to CakeML must ensure that any constructors and pattern matches it introduces are considered well-typed by CakeML's semantics. We achieve this straightforwardly: we emit CakeML data type declarations to introduce the types and constructors extracted from PureLang by parsing (§ 3.1); and we use simple syntactic invariants to verify that compilation preserves the type-checked constructor applications and pattern matches of PureLang. The latter form one of the syntactic well-formedness conditions in equation 4.4 (pg. 54). When emitting CakeML declarations we need only consider constructors and arities, and not attempt to translate PureLang types to CakeML types or to handle mutually recursive declarations. Therefore, we introduce dummy type declarations consisting of constructors applied to unit types only, neglecting even to assign unnecessary type names. For example, a standard binary tree data type in PureLang (left-hand side) becomes a CakeML type declaration (right-hand side, using OCaml-like syntax for accessibility):

```
data Tree a =
    Leaf | Node (Tree a) a (Tree a)
```

```
type _ =
    Leaf | Node of unit * unit * unit
```

These are introduced in a *compilation preamble*, prepended to the compiled program itself. One subtlety is that the list data type and subscript exception are considered built-in in CakeML. For simplicity, we consider them built-in in PureLang too: any attempt to redefine them will result in a compilation error.

5.1.2 Built-in operations

We must implement in CakeML both the effectful primitives introduced in StateLang (§ 4.4) and the primitive operations inherited from PureLang. The former were designed to permit direct translation to CakeML, though some wrapping with bounds checks is necessary to match semantics. Primitive division and modulo operators inherited from PureLang must also be wrapped to handle division/modulo by zero. Translation of primitive PureLang string operations is slightly more involved as CakeML offers fewer string primitives (*e.g.*, strings can be compared for equality but not lexicographic ordering), and incurs further bounds-checks. In particular, we implement lexicographic string comparison as a recursive function in CakeML, and bind it as a declaration once and for all in the compilation preamble.

The most significant challenge here is compilation of FFI calls. In CakeML, an FFI call **ffi** *ch s l* contains three pieces of information: a channel *ch* (*i.e.*, the name of the underlying C function), a string argument *s*, and a reference to a byte-array *l*. The FFI function reads the contents of the byte-array, so it effectively receives two arguments: the string and the contents of the fixed-length byte-array. The FFI function writes its response into the same byte-array, so making it available to the CakeML program.

However, PureLang has no byte-arrays, so its FFI calls accept messages containing two strings (FFI channel and argument): **action** (**msg** *ch s*). The response is a single string. When realising a PureLang FFI call in CakeML, we can directly pass in the channel *ch* and string *s* with some arbitrary byte-array. However, we must read the FFI response back from that same byte-array and convert it to a string so that the rest of the compiled PureLang program can use it. There is a mismatch here: PureLang semantics could naturally permit string FFI responses of any length, but the CakeML semantics enforces responses lengths that fill the fixed-length byte-array. When compiling to CakeML, we do not know how long a given PureLang FFI response will be. We impose three restrictions on FFI functions used with the PureCake compiler to solve this issue: their responses must be shorter than some fixed bound (4096 bytes); they do not read from their input byte-array; and they write the length of their response in the first two bytes of the input byte-array (followed by the response itself). Restricting FFI calls in this way does not modify our trusted computing base: just like CakeML, PureCake must trust its FFI implementation in C.

We have seen that PureLang semantics considers overlong FFI responses equivalent to an FFI error (figure 2.6, pg. 29). We can see now that this restriction is necessary to relate PureCake and CakeML FFI calls directly, and that the maximum FFI response length is responseBound = 4096. Some byte-array of length 4098 is passed to each FFI call, safe in the knowledge its contents will not be read; the response length can be encoded in the first two bytes leaving the remainder available for a maximum-length response. For simplicity and to reduce allocation overhead, we allocate a single byte-array (called `ffi_array`) in the compilation preamble, reusing this for each FFI call. A StateLang FFI call $\textbf{action}_{\text{prim}}$ *ch msg* is therefore compiled to the following CakeML (expressed as OCaml-like pseudocode for accessibility):

```
let s = msg in
  FFI ch s ffi_array;
  let len = 256 * int(ffi_array[1]) + int(ffi_array[0]) in
  let len = min(4096, len) in
  string(ffi_array[2:len])
```

The message is bound to a variable before invoking CakeML's FFI primitive with the correct channel, message, and byte-array. The external FFI function (whose name is given by the channel) executes and writes its response back to `ffi_array`. Reading the

```
(* BEGIN preamble *)

(* Type and exception declarations *)
type _ = ...
...
exception _ of ...
...

(* String operations *)
let rec strleq = ... in
...

(* FFI array *)
let ffi_array = allocate(byte, 4098) in

(* END preamble *)

let _ = compiled_program;
```

Figure 5.1. The overall structure of a CakeML program compiled by PureCake, expressed as OCaml-like pseudocode for accessibility.

first two bytes determines the response length, and the response is copied into a string for the surrounding context to use.

Figure 5.1 shows the overall compilation of a top-level PureLang program by prepending the preamble and wrapping with a trivial top-level declaration (`let _ = ...`). The latter is necessary because CakeML top-level programs can contain only declarations. The preamble declares data types and exceptions, defines implementations of primitive operations, and allocates the byte-array used by FFI calls.

5.2 Reconciling oracles and ITrees

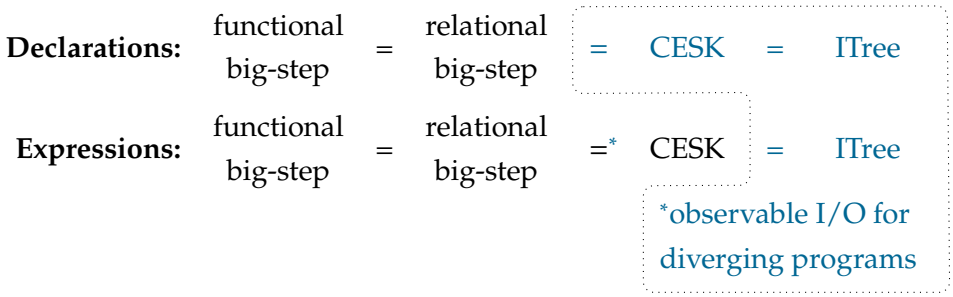
CakeML's correctness results are phrased in terms of its functional big-step semantics, which produces *linear* I/O traces by using an oracle to supply environment responses to each FFI call (§ 1.2.2). By contrast, PureLang's ITrees model all possible environment responses in their branching structure: each Vis node continuation accepts any response to produce the remaining computation.

In this section, I reconcile these differing semantic styles. First I build on existing results in CakeML to create a verified ITree-producing semantics for CakeML (§ 5.2.1). Next, I derive a new compiler correctness theorem with respect to this semantics (§ 5.2.2).

5.2.1 Interaction tree semantics for CakeML

Other than functional big-step, CakeML specifies versions of its semantics in two other styles: *relational* big-step and CESK machine [Felleisen and Friedman 1987]. The relational

big-step semantics is proved equivalent to the functional big-step one. However, the CESK semantics is partial: it is specified only for CakeML's expressions, not top-level declarations, so it cannot express the semantics of a whole CakeML program. The expression-level CESK semantics is proved mostly equivalent to the relational big-step semantics; however, observable I/O for diverging programs is not considered. We specify a CESK semantics for declarations and derive complete equivalence proofs with the relational big-step semantics. We then define an ITree-producing semantics for CakeML, verifying equivalence with the enriched CESK semantics. Below is an informal summary of the various CakeML semantics and their equivalence proofs, with new contributions outlined.



CESK semantics for CakeML declarations

Specifying top-level declarations in CakeML's CESK semantics requires careful management of environments for local declarations. Consider the following program:

```
local w = ...; x = ...
in y = ... end;

z = ...
```

Here, `w` is in scope when defining `x` and `y`, but only `y` is in scope when defining `z`. Any `w` defined above the keyword `local` is shadowed when defining `x` and `y`, but restored when defining `z`. Local declarations can also nest in CakeML.

This scoping has an imperative flavour: an ambient global environment of bindings is modified by each declaration, and definitions can escape their scopes (*e.g.*, `y` above). CESK machines are designed for functional languages: the "current" scope is represented by an environment, which is either augmented with newly-introduced bindings or replaced entirely during a change of scope (*e.g.*, evaluation of a closure application). To support `local`, we avoid maintaining such an explicit environment. Instead, the continuation stack stores *environment fragments*, which can be combined to determine the current scope: each variable lookup traverses the continuation stack to compute the

current environment, allowing newer fragments to shadow older ones. Each declaration produces an environment fragment when evaluated, and we maintain correct scoping by careful combining and discarding of fragments when returning them to the stack.

The resulting CESK machine expresses transitions (step) between *configurations*, triples of a control c (the current declaration or environment fragment), mutable store σ, and continuation stack κ:

$$\mathsf{step}\ \Delta\ \eta\ \langle c,\ \sigma,\ \kappa\rangle = (\langle c',\ \sigma',\ \kappa'\rangle,\ \Delta',\ io)$$

The initial environment is given by η, and as in § 1.2.2 oracle Δ is invoked (and updated) during an FFI call, adding to the trace of observable I/O *io*. Halting configurations ($\mathsf{halt}\ \langle\ldots\rangle$) cannot make further transitions, and iterating steps (denoted step_n) allows specification of terminating and diverging behaviours (as we will see below).

We establish equivalence with the existing relational big-step semantics for declarations, which is specified in two relations below. Each is parametrised by environment η, mutable store σ, and oracle Δ.

$$\begin{aligned}\text{Termination:}&\quad \eta;\ \sigma;\ \Delta \vdash \mathit{decls} \Downarrow r;\ \eta';\ \sigma';\ \Delta';\ \mathit{io}\\ \text{Divergence:}&\quad \eta;\ \sigma;\ \Delta \vdash \mathit{decls} \Uparrow \mathit{io}\end{aligned}$$

A series of declarations *decls* may terminate to some result r (either a value or error) and resulting environment/store/oracle, or diverge with no result. Either way, the declarations may produce a trace of observable I/O *io*. Equivalence is phrased in terms of these relations in theorem 5.1.

Theorem 5.1. Equivalence of relational big-step and CESK semantics.

$$\begin{aligned}&\vdash \eta;\ \sigma;\ \Delta \vdash \mathit{decls} \Downarrow r;\ \eta';\ \sigma';\ \Delta';\ \mathit{io} \iff\\ &\quad \exists n\ \mathit{conf}.\ \mathsf{halt}\ \mathit{conf}\ \wedge\ (r,\ \eta',\ \sigma') \nleftrightsquigarrow \mathit{conf}\ \wedge\\ &\qquad \mathsf{step}_n\ \Delta\ \eta\ \langle \mathsf{Decls}\ \mathit{decls},\ \sigma,\ \varepsilon\rangle = (\mathit{conf},\ \Delta',\ \mathit{io})\end{aligned}$$

$$\begin{aligned}&\vdash \eta;\ \sigma;\ \Delta \vdash \mathit{decls} \Uparrow \mathit{io} \iff\\ &\quad \big(\forall n.\ \exists \mathit{conf}\ \mathit{io}_n\ \Delta_n.\ \neg\,\mathsf{halt}\ \mathit{conf}\ \wedge\\ &\qquad \mathsf{step}_n\ \Delta\ \eta\ \langle \mathsf{Decls}\ \mathit{decls},\ \sigma,\ \varepsilon\rangle = (\mathit{conf},\ \Delta_n,\ \mathit{io}_n)\big)\ \wedge\\ &\qquad \mathit{io} = \bigsqcup_n \{\mathit{io}_n \mid \mathsf{step}_n\ \Delta\ \eta\ \langle \mathsf{Decls}\ \mathit{decls},\ \sigma,\ \varepsilon\rangle = (\ldots,\ \mathit{io}_n)\}\end{aligned}$$

Termination to a value r in the relational big-step semantics coincides with the ability to produce a suitably equivalent ($\nleftrightsquigarrow$) halting configuration *conf* in the CESK semantics after some number of steps n. Divergence coincides with a lack of halting configuration for any number of steps. In both cases, the CESK semantics starts with

an empty continuation stack ε, initial environment η, and a control containing the declarations *decls*. Observable I/O must also coincide; for divergence, the relational big-step semantics produces an I/O trace equal to the (potentially infinite) upper bound of all finite I/O traces produced by the CESK semantics.

Theorem 5.1 ports existing expression-level theorems to the level of declarations, further adding equivalence of observable I/O for divergence (its final line). Though this is a minor addition to the top-level theorem statement, it requires re-derivation of the entire expression-level proof with strengthened invariants.

ITree semantics for CakeML

We define a second CESK machine which produces ITrees. Its transition function istep is nearly identical to step above, but is not parametrised by an oracle and does not produce observable I/O. Wherever step requires input from its oracle, istep instead produces an **action** control, considered a halting configuration (ihalt).

$$\mathsf{istep}\ \eta\ \langle c,\ \sigma,\ \kappa\rangle = \langle c',\ \sigma',\ \kappa'\rangle \qquad \mathsf{ihalt}\ \langle \mathbf{action}\ a,\ \sigma,\ \kappa\rangle$$

Application of unfold (lemma 2.1, pg. 23) to next_halt below produces the final ITree. In particular, **action** halting configurations produce Vis nodes, and non-**action** configurations produce Ret nodes. Otherwise, no halting configuration exists, producing Div. I have elided the second argument to Vis', which accepts response r from the environment and produces an updated version of *conf*′ with r stored in memory. The semantics of a top-level CakeML program *prog* ($[\![\ prog\ ]\!]_{\gg}$) is the ITree produced by its declarations using empty initial environment/stack/store.

$$\mathsf{next\_halt}\ \eta\ conf \stackrel{\text{def}}{=} \begin{cases} \mathsf{Vis'}\ a\ (\lambda r.\ \ldots) & \exists n\ conf'.\ \mathsf{istep}_n\ \eta\ conf = conf' \wedge \\ & \quad conf' = \langle \mathbf{action}\ a, \ldots\rangle \\ \mathsf{Ret'}\ r & \exists n\ conf'.\ \mathsf{istep}_n\ \eta\ conf = conf' \wedge \\ & \quad \mathsf{ihalt}\ conf' \ \wedge\ conf' \leftrightsquigarrow r \\ \mathsf{Div'} & \forall n.\ \neg\,\mathsf{ihalt}\,(\mathsf{istep}_n\ \eta\ conf) \end{cases}$$

$$[\![\ prog\ ]\!]_{\gg} \stackrel{\text{def}}{=} \mathsf{unfold}\ \big(\mathsf{next\_halt}\ \varnothing\ \langle \mathsf{Decls}\ prog,\ \varnothing,\ \varepsilon\rangle\big)$$

To reconcile this semantics with the oracle-parametrised semantics, we must traverse the output ITrees with respect to an oracle to derive a comparable I/O trace. We define this traversal by an inductive relation, accepting the *tree* and oracle Δ to produce the

resulting trace tr and optional outcome o: $tree \overset{\Delta}{\rightsquigarrow} tr,\ o$.

$$\frac{}{\mathsf{Ret}\ r \overset{\Delta}{\rightsquigarrow} \varepsilon,\ \mathsf{Some}\ r} \qquad \frac{}{\mathsf{Div} \overset{\Delta}{\rightsquigarrow} \varepsilon,\ \mathsf{None}} \qquad \frac{\Delta(e) = (r,\ \Delta') \qquad k\ r \overset{\Delta'}{\rightsquigarrow} tr,\ o}{\mathsf{Vis}\ e\ k \overset{\Delta}{\rightsquigarrow} (e, r) :: tr,\ o} \qquad \frac{}{tree \overset{\Delta}{\rightsquigarrow} \varepsilon,\ \mathsf{None}}$$

Terminal nodes produce empty I/O traces, the Div node further produces no outcome. Traversal of a Vis node invokes the oracle to determine the environment's response r, and traverses the continuation ITree $k\ r$ to produce the final outcome o and a trace tr which is prepended with the current I/O event. Traces can be cut short at any point. Intuitively, this definition produces prefixes of the linear path identified by a given oracle through the many branches of an ITree.

We show per-oracle that these prefixes and outcomes are identical to those produced by CakeML's CESK semantics. By appealing to this result and the equivalences between CESK, relational big-step, and functional big-step semantics, we derive a phrasing of CakeML's top-level semantics in terms of ITrees.

Lemma 5.2. CakeML ITree semantics.

$$\vdash \mathsf{semantics}_{\Sigma}\ \Delta\ prog = \begin{cases} \mathsf{Terminate}\ r\ io & [\![prog]\!]_{\Sigma} \overset{\Delta}{\rightsquigarrow} io,\ \mathsf{Some}\ r \\ \mathsf{Diverge}\ io & \left(\neg\exists r.\ [\![prog]\!]_{\Sigma} \overset{\Delta}{\rightsquigarrow} \ldots,\ \mathsf{Some}\ r\right) \wedge \\ & io = \bigsqcup \left\{ tr \;\middle|\; [\![prog]\!]_{\Sigma} \overset{\Delta}{\rightsquigarrow} tr,\ \mathsf{None} \right\} \end{cases}$$

Terminating programs produce ITrees which can derive the appropriate I/O trace and outcome. Diverging programs produce ITrees with no derivable outcomes, and whose I/O traces have the appropriate upper bound.

5.2.2 Interaction tree compiler correctness for CakeML

Using lemma 5.2, we prove a novel formulation of CakeML's compiler correctness theorem, which concerns single traces in its current form (theorem 1.1, pg. 9). We define an ITree-producing semantics for machine code, deriving a formulation of machine semantics akin to lemma 5.2. By CakeML's existing trace-based correctness theorem, we know that each trace derivable in the source ITree which is not terminated by a type error gives rise to a trace through the machine ITree. The latter trace is either identical to the former, or a prefix terminated by an out-of-memory error. But ITrees can be defined precisely by their derivable traces, a form of extensionality:

$$\vdash t_1 = t_2 \iff \left(\forall \Delta\ tr\ o.\ t_1 \overset{\Delta}{\rightsquigarrow} tr,\ o \iff t_2 \overset{\Delta}{\rightsquigarrow} tr,\ o\right)$$

Quantifying over all derivable traces therefore lifts theorem 1.1 (pg. 9) to a coinductive *pruning* relation between ITrees:

$$\frac{}{conv \vdash \mathsf{Ret}\ r\ \ \mathsf{prunes}\ \ \mathsf{Ret}\ r} \qquad \frac{}{conv \vdash \mathsf{Div}\ \ \mathsf{prunes}\ \ \mathsf{Div}}$$

$$\frac{\forall r.\ conv\ r \implies conv \vdash k\ r\ \ \mathsf{prunes}\ \ k'\ r}{conv \vdash \mathsf{Vis}\ e\ k\ \ \mathsf{prunes}\ \ \mathsf{Vis}\ e\ k'} \qquad \frac{}{conv \vdash \mathsf{Ret\ OOM}\ \ \mathsf{prunes}\ \ t}$$

Pruning equates source and machine ITrees (right- and left-hand sides respectively), with two caveats: the machine ITree may terminate in an out-of-memory node at any point; we consider only paths reached by environment responses which satisfy an arbitrary convention *conv*. We will use such a convention in § 5.3.

ITree-based compiler correctness (theorem 5.3) is phrased in terms of pruning.

Theorem 5.3. ITree-based CakeML compiler correctness.

$$\begin{aligned}&\vdash \mathsf{target\_configs\_ok}\ config\ machine\ \wedge\ \mathsf{safe\_itree}_{\not\Rightarrow}\ conv\ [\![\ prog\ ]\!]_{\not\Rightarrow}\ \wedge\\ &\quad \mathsf{compile}_{\not\Rightarrow}\ config\ prog\ =\ \mathsf{Some}\ code\ \wedge\ \mathsf{code\_in\_memory}\ config\ code\ machine\\ &\quad \implies conv \vdash [\![\ machine\ ]\!]_{\mathrm{M}}\ \mathsf{prunes}\ [\![\ prog\ ]\!]_{\not\Rightarrow}\end{aligned}$$

The source ITree must satisfy $\mathsf{safe\_itree}_{\not\Rightarrow}\ conv$, that is, no type errors are derivable along any path reached by environment responses satisfying the convention *conv*. The definition of $\mathsf{safe\_itree}_{\not\Rightarrow}\ conv$ is nearly identical to that of safe_itree (definition 2.5, pg. 38), except its Vis rule considers only these convention-abiding responses. Note that omitting the convention would make this theorem strictly less general.

A weakness in ITree-based compiler correctness. Theorem 5.3 is technically weaker than theorem 1.1 (pg. 9). It holds only for source ITrees for which *all* convention-abiding traces do not produce type errors: there can be no sequence of environment responses which causes the source program to encounter such an error. However, theorem 1.1 can be applied to a program which encounters type errors with respect to some oracles, but not the particular one being used. This weakness does not limit us: we are concerned only with well-typed PureLang programs, which cannot encounter type errors no matter the environment's responses.

However, we might envisage proving correctness with respect to a particular execution environment, whose characteristics must be encoded in an oracle. In this situation, our current proof strategy would not suffice. By lifting single-trace correctness (theorem 1.1) to every-trace correctness, we inherit a weakness: single-trace correctness says nothing about programs which produce type errors, not even the I/O trace up to the point they encounter a type error. We conjecture that a stronger version of theorem 1.1 could be

proved. Though errors may not be preserved by compilation, it is likely that I/O traces are preserved until the moment the type-error is encountered. Proving this would be a significant undertaking, requiring re-specification and re-verification of most of the CakeML compiler back end.

5.3 Correctness of PureCake

We are now in a position to verify the top-level PureCake compiler, relying on our new ITree semantics for CakeML and its associated end-to-end correctness theorem (§ 5.2). We will later transport this verification to a compiler binary (§ 5.3.1).

First, we must compose the verification of each compiler pass, proving that the postcondition of one pass implies the precondition of the next (equation 4.5, pg. 54); successful type inference implies the first precondition. This produces PureCake's core correctness result, theorem 5.4.

Theorem 5.4. PureCake compiler correctness.

$$\vdash \mathsf{compiler}\ str\ =\ \mathsf{Some}\ ast_{\Rightarrow} \implies \exists ce\ cns.\ \mathsf{frontend}\ str\ =\ \mathsf{Some}\ (ce, cns)\ \wedge\ \mathsf{itree\_rel}\ [\![\ \mathsf{exp\_of}\ ce\ ]\!]_{\mathrm{pure}}\ [\![\ ast_{\Rightarrow}\ ]\!]_{\Rightarrow}$$

If the compiler converts an input string *str* to output CakeML AST $ast_{\Rightarrow}$, then the compiler frontend has *both* successfully parsed *str* to a PureLang compiler expression *ce* and a well-formed constructor environment *cns and* type-checked *ce* to deduce that it is safe. The observable semantics of *ce* and $ast_{\Rightarrow}$ are then related by itree_rel, which effectively equates PureLang and CakeML ITree-based semantics. It is defined as the coinductive interpretation of the rules below.

$$\frac{}{\mathsf{itree\_rel}\ \mathsf{Div}\ \mathsf{Div}_{\Rightarrow}} \qquad \frac{}{\mathsf{itree\_rel}\ (\mathsf{Ret}\ r)\ (\mathsf{Ret}_{\Rightarrow}\ r)}$$

$$\frac{\forall r\ r'.\ \mathsf{ffi\_response\_rel}\ r\ r' \implies \mathsf{itree\_rel}\ (k\ r)\ (k'\ r')}{\mathsf{itree\_rel}\ (\mathsf{Vis}\ (ch, s)\ k)\ (\mathsf{Vis}_{\Rightarrow}\ (ch, s, bs)\ k')}$$

$$\mathsf{ffi\_response\_rel}\ (\mathbf{ok}\ a)\ (\mathbf{ok}_{\Rightarrow}\ bs) \stackrel{\mathrm{def}}{=} \mathsf{length}\ bs = 4098 \wedge \exists l_0\ l_1\ junk.\ bs = [l_0, l_1] +\!\!+ \mathsf{bytes}(a) +\!\!+ junk \wedge \mathsf{length}\ a = 256 \times l_1 + l_0$$

Both ITrees must agree on terminal Ret and Div nodes, and neither may produce a type error. For Vis nodes (which represent observable I/O via FFI calls), this definition must reconcile PureCake and CakeML's differing FFI models (§ 5.1). Both ITrees must agree

on FFI channel *ch* and string argument *s*, but CakeML will also pass some byte-array *bs* to its FFI. Then, the Vis node continuations *k* and *k'* must produce related ITrees after receiving related FFI responses, as dictated by ffi_response_rel. At its heart, this latter relation enforces the FFI convention discussed in § 5.1: PureCake's FFI responses have a maximum length of 4096 bytes, realised in CakeML as the return of a byte array of length 4098 bytes, the first two bytes of which represent the length of the response held in the remainder. I have omitted clauses of ffi_response_rel which concern FFI errors; both ITrees must agree on these too.

The formulation of theorem 5.4 has a drawback: it hides that the PureCake compiler back end is total. Any program accepted by the compiler front end (*i.e.*, which parses and type-checks successfully) is guaranteed to compile correctly to CakeML code. Theorem 5.5 is an alternative formulation which emphasises this. Testing has shown that the front end accepts all of the well-formed programs we have written so far (§ 6.1).

Theorem 5.5. Alternative formulation of PureCake compiler correctness.

$$\begin{aligned}&\vdash \mathsf{frontend}\ str\ =\ \mathsf{Some}\ (ce,\ cns)\ \Rightarrow\\ &\quad \exists ast_{\mathbb{S}}.\ \mathsf{compiler}\ str\ =\ \mathsf{Some}\ ast_{\mathbb{S}}\ \wedge\ \mathsf{itree\_rel}\ \ [\![\ \mathsf{exp\_of}\ ce\ ]\!]_{\mathrm{pure}}\ \ [\![\ ast_{\mathbb{S}}\ ]\!]_{\mathbb{S}}\end{aligned}$$

To produce an end-to-end correctness theorem for PureCake, we first show that itree_rel implies safety with respect to our FFI convention:

$$\vdash\ \mathsf{itree\_rel}\ t\ t_{\mathbb{S}} \Rightarrow \mathsf{safe\_itree}_{\mathbb{S}}\ \mathsf{ffi\_convention}\ t_{\mathbb{S}}$$

$$\text{where } \mathsf{ffi\_convention}\ r_{\mathbb{S}} \stackrel{\text{def}}{=} \forall bs.\ r_{\mathbb{S}} = \mathbf{ok}_{\mathbb{S}}\ bs \Rightarrow \exists r.\ \mathsf{ffi\_response\_rel}\ r\ bs$$

In other words, as long as the CakeML ITree receives only well-formed FFI responses, it cannot go wrong. Now we can compose theorems 5.3 and 5.4 (pgs. 76 and 77) to produce theorem 5.6, expressing end-to-end guarantees from PureLang compiler expressions *ce* to machine code.

Theorem 5.6. End-to-end correctness.

$$\begin{aligned}&\vdash \mathsf{compiler}\ str\ =\ \mathsf{Some}\ ast_{\mathbb{S}}\ \wedge\ \mathsf{compile}_{\mathbb{S}}\ config\ ast_{\mathbb{S}}\ =\ \mathsf{Some}\ code\ \wedge\\ &\quad \mathsf{target\_configs\_ok}\ config\ machine\ \wedge\ \mathsf{code\_in\_memory}\ config\ code\ machine\\ &\quad \Rightarrow \exists ce\ cns.\ \mathsf{frontend}\ str\ =\ \mathsf{Some}\ (ce,\ cns)\ \wedge\\ &\qquad \mathsf{ffi\_convention}\ \vdash\ [\![\ machine\ ]\!]_{\mathrm{M}}\ \mathsf{prunes}\ \ [\![\ \mathsf{exp\_of}\ ce\ ]\!]\end{aligned}$$

If some machine code is generated by the successful composition of the PureCake and CakeML compilers, and correctly installed in a valid machine, that machine has a semantics which prunes the semantics of the PureLang compiler expression produced by the PureCake front end whenever the FFI convention is obeyed.

Later in this dissertation (pg. 130), I will revisit this end-to-end correctness theorem and compose it with the contributions described in Part II.

5.3.1 A verified compiler binary

To verifiably compile PureCake, we mimic CakeML's verified bootstrapping (§ 1.2.2): proof-producing synthesis generates CakeML AST which correctly implements the PureCake compiler, and in-logic evaluation of the CakeML compiler lowers this AST to a verified binary. This binary takes the entire step from PureCake concrete syntax to CakeML's S-expression syntax, *i.e.*, the step shown in theorem 5.4 (pg. 77) post-composed with CakeML's S-expression printing.

Note that we do not produce a binary which takes the full step to machine code: users must pipe the S-expression output of the above binary into a CakeML binary, along with flags to enable S-expression parsing and to disable type inference.

This indirection is an unfortunate trusted step, but one which exists only for superficial engineering reasons rather than any deeper verification concerns. In particular, the time taken to produce a verified binary is dominated by slow in-logic evaluation of the compiler: at the time of writing, PureCake takes four hours, but CakeML takes two days, which would slow down the PureCake development cycle significantly. This means we must be sure to use only compatible PureCake and CakeML binaries. There are no significant obstacles to composing the compilers in a single verified binary if we so wish.

Chapter 6

Discussion

I have now presented PureCake, an end-to-end verified compiler for PureLang, a featureful, Haskell-like language. To the best of my knowledge, PureCake lifts the achievements of CompCert and CakeML to the purely functional paradigm for the first time, and incorporates novel formalisations of indentation-sensitive PEG parsing, two-phase constraint-based type inference, demand analysis, and monadic reflection.

In this chapter I discuss the contributions presented in this part, in particular: PureCake's real-world usability (§ 6.1), its place amongst related work (§ 6.2), and potential strands of future work (§ 6.3).

6.1 Usability of PureCake

In this section, I discuss the real-world usability of the PureCake ecosystem. I consider the expressivity of PureLang (§ 6.1.1), measure the effectiveness of some of PureCake's optimisations (§ 6.1.2), and contextualise the performance of PureCake-generated code using the unverified Glasgow Haskell Compiler (§ 6.1.3).

I do not claim parity with existing industrial implementations of Haskell in expressivity, usability, or performance. Rather, I hope to identify key areas of improvement for future versions of PureCake, and reflect on the challenges of producing a realistic verified implementation of a Haskell-like language.

6.1.1 Expressivity of PureLang

We have written non-trivial programs to demonstrate expressivity of PureLang and usability of the PureCake compiler. These include the benchmark programs used in § 6.1.2 and code inspired by Haskell's `Prelude` (*e.g.*, functions over lists and binary trees). We have encountered no issues: PureCake accepts all the well-formed programs we have written.

QuviQ[1] have demonstrated PureCake's applicability to real-world Haskell code by compiling a virtual machine for smart contract evaluation from the Haskell-based Cardano blockchain platform. They have further compared the input-output behaviour of the resulting binary on automatically generated test cases with one produced by GHC, so increasing confidence both in GHC's compilation and in PureCake's capturing of Haskell semantics. More precisely, they have manually simplified the CEK machine of the Plutus Core language[2] to remove features incompatible with PureLang (mostly modules and type classes), validating correctness of the refinement using test cases generated by QuickCheck[3] (a property-based automatic test case generator). The simplified machine is in the fragment accepted by both GHC and PureCake, excepting differing I/O primitives and cryptographic primitives not yet implemented in PureCake. Further QuickCheck-generated test cases produce no discrepancies when run on both PureCake- and GHC-compiled binaries. Their work is open-source[4] at the time of writing.

However, PureCake does lack several widely-used Haskell features: type classes, richer types (*e.g.*, generalised algebraic data types), rich pattern matching (including function clauses and guarded patterns), and modules. PureCake's first version is a trade-off between tractable verification and language expressivity: it implements a minimal set of features sufficient for real-world usage. In future versions, we intend to add more features (§ 6.3).

6.1.2 Effectiveness of optimisations

We measure the efficacy of PureCake optimisations in the style of an *ablation study*: we disable optimisations individually to highlight their contributions to the performance of generated code. However, it is difficult to isolate optimisations in a verified compiler with multiple intermediate languages: some passes establish invariants relied on by future optimisations, and others bridge gaps between intermediate languages. Worse, single passes such as the inter-language compilation from PureLang to ThunkLang (§ 4.2.2) implement several intertwined optimisations at once. Therefore, we consider only the following, isolatable passes:

- pure, binding group analysis and the associated cleanup within PureLang (§ 3.2);
- demands, demand analysis within PureLang (§ 3.4);
- thunk, the mk_delay smart constructor (§ 4.2.2) and intra-language optimisations within ThunkLang (§ 4.2.3); *and*
- state, intra-language optimisation within StateLang (§ 4.4.3).

[1] http://www.quviq.com/
[2] https://github.com/input-output-hk/plutus/tree/master/plutus-core
[3] https://hackage.haskell.org/package/QuickCheck
[4] https://github.com/Quviq/plutus/tree/purecake/plutus-core

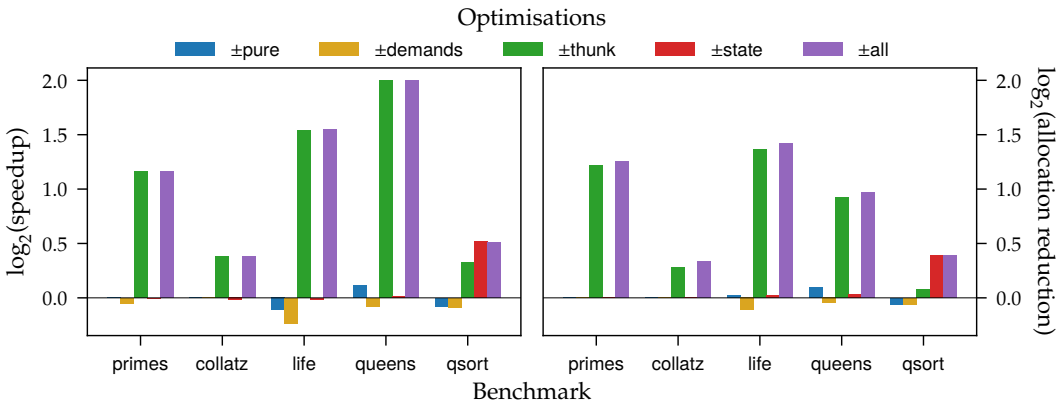


Figure 6.1. Graphs showing the performance impact of optimisations—the base-2 logarithm of a ratio of measurements (execution time or heap allocations in bytes) with/without the optimisation: $\log_2 (m_{\text{without}}/m_{\text{with}})$. The ±all bar shows impact with/without all optimisations considered. Negligible error bars are omitted.

Benchmarks and measurements. We use five benchmark programs to measure performance. Each accepts integer n from the command line and prints:

- primes, the nth prime calculated using two methods: testing each integer for divisors, and the sieve of Eratosthenes;
- collatz, the longest Collatz sequence of any number less than n;
- life, the nth iteration of Conway's Game of Life from a particular self-contained initial state;
- queens, the number of solutions to the n-queens problem; *and*
- qsort, quicksort of a reverse-sorted monadic array with length n.

Each program is implemented relatively naïvely to permit measurable run durations. 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 (3.8 GHz) and 64 GB RAM.

Results. Figure 6.1 shows our results, presented as a base-2 logarithm-scale graph of speedup and reduction in memory allocations. Optimisations reducing unnecessary thunk allocation and forcing in ThunkLang improve time and space efficiency considerably. Intra-StateLang optimisations improve efficiency for the monad-heavy qsort benchmark in particular, by reducing suspended computations and their applications to **unit**. Binding group analysis in PureLang has no significant negative effects.

Demand analysis can cause slight regressions as it is overly aggressive, inserting many **seq** operations in PureLang to produce many **force** operations in ThunkLang.

Many of these are not sufficiently handled by later optimisations. This could also explain why binding group analysis in PURELANG can cause slight negative effects: it converts non-recursive **letrec**-statements to **let**-statements, over which demand analysis operates more effectively. For now, we have disabled **seq**-insertion at call-sites pending development of better heuristics; without this modification performance is significantly worse (almost 3× for the life benchmark). Note that though demand analysis can cause increased allocations, it can also produce better *liveness properties* by ensuring data is live for shorter durations, which enables more effective garbage collection and a lowered heap footprint. For example, we can repeatedly apply the simple `add` function below to an infinite list of zeros and then index the first element of the resulting list. This quickly exhausts heap space unless first transformed by demand analysis. This function is inspired by calculations in the life benchmark: each element of a list is iteratively updated based on the values of its surrounding elements. Indeed, the life benchmark uses the `seq` operator twice: removing these usages slows it down, but disabling demand analysis at the same time causes it to exhaust heap space.

```
add :: Integer -> [Integer] -> [Integer]
add x l = case l of [] -> []
          h:t -> (x - h + head t) : add h t
```

6.1.3 Comparison with the Glasgow Haskell Compiler (GHC)

Using the same benchmark programs as § 6.1.2, we compare the efficiency of PureCake-generated code to GHC-generated code. This is possible because PureCake accepts a language almost identical to a subset of Haskell (§ 6.1.1). We need only: reconcile the monad operations of `IO` in PURELANG with those of its counterpart in GHC, including adapting `Array` to GHC's `IOArray`; replace PURELANG primitives with GHC primitives including inserting casts between `Int` and `Integer` where necessary; and use functions defined in GHC's `Prelude` rather than manually defining them as we do in PURELANG. These are minimal changes, mostly removing lines and slightly altering ~50 more.

Measurements. We measure execution time and memory footprint of PureCake programs in the same way as in § 6.1.2. We use two of the optimisation levels of GHC version 9.2.8: its default (`-O0`) which emphasises fast compilation with minimal optimisation; and its first level (`-O1`) which aims to generate high-quality code without excessive compile times. We do not use GHC's highest optimisation level (`-O2`).[5] We time execution of GHC-compiled programs straightforwardly, and use profiling features[6] to determine memory allocations. In particular, we compile programs with the options `-prof`

[5]The GHC developers use `-O1` "to get respectable speed ... when [they] want to measure something." (`https://downloads.haskell.org/ghc/9.2.8/docs/html/users_guide/using-optimisation.html`)

[6]`https://downloads.haskell.org/ghc/9.2.8/docs/html/users_guide/profiling.html`

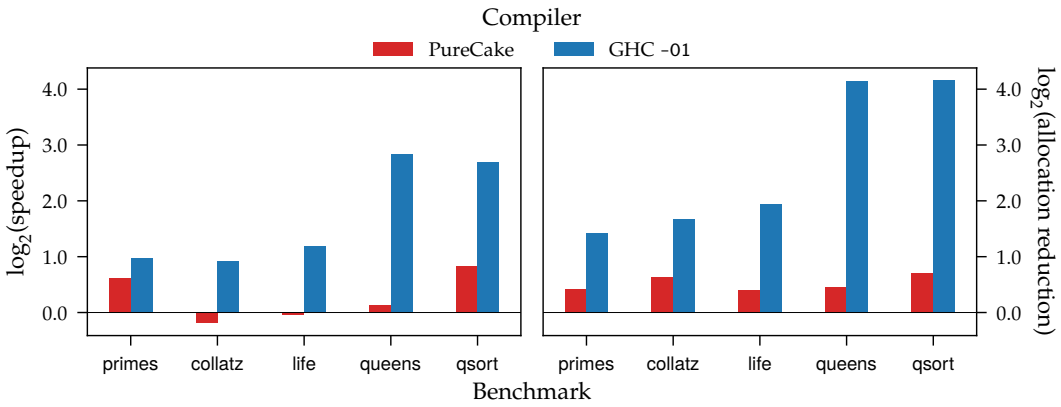


Figure 6.2. Graphs comparing the performance of PureCake- and GHC-generated code. Two optimisation settings are used for GHC: default (`-O0`) and the first level (`-O1`). Base-2 logarithm of speedup and allocation reduction compared to GHC `-O0` are plotted. Negligible error bars are omitted.

`-fprof-auto -rtsopts` and run them with the options `+RTS -p` to dump a `*.prof` file containing a measure of total byte allocations.

Results. Figure 6.2 shows the results, again presented as a base-2 logarithm-scale graph of speedup and reduction in memory allocations taking GHC `-O0` as a baseline.

Overall, PureCake-generated code performs better than code generated by GHC `-O0`, but falls short of `-O1`. In particular, PureCake-generated code is faster than `-O0`-generated code on most benchmarks and allocates less memory on all. When PureCake does produce slower code (collatz and life) the difference is slight. This is a promising result for an early iteration of PureCake, given the decades of development on GHC, including its well-designed Core language and usage of powerful graph reduction techniques. Note that each ecosystem self-reports its memory allocations, but there are no reasons to expect inaccuracies.

The queens and quicksort benchmarks show particular shortcomings of PureCake when compared to GHC `-O1`. Currently, PureCake implements no optimisations specific to data types (*e.g.*, deforestation/fusion) or arrays (all array naïvely hold only thunks). The effects are most pronounced for memory allocations, which is unsurprising given the high demand for memory in purely functional languages.

6.2 Related work

In this section, I review prior work which is most relevant to PureCake. More general work is described in § 1.3.

6.2.1 Verified compilation of Haskell-like languages

The CoreSpec project[7] has produced work concerning verified variants of Haskell. 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 mechanised reasoning about real-world Haskell library code. The tool itself is unverified, but effectively internalises Haskell's equational reasoning within Coq. Some progress has been made in implementing a naïve compiler[8] from GHC's Core to LLVM IR, aiming to permit verification of semantics preservation from CoreSpec's specification of GHC's Core to Vellvm, a formalisation of LLVM IR [Zhao et al. 2012]. However, no verification has been attempted so far, and the project appears abandoned at the time of writing. In future work, CoreSpec aims to derive an executable Coq model of Core automatically from GHC's implementation, permitting integration of Coq-verified optimisations in GHC as plugins. This would provide a rare link between the implementation of an industrial-strength compiler and a formally-specified semantics within a theorem prover. There is a discrepancy in our aims: CoreSpec focuses on accurately modelling GHC and so must compromise guarantees with unverified steps, where PureCake sacrifices some faithfulness to obtain end-to-end verification.

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 specifies a call-by-name semantics to enable straightforward equational reasoning while supporting a featureful language. We view lazy evaluation as an implementation strategy: a compiler optimisation which improves the performance of naïve call-by-name compilation.

McCreight et al. [2010] define GCminor, a reusable target for garbage-collected languages, and verifiably compile it to CompCert's Cminor. GCminor semantics encapsulates key aspects of garbage collection, distilling both the configurations and proof obligations that compilation from source languages must provide. To demonstrate their approach, they compile a version of GHC's Core to GCminor via Dminor: a strict, first-order, purely functional language with monadic effects. Dminor's novel type system provides memory safety by relying on runtime checks. Much of the pipeline from Dminor to GCminor is verified, focusing on allocations in particular. Allocation of thunks is therefore verified; however, compilation of **force** is not.

GHC's arity analysis pass [Breitner 2015] detects functions which can be η-expanded to permit more efficient code at call-sites. In particular, this enables effective list fusion

[7] `https://deepspec.org/entry/Project/Haskell+CoreSpec`
[8] `https://github.com/nomeata/veggies`

for left-folds, which would otherwise produce inefficient partially-applied closures. To ensure that η-expansion does not break GHC's value-sharing with repeated computations, the transformation is proved correct for a simplified Core language, indirectly relying on Launchbury's natural semantics for lazy languages [Launchbury 1993] to model value-sharing at a suitable level of abstraction (HOLCF [Müller et al. 1999] provides necessary domain-theoretic constructs). This work highlights the difficulty of developing an end-to-end verified compiler: verification of a single optimisation requires significant proof effort even with a much-simplified language and a carefully chosen semantic style. PureCake's design choices must instead compromise between each of its optimisations as well as its compilation to CakeML.

6.2.2 Translation validation of Haskell-like languages

Translation validation [Pnueli et al. 1998] is an alternative method to prove semantics preservation down to a binary. Unverified compilation is followed by an automatic proof that the resulting binary faithfully implements the original source code. This permits verification of code generated by industrial strength compilers, and can also provide an independently verifiable certificate for zero-trust applications. However, per-run automated proof can fail, and can be brittle with respect to evolving compiler implementations.

Krijnen et al. [2022] tackle this inflexibility by developing *translation certification*, which combines automated and manual proof. Their *translation relations* adopt a similar approach to § 4.1: syntactic relations are manually proven to encapsulate semantics-preserving transformations, so automated verification need only prove that code transformations inhabit these relations. As in PureCake, these syntactic proofs are much simpler than the semantic verification of the syntactic relations, so translation certification remains robust to evolving compiler implementations. The independent, convergent evolution of this approach gives us confidence in its importance.

Translation certification is applied to the Plutus Tx compiler from the Cardano blockchain platform, which targets Plutus Core (mentioned in § 6.1.1). Plutus Tx is a subset of Haskell, and its compiler implemented as a plugin for GHC: GHC machinery lowers Plutus Tx to a System F-like intermediate representation, which is then compiled to Plutus Core. The latter step is broken down into several syntactic relations, and compiler output can be proven to inhabit their composition automatically and reliably. Semantic verification of the syntactic relations is ongoing work at the time of writing.

6.2.3 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 [Landin 1964] 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 1992; 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.

6.2.4 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. 2021b].

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 (§ 2.4.2).

6.2.5 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 communications between a system of endpoints. Compilation projects individual endpoints out of the global specification, producing a program per endpoint such that simultaneous execution of all programs implements the global specification.

Hupel and Nipkow [2018] compile Isabelle/HOL functions to CakeML using a mixture of certifying and verified compilation, removing the formalisation gap when extracting programs verified in-prover (*cf.* CakeML's proof-producing synthesis, § 1.2.2, for similar motivation in HOL4). Initial certifying passes compile Isabelle's type classes to dictionaries and reify terms to a deeply-embedded data type. This data type is correctly compiled to CakeML via a series of optimisation passes. The net result is translation of shallowly embedded definitions to deeply embedded CakeML code, like the CakeML

translator. However, where CakeML steps directly from shallow HOL4 to deep CakeML, Hupel and Nipkow [2018] step from shallow Isabelle/HOL to deep Isabelle/HOL, then to deep CakeML. Only the first part of this is certifying (*i.e.*, per-run, and can fail); the second is proven correct once and for all.

6.2.6 Reconciling oracles and ITrees

As in § 5, 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 2011] separation logic for CompCert's Clight by specifying their permitted external interactions using ITrees. 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 § 5.2, their proofs rely on traversal of ITrees to derive traces. They need only 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, as this does not compromise soundness of the separation logic.

6.3 Future work

The work presented in this part is only a first version of PureCake, and we have identified several avenues of future work. So far, PureCake has grown organically, beginning with PureLang and its equational theory (§§ 2.2 and 2.4). With the insights gained from its complete implementation and end-to-end verification, we can carefully plan future work.

Compiler front end. PureCake's front end is the most obvious candidate for improvement, to bring it more in line with real-world Haskell implementations. For example, PureCake currently lacks user convenience features such as a module system.

Type class support will require a significant re-design. Parsing must be enriched to support the necessary syntactic constructs, and could further be verified to conform to a context-free grammar once PureLang concrete syntax has stabilised. We then envisage creating a typed AST (TAST) directly above PureLang to ensure static resolution of type classes. Type inference must then bridge the gap between the untyped AST produced by parsing and TAST (*i.e.*, annotating expressions with their types, not just checking them). Our proof-of-concept inference must gain type class support, inspired by Helium, and can further port more of Helium's constraint representations and precision. A standard dictionary construction [Hall et al. 1996] can lower TAST to the untyped PureLang, which must be enriched with records. Defining a semantics for TAST (which is typed) could

permit our end-to-end theorem to span from TAST to machine code, but in HOL4's simply typed logic we must carry around cumbersome invariants concerning well-typing instead of baking them in using generalised algebraic data types.

We could further consider a typed equational theory over TAST to permit clean verification of intra-TAST optimisations. For example, we could flatten nested pattern matches by relying on typing information. Alternatively, we could split the TAST-to-PureLang step in two by compiling TAST to a HOL4-mechanised GHC Core (which is also typed). This would permit direct lifting of Core-level optimisations from GHC (some are mentioned in § 6.2.3).

Our demand analysis requires better heuristics for **seq**-insertion, and implementation of supporting optimisations in ThunkLang (see below). Support for recursion too can be extended, *e.g.*, removing the requirement for A-normal form.

Compiler back end. We implement a small number of optimisations in ThunkLang which reduce unnecessary laziness, but much more work is required here. In particular, there is a disconnect between demand analysis' insertion of **seq**s and their usage by optimisations in ThunkLang. To simplify verification of intra-ThunkLang optimisations, we could consider formalising an equational theory for ThunkLang using step-indexed logical relations [Ahmed 2004]. Otherwise, we can port back end optimisations from GHC and other Haskell implementations.

Further afield, we envision a verified, `ghci`-like read-evaluate-print loop (REPL) for PureCake inspired by CakeML's own verified REPL [Sewell et al. 2022].

Part II

A realistic machine semantics for compiler verification

Outlook

Rigorously verifying the behaviour of software requires faithful modelling of the hardware on which it runs. An instruction set architecture (ISA) provides a convenient abstraction of hardware behaviour, carving out an envelope of permitted behaviours for the family of processors which implement it. Hardware designers are responsible for ensuring that each processor adheres to its ISA, so that software developers need only consider the ISA's specification to gain portability across an entire family of processors. Software verification with respect to a single ISA is then equally valid for all of its implementations, including ones which are yet to be designed.

However, modern, mainstream ISAs are highly complex. Documentation can run into thousands of pages and covers topics such as: processor state (operating mode, registers, and memory); instruction encoding, semantics, and assembly syntax; memory models, memory protection (virtualisation), memory attributes, and caching; synchronisation and semaphores; debug, trace, and monitoring; interrupts, timers, and exceptions.

Compiler verification efforts often rely on specifications which greatly simplify this complexity, compromising their connection to reality and so undermining the foundations of claimed results.

In this part, I develop techniques to handle the complexity of authoritative, high-fidelity specifications of Arm ISAs for use in compiler verification. I apply these techniques to CakeML by using such a specification to strengthen trust in the correctness of its compilation to Arm processors. First, I extract a near-complete formal specification of the Armv8 ISA from official Arm descriptions mostly automatically (§ 7). Second, I tame this complex specification once and for all so that compiler verification efforts can rely on its high fidelity without the burden of its complexity (§ 8). Third, I use this tamed specification to re-derive correctness of the CakeML compiler, producing the first compiler correctness theorem connected to an official ISA specification for a mainstream architecture (§ 9). Finally, I discuss the implications of these techniques and their place in the wider research landscape (§ 10).

Chapter 7

Instruction set specification

To reason about an instruction set with high confidence, we require a *machine-readable* specification of its semantics, as opposed to traditional prose and pseudocode. For verified compilation, this specification must be readable by a theorem prover. In this chapter, I use existing state of the art tools to generate a HOL4-readable Arm instruction set specification from official Arm sources.

First, I describe these existing tools: domain-specific languages for the engineering of instruction set specifications (§ 7.1), particularly those used to specify Arm architectures (§ 7.2). Then, I leverage the tools to produce an Arm specification in HOL4 mostly automatically (§ 7.3). I believe this is the first such specification used for interactive proof in HOL4. Therefore, aside from informing future chapters, this chapter is a summarised usage report aiding future (re)production and (re)use of such specifications.

7.1 Machine-readable specifications

Usage of machine-readable instruction set specifications has many benefits, including:

- Avoidance of ambiguities and errors detected by parsing and type-checking;
- Robust validation paths through simulation-based testing;
- Formal verification of processor designs through model-checking [Goel et al. 2020; Reid 2016d; Reid et al. 2016];
- Support for other formal modelling and verification activities, *e.g.*, work on:
  - memory and concurrency models [Flur et al. 2016, 2017; Gray et al. 2015; Pulte et al. 2018],
  - security [Bauereiss et al. 2022; Chlipala 2019; Nienhuis et al. 2020; Reid 2017f],
  - OS and hypervisor verification [Baumann et al. 2016; Gu et al. 2016; Guanciale et al. 2016; Klein et al. 2009; Leinenbach and Santen 2009; Li et al. 2021a],

- compiler and runtime verification [Fox et al. 2017; Leroy 2009; Myreen 2010; Myreen and Davis 2011; Myreen and Gordon 2009], and
- machine code verification [Armstrong et al. 2021; Fox 2015; Goel and Hunt 2013; Lindner et al. 2019].

In general, they do not cover *all* aspects of ISAs comprehensively, but provide at least a working reference semantics for machine code. As ISAs become more complex, it has become necessary to create domain-specific languages (DSLs) to streamline rigorous engineering of their specifications: *architecture specification languages*. The domain of architecture specification lends itself to languages with some common features.

First-order, imperative, statically typed. The simplicity of first-order languages is well-suited to the description of low-level processor behaviour, and permits efficient translations to *e.g.*, C simulation code. As with any DSL, there are implementation and readability penalties associated with unnecessary expressivity. Some form of looping construct is generally supported.

Imperative operations access a global mutable state, which represents the state of the processor (including registers, flags, memory, and configuration). Typically, the state can be declared incrementally, introducing each part in the context in which it is needed.

Strong static typing enables early error-detection, and type inference reduces the burden of type annotations. Users can often define records; some languages support polymorphic options/lists or even definition of algebraic data types. A built-in "undefined" value models architecturally unknown or underspecified values.

Scattered functions. Large function definitions can be split up and scattered through source files, effectively defined piecewise in amongst other definitions. For example, a specification must define the encoding, decoding, and execution behaviour of each instruction it models. In a language without scattered functions, this results in three large functions (encoder, decoder, and executor), each of which defines a clause per instruction. Using scattered functions, each such clause can be defined separately. To specify an instruction *instr*, we need only define its clauses in isolation:

$$\mathsf{encode}\ \mathit{instr} \stackrel{\mathsf{def}}{=} \ldots \qquad \mathsf{decode}\ \mathit{instr} \stackrel{\mathsf{def}}{=} \ldots \qquad \mathsf{execute}\ \mathit{instr} \stackrel{\mathsf{def}}{=} \ldots$$

In this way, all definitions relevant to *instr* can be localised, mimicking the documentation style of architecture reference manuals. Using a machine-readable specification as human-readable documentation is therefore straightforward.

In the restricted domain of architecture specification, scattered functions provide a solution to the *expression problem* [Wadler 1998], a tension in programming language

design: does a language permit straightforward extensibility of *both* data type syntax *and* data type behaviour? In other words, can we add new data constructors (*i.e.*, a new instruction type) and new behaviour (*e.g.*, an alternative decoder) without refactoring existing code? With scattered functions, this is straightforward: a new instruction need only define its entries in existing functions; a new function need only define clauses for each instruction. Critically, the new code can be appended to the existing specification.

There is a caveat: non-orthogonal/overlapping scattered clauses can be difficult to decipher. Often, clause priority is dictated by order of appearance.

Bit vector support. Strong bit vector support is necessary for streamlined specification of processor behaviour. Bit vectors are therefore available as primitives, with a wide array of built-in operations. Support for pattern matching on bit vectors also simplifies decoder specification. Bit vectors can be translated to efficient machine-words for simulation (*e.g.*, 64-bit bit vectors can be simulated by 64-bit unsigned integers).

Bit vector types ($\mathsf{bits}(n)$) usually support some lightweight dependent typing over the length of the bit vector (n). The degree of support varies considerably amongst architecture specification languages, but can permit reasoning about:

- bit vector concatenation and slicing;
- valid indexing of bit vectors and arrays;
- restricted domains of functions which *e.g.*, only accept whole-byte bit vectors;
- length dependencies, *e.g.*, one value describing the length of another, variants of instructions which operate on different bit vector sizes.

7.2 Arm specifications

Specifications of Arm ISAs are available in three architecture specification languages: ASL [Reid 2017e], Sail [Armstrong et al. 2019], and L3 [Fox 2012]. ASL emerged in 2011, building on the pseudocode language used in Arm documentation since the late 1990s. L3 and Sail were conceived in academia concurrently, with differing objectives; L3 was implemented in 2011 ahead of Sail.

In this section, I describe each of these languages at a high level to better inform the rest of this part. To give a flavour of each language, I show simple specifications of addition/subtraction (immediate) instructions (assembly shorthands ADD[s] and SUB[s]); readers need not understand every detail of these. Later, we will use ASL and Sail to obtain a high fidelity Arm specification in HOL4 (§ 7.3), and rely on L3's state of the art support for theorem proving in our own interactive proofs (§ 8).

7.2.1 ASL

ASL is Arm's own architecture specification language, with broad design objectives. Originally, it was intended to reduce errors in the informal pseudocode found in Arm documentation by providing parsing and type-checking [Reid 2016a, 2017a]. The language has since evolved and is now in extensive use within Arm.

Figure 7.1(a) shows an ASL specification of the execution of ADD[s]/SUB[s] instructions. ASL's type system provides expressive lightweight dependent types but does not bounds-check bit vector accesses. This avoids the need for flow-sensitive type inference, that is, type inference which takes control flow into account [Tobin-Hochstadt and Felleisen 2008]. It is therefore simpler to provide an efficient ASL implementation, clear error messages, and predictable behaviour for programmers. An exception-handling mechanism streamlines specification of error cases.

ASL's design prioritises fully-automated verification and testing (model-checking, SMT verification, and simulation). Arm produces near-complete ASL specifications for its ISAs via the same internal processes used to generate conventional architecture documentation. These are further subjected to rigorous internal testing to ensure full architecture compliance [Reid 2016d, 2017d,e]. They are then released publicly, allowing users to inspect and even run them.

7.2.2 Sail

The Sail language [Armstrong et al. 2019] and ecosystem is actively developed by the Rigorous Engineering of Mainstream Systems (REMS)[1] project. Initially designed for use in concurrency tools, it is now one-size-fits-all: it supports many ISAs, and its rich ecosystem permits extraction to diverse targets such as theorem proving (interactive and automated), symbolic evaluation, simulation, and documentation. Its design is therefore a careful balance of expressivity: it must idiomatically model real-world ISAs, while keeping translations simple.

I defer showing a Sail specification of ADD[s]/SUB[s] instructions until § 7.3.1. The language effectively supports a superset of ASL to permit automatic translation from ASL [Armstrong et al. 2018a] using a tool known as `asl_to_sail` (§ 7.3.1). Its type system tracks effects to ensure memory and register accesses are visible at the type level, supports dependently typed integer ranges, and statically bounds-checks all bit vector accesses. The latter is achieved using flow-sensitive type inference over liquid types [Rondon et al. 2008]. A bidirectional algorithm [Pierce and Turner 1998] generates constraints, propagates them according to program flow, and discharges them using the Z3 SMT solver [de Moura and Bjørner 2008]. In general, users need only provide

[1] `https://www.cl.cam.ac.uk/~pes20/rems/`

```
__instruction add_sub_immediate
  __encoding adds ...
  __encoding add ...
  __encoding subs ...
  __encoding sub ...
  __execute
    bits(datasize) result;
    bits(datasize) op1 = if n == 31 then SP[] else X[n];
    bits(datasize) op2 = imm;
    bits(4) nzcv;
    bit carry_in;
    if sub_op then
      op2 = NOT(op2);
      carry_in = '1';
    else carry_in = '0';
    (result, nzcv) = AddWithCarry(op1, op2, carry_in);
    if setflags then PSTATE.[N,Z,C,V] = nzcv;
    if d == 31 && !setflags then SP[] = result;
    else X[d] = result;
```

(a) ASL. I omit `__encoding` directives, which specify opcode variants and the decoding of each variant to input values *datasize*, `n`, `imm`, `sub_op`, `setflags`, and `d`. Operands $\texttt{op}_1$ and $\texttt{op}_2$ are read from registers `SP`/`X[n]` and immediate `imm` respectively, and added/subtracted by `AddWithCarry` according to the flag `sub_op`. The result is saved, and Arm's condition flags within processor state `PSTATE` are updated according to the flag `setflags`.

```
define Data > AddSubImmediate
       (sf::bits(S), sub_op::bool, setflags::bool,
        imm::bits(S), n::reg, d::reg) with S in 32, 64 = {
  op1 = if n == 31 then SP else X(n);
  op2 = imm;
  op2, carry_in = if sub_op then (~op2, true) else (op2, false);
  result`S, nzcv = AddWithCarry (op1, op2, carry_in);
  SetTheFlags (setflags, nzcv);
  if d == 31 and not setflags then SP ← result
  else X(d) ← result
}
```

(b) L3. This specification closely mirrors the ASL above.

```
dfn'AddSubImmediate (sf,sub_op,setflags,imm,n,d) state =
 (let
   (op2,carry_in) = if sub_op then (¬imm,T) else (imm,F);
   (result,nzcv) =
     AddWithCarry (if n = 31w then SP state else X n state,op2,carry_in);
   s = SetTheFlags (setflags,nzcv) state
  in if d = 31w ∧ ¬setflags then write'SP result s else write'X (result,d) s)
```

(c) L3-derived HOL4 (pretty-printed). This specification is readable and idiomatic, and maps directly to the original L3 above. Processor state is now passed as an argument and the definition of $\texttt{op}_1$ has been inlined at the call to `AddWithCarry`.

Figure 7.1. ASL, L3, and L3-derived HOL4 specifications of addition/subtraction (immediate) instructions (assembly shorthands: ADD[S], SUB[S]).

top-level type signatures for Sail to reconstruct full typing information.

Other notable language features include register references and bidirectional mappings. Register references allow users to pass handles to registers as arguments, simplifying specification of some ISA features. Bidirectional mappings permit definition of both directions of a bidirectional function at once, *e.g.*, conversions from assembly language syntax to their encodings and *vice versa*. This technique has been applied to RISC-V specifications, but Arm assembly syntax is considered too complex for now.

Sail's ecosystem provides toolchains to translate from ASL to Sail, and from Sail to specifications in Coq and Lem, executable simulators in OCaml and C, and SMT formats. Lem, in turn, is a lightweight language for engineering reusable semantic models, inspired by both functional programming languages and proof assistants [Mulligan et al. 2014; Owens et al. 2011]; its ecosystem provides translations to HOL4 and Isabelle/HOL, amongst others. However, Sail's featureful language and broad ecosystem make it less directly connected to theorem proving back ends than our final language, L3.

7.2.3 L3

L3 [Fox 2012] is designed to manage the complexity of writing ISA specifications for use in theorem provers: previous work constructed verbose specifications directly in HOL4 [Fox 2003; Fox and Myreen 2010]. All L3 specifications can be extracted directly (rather than via Lem) to valid HOL4 and Isabelle/HOL, producing streamlined and idiomatic definitions.

Figure 7.1(b) shows an L3 specification of ADD[s]/SUB[s] instructions. L3 has a simpler type system than ASL and Sail, supporting only concrete bit vector width restrictions on declared function arguments. Such restrictions produce functions which are defined only for bit vectors of certain widths. It additionally supports built-in finite maps and finite sets.

However, L3's ecosystem minimises the effort of specification-writing and maximises usability within prover back ends. Construction of an AST for instructions is prioritised, with support for a user-defined hierarchy of instruction classes. Instruction syntax, encoding, and semantics can be declared simultaneously using a packaged declaration form. Register types can be given named sub-fields for easy access, and reads/writes can be overloaded to hide complexity seamlessly. Two styles of specification can be extracted from L3: one which uses a state monad to keep track of processor state, another which uses HOL4's let-expressions. The latter produces highly idiomatic definitions in HOL4, *e.g.*, the extracted HOL4 specification of ADD[s]/SUB[s] instructions in figure 7.1(c).

L3 is designed to favour specification of valid code paths: exceptions can be declared and raised, but not handled. Extensive automation simplifies creation of *next-step libraries*: concise Hoare triples which provide tractable, rule-based instruction semantics

in interactive proof by explicitly stating conditions for well-definedness of instructions (*i.e.*, avoiding exceptions and unknown or underspecified behaviour). These Hoare triples are based on opcode patterns, tackling whole classes of instructions at a time.

Unofficial L3 specifications have been produced for many different architectures: Armv4 through to Armv8 (the latter AArch64 mode only), MIPS, x86 (core 64-bit mode instructions only), and RISC-V. The Armv7 specification in particular has been extensively validated against Arm hardware. To keep generated prover specifications idiomatic, L3 Arm specifications refactor official ASL ones in certain key areas, *e.g.*, preferring bit vectors to integers, sharing common logic where possible, and manually defining efficient instruction decoders.

L3's approach to trustworthiness is to implement a syntax similar to the pseudocode in architecture reference manuals, allowing specifications to mirror manuals closely. For example, the specification in figure 7.1(b) resembles the one in figure 7.1(a). The extracted theorem prover model is considered the trusted specification.

7.3 Generating a HOL4 specification for Armv8

We have seen that Sail's ecosystem includes an ASL front end to translate official ASL specifications into Sail, and a HOL4 back end to translate Sail specifications into HOL4 via Lem (§ 7.2.2). Put together, these provide a pathway from official ASL specifications to HOL4. To the best of my knowledge, there is currently no way to obtain a more faithful Armv8 specification in an interactive theorem prover: the extensive Arm-internal validation of ASL specifications provides the closest approximation to ground truth for the semantics of the Arm ISA.

In this section, I review the process of generating a HOL4 specification from official ASL. We rely entirely on the Sail ecosystem, but its design choices have important implications for the resulting specification. Armstrong et al. [2019] provide further details for the interested reader.

The diagram below depicts the pipeline for translation; in the upcoming subsections I describe each stage in turn and show the specification of ADD[s]/SUB[s] that it produces from the ASL in figure 7.1(a) (pg. 96). The Sail ecosystem is under active development, and at present, the process requires some manual intervention. I am indebted to the Sail developers for helping navigate this space.

$$\text{ASL} \xrightarrow[\S\,7.3.1]{\texttt{asl\_to\_sail}} \text{Sail} \xrightarrow[\S\,7.3.2]{\texttt{sail -lem}} \text{Lem} \xrightarrow[\S\,7.3.3]{\texttt{lem -hol}} \text{HOL4}$$

7.3.1 ASL to Sail

The tool `asl_to_sail`[2] translates ASL specifications into Sail. It uses ASLi ("ASL interpreter") [Reid 2020] to parse and type-check input ASL. The MRA tools collection [Reid 2017b] can be used to extract the input ASL from public, XML-format specifications released by Arm. However, we simply use an ASL specification provided by Arm.

Figure 7.2 shows the Sail specification of ADD[s]/SUB[s] instructions extracted from the ASL in figure 7.1(a) (pg. 96). As Sail can be considered a superset of ASL, the translation is relatively naïve. Some optimisations are made, for example turning mutable assignments into immutable let-bindings (*e.g.*, $\texttt{op}_1$ in the running example). Sail's richer type system complicates the extraction, and its flow-sensitive type inference is carefully designed to accommodate automated translation from ASL. For example, ASL's prevalent runtime assertions are taken into account as part of control flow: when checking code after an assertion, Sail can assume that the assertion holds. Even so, some interactive patching is required. In these cases, `asl_to_sail` halts to request a patch, displaying the original ASL, generated Sail, and the failed typing derivation. The changes required are often straightforward restrictions to permit inference of tighter typing constraints. For example, lifting subexpressions to immutable let-bindings, or specialising type signatures with effect annotations or bit vector width restrictions. However, Sail's type derivation output can be difficult to understand without in-depth knowledge of the type-checker.

Some ASL primitives are concretised in Sail. One important example is `SEE`, a construct used to disambiguate overlapping opcode spaces: the specifications of many Armv8 opcodes refer to another overlapping instruction, so `SEE` directs readers to "see" the other instruction. Therefore, in ASL it accepts a single instruction as an argument. In Sail, `SEE` is concretised using a designated exception, `Error_See`. It is up to the caller to handle this exception when invoking a function which might throw it: decoding/executing a particular opcode requires calling the decoder enough times to resolve the overlap. But we must be careful to avoid looping repeatedly on the same `Error_See` exception, so Sail introduces a global integer variable, also known as `SEE`. Each decoding/execution function clause is numbered by appearance, and is guarded: it can only execute if its assigned number is less than the current value of the `SEE` variable. Each clause body also updates `SEE` to its assigned number as its first action. Therefore, once we have entered a function clause, we cannot execute it again in the same decoding attempt because its guard will fail: `SEE` grows monotonically during repeated decoding attempts, and we progress through any chains of referrals systematically. Once a function clause has been fully executed successfully, `SEE` is set to `-1`.

We are grateful to Arm Limited and the Sail developers for providing an Armv8.6 A-class ASL specification and the necessary patches to translate it.

[2] `https://github.com/rems-project/asl_to_sail`

```
val execute_add_sub_immediate :
  forall 'd 'datasize 'n ('setflags : Bool) ('sub_op : Bool),
    (0 ≤ 'n & 'n ≤ 31 & 'datasize in {32, 64} & 0 ≤ 'd & 'd ≤ 31).
  (int('d), int('datasize), bits('datasize), int('n), bool('setflags), bool('sub_op))
    -> unit effect {escape, rreg, undef, wreg}

function execute_add_sub_immediate (d, datasize, imm, n, setflags, sub_op) = {
  result : bits('datasize) = undefined;
  let op1 : bits('datasize) =
    if n == 31 then SP_read(datasize) else X_read(datasize, n);
  op2 : bits('datasize) = imm;
  nzcv : bits(4) = undefined;
  carry_in : bits(1) = undefined;
  if sub_op then {
    op2 = not_vec(op2);
    carry_in = 0b1
  } else carry_in = 0b0;
  (result, nzcv) = AddWithCarry(op1, op2, carry_in);
  if setflags then (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv;
  if d == 31 & not_bool(setflags) then SP_set(datasize) = result;
  else X_set(datasize, d) = result;
}
```

Figure 7.2. Sail extracted from the ASL specification in figure 7.1(a) (pg. 96). The type signature uses liquid types to constrain bit vector widths and declares possible effects. ASL's implicit `undefined` values have become explicit.

7.3.2 Sail to Lem

Translation from Sail to Lem is more involved: Lem mirrors its HOL4 and Isabelle/HOL back ends, so imperative must become functional, and lightweight dependent types must become simple types. Sail also ships with hand-written Lem libraries encapsulating its built-in types and operations, building on the libraries that ship with Lem itself. I highlight some key aspects of the translation here; figure 7.3(a) (pg. 104) shows the Lem specification of ADD[s]/SUB[s] instructions translated from the Sail in figure 7.2.

State/exception/non-determinism monad

The Lem libraries for Sail include the implementation of a monad designed to represent imperative, effectful Sail code. During translation, Sail specifications are converted to A-normal form [Sabry and Felleisen 1992] to make explicit the calculation of intermediate values, and embedded into a state/exception/non-determinism monad of approximately the following type (where σ is the state type, α the return type, and ε the exception type):

$$(\alpha,\ \varepsilon,\ \sigma)\ \mathsf{M} \stackrel{\text{def}}{=} \sigma \to \big((\alpha,\ \varepsilon)\ \mathsf{Result} \times \sigma\big)\ \mathsf{set} \qquad (\alpha,\ \varepsilon)\ \mathsf{Result} ::= \mathsf{Value}\ \alpha\ \mid\ \mathsf{Ex}\ \varepsilon$$

This monad models non-determinism using a *set* of possible outputs. Monadic return, bind, and exception-handling operators are defined standardly (returnS, bindS, and throwS/try_catchS respectively). The running example uses standard shorthands $\gg$ and $\gg=$ for bindS.

Imperative early-return statements are modelled by extending the exception type to a sum to produce the "early-return monad", in which $(\alpha,\ \alpha + \varepsilon,\ \sigma)$ M is the type of a function which may return its result (of type α) early. In other words, early-return is considered another type of exception, and the early_returnS operator is a shorthand for throwS $\circ$ inj$_{\text{l}}$. Early-return functions are wrapped by catch_early_returnS to embed them back into the original monad; liftRS lifts from the original monad to the early-return one; and try_catchRS catches true exceptions in the presence of early-return:

$$\vdash \text{catch\_early\_returnS (early\_returnS } res) = \text{returnS } res$$
$$\vdash \text{catch\_early\_returnS (liftRS (returnS } res)) = \text{returnS } res$$
$$\vdash \text{catch\_early\_returnS (liftRS (throwS } err)) = \text{throwS } err$$
$$\vdash \text{try\_catchRS (liftRS (throwS } err)) \ f = f\ err$$

Where possible, Sail translates local mutable variables to let-bindings to avoid the need for local monadic state. However, this is much more limited than L3's let-expression extraction mode (§ 7.2.3).

Data representation

Users can choose to translate bit vectors as lists of three-value logic "trits" or machine-words from default Lem libraries. The Lem libraries for Sail define a type class for bit vectors: distinct trit-list and machine-word instantiations permit relatively straightforward switching between the two. There is a key trade-off here: trit-lists are a simpler extraction target, but require extra reasoning about bit vector widths (*i.e.*, list lengths); machine-words have type-backed widths but require further processing (monomorphisation, see below) to target Lem's simply typed setting. We use the machine-word representation (**mword** in the running example): this has stronger library support and more efficient in-logic evaluation procedures within HOL4.

The Lem libraries for Sail also define a record type to represent Sail's register references (§ 7.2.2). During translation, each register declared in the Sail specification produces an instance of the Lem record type. Each such instantiation *reg* has a type $(\rho,\ \nu,\ \tau)$ register_ref, where: the register contains a value of type τ; ρ is the type of the entire register state (*i.e.*, a data type which holds the values of all registers); and ν is an internal representation type gathering all possible register values (*i.e.*, all the instantiations of τ). In practice, ρ and ν are effectively fixed after extracting a Sail

specification to Lem. Register references contain the following components:

Component	Type	Purpose
reg.name	string	canonical register name
reg.read_from	$\rho \rightarrow \tau$	read register from register state
reg.write_to	$\tau \rightarrow \rho \rightarrow \rho$	write register to register state
reg.of_regval	$\nu \rightarrow \tau$ option	conversions to/from the internal ν type
reg.regval_of	$\tau \rightarrow \nu$	

Extracted specifications read from and write to a register *reg* using read_regS *reg* and write_regS *reg new_value* respectively. For example, the second half of the running example repeatedly reads/writes to processor state using `PSTATE_ref`.

Monomorphisation

Sail functions over bit vectors can be both dependently typed and polymorphic, which is incompatible with Lem's simple types. Arm specifications make heavy use of assertions and case splits on bit widths, which produce many dependently typed functions.

Sail attempts to partially *monomorphise* such functions: similar to C++ template expansion, each call-site of a polymorphic function could in theory produce a type-specialised version of its definition. For example, consider a Sail function accepting any bit vector of width divisible by 8, and returning its length in bytes:

$$\mathsf{lenBytes} : \forall n.\ \mathsf{bits}(n \times 8) \rightarrow \mathsf{int}$$

If this is applied to arguments of types bits(16) and bits(32), naïve monomorphisation could produce two type-specialised versions in Lem:

$$\mathsf{lenBytes}_{\mathsf{bits}(16)} : \mathsf{word16} \rightarrow \mathsf{int} \qquad \mathsf{lenBytes}_{\mathsf{bits}(32)} : \mathsf{word32} \rightarrow \mathsf{int}$$

Sail's monomorphisation is inspired partly by prior work translating ASL to Verilog [Reid 2017c]. Case splits on bit vector widths and integer type variables are introduced until these can remain constant throughout a function, and constant propagation through functions determines these widths. Sail's type-checker drives the process throughout.

However, full monomorphisation is not required as Lem permits polymorphism over bit vector widths (*i.e.*, functions over bit vectors can be width-agnostic), and is not desirable as it causes code duplication and alters the structure of the input specification. If a function can be case-split to maintain consistent bit widths over each case, the various cases can be recombined into a single polymorphic definition. In the example above, lenBytes treats bit vectors of types bits(16) and bits(32) uniformly, so $\mathsf{lenBytes}_{\mathsf{bits}(16)}$ and

$\mathsf{lenBytes}_{\mathsf{bits}(32)}$ will be recombined into one final, polymorphic Lem function:

$$\mathsf{lenBytes}_{\alpha} : \alpha\ \mathsf{word} \rightarrow \mathsf{int}$$

To achieve this, Sail inserts bit vector extension operations without changing values, and simplifies type signatures until they become Lem-compatible. It converts bit vector slicing operations (which rely heavily on dependent typing) into masking operations wherever possible. Note that $\mathsf{lenBytes}_{\alpha}$ accepts bit vectors of widths not divisible by 8, where the original Sail function (lenBytes) does not. However, as all call-sites of lenBytes type-check in Sail, we know it will never receive such inputs.

Simplification of type signatures prior to monomorphisation further reduces code duplication. The Sail developers provide the following example, which simplifies load to load' [Armstrong et al. 2019, §4.1]:

$$\mathsf{load} : \forall n,\ n \geq 0\ .\ \mathsf{bits}(64) \rightarrow \mathsf{bits}(8 \times n) \rightarrow \mathsf{bits}(64)$$
$$\mathsf{load'} : \forall n\ m,\ n \geq 0\ \wedge\ m = 8 \times n\ .\ \mathsf{bits}(64) \rightarrow \mathsf{bits}(m) \rightarrow \mathsf{bits}(64)$$

The Lem-inexpressible $\mathsf{bits}(8 \times n)$ type becomes the equivalent, expressible $\mathsf{bits}(m)$, where m is quantified at the top-level. Once again, the Sail type-checker ensures that the constraints on n and m are never violated, and drops them on translation to Lem.

In the running example, bit vector width constraints and declared effects in the Sail type signature of figure 7.2 (pg. 100) are dropped to produce the Lem type signature in figure 7.3(a). Though the Sail width `'datasize` takes two possible values (`32` or `64`), the output Lem is a single function which is polymorphic over `'datasize`.

While Sail remains under active development, the class of specifications which monomorphise successfully is a moving target.

Other code transformations

Scattered function clauses are collected into single, monolithic functions, with clauses ordered by appearance. Each function clause may be guarded in Sail, so the guards are converted to if-statements.

Both ASL and Sail support an "undefined" built-in, used extensively in Arm specifications to model architecturally unknown values. Lem must explicitly implement this built-in: for each declared type in a specification, Sail automatically generates a function to produce the corresponding undefined value. In the running example, `undefined_bitvector` implements undefined values for bit vectors.

Sail unrolls recursive functions whose recursion depth can be determined, removing the need for some termination proofs in theorem prover back ends. Users can also

```
val execute_add_sub_immediate : forall 'datasize. Size 'datasize ⇒
  integer -> integer -> mword 'datasize -> integer -> bool -> bool -> M unit

let execute_add_sub_immediate d datasize imm n setflags sub_op =
  (undefined_bitvector (__id datasize) : M (mword 'datasize)) ≫= fun result ->
  (if eq n 31 then SP_read datasize : M (mword 'datasize)
    else X_read datasize n : M (mword 'datasize)) ≫= fun op1 ->
  let op2 = imm : mword 'datasize in
  (undefined_bitvector 4 : M (mword ty4)) ≫= fun nzcv ->
  (undefined_bitvector 1 : M (mword ty1)) ≫= fun carry_in ->
  let (carry_in : mword ty1, op2 : mword 'datasize) =
    if sub_op then
      let op2 = not_vec op2 : mword 'datasize in
      let carry_in = 0b1 : mword ty1 in
      (carry_in, op2)
    else
      let carry_in = 0b0 : mword ty1 in
      (carry_in, op2)
  in
  let (tup0, tup1) =
    AddWithCarry op1 op2 carry_in : mword 'datasize × mword ty4 in
  let result = tup0 in
  let nzcv = tup1 in
  (if setflags then
    read_reg PSTATE_ref ≫= fun (w2 : ProcState) ->
    write_reg PSTATE_ref
      ◁ w2 with ProcState_N = (subrange_vec_dec nzcv 3 3 : mword ty1) ▷ ≫
    read_reg PSTATE_ref ≫= fun (w3 : ProcState) ->
    write_reg PSTATE_ref
      ◁ w3 with ProcState_Z = (subrange_vec_dec nzcv 2 2 : mword ty1) ▷ ≫
    read_reg PSTATE_ref ≫= fun (w4 : ProcState) ->
    write_reg PSTATE_ref
      ◁ w4 with ProcState_C = (subrange_vec_dec nzcv 1 1 : mword ty1) ▷ ≫
    read_reg PSTATE_ref ≫= fun (w5 : ProcState) ->
    write_reg PSTATE_ref
      ◁ w5 with ProcState_V = (subrange_vec_dec nzcv 0 0 : mword ty1) ▷
  else return ()) ≫
  if and_bool (eq d 31) (not setflags) then SP_set datasize result
  else X_set datasize d result
```

(a) Lem (simplified; most parentheses and type annotations removed).

```
execute_add_sub_immediate d datasize imm n setflags sub_op =def
 do
  result ← undefined_bitvector (id datasize);
  op1 ← if n = 31 then SP_read datasize else X_read datasize n;
  op2 ↞ imm;
  nzcv1 ← undefined_bitvector 4;
  carry_in ← undefined_bitvector 1;
  (carry_in,op2) ↞
   if sub_op then (let op2 = not_vec op2; carry_in = 1w in (carry_in,op2))
   else (let carry_in = 0w in (carry_in,op2));
  (tup0,tup1) ↞ AddWithCarry op1 op2 carry_in;
  result ↞ tup0;
  nzcv1 ↞ tup1;
  if setflags then
   do
    w2 ← read_regS PSTATE_ref;
    w3 ←
     do
      write_regS PSTATE_ref (w2 with ProcState_N := subrange_vec_dec nzcv1 3 3);
      read_regS PSTATE_ref
     od;
    w4 ←
     do
      write_regS PSTATE_ref (w3 with ProcState_Z := subrange_vec_dec nzcv1 2 2);
      read_regS PSTATE_ref
     od;
    w5 ←
     do
      write_regS PSTATE_ref (w4 with ProcState_C := subrange_vec_dec nzcv1 1 1);
      read_regS PSTATE_ref
     od;
    write_regS PSTATE_ref (w5 with ProcState_V := subrange_vec_dec nzcv1 0 0)
   od
  else returnS ();
  if d = 31 ∧ ¬setflags then SP_set datasize result else X_set datasize d result
 od
```

(b) ASL-derived HOL4 (pretty-printed).

Figure 7.3. Lem and HOL4 derived from the Sail specification in figure 7.2 (pg. 100). The HOL4 maps directly to the Lem, though do-notation hides monadic operations and type annotations are omitted.

manually tweak parts of a specification by providing alternative function implementations to splice in. We make use of this splicing feature to modify our specification (§ 8.1).

7.3.3 Lem to HOL4

Lem straightforwardly translates to HOL4, though the raw specifications are not human-friendly until parsed/pretty-printed by HOL4: all expressions are type-annotated and fully-bracketed. HOL4 libraries for Sail are automatically translated from the corresponding Lem libraries, and Lem's simple type classes become HOL4 record types. Figure 7.3(b) shows the HOL4 specification of ADD[s]/SUB[s] instructions extracted from the Lem in figure 7.3(a). Even for these simple instructions, the differences from the L3-derived HOL4 in figure 7.1(c) (pg. 96) are considerable.

Some manual intervention is required for well-foundedness checking: Lem can automatically generate simple well-foundedness proofs, but some Sail library functions are well-founded for non-trivial reasons. These usually require arguments about integer arithmetic. However, naïve definitions for pure and monadic while-looping constructs cannot be proved well-founded. We redefined these functions correctly, deriving theorems showing their behaviour is as originally intended.

We also corrected a minor HOL4 usability issue when working with this substantial specification.[3] We introduced a syntax for let-declarations within monads to enable clear printing of definitions which mix monadic and pure assignments. The two equivalent terms below show the original pretty-printing (left) and the new syntax (↞, right). Removing unnecessary indentation and reducing do . . . od blocks permits interactive inspection of the translated specification. Previously, cumulative indentation on each pure assignment would push definitions off the screen.

```
do
  y ← f x;
  let z = g y in do
    return z
  od
od
```

```
do
  y ← f x;
  z ↞ g y;
  return z
od
```

[3] https://github.com/HOL-Theorem-Prover/HOL/pull/825

Chapter 8

Proofs of semantics preservation

Note the distinction between my goal of enabling compiler verification and the goals of previous efforts built on ASL-derived specifications [Nienhuis et al. 2020; Reid 2017f; Reid et al. 2016]: I use our HOL4 specification as a semantics for Arm machine code, and do not consider (micro)architectural properties such as safety, security, or implementation correctness. Instead, I am interested in proofs of *semantics preservation*, relating the observable behaviour of Arm machine code programs with that of other programs (likely in another language). This is exactly the class of proofs for which L3 was designed.

However, the detailed HOL4 specification we have generated from official ASL is one of the largest and most complicated known ISA specifications in a theorem prover, matched only by other specifications produced from official Arm ASL via Sail. In this chapter, I develop techniques to tame it once and for all for use in proofs of semantics preservation. In particular, I prove that an existing L3 specification simulates the ASL-derived one. Verification efforts can then enjoy the ease-of-use of the former while retaining the faithful modelling of the latter: future work can avoid navigating the complexity of the ASL-derived specification, and other users of the L3 specification benefit from its validation against official ASL.

More specifically, I describe the following. First, I modify the ASL-derived specification conservatively to adapt it to our goal (§ 8.1). Second, I inspect the result more closely, focusing on: why it differs from other ISA specifications (§ 8.2); and how usable its semantics of instructions are within the theorem prover, particularly compared to L3's (§ 8.3). Finally, I prove that the L3 specification simulates the ASL-derived one (§ 8.4).

8.1 Modifying the specification

We modify two aspects of the specification derived in § 7.3: its monad, and its modelling of address translation.

Monad. We remove the set-based non-determinism from the state/exception/non-determinism monad (§ 7.3.2), preferring to use HOL4's Hilbert choice operator (§ 1.2.1) to express unknown values. This is more idiomatic, and streamlines interactive proof. The new monad type is therefore:

$$(\alpha,\ \varepsilon,\ \sigma)\ \mathsf{M} \stackrel{\mathsf{def}}{=} \sigma \to (\alpha,\ \varepsilon)\ \mathsf{Result} \times \sigma \qquad (\alpha,\ \varepsilon)\ \mathsf{Result} ::= \mathsf{Value}\ \alpha\ \mid\ \mathsf{Ex}\ \varepsilon$$

More precisely, set-based non-determinism is not compatible with in-logic evaluation (§ 1.2.1), which we will rely on heavily later (§ 8.4.2). For example, without this change an undefined 64-bit bit vector in Sail would be translated to a set of all 2^{64} possible 64-bit bit vectors in HOL4, which is intractable to evaluate. To generate undefined values for enumerated types, Sail libraries create such an undefined bit vector of sufficient width and cast it to a natural number. This provides an index into a list which enumerates all possible elements of the type. Undefined values for algebraic data types are then constructed by composing undefined values of their constituent types. This too is more cleanly and tractably represented as Hilbert choice. For example, an undefined 64-bit bit vector is simply $\varepsilon(x : \mathsf{word64}).\ \mathsf{T}$, and an undefined value of an enumerated type is $\varepsilon x.\ x \in \{\ldots\}$ (*i.e.*, any element from the set of possibilities).

We must be careful with manual changes in a high-fidelity specification. However, our modifications are conservative: we change only the monad implementation in the hand-written Lem libraries for Sail. This gives us confidence in their validity.

Address translation. We remove address translation, considering it to be a detail orthogonal to many proofs of semantics preservation. This is in keeping with L3 specifications and other specifications modelling the semantics of machine code [Erbsen et al. 2021; Sammler et al. 2022]: if hardware manufacturers ensure that their processors correctly implement address translation, general-purpose software can assume that physical memory has been abstracted correctly to virtual memory, and work only at the level of the virtual address space. Software which must interact directly with address translation (*e.g.*, hypervisors) cannot rely on this assumption.

Using Sail's user-splicing feature (§ 7.3.2), we stub out address translation functions to express an identity mapping between virtual and physical memory. However, in Armv8 AArch64 virtual addresses are 64-bit and physical addresses are up to 52-bit. User-splicing cannot modify types, so we must manually modify the specification to convert the type of physical addresses. Again, we must keep changes conservative to maintain trust in the specification. We rely entirely on the Sail type-checker to help identify a minimal set of required edits, which mostly affect type annotations.

Table 8.1. Metrics for various architecture specification sizes.

(a) Metrics for the extraction of L3 and ASL specifications via the L3 and Sail ecosystems respectively. HOL4 character counts are both as extracted (raw) and after pretty-printing to 80 columns. Timings taken using Intel® Xeon® E-2186G (3.8 GHz, 64 GB RAM).

Original specification		**L3**	**ASL**
Number of non-whitespace characters / 10^6	*Source*	0.053	4.2
	Sail	-	7.4
	Lem	-	19.9
	Raw HOL4	0.20	26.7
	HOL4	0.070	12.2
Size / kLoC	*Source*	2.4	168
	Raw HOL4	8.5	488
Total time to extract		1 s	~ 2 hrs
HOL4 build time		< 30 s	~ 3 hrs

(b) Metrics for other comparable specifications.

Architecture	**Language**	**No. non-whitespace characters / 10^6**
RISC-V	Sail	0.59
x86	ACL2	1.7

8.2 Inspecting the specification

Table 8.1 shows metrics taken throughout the extraction of L3 and ASL specifications to HOL4, as well as similar measurements for other architectures. The difference here is clear—to the best of my knowledge, the ASL-derived HOL4 specification is one of the most complex, unwieldy specifications to be used in interactive proof. Why is this?

Why is the ASL specification so much larger than its peers? It covers more of its intended ISA, modelling most modes/instruction sets, where other specifications tend to formalise a particular mode of operation. For example, the L3 specification covers only AArch64 mode, and also omits vector (SIMD) and floating-point instructions; in general, L3 specifications model mostly user-level code, omitting most system registers/instructions. Until recently [Coglio and Goel 2018; Goel et al. 2017] the ACL2 specification of x86 only covered 64-bit mode.

This is because ASL specifications have differing goals to their peers (§ 7.2.1): primarily they provide documentation, focusing on thoroughness rather than simplicity. Other efforts focus on verification: conciseness and usability are primary goals. For example,

the ASL specification defines IEEE floating-point semantics from scratch to ensure faithful behaviour, whereas other efforts outsource to libraries.

Why does extraction to HOL4 bloat the ASL specification? Representing sequential, imperative code monadically (§ 7.3.2) adds considerable bloat due to conversion to A-normal form and instrumentation with explicit monadic operations. Instead, the L3 specification is extracted using the ecosystem's let-expression mode, keeping nearly all definitions pure (looping constructs remain monadic).

Various translation artefacts (§ 7.3.2) are verbose, for example: succinct bit vector slicing in Sail is often converted to masking; Sail's undefined built-in primitive must be implemented for each declared type; registers passed by reference in Sail require explicit definitions in HOL4.

L3 couples tightly with prover back ends, using relatively simple types and HOL-flavoured constructs to provide an extremely sophisticated syntactic sugar for higher-order logic. Sail is more general-purpose, and not well-optimised for use with HOL4: bit-vector built-ins must be realised in libraries, and these often re-implement HOL4 operations unnecessarily and non-idiomatically. Their definitions often involve many auxiliary functions, and perform unusual conversions during operation. In particular, they typically convert each input machine-word to a list of booleans, manipulate the list to perform their operation, then convert back to an output machine-word. Often, this re-implements a well-supported, idiomatic feature from HOL4 machine-word libraries. A concrete example is given in figure 8.1 (§ 8.3, pg. 111).

Why are extraction and build times so long for the ASL pipeline? A significantly larger specification takes longer to extract and build. However, the increase is not proportional to size alone.

The ASL specification must be type-checked multiple times during extraction: in ASL, in Sail, and in Lem. Monomorphisation is a complex process, and (along with the lightweight dependent types in ASL/Sail) necessitates heavy use of Z3 to discharge constraints during extraction.

Sail produces concrete HOL4 syntax via Lem, which must be parsed and type-checked in HOL4. Instead, L3 relies on HOL4's accessible metaprogramming: it produces Standard ML which stitches together its definitions efficiently. Larger definitions built from Sail's scattered functions are particularly slow, including the decoder. By contrast, L3's decoder is manually defined to minimise its footprint; it tests opcodes in an order which avoids ambiguities from overlapping opcode spaces, thereby also avoiding the extra machinery required to handle `SEE` (§ 7.3.1).

8.3 Working with the specification

We can now evaluate the practicality of our ASL-derived specification for interactive proofs of semantics preservation. Unfortunately, there are some significant obstacles.

Opaqueness to inspection and interaction. The ASL-derived specification does not implement an AST for Arm instructions, in contrast to the L3 specification. Users must work directly with opcodes or manually define an AST.

Explicit monadic operations obfuscate high-level semantics. These are awkward and tedious in interactive proofs, which must step over each monadic operation. L3's let-expression mode mostly removes monadic sequencing instead.

Monolithic HOL4 functions coalesced from ASL's scattered functions are unwieldy, *e.g.*, the 23 kLoC decoding/execution function DecodeA64 (for which each instruction implements a clause). Investigating and interacting with the semantics of a particular instruction is therefore challenging.

ASL specifications and Sail libraries use many auxiliary functions and data manipulations that are not idiomatic in HOL4: many steps of definition expansion are required to inspect or use intended semantics.

Opaqueness to automated evaluation. Non-idiomatic bit vector operations have poor evaluation support in HOL4, often re-implementing functionality in libraries shipped with HOL4. Many operations even convert machine-word operands to bit-lists, perform operations on bit-lists, and convert back. Permeative bookkeeping of these list lengths requires automated discharging of necessary preconditions; by contrast, type-backed widths require no such reasoning. Integers are used throughout even for always-positive constants, despite HOL4's better support for natural number reasoning. I have already mentioned that HOL4 cannot derive various integer-based well-foundedness proofs automatically (§ 7.3.3).

Consider the L3 (left) and ASL-derived (right) HOL4 definitions in figure 8.1, which determine the index of the highest set bit of input bit vector x. The L3 definition uses HOL4 library definitions to convert the base-2 logarithm of the input word to an integer (w2i). The ASL-derived definition is more computational: it tests each bit from high to low, returning early if any is set. However, this intended definition is obfuscated: the early return necessitates embedding within the Sail monad and use of early-return monadic operations (catch_early_returnS and early_returnS); explicit monadic operations are used for looping and sequencing (made more palatable here by do-notation); integer loop variables are used over natural numbers (n2i casts from natural to integer); custom bit vector operations from Sail libraries are used (vec_of_bits and access_vec_dec). The

```
HighestSetBit_L3 x def=
  if x = 0w then −1
  else w2i (word_log2 x)
```

```
HighestSetBit_ASL x def=
  catch_early_returnS
   (let
     lower = 0;
     upper = n2i (word_len x) − 1
    in
     do
      foreachS (index_list upper lower (−1)) ()
       (λ i unit_var.
           if vec_of_bits [access_vec_dec x i] = 1w then
            early_returnS i
           else returnS ());
      returnS (−1)
     od)
```

Figure 8.1. L3 and ASL-derived HOL4 definitions for determining highest set bits.

definitions of these last operations involve a large number of auxiliary functions: they convert x to a list of booleans, access the boolean at index i, and use it to create a 1-bit bit vector. This re-implements HOL4's well-supported bit vector access operation. This small example exemplifies the complexity which pervades the ASL-derived specification. We have already seen an ASL-derived HOL4 specification of the execution of a simple instruction (figure 7.3(b), pg. 104).

8.4 Simulation proofs

In this section, I demonstrate an approach to managing the complexity of the ASL-derived, modified specification from § 8.1. The approach is to tame it using a once-and-for-all proof of simulation, abstracting it interactively to a more theorem-prover-friendly specification. I choose the existing Armv8 L3 specification as the abstraction target, rigorously validating it for the first time and so improving trust in the results of its other users. The results in this section are decoupled from any particular application, and so can be reused by future verification efforts.

8.4.1 Simulation relation

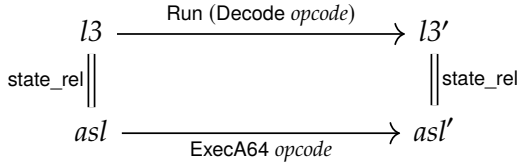
Definition 8.1 expresses our simulation relation, l3_models_asl, a predicate on the AST for instructions defined by the L3 specification. The instruction should successfully encode (using the L3-specified Encode) to produce a 32-bit *opcode*. Given L3 and ASL-derived machine states related by state_rel, if the L3 specification can decode and run

Definition 8.1. Simulation relation between L3 and ASL-derived specifications.

```
l3_models_asl opcode ≝
  Decode opcode ≠ Unallocated ∧
  ∀l3 asl l3′.
    state_rel l3 asl ∧ asl_sys_regs_ok asl ∧ Run (Decode opcode) l3 = l3′ ∧
    l3′.exception = NoException ⇒
    ∃v asl′.
      ExecA64 opcode asl = (Value v,asl′) ∧ state_rel l3′ asl′ ∧
      asl_sys_regs_ok asl′

l3_models_asl_instr instr ≝
  ∃opcode. Encode instr = ARM8 opcode ∧ l3_models_asl opcode
```

(Run (Decode *opcode*)) the opcode without failure, the ASL-derived specification should also run (ExecA64) it successfully, both producing resultant states that remain related by state_rel. In addition, the predicate asl_sys_regs_ok should hold of the ASL-derived state throughout. Intuitively, the relation specifies the following diagram:



Note that we rely on L3 machinery: its AST for instructions and its encoder. These are orthogonal to the semantics of instructions, but use of an AST is well-suited to interactive proof and makes it simpler to carve out classes of opcodes. I have already noted that the ASL-derived specification does not provide such an AST or encoder.

The state equality relation state_rel is effectively a simple inclusion: the ASL-derived specification models strictly more registers and processor state than the L3 one, so state_rel asserts that the specifications agree on the parts modelled by both. I omit the full definition, which is verbose due to superficial differences between the two specifications. In particular, L3 machine states (*l3*) are concise records, with clean access to registers, *e.g.*, *l3*.PC is the program counter. Instead, ASL-derived states (*asl*) group registers together by their types, *e.g.*, *asl*.regstate.bitvector_64_dec_reg of type string → word64 models all simple 64-bit system registers. The sheer number of registers forces this unusual approach: HOL4 struggles to cope with very large records in which each register is declared separately, so they must be grouped. Register references (§ 7.3.2) simplify indexing a particular register, for example PC_ref.read_from *asl*.regstate reads the program counter. This is equivalent to *asl*.regstate.bitvector_64_dec_reg "_PC".

Subtleties

Versioning. The L3 specification models Armv8.0, whereas the ASL-derived specification models Armv8.6. Though this is a "minor" version difference, there are observable effects, *e.g.*, certain system control registers are 32-bit in L3 but 64-bit in ASL. In Armv8.0, these registers were 64-bit with their upper 32 bits reserved, so only their lower 32 bits required modelling. However, ASL faithfully models all 64 bits regardless.

System registers. The predicate asl_sys_regs_ok is necessary as L3 specifications model mostly user-level operation, omitting most system registers. We fix 10 bits of these registers and clear one entirely in the ASL-derived specification to ensure it models a processor in a similar mode of operation:

Specified bit(s)		Purpose
PSTATE.nRW	clear	Using AArch64 mode
SCR_EL3[0]	set	EL0-2 are non-secure
SCR_EL3[10]	set	EL0-2 are not AArch32
SCR_EL3[18]	clear	Disable secure EL2
HCR_EL2[31]	set	EL1 is AArch64
HCR_EL2[34]	clear	Disable FEAT_VHE
TCR_EL1[51-52]	clear	FEAT_PAUTH flag
TCR_EL{2,3}[29]	clear	FEAT_PAUTH flag
HIGHEST_EL_AARCH32	clear	EL3 is not AArch32
CNTCONTROLBASE	clear	Generic timer control frame

The aim is to disable optional features not modelled in L3. The Arm Architecture Reference Manual [Arm Limited 2020] provides a full account of these features. Note that four of these bits are in the TCR_EL{1,2,3} registers which are also modelled in L3. A feature (FEAT_PAUTH) implementing pointer authentication codes (PACs) was made compulsory in Armv8.3 onward, supporting authentication of addresses stored in registers before targeting them for a branch or load. Fixing these bits in the Armv8.6 modelled by ASL aligns behaviour more to the Armv8.0 modelled by L3, by ensuring the feature is applied uniformly to data and instruction accesses.

Memory and registers. The L3 specification represents memory cleanly as a total function from addresses to bytes (word64 → word8). However, the ASL-derived specification models memory as a finite mapping from natural numbers to trit-lists (num ↦ trit list), each intended to represent a byte. It also models per-address validity tags (num ↦ trit). We impose restrictions on ASL-derived memory to ensure we can

equate it to L3 memory: its domain must be exactly the natural numbers representable by 64-bit words; its range must contain only *bit*-lists (*i.e.*, no "unknown" trits) of length 8; all addresses must have a valid tag.

To model Arm's 31 general-purpose registers and zero register, the L3 specification uses a total mapping from 5-bit words (word5 → word64). The ASL specification instead uses a list of registers (word64 list). To access the register n, it takes the nth index of the list. We must require the list to have length of exactly 32.

8.4.2 Proving the simulation

Establishing simulation for a particular instruction effectively requires execution of the instruction on both specifications. We leverage pre-existing automation to execute the L3 specification effectively. However, the difficulties detailed in § 8.3 complicate execution of the ASL-derived specification.

We adopt a partial, symbolic evaluation strategy: we use HOL4's customisable in-logic evaluation library (§ 1.2.1) to bypass the large, monolithic decoding machinery within ExecA64. This is possible because decoding tests only the known, concrete upper bits of opcodes, *i.e.*, those that distinguish it from other classes of opcode. We implement lightweight automation to direct the in-logic evaluation; once this has sufficiently narrowed down the semantics of the given instruction, we proceed by interactive proof. More precisely, our automation mostly bypasses HOL4 specifications derived from the `__encoding` directives mentioned in figure 7.1(a) (pg. 96). Then, interactive proof relates L3- and ASL-derived HOL4 specifications such as the simple examples in figure 7.1(c) (pg. 96) and figure 7.3(b) (pg. 104) respectively.

In interactive proof, we must minimise the complexity that the ASL-derived specification introduces into our goal. We use the L3 specification heavily to provide abstract representations of auxiliary functions, advancing the proof of simulation without unfolding complex ASL-derived definitions. For example, we prove the lemmas below for HighestSetBit from figure 8.1 (pg. 111), and AddWithCarry found in figure 7.1(c) (pg. 96) and figure 7.3(b) (pg. 104).

$$\vdash \mathsf{HighestSetBit}_{\mathrm{ASL}}\ w = \mathsf{returnS}\ (\mathsf{HighestSetBit}_{\mathrm{L3}}\ w)$$

$$\vdash \mathsf{AddWithCarry}_{\mathrm{ASL}}\ x\ y\ \mathit{carry} = \mathsf{AddWithCarry}_{\mathrm{L3}}\ (x,y,\mathit{carry})$$

A notable sub-proof relates the L3 and ASL-derived implementations of DecodeBitMasks, used in decoding to resolve immediate fields. Its 90 LoC ASL definition is obfuscated by its optimised implementation, but a comment block asserts an equivalent 7 LoC version. The L3 specification uses the shorter definition, so we must prove the asserted equivalence: we split up the large function into more manageable chunks, proving

manually-defined specifications for each chunk by brute force enumeration of inputs. The resulting lemma is shown below.

$$\vdash \mathsf{DecodeBitMasks}_{\mathrm{L3}}\ (n,s,r,b) = \mathsf{Some}\ res \Rightarrow \mathsf{DecodeBitMasks}_{\mathrm{ASL}}\ 64\ n\ s\ r\ b = \mathsf{returnS}\ res$$

We instruct HOL4's rewriting engine to perform a variety of automatic simplifications. We prove monad laws to enable automatic reassociation of bindS operations and removal of returnS operations, avoiding repeated manual reassociation and case splits on successful/exceptional return values. Unfolding simple but pervasive definitions (*e.g.*, various auxiliary bit vector operations, definitions of Arm's exception levels, *etc.*) reduces their repeated manual unfolding too.

In total, we prove l3_models_asl (definition 8.1, pg. 112) for the AArch64 instruction classes below, requiring 7.5 kLoC of proof and 0.5 kLoC of simple automation (~40 mins to build, limited by in-logic evaluation). Our reliance on the existing L3 specification keeps the proofs tractable, and no unexpected discrepancies were found in its semantics or encoder (we do encounter a known issue, see § 10.2.3). All of our definitions and proofs thus far are self-contained, and not tied to any particular usage of our ASL-derived specification. To aid reuse in future work, they have been integrated into the HOL4 public repository (§ 1.1.1).

Instruction class description	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.

Chapter 9

Compiler correctness

In this chapter, I use the simulation proofs of § 8.4 to re-derive correctness of CakeML's compilation to Armv8 with significantly increased trust. I believe this is the first compiler correctness result with respect to an official mainstream ISA specification. First, I review CakeML's use of unofficial L3 specifications to verify compilation to multiple architectures (§ 9.1). Then, I show how to derive the new compiler correctness result, which targets our ASL-derived specification (§ 9.2). Later in this dissertation, I will derive a similar correctness result for PureCake (pg. 130).

9.1 Target correctness proofs in CakeML

CakeML targets x86-64, Armv7, Armv8 (AArch64), RISC-V, MIPS, and Silver (a custom ISA for a verified processor [Lööw et al. 2019]), proving compiler correctness with respect to specifications for each. These are all L3-derived, but we are concerned with Armv8.

Both the compiler and its proofs are carefully structured to remain as target-agnostic as possible, reducing the implementation burden and proof obligation of supporting a new target. The following design decisions were made [Fox et al. 2017].

Generic assembly and assembly configurations. The final intermediate language of the CakeML compiler is LabLang, a target-neutral, labelled assembly language. A LabLang program is a series of sections, each composed of lines. A line is either a label or an Asm generic assembly instruction. The idea is that each Asm instruction can be encoded directly in each target language supported by CakeML. This necessitates *e.g.*, parametric machine-word widths to ensure support for differing targets. The type of an Asm instruction with width α is α asm.

However, the reality is not so straightforward: architectures differ considerably in the particular assembly features they support. Therefore, Asm is carefully designed to

trade off expressivity and simplicity; for example, it prefers the more widely supported compare/jump instructions over status registers, though these are idiomatic for certain architectures. The language is further parametrised by an *assembly configuration* (*aconf* : α asm_config) which encapsulates the set of valid features available for a given target. For example, each assembly configuration *aconf* contains the following components:

Component	Purpose
aconf.encode : α asm $\rightarrow$ word8 list	instruction encoder
aconf.big_endian	endianness
aconf.code_alignment	instruction address alignment
aconf.link_reg	optional saved program counter for calls
aconf.avoid_regs	special registers to avoid
aconf.reg_count	no. of general-purpose registers
aconf.two_reg_arith	two-register arithmetic only (*e.g.*, x86-64)
aconf.valid_imm	valid immediate predicate
aconf.addr_offset	min/max address offsets for loads/stores
aconf.jump_offset	min/max jump offsets

Compiler configurations. CakeML's compiler implementation is parametrised by another target-specific configuration record, known as a *compiler configuration* (*cconf*). This configures target-specific aspects of CakeML's various back end optimisations, including defining supported operations, specifying bit widths and padding, and a mapping from CakeML's registers to target registers. Each compiler configuration record also contains an assembly configuration, *cconf*.config.

The full details are not relevant for our purposes; suffice it to say that compiling to a new target requires definition of an appropriate compiler configuration (and therefore an assembly configuration). This ensures that the compiler generates valid code for the target. Defining a new encoder is the most involved step, but Asm's design permits relatively simple mappings into all of its target assemblies.

Generic machine semantics. A generic machine semantics ($\mathsf{semantics}_{\mathrm{M}}$) is specified in the functional big-step style (§ 1.2.2). It is parametrised by FFI oracle Δ and a machine state (*machine*), containing another kind of target-specific record known as a *machine configuration* (*machine*.mconf). Each machine configuration *mconf* contains the components below (where α is the machine-word width, σ the type of processor state, and π a type representing a projection of processor state). I omit FFI-, cache-, and floating point-specific configuration for brevity; they are not relevant for our purposes.

Component	Purpose
mconf.target.next : $\sigma \to \sigma$	next-step function
mconf.target.get_pc : $\sigma \to \alpha$ word	get program counter
mconf.target.get_reg : $\sigma \to$ num $\to \alpha$ word	get register value
mconf.target.get_byte : $\sigma \to \alpha$ word $\to$ word8	get byte from memory
mconf.next_interfer : num $\to \sigma \to \sigma$	interference oracle
mconf.target.state_ok : $\sigma \to$ bool	valid state predicate
mconf.target.proj : (α word $\to$ bool) $\to \sigma \to \pi$	state projection
mconf.callee_saved_regs : num list	callee-preserved registers
mconf.prog_addresses : α word $\to$ bool	set of instruction addresses
mconf.halt_pc : α word	program exit PC
mconf.target.config : α asm_config	assembly configuration

Inclusion of the next-step function (*mconf*.target.next) and the accessors of registers, the program counter, and memory (*mconf*.target.get_{reg,pc,byte}) permits generic specification of machine semantics for all targets at once: the semantics can step through successive machine states and refer to, *e.g.*, the program counter without any knowledge of how processor state is specified. I refer to the sub-record *mconf*.target as a *target configuration* (*tconf*).

The generic semantics also models interference from the execution environment by allowing the environment to change a subset of target state almost arbitrarily between instructions and on FFI calls. This is specified using *interference oracles*, such as *mconf*.next_interfer. Interference is subject to some conditions: it must preserve *both* a well-formedness predicate on target state (*mconf*.target.state_ok) *and* a projection of processor state (*mconf*.target.proj) exactly. The projection (of type π) is effectively the subset of target state which is left unmodified (or modified then restored) by the surrounding execution environment, *i.e.*, *mconf*.target.proj is always the same both before and after external interference. Note that projections are taken with respect to a *memory domain*, *i.e.*, a set of valid memory addresses (of type α word $\to$ bool). The semantics of LabLang requires all memory accesses to fall within this domain, which can therefore be considered the set of addresses owned by the LabLang program.

Other fields specify: calling convention information (*mconf*.callee_saved_regs); the set of instruction addresses (*mconf*.prog_addresses); the program counter for successful program exit (*mconf*.halt_pc); and the assembly configuration (*mconf*.target.config).

Compiler proofs. The use of a generic machine semantics allows much of compiler correctness to be proved once and for all, reducing duplication across multiple targets. All target-specific obligations are factored out into three preconditions:

Precondition	Purpose
backend_config_ok *cconf*	compiler configuration well-formedness
mc_init_ok *cconf mconf*	relation between compiler/machine configurations
mc_conf_ok *mconf*	machine configuration well-formedness

These mostly encapsulate natural well-formedness restrictions and correspondences between the various configuration records. We can see now that target_configs_ok (theorem 1.1, pg. 9) is defined in terms of these three:

$$\begin{aligned}\text{target\_configs\_ok } \mathit{cconf}\ \mathit{machine} \stackrel{\text{def}}{=} {}& \text{backend\_config\_ok } \mathit{cconf} \wedge \\ & \text{mc\_init\_ok } \mathit{cconf}\ \mathit{machine}.\text{mconf} \wedge \\ & \text{mc\_conf\_ok } \mathit{machine}.\text{mconf}\end{aligned}$$

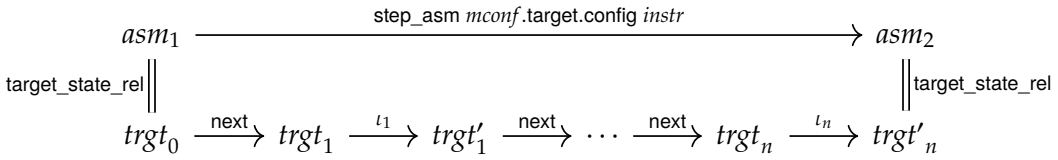
When proving target_configs_ok *cconf mconf* for a given compiler and target configuration, the key proof obligation is encoder_correct *mconf*.target, necessary to establish mc_conf_ok *mconf*.

Definition 9.1. encoder_correct.

$$\begin{aligned}&\text{encoder\_correct } \mathit{tconf} \stackrel{\text{def}}{=} \\ &\quad \text{target\_ok } \mathit{tconf} \wedge \\ &\quad \forall\, \mathit{asm}_1\ \mathit{instr}\ \mathit{asm}_2\ \mathit{trgt}. \\ &\qquad \text{step\_asm } \mathit{tconf}.\text{config } \mathit{instr}\ \mathit{asm}_1 = \text{Some } \mathit{asm}_2 \wedge \text{target\_state\_rel } \mathit{tconf}\ \mathit{asm}_1\ \mathit{trgt} \Longrightarrow \\ &\qquad \exists\, n.\ \forall\, \iota. \\ &\qquad\quad \text{interference\_ok } \iota\ (\mathit{tconf}.\text{proj } \mathit{asm}_1.\text{mem\_domain}) \Longrightarrow \\ &\qquad\quad \text{target\_state\_rel\_after\_n } n\ (\mathit{tconf}, \iota, \mathit{instr})\ (\mathit{asm}_1, \mathit{asm}_2, \mathit{trgt})\end{aligned}$$

For encoder_correct to hold of some target configuration *tconf*, the *tconf* must first satisfy target_ok. The remainder of the definition is quite verbose (particularly the definition of target_state_rel_after_n), and obscures the key intuition: given an Asm state asm_1, an Asm instruction *instr* which will successfully execute (step_asm) to produce state asm_2, any equivalent (target_state_rel) target state *trgt* with the encoded Asm instruction in memory must be able to execute (*tconf*.next) some number of steps n successfully, producing some final state equivalent to asm_2 (target_state_rel_after_n). The definition of target_state_rel_after_n allows for arbitrary intervening interference (ι) between execution steps, as long as the interference preserves *tconf*.proj asm_1.mem_domain (this is enforced by interference_ok). Existential quantification over n is necessary because single Asm instructions are often encoded as a sequence of target opcodes. Each tconf.next and interference step must also preserve *tconf*.state_ok and memory addresses which either

contain instructions or are not within asm_1.mem_domain. Overall, this corresponds to the diagram below, where I have abbreviated next $\stackrel{\text{def}}{=}$ *tconf*.next.



The target_ok predicate imposes further well-formedness restrictions on encoding and projection. In particular, *tconf*.proj should essentially identify a target state: if two states agree on *tconf*.proj for some memory domain, they must also agree on program counters and all memory addresses within that memory domain. Each should further be well-formed (*tconf*.state_ok) iff the other is, and should be equivalent to the same set of Asm states (according to target_state_rel).

At its heart, encoder_correct specifies a simulation relation. The natural way to prove this is by symbolic evaluation of instruction semantics on Asm and target state machines side-by-side. Therefore, proofs rely heavily on L3-enabled automation to repeatedly apply *tconf*.next and re-establish *tconf*.state_ok after each step and its associated interference.

9.2 Lifting simulation to compiler correctness

The CakeML compiler already targets Armv8, so we do not need to define a new compiler configuration or derive backend_config_ok. Therefore, to lift our simulation results (§ 8.4) to a top-level compiler correctness theorem, we must define a suitable machine configuration and establish mc_init_ok/mc_conf_ok.

Defining the machine configuration is mostly boilerplate: most definitions and proofs essentially mirror the existing ones for the L3 specification. One key difference is that the new machine configuration must incorporate asl_sys_regs_ok (definition 8.1, pg. 112) into *mconf*.target.state_ok, and extend the projection *mconf*.target.proj to ensure that these registers are preserved during interference too. Most other differences are much more minor: memory and registers must also be well-formed (§ 8.4). However, we must also define a next-step function (*mconf*.target.next). Though the ASL specification provides a function TopLevel, this covers unwanted extraneous details, such as processor interrupts and memory-mapped devices. Instead we define NextASL (definition 9.2), essentially TopLevel with the complexity stripped away. NextASL clears the branch-taken flag, reads the program counter, fetches the next opcode, and executes the opcode. The program counter is then updated only if no branch has been taken.

As discussed, the key proof obligation is encoder_correct (definition 9.1). We first prove theorem 9.3: l3_models_asl (definition 8.1, pg. 112) holds for any encodable Armv8

Definition 9.2. NextASL, a next-step function for the ASL-derived specification.

```
NextASL ≝
 do
  write_regS BranchTaken_ref F;
  pc ← PC_read ();
  instr ← Mem_read0 pc 4 AccType_IFETCH;
  ExecA64 instr;
  branch_taken ← read_regS BranchTaken_ref;
  if branch_taken then returnS () else do pc ← PC_read (); PC_set (pc + 4w) od
 od
```

Theorem 9.3. l3_models_asl for CakeML-generated instructions.

$$\vdash \mathsf{mem}\ \mathit{instr}\ (\mathsf{asm\_to\_arm8}\ \mathit{prog}) \wedge (\forall s.\ \mathsf{Encode}\ \mathit{instr} \neq \mathsf{BadCode}\ s) \Rightarrow \mathsf{l3\_models\_asl\_instr}\ \mathit{instr}$$

instruction produced by CakeML's compilation via Asm. Ideally, we would be able to reuse the encoder_correct result proved for the L3 specification directly, by deriving the following unproven ($\nvdash$) theorem statement:

$$\nvdash \mathsf{encoder\_correct}\ \mathit{l3\_mconf}.\mathsf{target} \Rightarrow \mathsf{encoder\_correct}\ \mathit{asl\_mconf}.\mathsf{target}$$

However, we are foiled by the interference from the surrounding execution environment. The definition of encoder_correct (definition 9.1, pg. 119) means we would require once and for all a transformation from interference with ASL-derived target states (satisfying interference_ok) to interference with L3 target states, such that the transformation preserves state_rel (§ 8.4.1, to allow use of the simulation proofs of § 8.4). In other words, we must derive the following result:

$$\begin{aligned}\nvdash\ & \mathsf{interference\_ok}\ \iota\ (\mathit{asl\_tconf}.\mathsf{proj}\ \mathit{mem\_domain}) \Rightarrow \\ & \exists \iota'.\ \mathsf{interference\_ok}\ \iota'\ (\mathit{l3\_tconf}.\mathsf{proj}\ \mathit{mem\_domain}) \wedge \\ & \quad \forall \mathit{l3}\ \mathit{asl}.\ \mathsf{state\_rel}\ \mathit{l3}\ \mathit{asl} \Rightarrow \forall n.\ \mathsf{state\_rel}\ (\iota'\ n\ \mathit{l3})\ (\iota\ n\ \mathit{asl})\end{aligned}$$

We cannot express this transformation: interference on ASL-derived specifications has a larger input space than that on L3 specifications, due to processor state not modelled in L3. The best we can do is transform interference *per-state* in lemma 9.4. Fortunately, this is sufficient to reuse our simulation proofs and avoid further symbolic evaluation with respect to the ASL-derived specification.

Proof of encoder_correct *for the ASL-derived specification.* From definition 9.1 (pg. 119), we have step_asm $\mathit{arm8\_aconf}\ \mathit{instr}\ \mathit{asm}_1$ = Some asm_2 and target_state_rel $\mathit{asl\_tconf}\ \mathit{asm}_1\ \mathit{asl}$,

Lemma 9.4. Transforming interference.

$$\begin{aligned}&\vdash \mathsf{interference\_ok}\ \iota\ (\mathit{asl\_tconf}.\mathsf{proj}\ \mathit{mem\_domain}) \wedge \mathsf{state\_rel}\ \mathit{l3}\ \mathit{asl} \Rightarrow \\ &\quad \exists\, \iota'.\ \mathsf{interference\_ok}\ \iota'\ (\mathit{l3\_tconf}.\mathsf{proj}\ \mathit{mem\_domain}) \wedge \\ &\qquad \forall\, n.\ \mathsf{state\_rel}\ (\iota'\ n\ \mathit{l3})\ (\iota\ n\ \mathit{asl})\end{aligned}$$

where *instr* is an Asm instruction produced by the CakeML compiler and *asl* is our initial ASL-derived state. We must alternately apply NextASL and some interference ι to produce a final state $\mathit{asl'}_n$ which satisfies target_state_rel $\mathit{asl\_tconf}\ \mathit{asm}_2\ \mathit{asl'}_n$. Choose n to be the length of the Armv8 encoding of *instr*.

Let *l3* be the L3 state which subsets *asl*, so then state_rel *l3 asl*. Now leverage existing automation for the L3 specification to symbolically execute the first instruction of the Armv8 encoding on *l3* to produce $\mathit{l3}_1$. Use theorem 9.3 (pg. 121) to derive the corresponding asl_1 for which state_rel $\mathit{l3}_1\ \mathit{asl}_1$.

Now asl_1 undergoes interference to produce asl'_1. It is not necessarily the case that state_rel $\mathit{l3}_1\ \mathit{asl}'_1$. However, we can use lemma 9.4 to interfere with $\mathit{l3}_1$ in a way which satisfies interference_ok, producing some $\mathit{l3}'_1$ for which we can re-establish state_rel $\mathit{l3}'_1\ \mathit{asl}'_1$.

We repeat this process a total of n times, reaching $\mathit{l3}'_n$, for which we must have target_state_rel $\mathit{l3\_tconf}\ \mathit{asm}_2\ \mathit{l3}'_n$. We can then straightforwardly derive the corresponding result for asl'_n. □

Overall, we produce theorem 9.5, our top-level compiler correctness proof. This is a specialised version of CakeML's generic compiler correctness theorem (theorem 1.1, pg. 9), where we have instantiated the compiler configuration to arm8_compiler_config and discharged the target_configs_ok precondition by showing it is implied by the asl_machine_config_ok predicate. The latter carves out a set of all possible valid ASL-derived machine configurations. It mimics an existing predicate defined for the L3 Armv8 semantics, but enforces the new *mconf*.target instead.

Theorem 9.5. CakeML compiler correctness for ASL-derived Armv8.

$$\begin{aligned}&\vdash \mathsf{asl\_machine\_config\_ok}\ \mathit{machine}.\mathsf{mconf} \wedge \\ &\quad \mathsf{semantics}_{\gg}\ \Delta\ \mathit{prog} \neq \mathsf{Terminate\ Error\ \_} \wedge \\ &\quad \mathsf{compile}_{\gg}\ \mathsf{arm8\_compiler\_config}\ \mathit{prog}\ =\ \mathsf{Some}\ \mathit{code} \wedge \\ &\quad \mathsf{code\_in\_memory\ arm8\_compiler\_config}\ \mathit{code}\ \mathit{machine} \\ &\quad \Rightarrow\ \mathsf{semantics}_{\mathrm{M}}\ \Delta\ \mathit{machine}\ \in\ \mathsf{extend\_with\_oom}\ (\mathsf{semantics}_{\gg}\ \Delta\ \mathit{prog})\end{aligned}$$

Chapter 10

Discussion

I have now showcased a compiler correctness proof which is backed by an official specification of an Arm ISA. The proof is made tractable by leveraging an existing L3 specification to abstract away complexity.

This approach rigorously validates the L3 specification, finding no new bugs. Most of the work is decoupled from CakeML, strengthening assurances in other projects which rely on the L3 specification of Armv8 and permitting reuse in future verification efforts. Usage of the Sail ecosystem demonstrates a novel application to proofs of semantics preservation, and further validates the Sail extraction process.

In this chapter I discuss the contributions presented in this part, in particular: high-level questions raised by them (§ 10.1), where they fit into the wider research landscape (§ 10.2), and how future work could build on them (§ 10.3).

10.1 High-level questions

This work raises some natural questions:

- What can be said about the trustworthiness of our results, particularly in comparison to those built on the L3 specification? (§ 10.1.1)
- Why did we find no new bugs in the L3 specification? (§ 10.1.2)
- Could we better adapt our specification for theorem proving, so avoiding the use of L3 as an intermediary? (§ 10.1.3)

10.1.1 Trustworthiness of our specifications

The ASL specification cannot cover all aspects of its ISA, instead providing a working reference. When considering more complex features, such as concurrency and interrupts, it remains an abstraction of the authoritative detail of the Arm Reference Manual.

However, even if a complete, machine-readable specification existed, proofs of semantics preservation with respect to it would be intractable without the abstractions. The ASL specifications and our work are a best-effort, modelling as much detail as currently feasible. Other proof goals (such as architecture security properties) may require different levels of detail.

Our root of trust is the extensive Arm-internal evaluation of the ASL specification, but extraction via Sail could introduce unintended semantic changes. Validating the generated HOL4 against Arm test suites or real hardware could improve trust (Sail's C back end has been tested in this way), but proving that the extraction preserves semantics is better still. This would be a significant undertaking, and require formal models of (at least) ASL and Sail. There has been some work into such models for both ASL (from personal communication) and Sail [Armstrong et al. 2018b]. Note that both Sail and Lem have been validated through heavy usage. For example, Sail-extracted ASL models have successfully been used to simulate a Linux boot, and the Lem ecosystem is well-exercised. However, Sail's HOL4 back end has received limited prior usage, so our work better validates this pathway too.

Our simulation proofs (§ 8.4) allow each specification to improve trust in the other. The L3 and ASL-derived specifications differ in their derivations considerably, yet are formally connected by our work. Therefore, any bugs found in one must be found in the other, and the low likelihood of this strengthens assurances in both. Previously, the L3 Armv8 specification had not been rigorously validated due to the scarcity of Armv8 hardware when it was written. By contrast, the L3 Armv7 specification was tested extensively against real hardware.

Even so, we strengthen trust in a single L3 specification, and a single extraction pathway via Sail. There are many such specifications that can be generated, via different extraction options, versioning differences, choices in manual modification, and so on.

10.1.2 Absence of bugs

We discovered no new bugs in the L3 specification semantics or encoder (we encountered one known issue, see § 10.2.3), despite the differing provenances of the L3 and ASL-derived specifications. This validates the approach of formalising a specification by using a DSL (L3) which can closely mirror it. However, the absence of bugs is surprising given that the ASL-derived specification covers implementation-defined behaviour and architecturally unknown values.

A partial explanation is our restricted domain of proofs of semantics preservation: we verify general-purpose instructions targeted by compilers, which avoid ambiguity to ensure portability. As a verified compiler, CakeML targets an even smaller subset of instructions to reduce proof overhead.

Removal of address translation and interrupts (§§ 8.1 and 8.4 respectively) further reduces ambiguity significantly. We also do not tackle exception-handling, assuming that instructions complete execution without failure in the L3 specification (definition 8.1, pg. 111). This precondition holds for all instructions generated by the CakeML compiler.

Furthermore, Arm intentionally reduced underspecification in Armv8 (compared to Armv7). For example: in Armv7 the program counter is a general-purpose register (R15) which can be modified unexpectedly by programmers; in Armv8 there is no direct access to the program counter. Architecturally unknown values are also mostly used as placeholders for variables which are declared and only later initialised.

10.1.3 The need for L3

We build our results on existing tools: an ASL specification and the Sail ecosystem. Neither is designed for interactive theorem proving, so we use a purpose-built L3 specification as a stepping stone in proof. Instead, could we obviate the need for this indirection by better adapting the tools to our domain?

One approach is to change the official ASL specification, though this would require support from Arm. Stylistic refactoring could reduce overly imperative code (*e.g.*, figure 8.1, pg. 111). Logical refactoring has recently streamlined address translation, and could be applied to other parts of the specification. Forbidding early-return statements could reduce embedding of otherwise pure functions into the monad, but this is a significant language change. Alternatively ASL-to-ASL transformations before translation to Sail could achieve similar effects, without compromising our root of trust: the resultant ASL can be subjected to the Arm-internal test suite. Semantics-preserving transformations are already used in Arm to produce model-checking-friendly Verilog (from personal communication).

Another approach is to strengthen Sail's extraction to HOL4, taking inspiration from L3. Streamlining of Sail libraries for HOL4 is a first step. Extraction would also need to produce an AST for instructions. The challenge here is reproducing L3's ease of use: its AST is handcrafted for theorem proving (*i.e.*, split into instruction classes for convenience and to avoid scaling issues with HOL4 types). However, Sail's extraction must be automatic and target-agnostic. Design choices made here will suit different domains, for example the AST could mirror assembly syntax or reference manual structure. A direct translation from ASL or Sail to HOL4 (cutting out Lem) could also streamline the extraction process and more closely target HOL4, but this is a significant undertaking.

Direct proof without L3 would also require new proof automation. No next-step libraries exist for the ASL-derived specification, but several have been painstakingly handcrafted for various other specifications. Automatic generation of certified next-step libraries using symbolic evaluation and symbolic execution [Campbell and Stark 2016]

is a promising approach. A key challenge will be navigating the large, monolithic decoding/execution functions.

Though we have identified some promising avenues to verification without L3, none provides a silver bullet and taken together they represent a significant body of work. As industrial specifications of the scale of Armv8 become more prevalent, such engineering issues will be critical.

10.2 Related work

In this section, I review prior work which is most relevant to this part. More general work is described in § 1.3.

10.2.1 Applications of the Sail toolchain

Isla [Armstrong et al. 2021] is an SMT solver-based symbolic execution engine for Sail. Given an opcode and constraints on processor state, it can produce instruction execution traces which are much simpler than the original Sail. This echoes the next-step libraries used with many L3 specifications (§ 7.2.3), which similarly prune irrelevant parts of the specification using preconditions; next-step libraries are created once and for all in-logic where Isla traces are generated on-the-fly using an external SMT solver. Islaris [Sammler et al. 2022] formalises Isla-generated traces in Coq and builds a separation logic for reasoning about their semantics, using automated proof search to simplify verification efforts considerably. Our shared goal is to derive simplified machine code semantics from the complex ASL-derived specifications, using the desired domain to constrain semantics appropriately. Islaris' domain is machine code verification so it relies on user constraints and an external SMT solver on a per-opcode basis, effectively handling multiple ISAs and system registers/instructions. However, use of external solvers reduces trust, and it can be difficult to find sufficient constraints to narrow down semantics. Our domain of semantics preservation instead allows us to abstract once and for all within the prover, sacrificing some portability and system-level behaviour for greatly increased assurances.

Sail is integral in key security proofs with respect to Morello, an Arm implementation of the Capability Hardware Enhanced RISC Instructions (CHERI) ISA [Bauereiss et al. 2022; Nienhuis et al. 2020]. CHERI extends conventional ISAs with new features enabling memory protection, defending against many memory-based security exploits [Watson et al. 2019, 2020]. The Isabelle/HOL proof (with some SMT solver oracles for bit vector operations) is complex, requiring hours to build with considerable computing power. For such security properties, monadic representation of specifications may be useful: these proofs consider not just input-output behaviour, but also intervening steps and

their memory accesses. In our case, these intervening steps obfuscate the high-level semantics of the instruction.

10.2.2 Verified compilation to verified hardware

Verifying individual processor implementations can produce impressively powerful end-to-end guarantees. Verified compilation to such a processor has a minimal trusted computing base, compared to other targets which must trust ISA specifications and processor implementations. However, their purpose-built specifications cannot approach modern, mainstream processors in complexity or efficiency; our focus on ISAs (which specify an envelope of behaviours to which processor implementations must adhere) therefore sacrifices some trust for greater applicability to common processors. Just as ISA verification relies on architecture specification languages, so too does hardware specification rely on *hardware description languages* (HDLs).

CakeML targets the Silver ISA and its verified implementation [Lööw et al. 2019]. Proof-producing synthesis of Verilog (an HDL) circuits from HOL4 *circuit functions* [Lööw and Myreen 2019] mirrors synthesis of CakeML from computable functions (§ 1.2.2), and is formalised with respect to a deep embedding of Verilog syntax and semantics. Guarantees can be further transported to netlists by another step of synthesis from Verilog [Lööw 2021]. Running CakeML applications directly on the processor without an intervening OS (*bare metal*) permits discharging of CakeML's assumptions concerning the behaviour of the external execution environment. Notably, the action of system calls and their side-effects on processor state (*e.g.*, preservation of CakeML's calling convention) no longer need to be asserted.

Choi et al. [2017] produce the Kami Coq library for specifying and verifying hardware designs written in the style of the HDL Bluespec. Transliteration to Bluespec permits automated unverified compilation to FPGAs and chip designs, though pairing with the Kôika [Bourgeat et al. 2020] verified compiler for a subset of Bluespec could port guarantees closer to these devices. A family of pipelined processors with coherent caches is implemented and verified using Kami; one concretised instance of this family implements a subset of RISC-V. Erbsen et al. [2021] build on this, verifying an application on a realistic embedded stack by implementing a C-based language and verified compiler targeting Kami's RISC-V implementation. These works are part of the Bedrock project, which prioritises clean, highly automated verification using modular interfaces and realistic I/O over performance and usability.

10.2.3 Validation of ISA specifications

The Provably Secure Execution Platforms for Embedded Systems (PROSPER) project has created Scam-V [Nemati et al. 2020], an automatic validator for ISA specifications. It searches for pairs of executions which behave identically according to the specification, but are distinguishable on hardware via some observation. If such a pair exists, the specification has failed to model an observable side-channel correctly.

Scam-V uncovered a bug in the L3 specification of Armv8: the compare and branch on non-zero instruction (CBNZ) incorrectly behaved like the compare and branch on zero instruction (CBZ). We replicated this finding in a failed attempt to prove l3_models_asl (definition 8.1, pg. 112) for an unpatched CBNZ. Furthermore, we have validated the corrected specification by successfully establishing l3_models_asl. Note that our CakeML proofs were unaffected: the CakeML compiler does not produce CBNZ instructions.

10.2.4 Verification with respect to the Arm architecture

Official Arm ISA specifications have been used within Arm to establish architectural properties below the ISA abstraction boundary. These efforts use automated techniques (SMT solving and bounded model-checking) to suit their problem domain and their setting in industry.

ISA-Formal [Reid 2016b; Reid et al. 2016] uses bounded model-checking to verify that Arm ISA implementations adhere to their intended specification. The project translates ASL specifications to reference Verilog implementations [Reid 2016c], comparing these to the actual implementation using an off-the-shelf bounded model-checker. Failures are output as counterexamples.

Secure-M [Reid 2017f] produces key security properties for the Arm M-class specification, using an automated SMT solver to verify they hold. It provides a more rigorous alternative to code review and testing for architecture modifications, focusing on whole-specification properties for continuous integration testing. There is also some support for developer-stated assertion-checking. A future goal is to prove that successive versions of specifications preserve necessary backwards-compatibility.

10.2.5 Other applications of L3

The seL4 [Klein et al. 2009, 2014] verified operating system microkernel uses a translation validation [Sewell et al. 2013] phase to extend its guarantees to the binary on Armv7. In particular, seL4's C implementation is verified with respect to a C semantics specified in Isabelle/HOL. To ensure this verification still holds for a compiled Armv7 binary, the binary is *decompiled* to a HOL4 function using the L3 specification of Armv7. The HOL4 function is ported to Isabelle/HOL, and proved equivalent to the C source program via

translation to a graph representation followed by SMT solving. Critically, decompilation is automatic and proof-producing [Myreen et al. 2008]: each run produces a theorem relating the semantics of the original binary to the extracted HOL4 function.

10.3 Future work

Removing the precondition to theorem 9.3 (pg. 121) is a clear next step. This could be complicated by further incompatibilities between the L3 and ASL-derived specifications, particularly their versioning differences (Armv8.0 vs. Armv8.6 respectively). One mitigation is to upgrade the L3 specification to support Armv8.6, and to model more system registers. The latter could further reduce the number of bits fixed by `asl_sys_regs_ok` (§ 8.4).

Further afield, we could augment the L3 specification to model a greater subset of the ISA. This is no small task, requiring close familiarity with both L3 and the Arm Architecture Reference Manual [Arm Limited 2020]. AArch32 and floating point instructions in particular are not currently modelled in L3 and so remain unsupported by CakeML on Armv8. Connecting ASL's custom floating point definitions with HOL4 libraries would be a significant challenge here. Fortunately, the simulation proofs can help by uncovering any discrepancies between the specifications.

An L3 specification provides the semantics for Armv7 used for seL4's translation validation (§ 10.2.5). Applying the simulation proofs of § 8.4 to seL4 would significantly strengthen this result. However, the versioning mismatch here is considerable: seL4 implementation and proofs would need need to be updated to target Armv8 first.

A more faithful account of address translation would improve trust in our work and widen its applicability. This could take inspiration from the Sail developers, who prove in Isabelle/HOL that address translation adheres to a hand-written specification under certain conditions [Armstrong et al. 2019, §8]. Recent simplifications to Arm's ASL specification of address translation may help here.

Conclusion

I have now presented two advancements in state of the art general-purpose, end-to-end compiler verification: an end-to-end verified compiler for a purely functional, Haskell-like language, and a formal connection between a realistic Armv8 instruction set semantics and a compiler correctness result. Each advancement extends the guarantees of the CakeML ecosystem outwards: PureCake lifts them to a Haskell-like language, reusing CakeML as an unmodified component; and a realistic machine semantics reduces the trust that they demand.

Composing theorem 5.6 (pg. 78) and theorem 9.5 (pg. 122) via theorem 5.3 (pg. 76) fulfils the title of this dissertation: an end-to-end correctness theorem for the compilation of a purely functional language to a realistic machine semantics.

$$
\begin{aligned}
&\vdash \mathsf{compiler}\ \mathit{str}\ =\ \mathsf{Some}\ \mathit{ast}_{\gg} \wedge \\
&\quad \mathsf{compile}_{\gg}\ \mathsf{arm8\_compiler\_config}\ \mathit{ast}_{\gg}\ =\ \mathsf{Some}\ \mathit{code} \wedge \\
&\quad \mathsf{asl\_machine\_config\_ok}\ \mathit{machine}.\mathsf{mconf} \wedge \\
&\quad \mathsf{code\_in\_memory}\ \mathsf{arm8\_compiler\_config}\ \mathit{code}\ \mathit{machine} \\
&\quad \Longrightarrow \exists \mathit{ce}\ \mathit{ns}.\ \mathsf{frontend}\ \mathit{str}\ =\ \mathsf{Some}\ (\mathit{ce}, \mathit{ns}) \wedge \\
&\qquad\quad \mathsf{ffi\_convention} \vdash [\![\ \mathit{machine}\ ]\!]_{\mathrm{M}}\ \mathsf{prunes}\ [\![\ \mathsf{exp\_of}\ \mathit{ce}\ ]\!]
\end{aligned}
$$

In this section, I will answer two questions. First, how much trust does this theorem require? Second, how much were we constrained by CakeML?

Trust in the theorem

What remains in our TCB?

We must still trust the logic and implementation of HOL4, though this can be mitigated somewhat using HOL4's ability to generate independently verifiable proof articles. Otherwise, most assumptions are inherited from CakeML, concerning isolation provided by the surrounding execution environment. That is, code generated by PureCake assumes that the operating system invisibly handles context switches, as if the code were running directly on hardware. In particular, the program is initialised correctly (code_in_memory) and the operating system respects a calling convention (*i.e.*,

does not clobber registers unexpectedly). Like CakeML, PureCake also trusts the C glue code implementing its FFI, and off-the-shelf linking which integrates the glue code into its verified binary.

Even with the use of a realistic machine semantics, PureCake still assumes that hardware implementations correctly adhere to ISA specifications, and that address translation is seamless so virtual memory correctly abstracts physical memory. But our Armv8 ISA specification offers a much-reduced TCB due to its derivation from a near-complete official specification. The translation of Arm ASL into HOL4 and our ensuing modifications still require some trust, but this is now measurable: a fixed number of lines of code and manual modifications, as opposed to unknown trust in human-written specifications which can at best be validated via testing. The only increase in TCB is the addition of a few extra system registers which we assume will be preserved by the operating system.

Usage of CakeML

We have built our work on CakeML—but does this limit its wider applicability?

Most of the techniques I have presented are orthogonal to CakeML, which serves only as a research vehicle. The challenges of verifiably compiling a purely functional language or taming an industrial instruction set specification are equally applicable when working with any other verified compiler. However, PureCake's implementation and verification are streamlined because CakeML already compiles a functional language, and using realistic semantics with CakeML is simpler because it already targets mainstream hardware. Applying these techniques to a compiler for a lower-level language or one which targets other hardware will require extra effort.

What about those techniques that are CakeML-specific—how do these constrain our work? PureCake cannot implement features such as separate compilation without support from CakeML. But it does make some considerably different design choices without adverse effects: for example its semantics is based on interaction trees, which need only be connected formally to CakeML's trace-producing one. Most of our work on realistic machine semantics remains independently reusable too.

However, both efforts have had to contend with CakeML's model of I/O and the surrounding execution environment. In particular, PureCake's FFI model is mostly inherited from CakeML, and it is non-trivial to use our realistic machine semantics alongside CakeML's modelling of environmental inference. Arguably, these are natural consequences of verifying a component of a trusted system (the compiler) rather than the whole system, and it is prohibitively difficult to verify entire systems which consist of realistic components. We have to make assumptions about the parts we do not verify: the TCB of any verified system will necessarily form part of its externally-presented

proof interface. But these assumptions are not necessarily set in stone. CakeML models its FFI essentially as uninterpreted function calls, permitting encoding of other FFI models, including PureCake's slightly higher-level one. Generalising the more concrete, low-level aspects of CakeML's model could simplify similar efforts in the future. CakeML introduces environmental interference only in its lowest-level target semantics too, suggesting that these assumptions can be modified without reworking the entire verified stack. This could permit, for example, composition with a verified operating system which guarantees exactly those properties that CakeML assumes about its surrounding execution environment.

Bibliography

O. Abrahamsson, S. Ho, H. Kanabar, R. Kumar, M. O. Myreen, M. Norrish, and Y. K. Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *J. Autom. Reason.*, 64(7), 2020. DOI 10.1007/s10817-020-09559-8.

O. Abrahamsson, M. O. Myreen, R. Kumar, and T. Sewell. Candle: A verified implementation of HOL Light. In *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI 10.4230/LIPIcs.ITP.2022.3.

S. Abramsky. The lazy λ-calculus. In *Research Topics in Functional Programming*. Addison Wesley, 1990.

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

A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, USA, 2004. URL https://www.ccs.neu.edu/home/amal/ahmedthesis.pdf.

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Principles of Programming Languages (POPL)*. ACM, 2009. DOI 10.1145/1480881.1480925.

R. M. Amadio, N. Ayache, F. Bobot, J. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. S. Coen, I. Stark, and P. Tranquilli. Certified complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis (FOPARA)*, volume 8552 of *Lecture Notes in Computer Science*. Springer, 2013. DOI 10.1007/978-3-319-12466-7_1.

A. Anand, A. W. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Bélanger, M. Sozeau, and M. Z. Weaver. CertiCoq: A verified compiler for Coq. In *Workshop on Coq for Programming Languages (CoqPL)*, 2017. URL https://popl17.sigplan.org/details/main/9/CertiCoq-A-verified-compiler-for-Coq.

A. W. Appel. Verified software toolchain. In *European Symposium on Programming (ESOP)*, volume 6602 of *Lecture Notes in Computer Science*. Springer, 2011. DOI 10.1007/978-3-642-19718-5_1.

Arm Limited. *Arm Architecture Reference Manual*. Arm Limited, F.c edition, 2020. URL https://developer.arm.com/documentation/ddi0487/fc.

A. Armstrong, T. Bauereiss, B. Campbell, S. Flur, K. E. Gray, P. Mundkur, R. M. Norton, C. Pulte, A. Reid, P. Sewell, I. Stark, and M. Wassell. Detailed models of instruction set architectures: From pseudocode to formal semantics. In *Proceedings of the Automated Reasoning Workshop*, 2018a. URL http://www.cl.cam.ac.uk/~pes20/sail/2018-04-12-arw-paper.pdf. Two-page abstract.

A. Armstrong, N. Krishnaswami, P. Sewell, and M. Wassell. Formalisation of MiniSail in the Isabelle theorem prover. In *Proceedings of the Automated Reasoning Workshop*, 2018b. URL https://www.cl.cam.ac.uk/~pes20/sail/arw18_mpew2.pdf. Two-page abstract.

A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3 (POPL), 2019. DOI 10.1145/3290384.

A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Computer Aided Verification (CAV)*, volume 12759 of *Lecture Notes in Computer Science*. Springer, 2021. DOI 10.1007/978-3-030-81685-8_14.

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*. Springer, 2005. DOI 10.1007/11541868_4.

B. Barras. Programming and computing in HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 of *Lecture Notes in Computer Science*. Springer, 2000. DOI 10.1007/3-540-44659-1_2.

A. Barrière, S. Blazy, O. Flückiger, D. Pichardie, and J. Vitek. Formally verified speculation and deoptimization in a JIT compiler. *Proc. ACM Program. Lang.*, 5(POPL), 2021. DOI 10.1145/3434327.

A. Barrière, S. Blazy, and D. Pichardie. Formally verified native code generation in an effectful JIT: turning the CompCert backend into a formally verified JIT compiler. *Proc. ACM Program. Lang.*, 7(POPL), 2023. DOI 10.1145/3571202.

G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4 (POPL), 2020. DOI 10.1145/3371075.

T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture. In *European Symposium on Programming (ESOP)*, volume 13240 of *Lecture Notes in Computer Science*. Springer, 2022. DOI 10.1007/978-3-030-99336-8_7.

C. Baumann, M. Näslund, C. Gehrmann, O. Schwarz, and H. Thorsen. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications (EuCNC)*. IEEE, 2016. DOI 10.1109/EuCNC.2016.7561034.

F. Besson, S. Blazy, and P. Wilke. A concrete memory model for CompCert. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*. Springer, 2015. DOI 10.1007/978-3-319-22102-1_5.

F. Besson, S. Blazy, and P. Wilke. CompCertS: A memory-aware verified C compiler using pointer as integer semantics. In *Interactive Theorem Proving (ITP)*, volume 10499 of *Lecture Notes in Computer Science*. Springer, 2017. DOI 10.1007/978-3-319-66107-0_6.

T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind. The essence of Bluespec: a core language for rule-based hardware design. In *Programming Language Design and Implementation (PLDI)*. ACM, 2020. DOI 10.1145/3385412.3385965.

J. Breitner. Formally proving a compiler transformation safe. In *Symposium on Haskell*. ACM, 2015. DOI 10.1145/2804302.2804312.

J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich. Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). *Proc. ACM Program. Lang.*, 2(ICFP), 2018. DOI 10.1145/3236784.

B. Campbell and I. Stark. Extracting behaviour from an executable instruction set model. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016. DOI 10.1109/FMCAD.2016.7886658.

V. Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005. DOI 10.2168/LMCS-1(2:1)2005.

Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation (PLDI)*. ACM, 2014. DOI 10.1145/2594291.2594301.

A. Chlipala. Algorithmic checking of security arguments for microprocessors. In *GOMACTech Conference*, 2019. URL https://apps.dtic.mil/sti/citations/AD1075652.

J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), 2017. DOI 10.1145/3110268.

A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2), 1940. DOI 10.2307/2266170.

A. Coglio and S. Goel. Adding 32-bit mode to the ACL2 model of the x86 ISA. In *Workshop on the ACL2 Theorem Prover and Its Applications*, volume 280 of *EPTCS*, 2018. DOI 10.4204/EPTCS.280.6.

L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008. DOI 10.1007/978-3-540-78800-3_24.

A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. Integration verification across software and hardware for a simple embedded system. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021. DOI 10.1145/3453483.3454065.

M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Principles of Programming Languages (POPL)*. ACM Press, 1987. DOI 10.1145/41625.41654.

A. Filinski. Representing monads. In *Principles of Programming Languages (POPL)*. ACM Press, 1994. DOI 10.1145/174675.178047.

A. Filinski. Monads in action. In *Principles of Programming Languages (POPL)*. ACM, 2010. DOI 10.1145/1706299.1706354.

S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Principles of Programming Languages (POPL)*. ACM, 2016. DOI 10.1145/2837614.2837615.

S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Principles of Programming Languages (POPL)*. ACM, 2017. DOI 10.1145/3009837.3009839.

S. Foster, C. Hur, and J. Woodcock. Formally verified simulations of state-rich processes using interaction trees in Isabelle/HOL. In *Conference on Concurrency Theory (CONCUR)*, volume 203 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI 10.4230/LIPIcs.CONCUR.2021.20.

A. C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*. Springer, 2003. DOI 10.1007/10930755_2.

A. C. J. Fox. Directions in ISA specification. In *Interactive Theorem Proving (ITP)*, volume 7406 of *Lecture Notes in Computer Science*. Springer, 2012. DOI 10.1007/978-3-642-32347-8_23.

A. C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*. Springer, 2015. DOI 10.1007/978-3-319-22102-1_12.

A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010. DOI 10.1007/978-3-642-14052-5_18.

A. C. J. Fox, M. O. Myreen, Y. K. Tan, and R. Kumar. Verified compilation of CakeML to multiple machine-code targets. In *Certified Programs and Proofs (CPP)*. ACM, 2017. DOI 10.1145/3018610.3018621.

S. Goel and W. A. Hunt, Jr. Automated code proofs on a formal model of the x86. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*. Springer, 2013. DOI 10.1007/978-3-642-54108-7_12.

S. Goel, W. A. Hunt, Jr, and M. Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering. Springer, 2017. DOI 10.1007/978-3-319-48628-4_8.

S. Goel, A. Slobodová, R. Sumners, and S. Swords. Verifying x86 instruction implementations. In *Certified Programs and Proofs (CPP)*. ACM, 2020. DOI 10.1145/3372885.3373811.

A. Gómez-Londoño, J. Å. Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. DOI 10.1145/3428272.

M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. ISBN 3-540-09724-4. DOI 10.1007/3-540-09724-4.

S. Graf and S. Peyton Jones. Selective lambda lifting. *CoRR*, abs/1910.11717, 2019.

K. E. Gray, G. Kerneis, D. P. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Microarchitecture (MICRO)*. ACM, 2015. DOI 10.1145/2830772.2830775.

R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Principles of Programming Languages (POPL)*. ACM, 2015. DOI 10.1145/2676726.2676975.

R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016. DOI 10.5555/3026877.3026928.

R. Guanciale, H. Nemati, M. Dam, and C. Baumann. Provably secure memory isolation for Linux on ARM. *J. Comput. Secur.*, 24(6), 2016. DOI 10.3233/JCS-160558.

A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish. Verified characteristic formulae for CakeML. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*. Springer, 2017. DOI 10.1007/978-3-662-54434-1_22.

J. Hackett and G. Hutton. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.*, 3(ICFP), 2019. DOI 10.1145/3341718.

C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2), 1996. DOI 10.1145/227699.227700.

P. G. Hancock and A. Setzer. Interactive programs in dependent type theory. In *Computer Science Logic (CSL)*, volume 1862. Springer, 2000. DOI 10.1007/3-540-44622-2_21.

B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, Netherlands, 2005. URL http://dspace.library.uu.nl/handle/1874/7297.

B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *Workshop on Haskell*. ACM, 2003. DOI 10.1145/871895.871902.

B. Heeren, J. Hage, and D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Department of Information and Computing Sciences, Utrecht University, 2022. URL https://www.open.ou.nl/bhr/GeneralizingHM.html.

J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146, 1969. DOI 10.2307/1995158.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124(2), 1996. DOI 10.1006/inco.1996.0008.

L. Hupel and T. Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*. Springer, 2018. DOI 10.1007/978-3-319-89884-1_35.

C. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Principles of Programming Languages (POPL)*. ACM, 2011. DOI 10.1145/1926385.1926402.

H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng. Towards certified separate compilation for concurrent programs. In *Programming Language Design and Implementation (PLDI)*. ACM, 2019. DOI 10.1145/3314221.3314595.

T. Johnsson. Efficient compilation of lazy evaluation. In *Symposium on Compiler Construction*. ACM, 1984. DOI 10.1145/502874.502880.

J. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *European Symposium on Programming (ESOP)*, volume 7211 of *Lecture Notes in Computer Science*. Springer, 2012. DOI 10.1007/978-3-642-28869-2_20.

H. Kanabar, A. C. J. Fox, and M. O. Myreen. Taming an authoritative Armv8 ISA specification: L3 validation and CakeML compiler verification. In *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI 10.4230/LIPIcs.ITP.2022.20.

H. Kanabar, S. Vivien, O. Abrahamsson, M. O. Myreen, M. Norrish, J. Åman Pohjola, and R. Zanetti. PureCake: A verified compiler for a lazy functional language. In *Programming Language Design and Implementation (PLDI)*. ACM, 2023. DOI 10.1145/3591259.

J. Kang, Y. Kim, C. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Principles of Programming Languages (POPL)*. ACM, 2016. DOI 10.1145/2837614.2837642.

O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Symposium on Haskell*. ACM, 2015. DOI 10.1145/2804302.2804319.

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Operating Systems Principles (SOSP)*. ACM, 2009. DOI 10.1145/1629575.1629596.

G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1), 2014. DOI 10.1145/2560537.

J. Koenig and Z. Shao. CompCertO: compiling certified open C components. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021. DOI 10.1145/3453483.3454097.

N. Koh, Y. Li, Y. Li, L. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In *Certified Programs and Proofs (CPP)*. ACM, 2019. DOI 10.1145/3293880.3294106.

A. Koprowski and H. Binsztok. TRX: A formally verified parser interpreter. *Log. Methods Comput. Sci.*, 7(2), 2011. DOI 10.2168/LMCS-7(2:18)2011.

J. O. G. Krijnen, M. M. T. Chakravarty, G. Keller, and W. Swierstra. Translation certification for smart contracts. In *Functional and Logic Programming (FLOPS)*, volume 13215 of *Lecture Notes in Computer Science*. Springer, 2022. DOI 10.1007/978-3-030-99461-7_6.

R. Kumar and M. Norrish. (Nominal) unification by recursive descent with triangular substitutions. In *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010. DOI 10.1007/978-3-642-14052-5_6.

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL)*. ACM, 2014. DOI 10.1145/2535838.2535841.

P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4), 1964. DOI 10.1093/comjnl/6.4.308.

J. Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages (POPL)*. ACM Press, 1993. DOI 10.1145/158511.158618.

O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4), 1998. DOI 10.1145/291891.291892.

D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*. Springer, 2009. DOI 10.1007/978-3-642-05089-3_51.

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7), 2009. DOI 10.1145/1538788.1538814.

X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1), 2008. DOI 10.1007/s10817-008-9099-0.

T. Letan and Y. Régis-Gianas. FreeSpec: specifying, verifying, and executing impure computations in Coq. In *Certified Programs and Proofs (CPP)*. ACM, 2020. DOI 10.1145/3372885.3373812.

S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A secure and formally verified Linux KVM hypervisor. In *Security and Privacy (SP)*. IEEE, 2021a. DOI 10.1109/SP40001.2021.00049.

Y. Li, L. Xia, and S. Weirich. Reasoning about the garden of forking paths. *Proc. ACM Program. Lang.*, 5(ICFP), 2021b. DOI 10.1145/3473585.

A. Lindner, R. Guanciale, and R. Metere. TrABin: Trustworthy analyses of binaries. *Sci. Comput. Program.*, 174, 2019. DOI 10.1016/j.scico.2019.01.001.

A. Lööw. Lutsig: a verified Verilog compiler for verified circuit development. In *Certified Programs and Proofs (CPP)*. ACM, 2021. DOI 10.1145/3437992.3439916.

A. Lööw and M. O. Myreen. A proof-producing translator for Verilog development in HOL. In *Formal Methods in Software Engineering, (FormaliSE@ICSE)*. IEEE / ACM, 2019. DOI 10.1109/FormaliSE.2019.00020.

A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. C. J. Fox. Verified compilation on a verified processor. In *Programming Language Design and Implementation (PLDI)*. ACM, 2019. DOI 10.1145/3314221.3314622.

W. Mansky, W. Honoré, and A. W. Appel. Connecting higher-order separation logic to a first-order outside world. In *European Symposium on Programming (ESOP)*, volume 12075 of *Lecture Notes in Computer Science*. Springer, 2020. DOI 10.1007/978-3-030-44914-8_16.

C. McBride. Turing-completeness totally free. In *Mathematics of Program Construction (MPC)*, volume 9129 of *Lecture Notes in Computer Science*. Springer, 2015. DOI 10.1007/978-3-319-19797-5_13.

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

R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3), 1978. DOI 10.1016/0022-0000(78)90014-4.

A. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Principles of Programming Languages (POPL)*. ACM, 1999. DOI 10.1145/292540.292547.

E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman. Œuf: minimizing the Coq extraction TCB. In *Certified Programs and Proofs (CPP)*. ACM, 2018. DOI 10.1145/3167089.

O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF=HOL+LCF. *J. Funct. Program.*, 9(2), 1999. DOI 10.1017/s095679689900341x.

D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *International Conference on Functional Programming (ICFP)*. ACM, 2014. DOI 10.1145/2628136.2628143.

M. O. Myreen. Verified just-in-time compiler on x86. In *Principles of Programming Languages (POPL)*. ACM, 2010. DOI 10.1145/1706299.1706313.

M. O. Myreen. A minimalistic verified bootstrapped compiler (proof pearl). In *Certified Programs and Proofs (CPP)*. ACM, 2021a. DOI 10.1145/3437992.3439915.

M. O. Myreen. The CakeML project's quest for ever stronger correctness theorems (invited paper). In *Interactive Theorem Proving (ITP)*, volume 193 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021b. DOI 10.4230/LIPIcs.ITP.2021.1.

M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011. DOI 10.1007/978-3-642-22863-6_20.

M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*. Springer, 2007. DOI 10.1007/978-3-540-71209-1_44.

M. O. Myreen and M. J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009. DOI 10.1007/978-3-642-03359-9_25.

M. O. Myreen and S. Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3), 2014. DOI 10.1017/S0956796813000282.

M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2008. DOI 10.1109/FMCAD.2008.ECP.24.

M. O. Myreen, K. Slind, and M. J. C. Gordon. Extensible proof-producing compilation. In *Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*. Springer, 2009. DOI 10.1007/978-3-642-00722-4_2.

G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *International Conference on Functional Programming (ICFP)*. ACM, 2015. DOI 10.1145/2784731.2784764.

H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs. Validation of abstract side-channel models for computer architectures. In *Computer Aided Verification (CAV)*, volume 12224 of *Lecture Notes in Computer Science*. Springer, 2020. DOI 10.1007/978-3-030-53288-8_12.

K. Nienhuis, A. Joannou, T. Bauereiss, A. C. J. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Security and Privacy (SP)*. IEEE, 2020. DOI 10.1109/SP40000.2020.00055.

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory Pract. Object Syst.*, 5(1), 1999.

S. Owens, P. Böhm, F. Z. Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011. DOI 10.1007/978-3-642-22863-6_27.

S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In *European Symposium on Programming (ESOP)*, volume 9632 of *Lecture Notes in Computer Science*. Springer, 2016. DOI 10.1007/978-3-662-49498-1_23.

S. Owens, M. Norrish, R. Kumar, M. O. Myreen, and Y. K. Tan. Verifying efficient function calls in CakeML. *Proc. ACM Program. Lang.*, 1(ICFP), 2017. DOI 10.1145/3110262.

M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6), 2019. DOI 10.1145/3280984.

D. Patterson and A. Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP), 2019. DOI 10.1145/3341689.

J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming (ESOP)*, volume 8410 of *Lecture Notes in Computer Science*. Springer, 2014. DOI 10.1007/978-3-642-54833-8_8.

S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer, 1993. DOI 10.1007/978-1-4471-3236-3_17.

S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. Funct. Program.*, 2(2), 1992. DOI 10.1017/S0956796800000319.

S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer, 1991. DOI 10.1007/3540543961_30.

S. L. Peyton Jones and J. Salkild. The spineless tagless G-machine. In *Functional Programming Languages and Computer Architecture (FPCA)*. ACM, 1989. DOI 10.1145/99370.99385.

B. C. Pierce and D. N. Turner. Local type inference. In *Principles of Programming Languages (POPL)*. ACM, 1998. DOI 10.1145/268946.268967.

M. Piróg and J. Gibbons. The coinductive resumption monad. In *Mathematical Foundations of Programming Semantics (MFPS)*, volume 308 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2014. DOI 10.1016/j.entcs.2014.10.015.

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

A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*. Springer, 1998. DOI 10.1007/BFb0054170.

J. Å. Pohjola, A. Gómez-Londoño, J. Shaker, and M. Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI 10.4230/LIPIcs.ITP.2022.27.

C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL), 2018. DOI 10.1145/3158107.

A. Reid. Arm's architecture specification language. Blog post, 2016a. URL https://alastairreid.github.io/specification_languages/.

A. Reid. Limitations of ISA-Formal. Blog post, 2016b. URL https://alastairreid.github.io/isa-formal-limitations/.

A. Reid. Verifying against the official Arm specification. Blog post, 2016c. URL https://alastairreid.github.io/using-armarm/.

A. Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016d. DOI 10.1109/FMCAD.2016.7886675.

A. Reid. Using ASLi with Arm's v8.6-A ISA specification. Blog post, 2017a. URL https://alastairreid.github.io/asl-lexical-syntax/.

A. Reid. Dissecting Arm's machine readable specification. Blog post, 2017b. URL https://alastairreid.github.io/dissecting-ARM-MRA/.

A. Reid. Formal validation of the Arm v8-M specification. Blog post, 2017c. URL https://alastairreid.github.io/validating-specs/.

A. Reid. Arm releases machine readable architecture specification. Blog post, 2017d. URL https://alastairreid.github.io/ARM-v8a-xml-release/.

A. Reid. Arm v8.3 machine readable specification released. Blog post, 2017e. URL https://alastairreid.github.io/arm-v8_3/.

A. Reid. Who guards the guards? Formal validation of the Arm v8-M architecture specification. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017f. DOI 10.1145/3133912.

A. Reid. ASL lexical syntax. Blog post, 2020. URL https://alastairreid.github.io/using-asli/.

A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi. End-to-end verification of processors with ISA-Formal. In *Computer Aided Verification (CAV)*, volume 9780 of *Lecture Notes in Computer Science*. Springer, 2016. DOI 10.1007/978-3-319-41540-6_3.

P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*. ACM, 2008. DOI 10.1145/1375581.1375602.

A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Lisp and Functional Programming (LFP)*. ACM, 1992. DOI 10.1145/141471.141563.

M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell. Islaris: verification of machine code against authoritative ISA semantics. In *Programming Language Design and Implementation (PLDI)*. ACM, 2022. DOI 10.1145/3519939.3523434.

M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. *Log. Methods Comput. Sci.*, 11 (1), 2015. DOI 10.2168/LMCS-11(1:7)2015.

I. Sergey, S. Peyton Jones, and D. Vytiniotis. Theory and practice of demand analysis in Haskell. Unpublished draft, 2014. URL https://core.ac.uk/display/357603019.

J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), 2013. DOI 10.1145/2487241.2487248.

T. Sewell, M. O. Myreen, Y. K. Tan, R. Kumar, A. Mihajlovic, O. Abrahamsson, and S. Owens. Cakes that bake cakes: Dynamic computation in CakeML. In *Programming Language Design and Implementation (PLDI)*. ACM, 2022. DOI 10.1145/3591266.

T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation (PLDI)*. ACM, 2013. DOI 10.1145/2491956.2462183.

K. Slind and M. Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008. DOI 10.1007/978-3-540-71067-7_6.

Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C. Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4 (POPL), 2020. DOI 10.1145/3371091.

G. Stelle and D. Stefanovic. Verifiably lazy: Verified compilation of call-by-need. In *Implementation and Application of Functional Languages (IFL)*. ACM, 2018. DOI 10.1145/3310232.3310236.

G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Principles of Programming Languages (POPL)*. ACM, 2015. DOI 10.1145/2676726.2676985.

N. Swamy, A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman, and G. Martínez. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP), 2020. DOI 10.1145/3409003.

Y. K. Tan, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, and M. Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29, 2019. DOI 10.1017/S0956796818000229.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*. ACM, 2008. DOI 10.1145/1328438.1328486.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2), 1990. DOI 10.1016/0304-3975(90)90147-A.

P. Wadler. The expression problem. Mailing list, 1998. URL https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture (FPCA)*, volume 274 of *Lecture Notes in Computer Science*. Springer, 1987. DOI 10.1007/3-540-18317-5_21.

X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014. DOI 10.5555/2685048.2685052.

Y. Wang, P. Wilke, and Z. Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL), 2019. DOI 10.1145/3290375.

Y. Wang, X. Xu, P. Wilke, and Z. Shao. CompCertELF: verified separate compilation of C programs into ELF object files. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. DOI 10.1145/3428265.

R. N. M. Watson, S. W. Moore, P. Sewell, and P. Neumann. An introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, 2019.

R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, 2020.

S. Weirich, A. Voizard, P. H. A. de Amorim, and R. A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP), 2017. DOI 10.1145/3110275.

L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL), 2020. DOI 10.1145/3371119.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*. ACM, 2011. DOI 10.1145/1993498.1993532.

J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Principles of Programming Languages (POPL)*. ACM, 2012. DOI 10.1145/2103656.2103709.