Improving Thunks in the Verified PureCake Compiler

Master's thesis in Computer science and engineering

NIKOLAOS ALEXANDRIS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Improving Thunks in the Verified PureCake Compiler

NIKOLAOS ALEXANDRIS

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Improving Thunks in the Verified PureCake Compiler
NIKOLAOS ALEXANDRIS

© NIKOLAOS ALEXANDRIS, 2025.

Supervisor: Magnus Myreen, Department
Examiner: Krasimir Angelov, Department

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Improving Thunks in the Verified PureCake Compiler
NIKOLAOS ALEXANDRIS
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Compiler construction is a research area that is constantly being developed, as the need for faster and more correct programming languages and implementations is steadily growing. While much work has been done to ensure compilers output efficient and semantically intact code, the complexity and size of the current optimising compilers lead to bugs that can be hard to find and diagnose. Situations where correctness is critical has therefore led to the development of formally verified compilers, that guarantee the correctness of the generated code. The price paid for developing verified compilers is that every step in the compilation process must be carefully implemented and formalised in order for the verification system to accept it, which makes optimisation pipelines much harder to implement.

This project's work is based on the PureCake verified compiler for pure, lazy functional languages. PureCake is an extension of the CakeML project, which itself is a verified compiler for strict functional languages, both developed using the HOL4 formal verification system. The PureCake compiler is already a complete verified compiler, but it lacks the optimisations that would make its generated code as efficient as the generated code of optimising non-verified compilers for lazy languages such as GHC. This project's goal is to enhance PureCake's and CakeML's language with features that will enable critical optimisations to be implemented in order to close the gap between verified and non-verified compilers for lazy languages in terms of runtime efficiency.

In this thesis, we take on the task of improving the quality of PureCake's generated code, by optimising the handling of thunks by the PureCake and CakeML compilers. We first target the representation of thunks inside PureCake, by enriching the semantics of the operations responsible for handling thunks and by proving these changes correct. Then, we implement thunks in CakeML and lay the foundations for future optimisations in the runtime. This work will help bring PureCake up to par with unverified compilers in terms of the generated executable's performance, while also making sure that the soundness of the compilation process remains intact.

Keywords: Computer science, thesis, functional programming, formal verification, lazy languages, compilers, compiler optimisations.

Acknowledgements

I would like to thank my supervisor Magnus O. Myreen for his support and guidance during this project. I would also like to thank the developers of CakeML for their help in learning HOL, especially Hrutvik Kanabar for his comments throughout the development of this work. I would also like to express my gratitude to my family for their encouragement, and my apologies to my friends for cancelled plans and the occasional nag about troublesome proofs.

Nikolaos Alexandris, Gothenburg, 2025-05-28

Contents

List of Figures **xi**

1 Introduction **1**
- 1.1 Formal Verification in Software Systems . . . 1
- 1.2 The PureCake Project . . . 2
- 1.3 PureCake's Compilation Process . . . 3
- 1.4 Motivation . . . 4
  - 1.4.1 Thunks in Function Application . . . 4
  - 1.4.2 Thunks in Data Structures . . . 5
- 1.5 Project Outline . . . 7

2 Logical Relation for `ThunkLang` **9**
- 2.1 The `ThunkLang` Intermediate Language . . . 9
  - 2.1.1 Syntax . . . 9
  - 2.1.2 Semantics . . . 10
- 2.2 Step-Indexed Logical Relations . . . 11
- 2.3 A Logical Relation for `ThunkLang` . . . 13
- 2.4 Properties . . . 15
- 2.5 Usage . . . 16

3 Strengthening the Semantics of `force` in PureCake **17**
- 3.1 Adapting `ThunkLang` to the New Semantics . . . 17
  - 3.1.1 Proving correctness of `ThunkLang` Optimisations . . . 18
  - 3.1.2 Side Effects in a Pure Language . . . 19
  - 3.1.3 Fixing `delay_force` . . . 20
- 3.2 The Stateful Thunk Semantics in `StateLang` . . . 22
  - 3.2.1 `StateLang`'s Stateful Semantics . . . 22
  - 3.2.2 The Dual Representation of Thunks in `StateLang` . . . 23
  - 3.2.3 Making Thunks Explicit in `StateLang` . . . 24
  - 3.2.4 Recursive forcing of Thunks . . . 26
- 3.3 Summary . . . 28

4 Adding Thunks to CakeML **29**
- 4.1 The Semantics of CakeML . . . 29
- 4.2 Carrying the Semantics Through CakeML . . . 31
  - 4.2.1 Transforming the State . . . 31

4.2.2 Closure Conversion 32
4.3 Summary 32

5 Conclusion **33**
5.1 Future Work 33
5.1.1 Project Extensions 33
5.1.2 Runtime Thunk Inlining 34

Bibliography **37**

List of Figures

1.1 PureCake Intermediate Representations . . . 4
1.2 Strict list memory representation . . . 5
1.3 Unevaluated lazy list memory representation . . . 6
1.4 Lazy list after evaluating the first element . . . 6
1.5 Lazy list after evaluating the spine . . . 6
1.6 Lazy list after full evaluation . . . 7

2.1 `ThunkLang` Syntax . . . 9
2.2 Example big-step evaluation rules . . . 10
2.3 Example `ThunkLang` evaluation rules . . . 11
2.4 Top level `ThunkLang` evaluation function . . . 11
2.5 Derivation rules of `ThunkLang`'s special operators . . . 11
2.6 `ThunkLang` Logical Relation . . . 14

3.1 Simplified derivation rules of `ThunkLang`'s special operators . . . 17
3.2 Stronger version of `force` . . . 17
3.3 Translation of monadic expressions from PureCake source to `ThunkLang` prior to this work . . . 21
3.4 Adapted translation of monadic expressions from PureCake source to `ThunkLang` . . . 21
3.5 Beginning of thunk evaluation . . . 26
3.6 Thunk evaluation conflict . . . 26

4.1 Compiler structure after `StateLang` . . . 29

5.1 Evaluated thunk before optimisation . . . 34
5.2 Evaluated thunk after garbage collection . . . 34
5.3 Memory layout after evaluating `sum [1,2,3]` . . . 34
5.4 Memory layout after GC has run and inlined the evaluated thunks . . . 35

1 Introduction

In this section we describe the setting of our work and motivate our goals. We also give a brief outline of our objectives and the structure of the rest of the thesis.

1.1 Formal Verification in Software Systems

Formal verification is defined as the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness [1]. Specifically, in the context of software development, a piece of software is said to be formally verified if it provably transforms its inputs to outputs based on a given specification. A specification is usually a mathematical model of the program. For example, one could formally verify a regular expression matching program by describing a mapping of the program's datatypes to finite state automata, and a mapping of the program's procedures to transformations over these automata. The specification then can use the mathematical theory of finite automata to prove theorems about the program, such as that the matcher only accepts inputs included in the language of the regular expression.

However, for sufficiently complex programs, doing this transformation on paper would be impractical and very error-prone. Therefore, the preferred way is to use software tools called theorem provers or proof assistants. These tools allow the programmer to state theorems about the behaviour of their code directly in the programming language and then interactively prove them. In our work for this thesis, we have used the HOL theorem prover [2], a proof assistant implemented on top of the Standard ML language which supports specification and proof development in higher-order logic.

This thesis is concerned with formal verification of programming language compilers. Compilers are some of the most complicated pieces of software, and are notoriously difficult to implement. However, their structure makes them a particularly good fit to develop in a theorem prover, which presents its own difficulties, but can guarantee that the compiler is sound and bug-free. Formally verified compilers already exist and have been successfully used in real-world systems, where code correctness is critical. A prominent example of a practical verified compiler is CompCert [3], a fully-featured, formally validated compiler for a large subset of the C99 language.

1.2 The PureCake Project

While verified compilers such as CompCert exist, they compile low level languages. This is the case because these languages have been historically used in critical systems due to their time, memory, and energy efficiency. Nevertheless, with the constant progress in hardware and software research, higher-level languages are now also viable to be used in even the most constrained environments. This is preferred when possible, as high level languages provide better programming facilities and more guarantees about the behaviour of the code.

In contrast to low level and imperative languages such as C, high level languages strive to be more declarative and aid the programmer to only have to think about the task at hand. This is achieved by providing powerful features both during the compilation of the programs, such as type inference and pattern matching, and during the runtime such as garbage collection. One such class of languages is functional languages. Functional programming's main feature is the treatment of functions as first-class entities, meaning that any function can be passed as an argument, returned from another function or stored in a data structure. Whereas in imperative programming procedures are sequences of instructions, in functional programming functions are expressions that transform inputs to outputs in a modular manner.

Writing programs in a functional style leads to fewer bugs and easier to test programs, so having a verified compiler for a functional language would provide strong guarantees about a program's behaviour. The CakeML project [4] implements the first fully-featured formally verified compiler for a high level, functional programming language. It compiles an ML-like language complete with type inference, pattern matching, and garbage collected runtime, targeting various hardware architectures. CakeML is continuously developed and optimisations are implemented, bridging the gap in runtime efficiency with the mainstream unverified alternatives.

Even though CakeML is a step towards a more modular and high level verified language, it follows the tradition of the Standard ML language; code usually conforms to the functional programming style, but side effects such as mutating variables and printing to the terminal are still allowed. Pushing the functional paradigm even further are languages such as Haskell, commonly described as pure, lazy functional languages. In this context, pure means that calling a function with the same inputs will always yield the same results, thus implying absence of side effects, and lazy means that every expression is only evaluated when needed. This means that computation in these languages directly maps to evaluation of mathematical functions, which makes reasoning about programs very intuitive.

Building on top of CakeML, the PureCake project [5] has much the same goals as CakeML, but for the compilation of lazy pure functional languages. The PureCake compiler handles the front-end of the compilation process, such as parsing and type checking for PureCake's source language. It then gradually transforms the source program through a series of intermediate languages applying optimisations and simplifications, and ends up with an equivalent CakeML source program. To complete the compilation process, this program is handed off to the CakeML compiler

which takes care of the rest of the compilation, therefore creating a verified end-to-end process.

1.3 PureCake's Compilation Process

For PureCake to use the CakeML compiler as a backend to handle the verified generation of machine code, it first has to translate its language to the CakeML compiler's source language. The semantics of these languages differ mainly in two ways: purity and evaluation strategy. Our work is focused on the transformation of the evaluation strategy.

CakeML uses the call-by-value evaluation strategy, therefore CakeML is called a strict language. In a strict language, when a function is called its arguments are evaluated before the body of the function executes. On the other hand, PureCake uses the call-by-name evaluation strategy, making PureCake a lazy language. In PureCake, when a function is called its arguments are not immediately evaluated. Instead, the function body is evaluated and, if the value of some argument is demanded for the evaluation of the function body, only then is the value of the argument computed.

To transition from one evaluation strategy to the other, the PureCake compiler transforms the lazy program to a strict one by introducing thunks. Thunks are containers that wrap a single unevaluated expression, and evaluate their expression when forced to do so. For example, consider the following simple Pure function definition:

```
suc n = n + 1
```

When a call like `suc (f x)` is evaluated, the `f x` call is only evaluated when the + operator demands its value in the body of `suc`. In order to transform the program's evaluation strategy, the compiler extends the language with two special expressions: `delay` and `force`. Here, `delay` wraps an expression in a thunk, and `force` queries a thunk for its value. Therefore, the `suc (f x)` lazy call could be simulated in a strict language by transforming the call to `(force suc) (delay ((force f) (delay x)))` and the function to

```
suc n = (force n) + 1
```

One can think of `delay` as wrapping an expression in a function that expects a dummy value, and `force` as calling such a function with the dummy value, thus evaluating the contained expression.

PureCake achieves the evaluation strategy transformation by passing the source program through a series of intermediate languages, at each step changing the semantics of the language to close the gap between PureCake and CakeML's semantics. The three intermediate languages are `ThunkLang`, `EnvLang`, and `StateLang`.

In `ThunkLang`, the `delay` and `force` operators are introduced and placed around the appropriate expressions as described above, and the evaluation strategy is converted to call-by-value. While this is semantically enough, it would be inefficient in practice. In `ThunkLang`, when a thunk is forced multiple times, its contained expression will

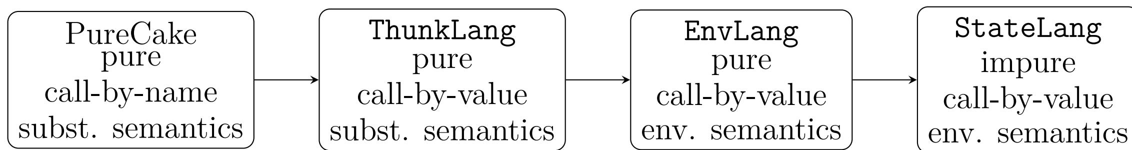

Figure 1.1: PureCake Intermediate Representations

be evaluated for each `force` separately. Since PureCake is a pure language, we know that the value of the thunk will always be the same regardless of when we evaluate it. So, ideally, we should only compute it once and then memoize its value for the rest of the forces.

This problem is solved in `StateLang`. `StateLang`'s semantics employ a mutable memory called a store. Each time a `delay` is executed, a thunk is produced and appended to the store along with a flag stating that the thunk has not yet been evaluated. Then, the first `force` demanding the thunk's value will get a reference to the thunk in the store and evaluate its expression. It will then set the flag to state that the thunk has been evaluated, and overwrite the unevaluated expression with the result it computed. Any subsequent `force` will then directly use the value instead of re-computing the expression.

In between these two intermediate languages is `EnvLang`, whose purpose is to change the semantics from substitution-based to environment-based. Since this language does not change how thunks behave, it will not concern us significantly throughout this thesis.

1.4 Motivation

Even though lazy languages provide the facilities to write very high-level code, laziness comes at a runtime cost. We will discuss two examples of how unoptimised thunks unnecessarily slow down simple programs:

1.4.1 Thunks in Function Application

In a strict conventional language, a function call usually just involves a few machine instructions to move the arguments to the right registers and push a stack frame. In a lazy language however, we need to allocate space in the main memory to store a thunk for each argument, which is orders of magnitudes slower than the strict function calls. Since functions are invoked constantly in a functional language, it is critical to optimise their usage as much as possible.

As a case study, consider the following PureCake function that sums the numbers from 1 to `n` using an accumulator argument:

```
count n acc =
  if n == 0 then acc else count (n - 1) (n + acc)
```

We observe that the values of both arguments are always demanded; the value of `n`

is needed for the conditional, and the value of `acc` is demanded either directly in the `then` case, or through the application to + in the `else` case (this can be inferred by the compiler using strictness analysis [6]). Therefore, even though this function is used in a lazy language, its behaviour will always be strict. We also note that a call-by-value language will optimise this function even further by noticing it's tail recursive [7], turning it into a single loop and thus avoiding even the stack allocations.

Since `count` is effectively strict in PureCake too, the compiler should ideally achieve the same degree of optimisation. Currently, the compiler initially translates the source language to `ThunkLang`, by adding delays to arguments in each function application and forces to every use of a parameter:

```
count n acc =
  if (force n) == 0
    then (force acc)
    else (force count) (delay ((force n) - 1))
                       (delay ((force n) + (force acc))
```

It then applies some `ThunkLang-to-ThunkLang` optimisations, namely demand analysis and a worker-wrapper transformation [8], and produces the following code:

```
count' n acc =
  let n' = force n in
  let acc' = force acc in
  if n' == 0 then acc' else
    count' (delay (n' - 1)) (delay (n' + acc'))
count = delay count'
```

The compiler detects that `n` and `acc` will always be demanded, so it forces them in the beginning of the function, thus avoiding work duplication. Still, each recursive call has to allocate unnecessary thunks for its arguments, but further work is planned to make the behaviour eventually be optimal. Even so, we observe the importance of efficiently handling thunks, since many of them are allocated and forced even when dealing with simple functions.

1.4.2 Thunks in Data Structures

Another important part of functional programming where thunks come up is data structures. Consider the memory representation of evaluating the list `xs = [1,2,3]` in a strict language. The resulting data structure would look like a linked list, with a list of pairs each storing a number and a reference to the next pair in the list.

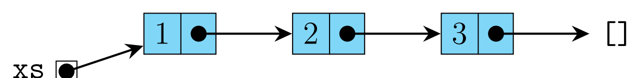

Figure 1.2: Strict list memory representation

However, in a lazy language the result is quite different. Initially, the result of evaluating `xs = [1,2,3]` is only an unevaluated thunk (we denote the evaluated or unevaluated flag with `E` and `NE` respectively).

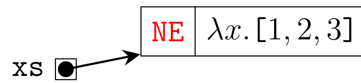

Figure 1.3: Unevaluated lazy list memory representation

How much the data structure will be evaluated is only determined by the operations we apply to it.

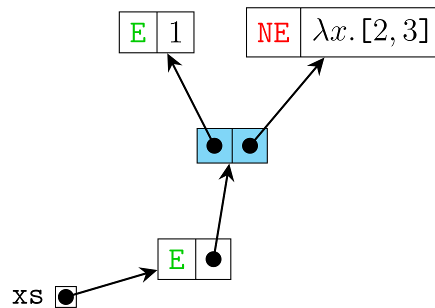

Figure 1.4: Lazy list after evaluating the first element

For example, evaluating the expression `head xs` demands the evaluation of the list's first element, but the rest of the list remains unevaluated. Another possibility is to only evaluate every second element of each list cell, also known as evaluating the spine of the list. This happens when we examine the structure of the list without caring about its contents, for example when evaluating `length xs`.

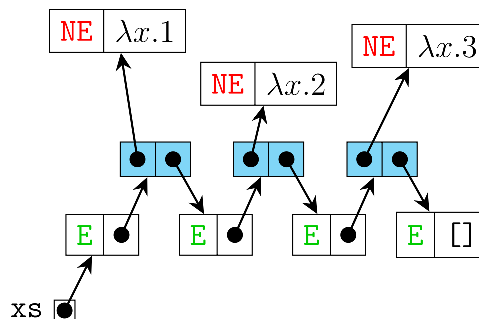

Figure 1.5: Lazy list after evaluating the spine

Lastly, a full evaluation would happen when we traverse the list (thus evaluating the spine) and also examining each stored element and therefore demanding its value too. An example of such an operation is `sum xs` which returns the sum of every element in the list, therefore both traversing the whole list and demanding all its values.

Conceptually, the result of fully evaluating the lazy list should be the same as having a fully evaluated strict list. However, because of the gradual evaluation process, the actual list cells exist in between two layers of thunks, which adds unnecessary indirections when we try to later access the elements.

Our goal in this project is to optimise PureCake's handling of thunks as much as possible, since it's a critical point for bringing the efficiency of PureCake's generated code up to par with that of non-verified compilers for lazy languages.

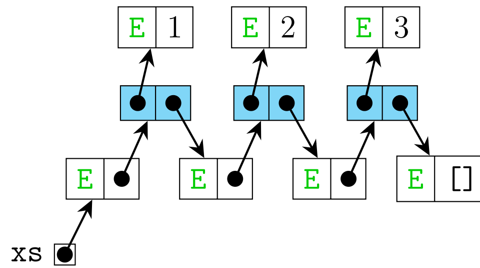

Figure 1.6: Lazy list after full evaluation

1.5 Project Outline

Our project consists of three parts:

1. Creating and formalising a logical relation for `ThunkLang`;
2. Enriching the semantics of thunk operations in PureCake;
3. Adding thunks as first-class constructs to CakeML.

The first part is mainly a quality-of-life improvement for proofs in `ThunkLang`. Developing a logical relation for the intermediate representation will greatly reduce the complexity and improve the maintainability of the proofs, by providing a framework for developing them that reduces boilerplate and promotes smaller proofs that compose general rules for the language constructs instead of big simulations.

The second part in chapter 3 is concerned with enriching the semantics around the handling of thunks in PureCake, so certain needed invariants will hold when we make the transition to CakeML. We will also discuss the necessary changes we had to make in the PureCake codebase to support this strengthened semantics.

Lastly, the third part in chapter 4 aims to add native thunks to CakeML. Even with optimisations which try to minimise the generation of unnecessary thunks, many thunks will eventually make it to the runtime. Prior to this project CakeML didn't support thunks, so PureCake compiles them to 2-valued arrays during the transition to CakeML, where the first cell is used as a flag to distinguish between evaluated and unevaluated thunks, and the second as the value of the thunk. This prevents CakeML to differentiate them from any other array, thus blocking the compiler from performing any meaningful optimisations on them. Having a robust implementation of thunks in CakeML will enable optimisations to be implemented in the runtime. We discuss some of them in detail in Future Work.

2

Logical Relation for ThunkLang

This chapter describes the development of a logical relation for the ThunkLang intermediate language of the PureCake compiler. We start with an analysis of ThunkLang's semantics and the proof methods currently used for proving optimisation transformations in it, followed by an overview of logical relations. Then, we motivate the introduction of a logical relation for ThunkLang and give our definition along with its properties and applications.

2.1 The ThunkLang Intermediate Language

Each one of PureCake's intermediate languages has its own syntax and semantics. ThunkLang is the first one, and its structure resembles that of the source language closely with just two differences: conversion of the calling convention from call-by-name to call-by-value and introduction of the `force` and `delay` expressions. We describe its syntax and semantics in depth to help with the understanding of the logical relation in the next section.

2.1.1 Syntax

ThunkLang's syntax is described in the following figure:

$$
\begin{array}{rcl}
exp & ::= & \texttt{var}\ x \\
& | & \texttt{Prim}\ op\ \overline{e_n} \\
& | & \texttt{Monad}\ mop\ \overline{e_n} \\
& | & e_1 \cdot e_2 \\
& | & \lambda x.\ e \\
& | & \texttt{letrec}\ \overline{x_n = e_n}\ \texttt{in}\ e \\
& | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
& | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& | & \texttt{delay}\ e \\
& | & \texttt{force}\ e \\
& | & \texttt{value}\ v \\
& | & \texttt{tick}\ e
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \texttt{Constructor}\ \mathrm{C}\ \overline{v_n} \\
& | & \texttt{Monadic}\ mop\ \overline{e_n} \\
& | & \texttt{Closure}\ x\ e \\
& | & \texttt{Recclosure}\ \overline{(x_i, e_i)}\ x \\
& | & \texttt{Thunk}\ e \\
& | & \texttt{Atom}\ lit \\
& | & \texttt{DoTick}\ v
\end{array}
$$

Figure 2.1: ThunkLang Syntax

The *exp* type describes the language's expressions. We denote the use of variables by name with `var`. `Prim` and `Monad` apply either a pure or a monadic operator to a list of expressions. Application and abstraction are denoted as $e_1 \cdot e_2$ and $\lambda x.\ e$. `let` and `letrec` denote name binding, `if-then-else` the usual conditional operator, and `value` embeds a value into an expression. The `delay`, `force` and `tick` operators are `ThunkLang`'s special operators. `delay` and `force` work as described in chapter 1 and the tick operator will be explained in the next section.

The set v describes the language's values. `Constructor` and `Monadic` construct pure and monadic values respectively, `Closure` and `Recclosure` represent the closures created by `let` and `letrec`, and `Atom` represents a primitive value. `Thunk` and `DoTick` are the new values added in `ThunkLang` when compared with the source language. `Thunk` wraps an expression in an unevaluated thunk, and `DoTick` will be explained along with `tick`.

We also observe that *exp* and v are mutually recursive by means of the `value` constructor in *exp*. This is due to `ThunkLang`'s semantics being substitution based. This means that whenever a function call `f` e is encountered, the resulting value v of evaluating e is substituted into each appearance of the corresponding formal parameter p of `f`, turning every `var` p expression into `value` v. Primary consequence of this is that when evaluating a `ThunkLang` expression, encountering a variable name is always an error, since the only way we can run into a variable expression is if it hasn't been bound to a value.

2.1.2 Semantics

`ThunkLang`'s semantics is defined in a big-step functional style [9]. Big-step semantics is defined by derivation rules for each construct of the language, and these derivation rules is defined in terms of a function $[\![\cdot]\!]$ that maps expressions of the language to values.

Usually in big-step semantics, $[\![\cdot]\!]$'s type would be $exp \ \rightarrow \ v \mid \texttt{Fail}$, where `Fail` handles the undefined cases such as encountering a variable or applying a value that is not a closure. Some derivation rules in such semantics would be:

$$\frac{}{[\![\texttt{var}\ x]\!] = \texttt{Fail}} \qquad \frac{[\![e_1]\!] = \texttt{true} \qquad [\![e_2]\!] = v}{[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = v}$$

Figure 2.2: Example big-step evaluation rules

Notably, this semantics doesn't capture non-termination. In the `if-then-else` case, for example, if $[\![e_2]\!]$ doesn't terminate, we have not defined how $[\![\cdot]\!]$ should behave. Usually this is not problematic, as translating this semantics to a function in any programming language will imply termination if any part of the computation terminates.

However, when developing proofs in a formal verification tool such as HOL, every function defined must be shown to terminate for all possible inputs. Therefore,

writing an evaluation function for any becomes impossible. To mitigate this, we use a technique called clocking [10]. This approach works by adding a natural number called a clock as an extra argument to $[\![\cdot]\!]_k$, and whenever the function recurses the clock decreases. The result type is also augmented with a constructor to represent non-termination, which gets produced any time the function is called with a zeroed clock. With this transformation, the evaluation function's type becomes $[\![\cdot]\!]_k : \mathbb{N} \to \mathit{exp} \to v \mid \texttt{Fail} \mid \texttt{OOT}$. Proving that $[\![\cdot]\!]_k$ terminates is now trivial, using the fact that the clock is strictly monotonic decreasing. With this transformation, the actual semantic derivation rules presented above for ThunkLang become:

$$\frac{}{[\![\texttt{var}\ x]\!]_k = \texttt{Fail}} \qquad \frac{[\![e_1]\!]_{k-1} = \texttt{true} \qquad [\![e_2]\!]_{k-1} = v}{[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_k = v}$$

$$\frac{}{[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_0 = \texttt{OOT}}$$

Figure 2.3: Example ThunkLang evaluation rules

In addition to the clocked function $[\![\cdot]\!]_k$, a regular evaluation function $[\![\cdot]\!]$ is also usually defined as

$$\frac{\nexists k.\ [\![e]\!]_k \neq \texttt{OOT}}{[\![e]\!] = \texttt{OOT}} \qquad \frac{\exists k.\ [\![e]\!]_k \neq \texttt{OOT} \qquad [\![e]\!]_k = v}{[\![e]\!] = v}$$

Figure 2.4: Top level ThunkLang evaluation function

We now have enough context to explain the meaning of the `tick` expression and `DoTick` value. Similarly to how the `delay` expression wraps its contained expression in a thunk value for `force` to unwrap, the `tick` expression wraps its expression in a `DoTick` value. When `force` encounters a `DoTick` value, it recurses with the underlying value wrapped in a `force` ∘ `value`, essentially repeating the force opeation with one less clock and the unwrapped `DoTick` value.

$$\frac{[\![e]\!]_k = v}{[\![\texttt{tick}\ e]\!]_k = \texttt{DoTick}\ v} \qquad \frac{}{[\![\texttt{delay}\ e]\!] = \texttt{Thunk}\ e}$$

$$\frac{}{[\![\texttt{force}\ e]\!]_0 = \texttt{OOT}} \qquad \frac{[\![x]\!]_k = \texttt{Thunk}\ e \qquad [\![e]\!]_{k-1} = u}{[\![\texttt{force}\ x]\!]_k = u}$$

$$\frac{[\![x]\!]_k = \texttt{DoTick}\ v \qquad [\![\texttt{force}\ (\texttt{value}\ v)]\!]_{k-1} = u}{[\![\texttt{force}\ x]\!]_k = u}$$

Figure 2.5: Derivation rules of ThunkLang's special operators

2.2 Step-Indexed Logical Relations

Step-indexed logical relations [11] are a useful tool for proving properties of programs. When working with untyped clocked big-step semantics, proving basic properties

about programs becomes repetitive and error-prone, with reasoning about clocks taking up the majority of the proofs. Logical relations are already used in CakeML to make reasoning about programs easier.

A (binary) logical relation relates two constructs that logically match each other. For example, when considering the results of `ThunkLang`'s evaluation function above, we could say that a `Fail` result is related with a `Fail` result, but not with an `OOT` result. This, of course should extend to all the constructs of the language. When evaluating two expressions successfully with results v_1 and v_2, are the results related or not? It becomes evident that we need to construct a similar relation for values and for expressions. After defining the relations for every construct of the language, we have to prove various compatibility theorems about the relation. These are the theorems that proofs using the relation depend on to reduce the amount of work they have to manually do. These compatibility theorems prove that the relation is closed under the constructs of the language. For example, if R is a logical relation and we have $R(x_1, x_2)$ and $R(f_1, f_2)$, then $R(f_1 \cdot x_1, f_2 \cdot x_2)$ should hold.

The constructs we usually want to relate using these relations are programs that differ in a way that implements some form of optimisation or otherwise useful transformation, but equational reasoning of the two programs is not enough to show that their meaning before and after the transformation does not change. For example, as we will discuss shortly, the two programs $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ and $e_2[e_1/x]$ cannot be show equivalent directly and a more general relation must be used to relax this constraint.

Currently, proofs concerning `ThunkLang` in PureCake are carried out using syntax-based relations that require explicit handling of all expression types and manually matching up the clocks by inserting `tick` expressions. This adds a lot of boilerplate code obfuscating the essence of the proof, and requires crafting a new relation every time which comes at the cost of time, complexity, and duplication. Although logical relations already exist in CakeML, the user of the relation still needs to match up the clocks when carrying out proofs. For example, consider the two programs $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ and $e_2[e_1/x]$. These are obviously equivalent, but the first program takes an extra step compared to the second one. Because the relation is step-indexed, meaning that programs are not only related to each other but also to a mutual clock, these two programs are not directly related. This is the reason intermediate languages in CakeML and PureCake contain `tick` expressions. The two programs can be show to be equivalent by relating $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ and $\texttt{tick}(e_2[e_1/x])$. Our goal is to develop a logical relation that will completely decouple the proofs from the clocking mechanism.

Previous work on developing a logical relation that decouples clocks has been done in Paraskevopoulou et al. [12]. The authors describe a logical relation for CertiCoq's λANF intermediate representation that doesn't require the clocks to match exactly, but accepts any relation over the clocks that satisfies a set of properties. We take inspiration from their implementation and adapt it to `ThunkLang`.

To adapt the logical relation defined in Paraskevopoulou et al. [12], we first need to develop an alternative semantics for `ThunkLang`, because the clocking in our

language is different from the clocking of their λANF language. As explained in Kumar et al. [13], there are two ways of embedding clocks in evaluation functions: environment-like clocks and state-like clocks. Environment-like clocking decrements the clocks on each recursive call and doesn't return them, while state-like clocking passes the clocks around as state, thus limiting evaluation in terms of length instead of in terms of depth. For example, when evaluating an `if-then-else` expression with an environment-like clock k, the recursive calls to expressions e_1, e_2, and e_3 will all be passed the same clock, $k-1$, whereas with a state like clock, each e_i would get a clock k_i such that $\sum k_i \leq k$. `ThunkLang`'s semantics use an environment-like clock.

We define an alternative, yet equivalent, semantics that uses a state-like clock for the purpose of being compatible with the logical relation. We also add a second clock called a trace, which only decrements when a function application occurs instead of at every step. This is used to track function application depth when proving theorems using the relation about transformations that add or remove function applications from the programs. This results in the alternative semantics $[\![\cdot]\!]_k^t$.

2.3 A Logical Relation for `ThunkLang`

To develop a logical relation for a language, we need to define how any two constructs of the language are allowed to be related. Therefore, we create a family of relations, one for every construct of the language. Each of these relations will then be used to prove that the relation as a whole behaves correctly according to the semantics of the language. For example, we prove that for any two related expressions under the expression relation, their evaluations produce related results under the value relation, lifting that weight from the user of the logical relation.

The logical relation, given in whole in Figure 2.6, consists of a value relation $\mathcal{V}$, a result relation $\mathcal{R}$, an expression relation $\mathcal{E}$, a variable relation $\mathcal{X}$, and a substitution relation $\mathcal{C}$. Each relation is indexed by a clock k and accepts either one or two invariants. The expression relation takes two invariats, a local invariant Q_L and a global invariant Q_G, while the others take just one invariant Q.

The value relation $\mathcal{V}$ relates two `ThunkLang` values. Two constructor values are related if their constructors are the same and their argument lists have the same length and are pairwise related. Two closure values are related if for any pair of values related at a strictly smaller step index the closure bodies are related at this index after substituting their formal parameters with the values. Two recursive closure values are related if for any strictly smaller step index the selected closure bodies are related at that index after substituting the recursive functions into their bodies. Two thunks are related if for any strictly smaller step index their enclosed expressions are related. Finally, two tick expressions are related if for their values are related.

The result relation $\mathcal{R}$ relates two `ThunkLang` results. Two `OOT` results and two `Fail` results are trivially related, and two value results are related if their values are related.

Value Relation

$\mathcal{V}^k(\texttt{Constructor}\ n_1\ \overline{v_1}, \texttt{Constructor}\ n_2\ \overline{v_2})\{Q\} \equiv$

$\quad n_1 = n_2 \wedge \mathcal{V}^k(\overline{v_1}, \overline{v_2})\{Q\}$

$\mathcal{V}^k(\texttt{Monadic}\ op_1\ \overline{v_1}, \texttt{Monadic}\ op_2\ \overline{v_2})\{Q\} \equiv$

$\quad op_1 = op_2 \wedge \mathcal{V}^k(\overline{v_1}, \overline{v_2})\{Q\}$

$\mathcal{V}^k(\texttt{Closure}\ x_1\ e_1, \texttt{Closure}\ x_2\ e_2)\{Q\} \equiv$

$\quad \forall i < k\ v_1\ v_2.\ \mathcal{V}^i(v_1, v_2)\{Q\} \wedge \mathcal{E}^i(e_1[v_1/x_1], e_2[v_2/x_2])\{Q, Q\}$

$\mathcal{V}^k(\texttt{Recclosure}\ \overline{f_1}\ x_1, \texttt{Recclosure}\ \overline{f_2}\ x_2)\{Q\} \equiv$

$\quad \forall i < k\ e_1.\ e_1 = \overline{f_1}.x_1 \Rightarrow \exists e_2.\ e_2 = \overline{f_2}.x_2 \wedge \mathcal{E}^i(e_1[\overline{f_1}], e_2[\overline{f_2}])\{Q, Q\}$

$\mathcal{V}^k(\texttt{Thunk}\ e_1, \texttt{Thunk}\ e_2)\{Q\} \equiv \forall i < k.\ \mathcal{E}^i(e_1, e_2)\{Q, Q\}$

$\mathcal{V}^k(\texttt{Atom}\ l_1, \texttt{Atom}\ l_2)\{Q\} \equiv l_1 = l_2$

$\mathcal{V}^k(\texttt{DoTick}\ v_1, \texttt{DoTick}\ v_2)\{Q\} \equiv \mathcal{V}^k(v_1, v_2)\{Q\}$

$\mathcal{V}^k(v_1, v_2)\{Q\} \equiv \text{False}$ Otherwise

Result Relation

$\mathcal{R}^k(\texttt{Fail}, \texttt{Fail})\{Q\} \equiv \text{True}$

$\mathcal{R}^k(\texttt{OOT}, \texttt{OOT})\{Q\} \equiv \text{True}$

$\mathcal{R}^k(v_1, v_2)\{Q\} \equiv \mathcal{V}^k(v_1, v_2)\{Q\}$

$\mathcal{R}^k(r_1, r_2)\{Q\} \equiv \text{False}$ Otherwise

Expression Relation

$\mathcal{E}^k(e_1, e_2)Q_L, Q_G \equiv \forall c_1 \leq k\ t_1\ r_1.\ [\![e_1]\!]_{c_1}^{t_1} = r_1 \Rightarrow$

$\quad \exists c_2\ t_2\ r_2.\ [\![e_2]\!]_{c_2}^{t_2} = r_2 \wedge Q_L(c_1, t_1)(c_2, t_2) \wedge \mathcal{R}^{k-c_1}(r_1, r_2)\{Q_G\}$

Variable Relation

$\mathcal{X}^k(sub_1, sub_2, x, y)\{Q\} \equiv$

$\quad \forall v_1.\ sub_1(x) = v_1 \Rightarrow$

$\quad\quad \exists v_2.\ sub_2(y) = v_2 \wedge \mathcal{V}^k(v_1, v_2)\{Q\}$

Substitution Relation

$\mathcal{C}^k(S, sub_1, sub_2)\{Q\} \equiv \forall x.\ x \in S \Rightarrow \mathcal{X}^k(sub_1, sub_2, x, x)\{Q\}$

Figure 2.6: ThunkLang Logical Relation

The substitution relation $\mathcal{C}$ relates two substitutions over a set of variables S, if for any variable in S the substitutions map the variable to related values, which is expressed by the variable relation $\mathcal{X}$.

2.4 Properties

The invariants the relations range over are binary relations between pairs of clocks and traces. Using any relation with the equality invariant ($k_1 = k_2 \wedge t_1 = t_2 \Rightarrow Eq(k_1, t_1)(k_2, t_2)$) is equivalent to using a standard logical relation. However, more complicated invariants can be used to relax the restrictions on the clocking. Coming back to the problem of relating `let` x `=` e_1 `in` e_2 and $e_2[e_1/x]$, we can now encode their relation into the invariants. When working with a syntactic relation as `ThunkLang`'s optimisations currently do, a relation between the two expressions would not work directly. One would have to relate `let` x `=` e_1 `in` e_2 with `tick`$(e_2[e_1/x])$ to make the clocks match up, which complicates both the relation and the proofs needed to show it's sound. With the logical relation though, one could use a local invariant Q_L defined as $k_1 = k_2 + 1 \wedge t_1 = t_2 \Rightarrow Q_L(k_1, t_1), (k_2, t_2)$. Using this relation, we can directly relate the two expressions since

$$\mathcal{E}^k(\texttt{let } x = e_1 \texttt{ in } e_2, e_2[e_1/x])\{Q_L, Eq\}$$

This flexibility is what enables users of the logical relation to decouple their proofs from the handling of the clocks by coming up with an appropriate invariant for their transformation.

However, the invariants are subject to some restrictions. We define the predicate `Compat` Q_L Q_G to mean that the local and global invariants satisfy a set of compatibility rules. These compatibility rules arise from the compatibility theorems of the logical relation. Specifically, we need to show that the relation is closed under `ThunkLang`'s constructs. For example, we need to prove a lemma stating that if $\mathcal{E}^k(f_1, f_2)\{Q_L, Q_G\}$ and $\mathcal{E}^k(x_1, x_2)\{Q_L, Q_G\}$, then $\mathcal{E}^k(f_1 \cdot e_1, f_2 \cdot e_2)\{Q_L, Q_G\}$. For this to be true, we need to assume some relations about the invariants. Finally, the complete lemma for application compatibility is:

Lemma 2.4.1. *Assume that* $Q_L(0,0)(0,0)$ *and that* $Q_L(c_1, t_1)(c_2, t_2) \wedge Q_L(c_3, t_3)(c_4, t_4) \Rightarrow Q_L(c_1 + c_3, t_1 + t_3)(c_2 + c_4, t_2 + t_4)$. *If*

- $\mathcal{E}^k(x_1, x_2)\{Q_L, Q_G\}$
- $\mathcal{E}^k(f_1, f_2)\{Q_L, Q_G\}$

then $\mathcal{E}^k(f_1 \cdot x_1, f_2 \cdot x_2)\{Q_L, Q_G\}$

We assume that the local invariant holds when both programs time out, and that it holds for the clocks and traces of the combination of the applications. Similarly, we need to prove compatibility lemmas for every other language feature. Each one of these lemmas requires some relation for the invariants, but the minimum set of relations needed to satisfy all compatibility theorems turns out to be the following:

1. $\dfrac{}{Q_L(0,0)(0,0)}$ This base condition is required for when both expressions time out.

2. $\dfrac{Q_L(c_1,t_1)(c_2,t_2)}{Q_L(c_1+1,t_1)(c_2+1,t_2)}$ This condition is required for operations that decrease the clock but do not perform any applications.

3. $\dfrac{Q_L(c_1,t_1)(c_2,t_2)}{Q_L(c_1+1,t_1+1)(c_2+1,t_2+1)}$ This condition is required for the case where a single application takes place along with the decrease in the clock.

4. $\dfrac{Q_L(c_1,t_1)(c_2,t_2) \quad Q_L(c_3,t_3)(c_4,t_4)}{Q_L(c_1+c_3,t_1+t_3)(c_2+c_4,t_2+t_4)}$ This last more complicated condition is used during the general application case.

2.5 Usage

Currently, each proof in `ThunkLang` has to define a new syntactical relation and prove the same lemmas about the relation. Also, an explicit handling of clocks is required making each proof that adds or removes clock even more tedious and error-prone. Even worse, `ThunkLang` has a whole `MkTick` expression that wraps any other expression and just consumes more clock, dedicated to aiding in keeping the clocks synchronised across transformations. Using this logical relation as a framework with appropriate invariants for each proof, will aid in their development and simplify the language in whole.

We have not yet applied this logical relation to re-write any of the proofs in `ThunkLang`. Our work was focused on providing the core of the relation and the proofs of the compatibility lemmas for the language's constructs. To make the logical relation practical, a comprehensive summary of the techniques required for all the proofs must be made, and a library around the relation should be developed to make the re-writing as smooth as possible, and the results optimal.

3

Strengthening the Semantics of `force` in PureCake

As discussed in chapter 1, `ThunkLang` is used as a phase to change the evaluation strategy from call-by-name to call-by-value with the addition of thunks and operations for thunks to simulate laziness. The semantics for the thunk operations are quite simple (we ignore clocking for the sake of clarity):

$$\frac{}{[\![\texttt{delay}\ e]\!] = \texttt{Thunk}\ e} \qquad \frac{[\![x]\!] = \texttt{Thunk}\ e \qquad [\![e]\!] = u}{[\![\texttt{force}\ x]\!] = u}$$

Figure 3.1: Simplified derivation rules of `ThunkLang`'s special operators

A `delay` boxes an expression into a thunk and `force` queries a thunk for its expression and evaluates it. However, we would like to have a stronger version of `force` which asserts that the result is not itself a thunk. This provides a very useful invariant that we use in proofs throughout the compilation pipeline. Therefore, we change the semantics of `force` to include that check.

$$\frac{[\![x]\!] = \texttt{Thunk}\ e \qquad [\![e]\!] = u \qquad \nexists t.\ u = \texttt{Thunk}\ t}{[\![\texttt{force}\ x]\!] = u}$$

Figure 3.2: Stronger version of `force`

To see how this small change affects compilation, we need to inspect the `ThunkLang` phase of the compiler closer.

3.1 Adapting `ThunkLang` to the New Semantics

Before the PureCake compiler transforms a `ThunkLang` program into the next intermediate representation `EnvLang`, it first goes through a number of `ThunkLang`-to-`ThunkLang` transformations that perform optimisations on the program. For instance, the simple transformation `force_rel` removes `force`s around variables that are known to already be `force`d. This is useful when we have an expression like `let m = force n in ...`, where any occurrence of `force n` inside is rewritten to just refer to `m`. The transformation that concerns us the most is the `delay_force`

pass, which transforms expressions of the form `delay (force (`e`))` to just e. This transformation in its current form cannot be proved correct with our new assertion. To explain why, we must examine how exactly we prove the correctness of these optimisation passes.

3.1.1 Proving correctness of ThunkLang Optimisations

Formally, a transformation is a pair of binary relations (R_e, R_v). The relation R_e describes the transformation of expressions, and R_v the appropriate transformation of the values resulting from evaluating a transformed expression. Therefore, the main theorem to prove for each transformation is that evaluating related expressions results to related values.

Theorem 3.1.1 (`exp_rel_eval`)**.** *For a transformation* (R_e, R_v) *and any expressions* e_1, e_2*, if* e_1 *and* e_2 *are related under* R_e *and* $eval(e_1)$ *is not an error, then the results of* $eval(e_1)$ *and* $eval(e_2)$ *must be related under* R_v*.*

Then, the whole process of translating a `ThunkLang` program to an `EnvLang` program consists of a chain of transformations $(R_{e_1}, R_{v_1}), \cdots, (R_{e_n}, R_{v_n})$ such that each one satisfies `exp_rel_eval` and that for any `ThunkLang` expression e, there exist `ThunkLang` expressions $e_1, \cdots, e_n$ such that $R_{e_1}(e, e_1) \wedge R_{e_2}(e_1, e_2) \wedge \cdots \wedge R_{e_n}(e_{n-1}, e_n)$. Finally, a transformation $R_{te}(e_n, e_e)$ exists that transforms the last `ThunkLang` expression to an `EnvLang` expression.

On their own, these relations are not strong enough to prove program equivalence, they only prove a relation between the starting and the final program. To prove equivalence from these relations, we show that they are bisimulations [14]. A binary relation is a bisimulation if its two parts behave the same according to some rules. Our relations are over `ThunkLang` expressions and the rules are the evaluation function, and using this theory we can prove that the evaluation of two related expressions cannot be distinguished, therefore making them equivalent.

It is evident that each expression relation must consider every expression on the left hand side that can be generated by its previous relation. Therefore, each relation has boilerplate cases that just carry over each expression unchanged, and some interesting once that apply the transformations. For example, the `delay_force` relation has two interesting cases

$$\frac{v \text{ variable}}{R_e(\texttt{delay (force (}v\texttt{))}, v)} \qquad \frac{R_v(v_1, v_2)}{R_e(\texttt{delay (force (}v_1\texttt{))}, v_2)}$$

but also all the trivial cases such as

$$\frac{R_e(x_1, x_2) \quad R_e(y_1, y_2) \quad R_e(z_1, z_2)}{R_e(\texttt{if } x_1 \texttt{ then } y_1 \texttt{ else } z_1, \texttt{if } x_2 \texttt{ then } y_2 \texttt{ else } z_2)}.$$

We note that transformations like `delay_force` are why Theorem 3.1.1 is relaxed with the error condition for the evaluation of e_1. The reason we allow the first expression in each relation to be an error is because we only care not to generate a

failing program from a correct one, and not the other way around. In the case of `delay_force`, for a variable v that expands to a thunk that fails when forced, the program `delay (force (`v`))` will fail, however the transformed program which is just v may not fail if the result of expanding v is never forced.

Since we added this new assertion in the semantics of `force`, we need to prove that none of the optimisations violates it. For each of these transformations, our change in the semantics now requires a new theorem to be proved:

Theorem 3.1.2 (`v_rel_thunk`)**.** *For a transformation* (R_e, R_v) *and any values* v_1, v_2*, if* v_1 *and* v_2 *are related under* R_v*, then* v_1 *is a thunk value if and only if* v_2 *is also a thunk value*

For all the transformations implemented this was trivial to prove, except for `delay_force`. To see why, we consider what happens when we try to prove the evaluation theorem for the interesting case of $Re($`delay (force (`v_1`))`$, v_2)$ when $R_v(v_1, v_2)$: The expansion of $eval($`delay (force (`v_1`))`$)$ will eventually assert that the result of the inner `force` is not a thunk. However, for v_2 no such assertion exists, therefore **`v_rel_thunk`** is not true and the evaluation proof fails. To explain how we correct `delay_force`, we first need to briefly dive into how PureCake handles side effects.

3.1.2 Side Effects in a Pure Language

So far we have only concerned ourselves with the pure semantics of `ThunkLang`. That is, the semantics of expressions that are perfectly referentially transparent and don't rely on any implicit state. This is the core of pure functional programming, but in any programming language there needs to be a way to interact with the outside world. Any operation that relies on such a state is called a side effect, and examples of side effects include user input and output, global state, exception handling and many others.

The way pure languages handle side effects is by introducing monads to the language [15]. A monad in the context of programming languages is a wrapper around a computation which also produces some additional information about that computation, like an execution trace or non-determinism. PureCake also employs monads to handle side effects. The evaluation of a `ThunkLang` program at the top level looks like a loop over monadic actions that might possibly interact with state, such as writing and reading from a mutable array or writing to the console. Internally, these operations call the usual pure semantics that we have discussed in depth without exposing any of the state that is being handled.

To be able to reason about stateful computation, PureCake uses Interaction Tree Semantics [16]. Interaction trees are a data structure for representing the behaviour of programs that interact with their environments. Interaction trees support the definition of interpreters that handle actions using monads, and bisimulation proofs can be used to show equivalence between interaction trees, similarly with how we described equivalence proofs for program transformations.

For example, we can consider a simple function that interleaves pure computation with side effects

```
foo :: Int -> IO Int
foo x = do
  let y = f x
  print y
  z <- input
  return z
```

When this function is run, the stateful `print` function will be evaluated first, demanding the value of `y` and printing it to the terminal. Then, `input` is used to read an integer from the terminal and unwraps it to the pure value `z`, which is then wrapped in a monadic value by `return` so it matches the return type of the function. This back-and-forth between pure and impure evaluation is modeled by the IO monad, which stands for Input/Output.

Because the results of stateful computations can be used by regular functions, they must be wrapped in thunks. Therefore, any value acquired by `input` or `return` or any other returning monadic action must be delayed. However, these monadic actions can also be called and chained by other monadic actions, where it's not necessary to wrap the results in thunks. Thus, in `ThunkLang` the semantics of the monadic operators themselves are strict, and there is a step during the translation from PureCake's source language to `ThunkLang` that wraps the program's monadic actions that need thunks with `delay`s. For example, whenever there is a call to `input` in the source language, in `ThunkLang` it is translated to `input >>= \v-> return (delay v)`, meaning that instead of directly returning the result of `input`, we first wrap it in a thunk since the monadic interpreter does not do this on its own.

This optimisation is crucial for avoiding the allocation of unnecessary thunks during monadic computations, but complicates the semantics of the language and is the root of our problem with `delay_force` as we will now see.

3.1.3 Fixing `delay_force`

We saw that the interesting cases in `delay_force` are concerned with value and variable expressions. As mentioned in chapter 1, `ThunkLang` uses a substitution based semantics. During `ThunkLang`'s evaluation, encountering a variable expression immediately results in an error. Instead, before evaluating an expression, every free variable in its body is substituted for some value expression. A value expression is just the embedding of a value inside the program, so when the evaluation encounters it, it just returns the value as is.

We can fix the failing proof by making two assumptions:

- Every value expression surrounded by a `force` is a thunk
- Every variable expression is substituted with a value expression of a thunk

These might seem like strong assumptions, but we just have to look at how the

compiler transitions from PureCake's source language to `ThunkLang` to see that this holds, except for one place, the handling of the monadic side effects. Since the compiler optimises away certain `delay`s in the monadic interpreter, this invariant doesn't hold.

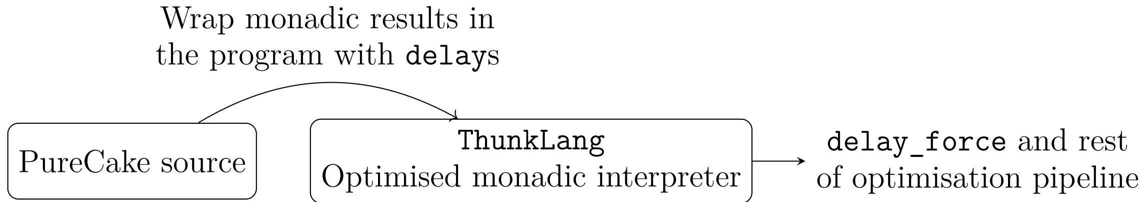

Figure 3.3: Translation of monadic expressions from PureCake source to `ThunkLang` prior to this work

To mitigate this issue we carried out a 4-step plan:

- We implemented an alternative unoptimised monadic interpreter that adds `delay`s around the results of all the monadic actions.
- Then, we removed the wrapping of monadic results during the PureCake source to `ThunkLang` pass to avoid duplicated `delay`s.
- Next a transformation from the unoptimised monadic interpreter to the optimised monadic interpreter was implemented by adding back the wrapping of monadic results in `delay`s.
- Lastly, we moved `delay_force` before the transformation to the optimised interpreter happens.

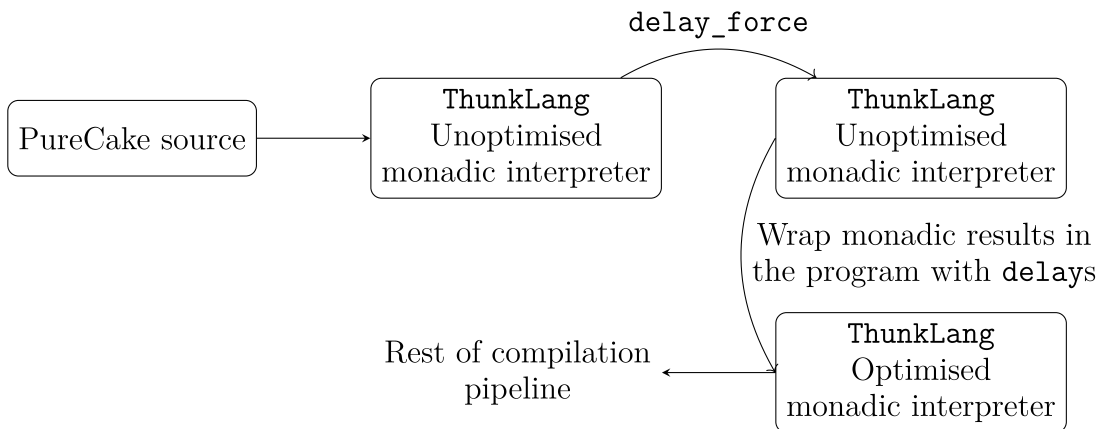

Figure 3.4: Adapted translation of monadic expressions from PureCake source to `ThunkLang`

With this translation scheme, we were able to carry on our invariant that the result of `force` never returns a thunk without sacrificing any optimisations. This also serves as a prime example of the usefulness of formal verification in compiler construction. All this rearranging of fragile parts would be very difficult to achieve correctly and optimally in a conventional compiler. Since we implement everything in a theorem

prover we can be sure that we achieved our goal and didn't lose any optimisations in the process.

3.2 The Stateful Thunk Semantics in `StateLang`

The next intermediate representation from `ThunkLang` is `EnvLang`. This language doesn't transform thunks or their operations in any way, so we just had to add the semantic check after the `force` and everything else stayed the same. Therefore we can immediately move on to the more interesting `StateLang`.

3.2.1 `StateLang`'s Stateful Semantics

`StateLang` is the last intermediate representation in the PureCake pipeline, and the boundary between PureCake and CakeML. It is also the only intermediate representation with a stateful semantics which means it can represent stateful thunks. Before `StateLang`, thunks are stateless; when evaluating an expression like `(force f) (delay ((force g) x))`, the argument `g x` is turned into a thunk value and whenever the corresponding parameter is used in the body of f, the expression `g x` is computed anew without the possibility of memoising the result, as that would require some stateful operation.

`StateLang`'s semantics provide a solution for this problem by employing a mutable state. This state is a store for mutable arrays, where each array element can be mutated in place. `StateLang` also provides primitive operations for manipulating arrays, namely `Alloc` for allocating an array, `Sub` for dereferencing an array element and `Update` for updating the value of an array element.

`StateLang` uses a continuation-based small-step semantics [17], in contrast to `ThunkLang`'s big step semantics. In small-step semantics, the semantic function does not walk the expression structure recursively. Instead, it breaks down the evaluation process into atomic steps, and takes one step at a time, returning the rest of the computation as a result. Concretely, the evaluation function takes a state, a clock, and an expression and returns one of three results: An expression, a value, or an error. In actuality, there are two more results, namely exception and action values for performing cross-language calls but they are not important for our treatment of the semantics. Along with the result, the `step` function returns a new state and a continuation stack. The continuation stack is a list of remaining operations waiting to be executed in the rest of the steps. As an example, let's consider how `StateLang`'s semantics handle application expressions. The (simplified) `step` function looks like this:

```
step st k (Exp env (App op (e:es))) =
  if not (num_args_ok op (length (e:es)))
    then error st k
    else push env e st (AppK env op [] es) k
```

We see that first the function checks if the number of arguments is the correct for the given operation. In `StateLang`, the term "application operator" is used to refer to

many things, one of them being a function application, but for example another one being the "application" of the `Sub` operator to an index and an array. So, assuming that we are talking about function application for this example, the `step` function should have received two expressions else it will return an error. If the number of arguments is correct, the function calls `push`, which just constructs a result consisting of `e` as the expression, `st` as the state and the continuation stack `k` prepended with the `AppK` continuation.

Intuitively, the `step` function takes away arguments from the application expression one by one, evaluates them and has stored a "promise" in the form of a continuation that it will eventually apply the operator to the resulting values. When all the operands have been evaluated, the `AppK` continuation will come to be handled, which will apply the operator to the list of values produced by the evaluation of the arguments. In the case of the application operator, it will check the head in the list of values to make sure it is a closure; then it will bind the second value in the list to the closure's environment, and finally it will return the closure's expression with the new environment for the `step` function to start evaluating it.

We can now explain the semantics of the stateful operators `Alloc`, `Sub` and `Update`. These operators are all application operators, therefore their evaluation will take a similar course with the application operator. When applied to the list of values, `Alloc` expects two values: a number n and an arbitrary value v. The `application` function then will append a new array of length n to the state and fill it with n occurrences of v. It will then return the length of the state as the index of the allocated array, assuming that accesses to the state are zero-based. The `Sub` operator also expects two values: an index n to select an array from the state, and an index m to select an element from the nth array of the state. The `application` function returns the selected element and leave the state intact. Lastly, the `Update` operator expects three values: two indices to select an element like `Sub`, and a third value to replace that element. The `application` function will update that element in the state with the third value, and return the unit constructor.

3.2.2 The Dual Representation of Thunks in `StateLang`

Thunks in `StateLang` exist in two forms: stateless and stateful. The stateless thunks are the same as in the previous intermediate representations, and exist so that compilation from `EnvLang` can directly target them. Prior to this work, `StateLang`'s store is used to implement stateful thunks in an ad-hoc way. Instead of having a proper representation for thunks, a thunk is represented as a two value array: the first value is a boolean flag and the second value is the actual value. When a thunk is needed, the array is allocated with the flag set to signal that the thunk is not yet evaluated, and the second value being a closure that when applied to unit computes the thunk's value. When a `force` encounters such a value, it checks the flag in the first place of the array and, if it is set it immediately returns the value in the second place. If it is not set, it applies the second value to unit, replaces the function with the value, sets the flag and finally returns the value. To make these operations more efficient, there are dual unsafe operators of `Sub` and `Update` called

UnsafeSub and UnsafeUpdate respectively. Since the semantics know the shape the array argument of a force will have, there is no need to check if the two indices are in range. The transition from the stateless thunks of ThunkLang to the stateful thunks of StateLang is done in a StateLang-to-StateLang transformation called unthunking. During StateLang's unthunking, every delay and force operation is replaced with the appropriate stateful operations. The generated code for force of a value stored in variable v prior to this work looked like this:

```
let v1 = unsafeSub v 0
let v2 = unsafeSub v 1
if v1 then v2 else
  let wh = v2 ()
  let _ = unsafeUpdate v 0 True
  let _ = unsafeUpdate v 1 wh
  return wh
```

This code mimics the desired behaviour, but results in the semantics beign hidden as a code generation in a proof instead of being explicit in the semantics of the language. With the addition of thunks to the language this whole construct compiles to just one stateful operation and the semantics becomes a lot clearer and robust as we will now see.

3.2.3 Making Thunks Explicit in StateLang

The way StateLang's store is implemented is not arbitrary; it has the same form as CakeML's store. However, both of them need to change to make the handling of thunks explicit. Therefore, we first change the type of StateLang's store from [[Value]] (an array of arrays of values) to [StoreV], where StoreV is defined as

```
data StoreV
  = Array [Value]
  | ThunkMem ThunkMode Value
```

and ThunkMode is defined as

```
data ThunkMode
  = Evaluated
  | NotEvaluated
```

Three new application operators were also added, namely AllocMutThunk, for appending a new mutable thunk in the store, UpdateMutThunk for updating a thunk on the store, and ForceMutThunk to replace the stateless force operation. The previous unsafe array operators were removed completely, as they were only used to work with the ad-hoc thunk arrays. Also, a new continuation ForceMutK was added, whose use will become clear with the explanation of the operators' semantics.

The semantics of AllocMutThunk and UpdateMutThunk are straightforward and are used as building blocks by the other operations. If v is a value, st is the store and k is the continuation stack, evaluating AllocMutThunk consists of returning an

updated store with the new thunk appended at the end, and the length of the stores as a `Value` result, which is used as an index of the thunk in the store.

```
application (AllocMutThunk mode) [v] st k =
  (Value (length st), st ++ [ThunkMem mode v], k)
```

Similarly, the `UpdateMutThunk` operator expects two values an index in the store and a value and updates the store at that index with a new thunk, after checking that the index is in bounds and that the current value at that index is an unevaluated thunk.

```
application (UpdateMutThunk mode) [i,v] st k =
  if i < length st && Thunk NotEvaluated _ = st[i] then
    (Value Unit, update st i (ThunkMem mode v), k)
  else Error
```

We notice that we assert in the semantics that an evaluated thunk will never be updated. This is important, as this invariant will be carried along in CakeML and help us implement optimisations which need to know that once a thunk is evaluated its value will never change.

More interestingly, the mutable force evaluation first checks that the index is in bounds of the store and then checks the value of the store. If the index does not point to a thunk it's trivially an error. If the value is an evaluated thunk, we just return the stored value. Lastly, if the value is an unevaluated thunk, we first want to apply the stored closure to `Unit` to evalaute the thunk and then update the thunk in the store.

```
application ForceMutThunk [i] st k =
  if i >= length st then Error else
  case st[i] of
    ThunkMem Evaluated v -> (Value v, st, k)
    ThunkMem NotEvaluated f ->
      application App [f, Unit] st (ForceMutK i :: k)
    _ -> Error
```

Because of the small-step semantics this cannot be done directly in the same function. First, the evaluation must take enough steps that the application of `f` to `Unit` will complete, and this is why we push a `ForceMutK` continuation which will be evaluated by the `return` function which pops a continuation off the stack on the return of an application and evaluates it with the result of the application as the first argument `v`:

```
return v st (ForceMutK i :: k) =
  if i >= length st || (is_thunk v) then Error else
  case st[i] of
    ThunkMem NotEvaluated _ ->
      (Value v, update st i (ThunkMem Evaluated v), k)
  _ -> Error
```

We can see that it takes the result of the application and updates the store with an evaluated thunk storing the value. We also note the `is_thunk` check, which is the

same assertion we have added to all the `force` semantics from `ThunkLang` onward.

This is the complete semantics for the stateful thunk operations in `StateLang`, but there is a subtle detail that has not been explained. We saw that **`ForceMutThunk`** and **`ForceMutK`** work in pairs; the former initiates the evaluation of a thunk and the latter stores its result. They also work with the same index in the store, therefore with the same thunk. However, they both seem to check that the thunk stored is not evaluated before proceeding. This seems redundant for **`ForceMutK`** to do, but there is a special case which we will examine now that prevents **`ForceMutK`** from assuming the thunk is still not evaluated.

3.2.4 Recursive forcing of Thunks

To explain this problem we consider the simple recursive declaration `x = x + 1`. When `x` is created, an unevaluated thunk is allocated in the store with the expression `x + 1`. When `force` is called on `x`, the `+` operator will again cause a `force` of `x`

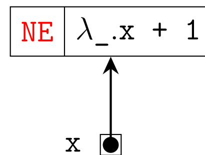

Figure 3.5: Beginning of thunk evaluation

leading to two evaluations trying to update the same thunk in the store. If one of

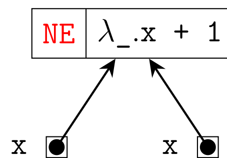

Figure 3.6: Thunk evaluation conflict

these computations were to finish, the other would find an evaluated thunk in the store and try to update it which is undesirable.

We observe here that once this situation arises, it is impossible for one of the computations to terminate, because each `force` would have to wait for the next `force` to finish, which essentially creates an infinite loop. This is also evident from the program, since the value of `x` depends directly on the `force`d value of itself. Therefore, this condition is not problematic on its own since the computation will never finish and the thunk will never be evaluated, but the behaviour must nevertheless be modelled so the semantics can be proved equivalent.

This is reminiscent of the black hole problem in Haskell [18]. In Haskell, the runtime representation of thunks does not have a binary flag, evaluated and not evaluated, as we have described for our thunks. Instead, it has three possible values: evaluated, not evaluated and evaluating. When a thunk starts getting evaluated, the evaluating flag is set. Then, if a `force` tries to update an evaluating thunk, the runtime system

detects that as an infinite loop and throws an exception. However, in our semantics we prove that if we try to force an evaluating thunk we will eventually diverge, so we only have to handle the evaluating and not evaluating cases in the rest of our proofs and in the runtime.

To prove this situation is impossible to arise, we turn to the unthunking pass of `StateLang` where, as mentioned earlier, the stateless thunks are compiled to stateful ones. The `step` function that implements the small-step semantics evaluates the stateless thunks in a similar way to the stateful semantics we described. When the `step` function encounters a `force`, it evaluates the enclosed expression and pushes a `ForceK` continuation to the stack. Then, the `return` function pops the continuation, asserts that the result is not a thunk and returns the computed value.

Our solution to the problem was to create wrapper functions `step'` and `return'` around `step` and `return`, that remember which thunks are currently in the process of evaluation.

```
return' avoid v st (ForceK :: k) =
  if member v avoid then Error else return v st (ForceK :: k)

step' avoid st k (Value v) = return' avoid v st k
step' avoid st k x = step st k x
```

These wrapper functions essentially intercept the evaluation of `force`s and check for each one if the thunk they are trying to evaluate is in the set of currently evaluating thunks, called the *avoid* set. We then prove the following theorem:

Theorem 3.2.1 (`add_to_avoid`)**.** *For any unevaluated stateless thunk t, if computing the result of its value with set avoid terminates successfully after m steps, then the computation terminates with the same result after the same number of steps with set $avoid \cup \{t\}$.*

Proof outline. If $t \in avoid$ then it's trivial since $avoid \cup \{t\} = avoid$. If $t \notin avoid$ then we distinguish two cases:

- If there exists $n < m$ such that in the n'th step we call `force` on t then our theorem would not hold, because adding t to *avoid* would trigger an error at this step. However, if such n existed, then the result of $n + 1$ steps would be the same expression as we started with, namely the evaluation of the thunk expression. Thus, every n steps the computation loops, which leads to contradiction because we started with the assumption that the evaluation of the thunk terminates after m steps.
- On the other hand, if no such n exists, the `step'` function does not encounter a `force` for t, and therefore it is safe to add it to *avoid* without producing any errors that the wrapped `step` function didn't produce before.

□

Finally, we modified the unthunking proof to use the `step'` semantics and proved the

equivalence of the stateless and stateful semantics, bypassing the non-termination problem by using `add_to_avoid`.

3.3 Summary

To summarise, this chapter was concerned with making the semantics of forcing stricter in PureCake, so the equivalent semantics in CakeML can have this strict behaviour. We described the changes to the semantics and optimisation pipelines throughout the PureCake project, and gave a detailed account of the challenges that arose naturally due to the strengthening of the semantics. We also outlined our solutions and proof processes we used to overcome these obstacles and verify our semantics are correct. We will now move over to the CakeML side of the project and explore the changes we had to implement there to complete our work.

4

Adding Thunks to CakeML

Moving downwards on the compilation pipeline, this chapter describes our work for the last part of the thesis, adding thunks to CakeML. The CakeML compiler was not built for a lazy language, so it rightly doesn't consider thunks natively. However, when considered as a target for PureCake, it is critical that it can handle thunks effectively, since they are the main source of inefficiency between lazy and strict programs.

4.1 The Semantics of CakeML

The CakeML source semantics is defined in functional big-step style like PureCake source semantics. Its functional big step semantics is easy to understand at a high level, but is not immediately appropriate for compilation from PureCake. For this reason, an alternative semantics exists based on Interaction Trees, that PureCake can directly target to compile its stateful operations.

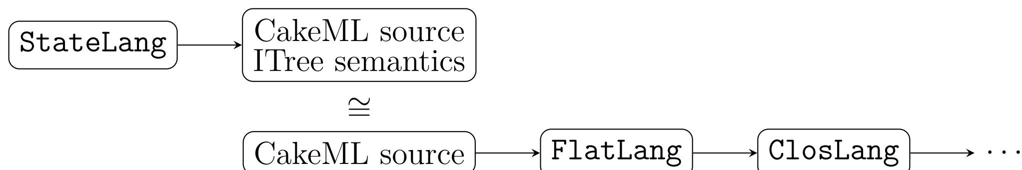

Figure 4.1: Compiler structure after `StateLang`

The CakeML compiler has many intermediate languages, each one serving the purpose of performing some transformations or optimisations, similar to how the PureCake compiler is structured. CakeML's source language has a big step semantics like `ThunkLang`, but also a state like `StateLang`, so we need yet another semantics for our operations to adapt to this model.

Similarly to our `StateLang` changes, we define three stateful thunk operations **`AllocThunk`**, **`UpdateThunk`**, and **`ForceThunk`**. A **`Thunk`** constructor is also added to the mutable store, as we did in `StateLang`, with an **`Evaluated`** or **`NotEvaluated`** flag. We do not need stateless thunks here as we did in `StateLang`, as they were merely a stepping stone between the semantics of `EnvLang` which only has stateless thunks and CakeML which we only wanted to have stateful thunks. More explicitly, thunks in CakeML are only added as targetable operations when constructing a CakeML program as a syntax tree and not by pasing concrete syntax.

We now describe the semantics of our operations in CakeML. This semantics stays the same between most of the intermediate languages. `AllocThunk` is constructed with a thunk mode flag (`Evaluated`/`NotEvaluated`), denoted with `m` in the snippet, and accepts a single value, appends it to the state and returns the updated state and a reference to the newly allocated thunk as a result. The `False` flag in the reference states that equality of references for this type should be done by index and not structurally. Two thunks are the same if and only if they refer to the same exact thunk in the store. CakeML's store supports other types such as byte arrays that can be compared by value.

```
eval (AllocThunk m) st [v] =
  let st' = st ++ [Thunk m v] in
  (st', Ref False (length st))
```

Similarly to `StateLang`, `UpdateThunk` is also constructed with the thunk mode flag, and accepts an index to the store and a value, checks if the reference is an unevaluated thunk and returns the updated store and `Unit` as the result.

```
eval (UpdateThunk m) st [Ref _ i, v] =
  if i >= length st then Error else
  case st[i] of
    Thunk NotEvaluated _ ->
      (update st i (Thunk Evaluated v), Unit)
    _ -> Error
```

The remaining operator, `Force`, is more contrived. We need to make sure that all the invariants we proved correct during the PureCake compilation are present here too. First, we define a function that decides whether a reference in the store is a thunk or not:

```
dest_thunk [Ref _ i] st =
  if i >= length st then BadRef else
  case st[i] of
    Thunk Evaluated v -> IsThunk Evaluated v
    Thunk NotEvaluated f -> IsThunk NotEvaluated f
    _ -> NotThunk
```

We discriminate between three possible results: `BadRef` when the reference is out of bounds, `NotThunk` when the reference points to something other than a thunk, and `IsThunk` which returns the thunk's mode and value. The reason we have different failing modes for out of bounds accesses and references to wrong values will be explained shortly. Then, we define a function that updates a thunk:

```
update_thunk [Ref _ i] st [v] =
  case dest_thunk [v] st of
    NotThunk -> Just (update st i (Thunk Evaluated v))
    _ -> Nothing
```

Finally, the semantics of `ForceThunk` are implemented using these helper functions. First, we check that the input value is a thunk. Then, if it's an evaluated thunk we

return the value. Otherwise we apply the unevaluated function to **Unit** to evaluate it, and check the result: if it's a thunk, then it's an error because force mustn't result in a thunk, otherwise we return the result with the updated state.

```
eval ForceThunk st vs =
  case dest_thunk vs st of
    BadRef | NotThunk -> Error
    IsThunk Evaluated res -> (st, res)
    IsThunk NotEvaluated f ->
      case eval st [App f Unit] of
        Error -> Error
        (st', res) -> case update_thunk vs st' res of
                        Nothing -> Error
                        Just st'' -> (st'', res)
```

The semantics of forcing stateful thunks here is the most important code sample in this thesis, as it captures exactly our desired semantics.

We will not describe the semantics of these operations in the rest of the CakeML's intermediate languages since they are very similar, but we will discuss some cases that needed extra attention during the transformations.

4.2 Carrying the Semantics Through CakeML

Through the rest of the compilation from CakeML's source to machine code lots of optimisations and transformations take place. The methods used to prove these transformations is akin to the ones we described for PureCake. In this section, we will discuss two families of transformations that required special attention to prove correct for our semantics.

4.2.1 Transforming the State

We described the PureCake transformations as a pair of relations R_e and R_v over the expressions and values of the language. Since PureCake also has a state, we also need a third relation R_s that describes how a transformation translates states. This R_s is usually straightforward. For index i and states s_1 and s_2, $R_s(s_1, s_2)$ is true if and only if i is in bounds for both states, and the store values at the index are related. In the case of mutable thunks `Thunk m v` and `Thunk m' v'`, they are related if and only if $\mathtt{m} = \mathtt{m'}$ and $R_v(\mathtt{v}, \mathtt{v'})$.

However, one proof shifts the whole state by a certain amount to prepend some fixed state to the start of the global store. Therefore, the state relation R_s has an extra argument b that maps indices before the transformation to their corresponding indices after the transformation. For example, if index i has stored a `Thunk m v`, then the related thunk after the transformation will not be at index i, but rather at index $b(i)$. Importantly, b is an injective function, meaning that it maps distinct elements from its domain (the indices of s_1) to its codomain (the shifted indices of s_2). What b does not concern itself with is indices that are out of bounds in s_1,

because it doesn't make sense to map an invalid index to anything in the transformed array.

This is exactly the case that forced us to distinguish between failures in `dest_thunk`. Remember from the semantics that `update_thunk` succeeds only when `dest_thunk` fails with a **`NotThunk`** result. If the queried index is out of bounds, then technically it is not a thunk and `update_thunk` could store the invalid reference in an evaluated thunk. However, this would make this proof impossible, since if `update_thunk` succeeded for state s_1 and out of bounds index i, we would need to show that it also succeeds for the transformed state s_2 at index $b(i)$. But i is not in the domain of indices of s_1, so b does not provide any information about where i is mapped.

4.2.2 Closure Conversion

At one point in the compilation of CakeML, closure conversion [19] is performed. This transformation achieves the separation between code and data. Whereas closures carry both an environment and an expression to evaluate, after closure conversion all functions are static and the environment is explicitly passed by the caller of the function. Another important optimisation performed is compilation to multi-argument functions. In functional programming languages, functions usually accept only one argument, and functions with multiple arguments are simulated through tuples or currying. However, both of these simulations require unnecessary allocations during runtime. Therefore, the compiler tries to group curried arguments and construct calls to apply them all at once.

This becomes problematic for our force semantics. When we extract an unevaluated thunk from the store to evaluate its closure, we apply it to a unit argument. However, we don't know prior to extracting the thunk if the enclosed value was a simple closure, or if it has been merged with other curried arguments to make a multi-argument function. Therefore, during the closure conversion pass we generate a piece of code replacing the simple application to unit that performs this check.

In the simplest case, the call has not been merged with any other curried arguments. Then, a call instruction is generated that enters the code in the closure with a unit argument. On the other had, if the closure expects more arguments then a different call instruction is generated that will allocate a partial application closure or perform repeated applications to accommodate the extra arguments.

4.3 Summary

Summarising, in this chapter we discussed our addition of thunks to the CakeML intermediate languages and how transformations that deal with the state and the calling convention of the languages interacted with our semantics and how we dealt with them. Some work is still needed to complete this part of our work, as calling functions in the lowest level language in CakeML has a very intricate semantics and we are working out how it is best to compile down to it, since calls to unevaluated thunks can become complicated as we saw in the previous section.

5

Conclusion

The main results of our work can be summarised as:

- Developing a logical relation for `ThunkLang`.
- Strengthening the semantics of `force` in PureCake.
- Adding thunks to CakeML.

We believe this work will help the PureCake project both in usability and maintainability, by providing the logical relation core for developing its proofs, as well as by better specifying the semantics of working with thunks, which is the construct at the center of laziness, and whose optimal handling is essential for a lazy language to be efficient. Furthermore, the necessity of this project emerged from the inability of the project's state prior to our work to support the development of critical optimisations which are now possible.

5.1 Future Work

Here, we describe how we plan to continue and extend this project, and we propose ways the PureCake project can use our work to bridge the gap between the efficiency of its generated executables and the efficiency of executable's generated by mainstream compilers.

5.1.1 Project Extensions

Firstly, we plan on extending this project by making the logical relation practical. By examining the `ThunkLang` transformations we can develop an ecosystem of definitions and theorems around the logical relation that all the proofs can use to factor out common patterns. Then, we would like to re-write some of the transformations using this relation and compare the results to the way they are currently implemented. If the consensus between the PureCake developers is that this is indeed a more composable and maintainable way of developing transformations, then we can progressively convert all transformations to make use of it, and even develop logical relations for the other intermediate languages of PureCake.

On the side of CakeML, we still have to sort out the exact way we will make the call to an unevaluated thunk's contained function, and maybe try to develop some

optimisations in this regard.

5.1.2 Runtime Thunk Inlining

The most critical optimisation this work allows the compiler to make, is an optimisation called updating [20].

CakeML, like every high-level functional language, has a garbage collected runtime. Each time a certain threshold of allocated memory is surpassed, the program stops running and the garbage collector traverses the memory starting from known used locations called roots, copying the reachable data into a new heap and disposing of the old one. The details of garbage collection in CakeML are described in Sandberg et al. [21], but this mental model is enough for our description here.

Having thunks implemented explicitly in the CakeML source language, gives the ability to the garbage collector to discriminate between chunks of memory that represent thunks, and tweak its behaviour when encountering them.

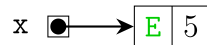

Figure 5.1: Evaluated thunk before optimisation

To illustrate the idea of the optimisation, suppose we have a thunk that has evaluated to the value 5 stored in variable location x. The memory block at x will hold a pointer to the thunk block, which will hold the actual thunk value. Since we now know that the two cells at x points to are a thunk and not just an array, we can instruct the garbage collector to replace the thunk with the evaluated value when it encounters it. When the garbage collector is run, instead of copying both values to the second heap, we can instead drop the thunk value and replace the pointer in x with the result directly.

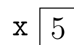

Figure 5.2: Evaluated thunk after garbage collection

Going back to the data structure example in chapter 1, this optimisation would turn a fully evaluated lazy list (Figure 5.3) into a strict list by removing the two layers of evaluated thunks and inlining their values (Figure 5.4).

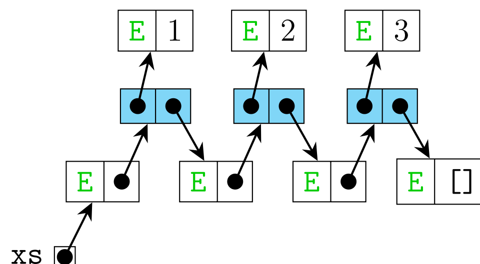

Figure 5.3: Memory layout after evaluating `sum [1,2,3]`

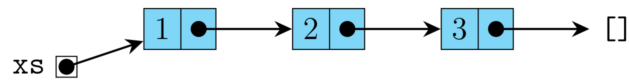

Figure 5.4: Memory layout after GC has run and inlined the evaluated thunks

This optimisation is crucial for both the memory and time efficiency of the program. When an evaluated thunk is in memory two words of memory are wasted, the one holding the pointer to the thunk and the thunk tag. Getting rid of these makes for more compact heaps that help the runtime efficiency of the program. Furthermore, having the value directly available results in one less indirection that the execution of the program has to traverse each time the value is demanded, making accesses to it have the same performance characteristics as if it were unboxed to begin with.

Bibliography

[1] P. Bjesse, "What is formal verification?" *ACM Sigda Newsletter*, vol. 35, no. 24, 1–es, 2005.

[2] K. Slind and M. Norrish, "A brief overview of hol4," in *International Conference on Theorem Proving in Higher Order Logics*, Springer, 2008, pp. 28–32.

[3] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782. DOI: `10.1145/1538788.1538814`. [Online]. Available: `https://doi.org/10.1145/1538788.1538814`.

[4] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Principles of Programming Languages (POPL)*, ACM Press, Jan. 2014, pp. 179–191. DOI: `10.1145/2535838.2535841`. [Online]. Available: `https://cakeml.org/popl14.pdf`.

[5] H. Kanabar, S. Vivien, O. Abrahamsson, *et al.*, "PureCake: A verified compiler for a lazy functional language," in *Programming Language Design and Implementation (PLDI)*, ACM, 2023. [Online]. Available: `pldi23-purecake.pdf`.

[6] I. Sergey, S. P. Jones, and D. Vytiniotis, "Theory and practice of demand analysis in haskell," *Unpublished draft, June*, 2014.

[7] W. D. Clinger, "Proper tail recursion and space efficiency," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998, pp. 174–185.

[8] A. Gill and G. Hutton, "The worker/wrapper transformation (extended version)," *J. Funct. Program.*, vol. 19, Jan. 2009.

[9] G. Kahn, "Natural semantics," in *Annual symposium on theoretical aspects of computer science*, Springer, 1987, pp. 22–39.

[10] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, "Functional big-step semantics," in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*, Springer, 2016, pp. 589–615.

[11] D. Dreyer, A. Ahmed, and L. Birkedal, "Logical step-indexed logical relations," *Logical Methods in Computer Science*, vol. 7, 2011.

[12] Z. Paraskevopoulou, J. M. Li, and A. W. Appel, "Compositional optimizations for certicoq," *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, pp. 1–30, 2021.

[13] R. Kumar and M. O. Myreen, "Clocked definitions in hol," *arXiv preprint arXiv:1803.03417*, 2018.

[14] D. J. Howe, "Proving congruence of bisimulation in functional programming languages," *Information and Computation*, vol. 124, no. 2, pp. 103–112, 1996.

[15] N. Benton, J. Hughes, and E. Moggi, "Monads and effects," in *International Summer School on Applied Semantics*, Springer, 2000, pp. 42–122.

[16] L.-y. Xia, Y. Zakowski, P. He, *et al.*, "Interaction trees: Representing recursive and impure programs in coq," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[17] K. Nakata and M. Hasegawa, "Small-step and big-step semantics for call-by-need," *Journal of Functional Programming*, vol. 19, no. 6, pp. 699–722, 2009.

[18] J. H. van Groningen, "Optimising recursive functions yielding multiple results in tuples in a lazy functional language," in *Symposium on Implementation and Application of Functional Languages*, Springer, 1999, pp. 59–76.

[19] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the ACM annual conference-Volume 2*, 1972, pp. 717–740.

[20] J. Seward, "Generational garbage collection for lazy graph reduction," in *Memory Management: International Workshop IWMM 92 St. Malo, France, September 17–19, 1992 Proceedings*, Springer, 1992, pp. 200–217.

[21] A. Sandberg Ericsson, M. O. Myreen, and J. Åman Pohjola, "A verified generational garbage collector for cakeml," *Journal of Automated Reasoning*, vol. 63, pp. 463–488, 2019.