CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

PureCake: Towards a formally verified non-strict language compiler

Master's thesis in Computer science and engineering

RICCARDO ZANETTI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

PureCake: Towards a formally verified non-strict language compiler

RICCARDO ZANETTI

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

PureCake: Towards a formally verified non-strict language compiler

RICCARDO ZANETTI

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: John Hughes, Department of Computer Science and Engineering

Master's Thesis 2021
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 2021

PureCake: Towards a formally verified non-strict language compiler

RICCARDO ZANETTI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

CakeML is an end-to-end formally verified compiler for a subset of the Standard ML language. Its provenly correct compilation chain prevents several classes of software bugs from seeping into the project, while drastically reducing the chances of many others to occur. In turn, other software projects can be built on top of CakeML; their correctness can be proven with respect to CakeML semantics, and ultimately produce software that is itself end-to-end verified.

The aim of this thesis is to bring a pure non-strict language variant (PureCake) into the project. The peculiar properties of the language, which include purity, referential transparency and equational reasoning facilitate the proof process, ultimately encouraging the spread of formal methods.

Keywords: compilers, formal verification, interactive theorem provers, functional programming, formal semantics of programming languages, code optimizations, inlining.

Acknowledgements

I would like to thank my supervisor Magnus for the chance he gave me: working with him has been a gratifying and insightful experience that I will always treasure.

I would also like to thank my examiner John Hughes for the valuable feedback provided and the rest of the PureCake team: Oskar Abrahamsson, Hrutvik Kanabar, Michael Norrish and Johannes Åman Pohjola. Without their contribution this thesis would have not been possible.

Finally, as I approach the end of my journey at Chalmers, I realize the great opportunity I have been given: the competence and expertise of the department exceeded my wildest expectations and I am grateful to all the passionate teachers I have found on my way.

Riccardo Zanetti, Gothenburg, February 2021

Contents

1

Introduction

Formal methods is a mathematical discipline that aims to *(i)* define rigorous specifications under which to develop software *(ii)* verify the compliance of such programs to the given specifications [16]. Recently, advancements in computer's hardware, better SMT/SAT solver algorithms [17], and more comprehensive infrastructures made advanced techniques in formal methods accessible for larger programs [7].

As a result, more and more projects flourished under the rigorous setting of this methodology [21, 18]. The employment of formal methods has been shown to substantially increase the overall quality of the products, to an extent unrivaled by any other technique [46]. CakeML [1] is one of those projects. It provides (among other things) a proven-correct compiler for a subset of Standard ML.

Despite its efficacy, a series of drawbacks stand in the way of the spread of formal methods in software development. Proving the correctness of software artifacts with respect to a formal specification is a time consuming task, that often exceeds the time dedicated to the code development. Also, formal methods require specialized knowledge in order to make a profitable use of them.

From this awareness emerges the PureCake [2] project which aims to produce a provenly correct compiler for a pure, non-strict language variant of CakeML. The spirit of the PureCake project is explorative and the intent is dual: to better understand how and to which extent non-strict functional languages fit into a rigorous formal setting and, on the other side, what opportunities their new semantics brings in the formal verification of programming languages, with an emphasis on ease of use.

This thesis is divided into two parts. The first part describes implementation and design choices for PureCake, with an emphasis on the aspects that contribute to the final proving process for the language.

The second part focuses on the practical verification of a specific class of properties: *code equalities*. As we will see, code equalities cover a major role in the code optimization process of the language. We will argue why PureCake is particularly well suited for the task and show the lemmas and definitions we developed for this class of proofs.

[1] `https://cakeml.org`
[2] `https://github.com/CakeML/pure/`

1.1 Motivation

The scope of this work spans over two different fields and brings together distinct objectives. Firstly, the work contributes to the implementation of the language PureCake, and serves the interests of the PureCake development team: to implement a provenly correct non-strict language compiler with a simple proof reasoning infrastructure. Secondly, this work sets the groundwork for an important code optimization within the context of a end-to-end provenly correct compiler, ultimately suggesting for additional research in the field. We will explore those two distinct aspects in order.

1.1.1 The PureCake language

Research over rigorous formalization of non-strict languages is abundant [4], but a review of the literature suggests that no verified end-to-end compiler has ever been produced. Stelle and Stefanovic recently formalized a call-by-need semantics in Coq [37]: the language is proven to be correct over a small abstract machine (Cactus Environment Machine [38]), but, as pointed out in their work, the proven compilation chain does not reach a concrete hardware architecture (as it is the case for CakeML). The novelty of our attempt makes it academically relevant and constitutes a first motivation for the project.

Moreover, as will be discussed in depth in Chapters 2 and 3, the type of the semantics adopted to define the language plays a major role over the final complexity of the system (hence, its proof-derivation endeavour). Denotational semantics, thanks to its abstractness, is remarkably suited for the task, and we consistently lean towards it. Non-strict evaluation is also very mathematically-oriented and, as we show later on, it blends naturally in the denotational framework. This is not entirely the case for strict evaluation, which has some operational residues that need to be addressed (Section 3.2).

Apart from semantics, PureCake features a series of properties that also contribute to the simplification of the language, like purity and referential transparency. The former distinguishes program's observable behavior from the strategy in which computation is performed, promoting proofs compositionality and reuse, the latter allows each definition in the code to be freely inlined without risking to alter the program's final outcome, allowing the programmer to re-shape a piece of code to more convenient forms while proving properties about it.

1.1.2 Provenly correct code rewriting

Code optimizations in compilers often rely on non-trivial correctness arguments and are, hence, likely to hide bugs [31].

Testing is also often not a satisfactory option: in order to be effective a compiler

needs to combine several optimizations together. While testing the correctness of a single code transformation might seem feasible, with unit testing for example, testing the correctness of several transformations together is much harder: the application of a code transformation often exposes the chance for other transformations to take place. In the end, the final expected result of several optimizations together is hard to determine by hand, making examples-based testing unfeasible.

The unfeasibility of examples-based testing lead to research in other forms of testing, for example property based ones. Even though more effective for our problem, this class of testing requires the definition of a generator for input programs, a far from trivial task. Addressing this challenge, Pałka et al. [29] developed a generator for a rather small subset of the Haskell language (simply typed lambda calculus), that was nonetheless sufficient to expose bugs in GHC.

The difficulties that seem to arise from testing the correctness of compiler optimizations suggest formal verification could be an effective alternative for the task. The PureCake project is a good fit for this attempt: its formal setting allows us to push the extent of the optimizations even further, without worrying about potential bugs.

Among all the possible optimizations that can be implemented, one stands out for simplicity and efficacy: *code rewriting*, which is a concept that covers all source-to-source code transformations that can be expressed in terms of *rewrite rules*, i.e. equalities that relates two expressions having the same meaning.

Thanks to *referential transparency*, pure, non-strict programming languages (like PureCake) are particularly amenable to code rewriting. Several important optimizations can be stated in terms of rewrite rules, including: inlining, function specialization, let-floating and list fusion [23]. In particular, inlining has been referred to as the most important code optimization implemented in the Glasgow Haskell Compiler (GHC), because of its impact over the final program speed and its ability to reveal other important optimization instances [31].

Finally, interactive theorem provers (like HOL) and compilers that perform code rewriting (like GHC) share the same operational mechanics: given a term/program and an equation that relates two entities in an expression, find occurrences of one side and replace them with the other one. HOL uses those equalities to prove theorems, while GHC makes use of those equalities to improve the speed of programs. Recently, Breitner [5] showed how to make use of the GHC rewrite engine to prove small code equations in Haskell, suggesting the proximity of those two fields might hide even more promising relations.

1.2 Contributions

This thesis brings:

- Contributions to the PureCake project, both in design and practical implementation.

- A set of theorems and lemmas that support the proof of source-to-source code transformations for the PureCake language, with the ultimate aim to implement an optimization pass.

The definitions and proofs presented in this thesis are the outcome of efforts by the whole PureCake development team, which I have been part of during the course of this Master's thesis.

In this document I present the main concepts and ideas of the project, supporting them with comments and reflections. All the definitions and proofs presented should not be attributed to me personally, but to the whole development team, to which I will refer to as "we" throughout the thesis.

For reference, the main contributions over the project that can be attributed completely to me are:

- the parametrization of literals and primitive operators in the language, and the associated proofs in the semantics `eval` (Section 3.3)
- the implementation of the capture avoiding substitution `CA_subst` and the associated `beta_equivalence` theorem, that states beta-reduction preserves the expressions' meaning (Section 4.3)
- the (partial) list fusion transformation proof (Section 4.5)

1.3 Overview

This thesis gives an overview of the PureCake project, the main features of the language and a commentary on its use in the context of provenly correct code transformations.

- Chapter 2 introduces some basic notions in the field of formal methods, semantics, and provenly correct software.
- Chapter 3 gives a description of the compiler implemented in this project. The description includes: the PureCake language, its semantics and various comments over features and implementation details.
- Chapter 4 goes through some source-to-source code transformations and shows how those can be proven in PureCake. It discusses some practical difficulties we encountered, and presents the main lemmas that we wrote in order to overcome them. There will be an emphasis over the ease of producing the proofs involved, together with a commentary on the alternative languages for the task.
- The thesis concludes with a few final remarks in Chapter 5.

2

Background

In this chapter we introduce formal methods as a discipline and how it is employed in the CakeML project. We will then characterize the process of proving properties in a formal setting and give a gist of the practical difficulties that might arise doing so. This will give us the chance to introduce functional languages and in particular some interesting properties that loosely relate with the proof process. We conclude with a brief introduction of formal semantics and a comparison between the three most commonly used ones.

2.1 Context

Formal methods is an umbrella term that covers any technique, method or framework that makes use of theoretical computer science notions to aid the development of software artifacts [16]. Formal methods put theory into practice: they unveil the mathematical nature of code, making it surface out of tangled programs. Ultimately carving out some of that complexity and revealing important properties and relations of the system.

In CakeML, formal methods are being used to verify the functional correctness of the compiler implemented. Specifically, it means that the compiler has been proven to translate CakeML programs into target code *preserving* their meaning.

But mathematical proofs only exist within the frame of the logic in which they have been stated. It follows that, in order to prove the correctness of a software artifact, like a compiler, this has to be written *within* a formal logic, together with models representing source and target languages and a "meaning" relation (*formal semantics*), that, for any program in either source or target language, it associates a common representation for the meaning of these. This relation is often referred to as *formal language specification*.

In this setting, the correctness property, for example, can be stated in this way:

$$\forall\ e.\ semantics(e) = semantics(compile(e))$$

with e any source program. Another example is whether two programs are observationally equivalent:

$$semantics(e_1) = semantics(e_2)$$

All these statements have to be proven true by the compiler designer, and, depending on the complexity of the objects involved, it might require some efforts. It is also clear the semantics covers a major role in the whole proving process.

2.1.1 The HOL theorem prover

Regarding the CakeML project, it uses higher-order logic as formal framework where languages, compiler and correctness proofs are formalized. The logic, together with the tools that support the proofs development process, are provided by the HOL theorem prover [1]. HOL relies on a small library (*the kernel*) in which higher-order logic is modeled as an *embedded domain-specific language (EDSL)* within the Standard ML's type system (Hindley-Milner type system). The library provides the primitive lexemes (axioms) together with the combinators to construct new theorems from the primitive ones (inference rules).

2.1.2 Proving is (sometimes) hard

The process of proving a property over a program is a task that requires a certain level of expertise: newcomers often have to face a counterintuitive learning curve, in which trivial theorems are hard to prove due to either:

- *declarative style theorem provers:* a divergence between how the logical system works and their way of arguing why something should be true (Isabelle, Agda, Epigram), or,
- *procedural style theorem provers:* the proof structure is clear, but there is no intuitive way for constructing it. The feasibility of a proof relies on high-level user directives (tactics) that need to be learnt (Coq, HOL) [43].

Learning how to effectively prove properties over programs requires effort, but the process of proving itself, i.e. correctly composing the logical lexemes to construct the proof object is, in itself, very simple.

So simple that it can often be performed by computer programs automatically. In fact automatic reasoners are extensively used in this context, especially when the search space for a proof is modest. Despite the progresses in the field, proving moderately complex properties over moderately complex languages, instead, still requires the expert intuition of the programmer.

This aspect highlights the resonance that language design choices can have over the final proof complexity: small simplifications have the potential to drastically decrease the subsequent proof efforts, which brings us to the next topic: purity and referential transparency.

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

2.2 Purity and referential transparency

One important property of pure functional languages is *referential transparency*. Roughly, it means that any term in the language, at any time, can be substituted with any other semantically equivalent term, without altering the evaluation's final result. For example,

```
(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3)
```

with `++` the list concatenation operation, as defined in Haskell. We can substitute any occurrence of one side of the equation with the other one

```
("purity"++"is")++"great!"
⇒
"purity"++("is"++"great!")
```

leading, in this case, to a more efficient program [9]. More generally:

$$\forall\ e_1\ e_2.\ semantics(e_1) = semantics(e_2) \Rightarrow$$
$$\forall\ e.\ semantics(e[e_1/e_2]) = semantics(e)$$

With $e[e_1/e_2]$ defined as the non-capturing substitution of the lambda *term* e_1 in e with e_2.

Referential transparency allows both the programmer and the compiler to reason over a program as an element of an *equivalence class* defined by the beta-reduction relation over the space of all possible programs. In substance, whenever we want to prove a property over a program's result, we can either do it directly, or we can pick any other program in this equivalence class, and prove it for it. The property holds for both:

$$\forall\ e_1\ e_2.\ e_1 \simeq e_2 \Leftrightarrow semantics(e_1) = semantics(e_2)$$
$$\forall\ P.\ \ e_1 \simeq e_2 \Rightarrow P\ semantics(e_1) \Leftrightarrow P\ semantics(e_2)$$

This, combined with a compositional semantics allows us to break down the proofs over the code into logical pieces. As we will see in Chapter 4, this also holds for functions in the language, making it even more useful during proofs.

Finally, referential transparency is a well understood concept in functional programming, being the mathematical framework over which computation is often described (beta-reductions in term rewriting systems), and it is extensively used in the context of informal proofs over the code [14, 11]. It is also used for optimizations purposes: in GHC end users can specify code equalities (*RULES* pragma) that are latter used by the rewrite engine to speed up programs.

2.3 Formal Semantics of programming languages

In semiotics, the study of sign processes, every *symbol* (a word in a text, a figure in a drawing), is distinguished between its concrete representation as a visual mean, and

its *denotation*, i.e. the associated meaning that the interpreter (reader, observer) gives to it.

Formal Semantics of programming languages is a branch of theoretical computer science that studies different possible ways to give rigorous meaning to the lexemes (symbols) of programming languages. One important field of application is of course in compilation technology: compilers are programs that transform entities (code) written in a language (source language) into another (target language) preserving their meaning.

Since its conception in the 1960s [8], three main frameworks have emerged: Operational semantics, Denotational semantics and Axiomatic semantics.

2.3.1 Operational Semantics

Operational semantics first introduces a mathematical model for a computational system, often an abstract machine (e.g. SECD machine [20]). Then gives meaning to the lexemes of the language as state transitions over the system.

Given a program and a starting configuration for the abstract machine, an *evaluation* is defined by the repeated application of those transition rules over the abstract machine.

Operational semantics gives great control over the program evaluation dynamics and brings into context practical aspects such as memory and clocks, allowing the designers to prove properties about them. Another important advantage is that operational semantics is often detailed enough to derive an interpreter for the language from it.

On the other side, the abundance of such details makes often hard to generalize the language meaning, and forces the proofs about it to take these into account, even when not strictly relevant. For example, whenever we want to prove that two programs are equivalent in meaning under a given semantics, the proof often needs to be unfolded for every possible machine's state [44, Section 5.1].

Abstract rewrite systems can be seen as a particular instance of operational semantics where the computational model is the language itself. In this case, the semantics provides the rewriting rules for it, and computation coincides with the continuous application of the rewriting rules over a given expression.

2.3.1.1 Small-step vs big-step operational semantics

Operational semantics is often distinguished in two classes: *small-step* and *big-step*. These two refer to the particular shape that the transition rules assume in the specification of the computational model.

In the small-step semantics, the evaluation of a program produces a sequence of machine's state transitions from the initial configuration to the final one. The big-

step semantics is instead defined as the reflexive, transitive closure of the small-step one. This means that the evaluation relation directly binds the whole input program to its final machine configuration.

This makes the big-step semantics slightly simpler to use in practice, but on the other side the lack of details complicates or even makes impossible to prove certain properties: for example in case of non-terminating programs the big-step semantics must be undefined, trivially because no transitive closure exists, while a small-step approach would allow us to make use of the state transitions' sequence to show it repeats infinitely often a certain pattern, practically proving the non-termination of the program.

2.3.2 Denotational Semantics

Denotational semantics takes a more abstract approach over the task: each element in a program is associated with a mathematical object, called *denotation*. Denotations are defined as (partial) functions from the expressions language ($\mathbb{L}$) and (when needed) an evaluation context (Γ) over the set of language values ($\mathbb{V}$):

$$[\![\_]\!]\_ : \mathbb{L} \times \Gamma \rightharpoonup \mathbb{V}$$

An evaluation context is usually defined as a map from variables names to values, that keeps track of the assignments during the expression's unfolding. Denotations are defined inductively over the structure of the language $\mathbb{L}$, e.g. for addition:

$$[\![add\ n_1\ n_2]\!]_\Gamma = [\![n_1]\!]_\Gamma + [\![n_2]\!]_\Gamma$$

One of the main features of this approach is that semantics is inherently *compositional*. This means that the denotation of an expression is directly derived by the denotation of its sub-expressions. Proving a property over such structures becomes easier; for example the commutativity of 'add' in the language directly follows from the fact that addition ('+'), as used in the denotation, is itself commutative. Another helpful feature that makes the proofs less cumbersome, is the abstractness of the evaluation context, that often remains unaltered in the various denotation's definitions, in contrast with the operational semantics, that modifies the machines' state at each step. In the latter, a property over a program needs to always be proved against the state transformations performed.

2.3.3 Axiomatic Semantics

Axiomatic Semantics closely relates with Hoare Logic. In the latter, *program constructs* (lexemes) of a language are treated as axioms (more accurately, inference rules) of a logical system. Those rules come with *assertions* (properties) about the state of the evaluation of a program. The assertions are divided between *pre-conditions*, and *post-conditions*. The rules have the following shape:

$$\{A\}\ c\ \{B\}$$

And they should be interpreted as follows: for any state σ of an execution system (e.g. an abstract machine) that satisfies A (the pre-condition), if the execution of c terminates in state σ', then σ' satisfies B (the post-condition) [44].

Axiomatic semantics, together with denotational semantics, are considered to be more abstract than operational semantics. This reputation comes from the fact that these are often not descriptive enough to derive a concrete implementation for the language.

While denotational semantics aims at giving a *mathematical meaning* to the programs in the language, axiomatic semantics focuses on describing the *effects* (the properties) that the evaluation of such programs should produce in a concrete implementation.

This shift makes axiomatic semantics very flexible and concise: whenever we want to prove a certain property over a system, there is no need to define the full meaning of the language, instead, it suffices to state the relation of that property with the lexemes in the language, and proceed deriving the desired conclusions.

The flexibility of axiomatic semantics comes with the need to prove the stated axioms to be sound against a model for the language, either denotational or operational.

In the end, in the context of compilers design, where unambiguous meanings and full implementations have to be derived for the language, other semantics are usually preferred, either operational or denotational.

The next chapter will introduce the PureCake language and its associated semantics. While describing it, we will refer back to the properties of the various semantics we discussed in this section, showing how those relate with the formal verification of PureCake.

3

The PureCake language

PureCake is a pure, non-strict functional language that comes with an associated formal semantics expressed in higher-order logic. In this chapter we introduce its syntax and denotation, together with high-level design choices and some implementation details.

The language semantics is divided in two parts: the first one concerns the meaning of the expressions in the language, and is represented by the function `eval`; the second one, instead, gives meaning to the side-effecting actions of the programs (`interp`) that defines its externally observable behavior.

3.1 Design choices

For PureCake we opted for call-by-name evaluation, defined through an operational semantics (*functional big-step semantics* [28]) paired with a series of equations that provide a denotational feel to the evaluation function. The operational semantics adopted is the same that has been used for CakeML, and its employment brings a couple of advantages: *(i)* the operational approach gives us fine-grained control over the evaluation, and consequently over the properties we can later state and prove *(ii)* a formal semantics logically close to the one used in CakeML makes the integration of PureCake with the compilation chain easier.

At the same time, employing a denotational semantics would bring a series of other advantages, mostly related with the formal verification of PureCake expressions (see Section 2.3). For this reason we developed a series of theorems and equations on top of the operational semantics that provides a compositional interface for evaluate of the expressions of the language, mimicking what would be a denotational semantics.

Another important aspect related with semantics is the evaluation strategy: either strict or non-strict (lazy). Non-strict fits slightly better, making the denotational equations we wrote fully compositional. We discuss this aspect in Section 3.2.

Finally, we need to decide whether to use an evaluation context or not. We opted for the absence of it, so that the whole evaluation state is kept within the expression being computed, resulting in a simple semantics function that maps expressions to values: `exp -> v`.

3.2 Evaluation style

PureCake, due to its Turing-completeness, is not *strongly normalizing*. As a consequence, the order in which the expressions are reduced into values might have an impact over the final result (either a proper value or non-termination) [3]. To keep the language evaluation deterministic, we need to pick a strategy to reduce our lambda terms, and stick with it. Two popular alternatives are *strict* (or *applicative*) and *non-strict* (or *normal*) order.

The strict order is loosely tied to the operational nature of abstract rewriting systems, which is the mathematical formalism where both evaluations (strict and non-strict) originated. This relationship can be clearly appreciated observing the denotational semantics of lambda functions that discard some arguments, like the `K` combinator:

$$\mathrm{K\ x\ y} = \mathrm{y}$$

non-strict semantics:

$$[\![\mathrm{K\ x\ y}]\!] = [\![\mathrm{y}]\!]$$

strict semantics:

$$[\![\mathrm{K\ x\ y}]\!] = \text{if } [\![\mathrm{x}]\!] = \bot \text{ then } \bot \text{ else } [\![\mathrm{y}]\!]$$

Under strict evaluation, all function's arguments must be fully evaluated before the function's body. If one of them diverges, also the body must. This detail is the only semantical difference between the two evaluation strategies. The final semantics becomes slightly less compositional; in the strict case, it has to keep track of the arguments a program comes across during the computation and conditionally diverge in case any of these diverge as well.

We will now introduce the expressions language, the values and finally the semantics.

3.3 PureCake expressions

PureCake programs have the following shape:

```
exp =
   Var vname
  | Prim op (exp list)
  | App exp exp
  | Lam vname exp
  | Letrec ((string × exp) list) exp
```

The language is purposefully inspired by GHC's intermediate language 'Core' [40, 39], and represents an untyped version of it.

The language consists of the three classical lambda calculus lexemes: function abstraction (`Lam`), application (`App`), and variable identifiers (`Var`). `vname` represents variable identifiers that are practically implemented as `string`s. Beside those, for convenience, a let binder (`Letrec`) has been added, for mutually recursive definitions. It takes as input a list of pairs (`string` × `exp`), which represent respectively an identifier and the expression that it binds to, and an expression that covers the role of letrec body. All the expressions mentioned can refer to any identifier (`string`) in the list of pairs, making the whole set of definitions (mutually) recursive.

Finally, there is the wrapper for the PureCake's primitive operators `Prim`, which gives support for standard language features, such as conditional branching (`If`) and datatype's constructors (`Cons`), projections (`Proj`) and equality (`IsEq`).

```
op =
  If
 | Cons string
 | IsEq string num
 | Proj string num
 | AtomOp atom_op
 | Lit lit
```

Together with the operation to be performed (an element of type `op`), `Prim` takes a list of expressions that represent the arguments to be passed to those primitives.

Worth of mention are the *parametrized* literals (`Lit`) and the associated operations over them (`AtomOp`). Parametrized in this context means that the two types `lit` and `atom_op` act as fixed polymorphic types with no concrete implementation. The rationale behind it is: during the design of a language, the operations over literals are subjected to continuous changes and redefinitions. In a classical compiler, adding a new primitive would simply require to extend the language with a new token, for example `'mul'`, adding it as new constructor for `atom_op`, and provide the associated meaning for it in the semantics. In a formal setting, in order to do so we also need to extend the compiler correctness proof to reflect those changes. So, with the intent to simplify the language design process and increase reusability, primitive operators have been parametrized: their semantics has been abstracted to force the proofs over the language (like, for example, the compilation correctness) to not rely on the specific implementation.

Finally, new PureCake lexemes can be introduced as macros (`Overload`s in HOL) defined in term of the other objects in the language. This is the case for `'Let'`, `'Case'`, and other operators. The main advantage of this approach is that it keeps the expression's datatype small. During an inductive proof on the structure of `exp`, we, in turn, only need to show it holds for the core language lexemes. At the same time, equations and properties can still be proved for the defined macros, making the language extension process ultimately easier: whenever we want to support new language features, macros exempts us from digging into the PureCake's technicalities and refactor all the core proofs.

3.4 PureCake values

The values in the language have the following shape:

```
v =
  Atom lit
 | Constructor string (v list)
 | Closure vname exp
 | Diverge
 | Error
```

The datatype (or more correctly co-datatype) is self-explanatory: `Atom`s are used to represent the parametrized literals. `Constructor` represents datatypes and co-datatypes within the PureCake language and `Closure`s represent function values (with `vname` the bound variable). The only peculiarity worth of notice is the distinction between `Diverge`, representing non-terminating programs' value, and `Error`, which, instead represents the outcome of ill-typed but terminating programs. An example for the latter would be a program that tries to apply a primitive operation over an incorrect number of arguments. This distinction is justified in Section 3.5.

Almost all these lexemes can be defined directly. `Constructor` is recursively stated in terms of a list of values. This shape implicitly allows for non-converging computation to produce an infinite data structure of type `v`. This is not a concerning aspect by itself. In fact, under a strict semantics, the evaluation of an expression that produces an infinite constructor would simply not terminate, and the semantics function would then give $\perp$ as meaning object to the whole computation, which is exactly the expected value for any strict program containing a non-terminating sub-expression.

But under non-strict semantics, those infinite sequences are allowed to be partially produced and consumed during an evaluation: programmers rely on the language laziness in order to avoid the full evaluation of the value, that would otherwise result in a non-terminating computation.

PureCake semantics needs to reflect this behavior. It has to cautiously compute the expressions into values only when actually needed, and even then it should compute the values up to its outermost data constructor, leaving the inner expressions uncomputed. A way to address this problem is model the values of the language using a lazy co-datatype. Doing so, the application of the language semantics over an expression would not result into a potentially infinite evaluation, but instead the actual computation would be suspended until when needed. The lazy data structure would also provide access to finite prefixes of the potentially infinite values, and the semantics could in turn be defined to make use of those.

For PureCake we opted for rose trees as lazy co-data structure (part of the HOL standard library: α `ltree`):

$$\alpha \texttt{ ltree} = \texttt{Branch } \alpha \texttt{ ((}\alpha \texttt{ ltree) list)}$$

The final type for our language values becomes `v_prefix ltree`, where `v_prefix` is defined as:

```
v_prefix =
  Atom' lit
 | Constructor' string
 | Closure' vname exp
 | Diverge'
 | Error'
```

`v_prefix` consists in all possible *finite* values of our language. Comparing it with `v` above, we can notice `Constructor` now lacks its second argument, which has been lifted into the `ltree` co-datatype, to take advantage of the lazy features offered. The `Constructor`'s arguments can in turn be accessed trough the exposed `ltree` interfaces, that ensure a lazy consumption of the structure.

3.4.1 Functions as values

One of the main characteristics of functional programming languages is that functions are *first class objects*. This means that functions are treated as any other value in the language and can hence be bound to identifiers, passed as arguments and returned.

In order to do so, we need to define a representation for functions in the values domain of our language. This representation is commonly known as *Closure*, and its concrete implementation (as data structure) is up to the compiler designer.

Our concrete representation is structurally equivalent to a lambda term in our expression's type `exp`:

$$\texttt{eval}\ (\texttt{Lam}\ n\ e)\ =\ \texttt{Closure}\ n\ e$$

When a closure is applied to an expression during the evaluation of a PureCake program, the substitution-based call-by-name semantics replaces each occurrence of the closure's bound variable n in e with the expression passed as argument, in practice absolving us from keeping explicit track of the bindings occurred.

3.4.1.1 Lambda terms and computable functions

The semantics for lambda terms, in the way it has been defined above, walks away from the denotational method, in favour of an operational one. It is clear that the proper mathematical object that should be associated with a lambda term is the function. Then why do not `Closure`s have the following type?

$$\texttt{v}\ =\ \texttt{Closure}\ (\texttt{exp}\ \rightarrow\ \texttt{v})$$

The reason is technical rather than conceptual: the objective of a function `f : exp` $\rightarrow$ `v` in `(Closure f)` is to provide a mathematical model for the computation that the closure represent. But the HOL logic can also model non-computable functions, and

there is no easy way to restrict the type of `f` to the strictly computable ones. Moreover, there is similarly nothing stopping the function `f` from syntactically inspecting the structure of the given input expression, e.g. it could treat $1+1$ and 2 differently because it has access to the explicit syntax.

For this reason we went for the operational representation of lambda terms.

3.5 PureCake language semantics

Following the functional big-step semantics style, we start defining the evaluation of PureCake expressions as a clocked interpreter function (with term rewriting as abstract machine for the evaluation). The function `eval_wh_to :num → exp → wh` assumes this role: it takes a natural number (the *clock*) and an expression as input, and returns as result the weak-head normal form of the input expression (of type `wh`).

The datatype for weak-head normal form expressions, shown below, is similar to the values `v` introduced in the previous section, with an important difference: the type is not recursive; the `Constructor` case holds a list of uncomputed expressions as its payload. This is consistent with the definition of weak-head normal form terms in term rewriting systems [30].

```
wh =
   wh_Constructor string (exp list)
 | wh_Closure string exp
 | wh_Atom lit
 | wh_Error
 | wh_Diverge
```

From the semantics function `eval_wh_to` (Figure 3.1) we can appreciate the role of the clock k. It acts as a "fuel" argument, that is decreased at each and every recursive call; when the clock reaches 0, the semantics returns a `wh_Diverge`.

The clock is needed in order to keep the function definition total, in fact, HOL is a logic of total functions where most recursive functions must follow well-founded recursion.

In the `App` case, for example, the recursive call does not always recur on a structurally smaller argument, because of the `bind` call that expands `body`. A similar argument applies for `Letrec`. The decreasing clock allows us to define the PureCake semantics without worrying about its totality.

The value returned by `eval_wh_to`, as defined, varies on the `k` passed, and it basically reduces to two cases: if `k` is "large enough", i.e. k is greater than the number of `App`s, `Letrec`s and `Prim`s encountered by `eval_wh_to` during the expression's evaluation, then `eval_wh_to` will return the actual value of the expression, which can be anything but `wh_Diverge`. Instead, if k is not large enough, or the program itself is not supposed to terminate, `eval_wh_to` will return `wh_Diverge`.

```
eval_wh_to k (Var s) ≝ wh_Error
eval_wh_to k (Lam s x) ≝ wh_Closure s x
eval_wh_to k (App x y) ≝
  let v = eval_wh_to k x
  in if v = wh_Diverge
      then wh_Diverge
      else case dest_wh_Closure v of
             None ⇒ wh_Error
           | Some (s, body) ⇒ if k = 0
                              then wh_Diverge
                              else eval_wh_to (k − 1) (bind s y body)
eval_wh_to k (Letrec f y) ≝
  if k = 0
   then wh_Diverge
   else eval_wh_to (k − 1) (subst_funs f y)
eval_wh_to k (Prim p xs) ≝
  if k = 0
   then wh_Diverge
   else let vs = map (λ a. eval_wh_to (k − 1) a) xs
        in case p of
             If ⇒ ... (omitted) ...
           | Cons s ⇒ wh_Constructor s xs
           | IsEq s'' i' ⇒ ... (omitted) ...
           | Proj s' i ⇒ ... (omitted) ...
           | AtomOp a ⇒ ... (omitted) ...
           | Lit l ⇒ ... (omitted) ...
```

Figure 3.1: `eval_wh_to` definition. For convenience, the semantics of most of the operations p in Prim p xs has been omitted.

The idea to use a clock to turn a partial function into a total one is not new, and it has been used in a substantial way in CakeML [28]. This approach, as pointed out by Xia et al. [45], is similar to the one proposed for denotational semantics by Dana Scott in one of his first formalizations [34], in which he suggests to construct the semantics of programming languages' functions through increasingly accurate approximations of partial functions, until the reach for a least fixed point.

Equipped with the clocked computation of the weak-head, the computation of the weak-head can hence be defined as the value assumed by `eval_wh_to` with k that extends towards infinity, practically removing the cases in which an expression e in `eval_wh_to k e` is mapped to a `wh_Diverge` because of a not large enough `k`.

This is done defining a function `eval_wh` as follows:

$$\begin{array}{l}\texttt{eval\_wh}\ e \stackrel{\text{def}}{=} \\ \quad \textsf{case}\ \texttt{some}\ k.\ \texttt{eval\_wh\_to}\ k\ e \neq \texttt{wh\_Diverge}\ \textsf{of} \\ \quad\quad \texttt{None} \Rightarrow \texttt{wh\_Diverge} \\ \quad\quad \mid \texttt{Some}\ k \Rightarrow \texttt{eval\_wh\_to}\ k\ e\end{array}$$

With `some` being wrapped around the choice operator ε [2]:

$$\texttt{some}\ x.\ P\ x \stackrel{\text{def}}{=} \textsf{if}\ \exists x.\ P\ x\ \textsf{then}\ \texttt{Some}\ (\varepsilon x.\ P\ x)\ \textsf{else}\ \texttt{None}$$

With the definition of `eval_wh` we conclude the formalization of the language semantics using the functional big-step framework, but we are not done yet: we also need to convert the `eval_wh` result from `wh` into `v`. To do so, we need to *(i)* force the evaluation of the expressions beyond the weak-head normal form, thus in practice calling `eval_wh` recursively over the arguments of a constructor *(ii)* wrap this potentially infinite resulting evaluation into the lazy datatype `v`.

The function `eval` is defined as follows:

$$\texttt{eval}\ x \stackrel{\text{def}}{=} \texttt{v\_unfold}\ \texttt{eval\_wh}\ x$$

And `v_unfold` in turn:

$$\begin{array}{l}\texttt{v\_unfold}\ f\ x \stackrel{\text{def}}{=} \\ \quad \textsf{case}\ f\ x\ \textsf{of} \\ \quad\quad \texttt{wh\_Constructor}\ s\ xs \Rightarrow \texttt{Constructor}\ s\ (\texttt{map}\ (\texttt{v\_unfold}\ f)\ xs) \\ \quad\quad \mid \texttt{wh\_Closure}\ s'\ x \Rightarrow \texttt{Closure}\ s'\ x \\ \quad\quad \mid \texttt{wh\_Atom}\ a \Rightarrow \texttt{Atom}\ a \\ \quad\quad \mid \texttt{wh\_Error} \Rightarrow \texttt{Error} \\ \quad\quad \mid \texttt{wh\_Diverge} \Rightarrow \texttt{Diverge}\end{array}$$

The function applies the argument f (in our case `eval_wh`) to the expression e and pattern matches over the result. In case of a `wh_Constructor`, it calls recursively itself over the xs arguments list and wraps the whole computation into the `v`'s lazy operator `Constructor`. For all the other cases, the `v` object creation follows trivially from the weak-head normal form conterpart.

3.5.1 `eval` equations

From this rather complex `eval` definition, we can derive a series of compositional equations that describe the relationship between PureCake expressions, values, and the semantics:

```
⊢ eval (Var s) = Error
  eval (Lam s x) = Closure s x
  eval (App x y) = let v = eval x
   in if v = Diverge then Diverge
      else case dest_Closure v of
             None ⇒ Error
           | Some (s, body) ⇒ eval (bind [(s, y)] body)
  eval (Letrec f x) = eval (subst_funs f x)
```

For the `Prim` case, we went a bit further, and proved the following equivalences:

```
⊢ eval (Prim (Cons s) xs) = Constructor s (map eval xs)
  eval (Prim (IsEq s n) [x]) = is_eq s n (eval x)
  eval (Prim (Proj s i) [x]) = el s i (eval x)
  eval (Prim (Lit l) []) = Atom l
  eval (Prim (If) [x; y; z]) =
    if eval x = Diverge then Diverge
    else if eval x = True then eval y
    else if eval x = False then eval z
    else Error
```

In the `Cons` case, the application of `eval` over the list of expressions `xs` could, potentially, not terminate. As we can see, this scenario is handled wrapping the application `map (eval c) xs` into the lazy operator `Constructor`, which is just an overload for the rose tree co-datatype constructor `Branch`:

$$\texttt{Constructor}\ s\ xs \stackrel{\text{def}}{=} \texttt{Branch}\ (\texttt{Constructor'}\ s)\ xs$$

3.5.2 Expressions evaluation: an example

The `eval` equations proved practically implement the `eval` function and provide a computable alternative for the evaluation of PureCake expressions. We can, for example, prove the following beta-reduction:

$$\vdash \texttt{closed}\ y \Rightarrow \texttt{eval}\ (\texttt{App}\ (\texttt{Lam}\text{“x”}\ (\texttt{Var}\ \text{“x”}))\ y) = \texttt{eval}\ y$$

only relying upon equational reasoning (*rewriting tactics* in HOL).

We can also evaluate infinite data-structures, like lists. Let us define an infinite list of identity functions `ids` using a `Letrec`:

$$\texttt{ids} = \texttt{Letrec}\ [(\text{“z”}, \texttt{Cons}\ \text{“cons”}\ [\texttt{Lam}\ \text{“x”}\ (\texttt{Var}\ \text{“x”});\ \texttt{Var}\ \text{“z”}])]\ (\texttt{Var}\ \text{“z”})$$

Making use of the equations proved, we can in turn unfold the datatype as follows:

```
⊢ eval ids =
Constructor “cons” [Closure “x” (Var “x”); eval ids]
```

The extent of the evaluation is explicitly controlled through the rewrite tactics offered in HOL.

3.6 Purity and I/O

As we said, in order to keep the observable behavior of programs deterministic, PureCake needs to be pure: the evaluation order of terms under non-strict semantics is data-dependent; this means that terms might, or might not, be evaluated depending on the outcome of other evaluations in the expression. The programmer has no explicit control over it, and the execution of side-effecting operations would potentially depend on the program's input data, making it non-deterministic. To avoid such scenarios, an explicit order between side-effecting operations needs to be enforced. This is done by making use of monads and preventing the programmer from placing side-effecting terms outside the monadic IO context. This is ultimately enforced by the language's type-system.

The monadic output of `eval` could be directly unfolded and performed by an effectful interpreter, but for the project we went for another path. Instead, we translate the monadic output value into an interaction tree [45]. Interaction trees are a recent formalization that allows to model (among other things) the I/O behavior of a program within the HOL logic: making it easier to prove properties about the interaction of the program with the environment, like liveness, for example. Interaction trees, as presented, consist in 3 constructors: `Ret`, terminates a program returning a value, `Vis`, used for interacting with the environment and `Tau`, that represent the execution of "silent" chunks of computation, needed to model programs' non-termination. Since we already model non-termination *within* our semantics, the Tau operator is not need. For this reason, our interaction tree co-datatype (`io`) consists only on the first two constructors:

$$(\epsilon,\ \alpha,\ \varsigma)\ \texttt{io}\ =\ \texttt{Ret}\ \varsigma\ \mid\ \texttt{Vis}\ \epsilon\ (\alpha\ \rightarrow\ (\alpha,\ \epsilon,\ \varsigma)\ \texttt{io})$$

The type `io` is parametrized over the three polymorphic type variables `(`ϵ`,` α`,` ς`)`.

The function `interp` translates the monadic actions into an interaction tree. The translation is roughly as follows: the monad is opened, revealing the sequence of bind-folded, side-effecting, chunks of code the program is made of. Those chunks contain exactly one side-effecting operation each. This operation can either be

- a `return` applied to a value `v`, i.e. a non side-effecting value that has been lifted to `IO`
- a proper side-effecting function, like `read`, `print` etc.

In case of `return`, the lifted value `v` is simply passed to the next code chunk, since no proper IO action occurred. Any other case is placed into a `Vis`, to perform the needed environment interaction. When no further interactions are supposed to happen, a `Ret` is produced to terminate the program.

The whole program semantics is finally defined as the composition of `eval` and `interp`:

$$\texttt{semantics}\ e \stackrel{\text{def}}{=} \texttt{interp}\ (\texttt{eval}\ e)$$

4

Code equalities with PureCake

In the previous chapter we presented the PureCake language and semantics. Here we will instead discuss the formal verification aspects of it. In particular, we will show the logical framework we designed to prove code equalities in PureCake and discuss some implementation details.

This Chapter is divided into two parts. In the first one (Section 4.1), we will argue why there is currently only a very limited support for code equalities proofs in the CakeML project, thus implicitly justifying the need for PureCake. In the second part, we will go through the practical matters of proving code equalities.

First, we will formalize what it means for two expressions to be equivalent, by giving context to the idea under both denotational and operational semantics. We will then move on introducing an important code equality transformation, *beta-reduction*, that will allow us to prove some code equalities, with a small example.

We will then try to generalize such equations through the use of forall quantifiers, and show the practical difficulties that this might bring, together with the solutions we adopted.

Ultimately, we will talk about how we handled proofs over co-inductive data structures, such as lists, and discuss their pros and cons. We finish with a comment about equalities and types.

4.1 Code equalities without PureCake

The two language alternatives offered by the CakeML project that we will discuss are the CakeML language itself, and HOL.

While CakeML is a natural choice for the task, using HOL, the logic used to develop all the proofs might seem peculiar. The usage of HOL as a programming language is made possible through a program called the "translator", which converts HOL functions into CakeML ones. The conversion, when successful, is also proven to be correct (proof-producing translation).

Finally, it is obligatory to mention we are limiting our comparison to the obvious alternatives *within* the CakeML project. There are several other languages and

logics that would be worth considering for this task, but most of these cannot take advantage of an end-to-end provenly correct compilation chain. This in itself does not make those languages inadequate for the task, but in such relaxed context the necessity for the code transformation to be proven correct vanishes. Proving a transformation does not provide any added value to the task except having shown the correctness in itself. With PureCake, instead, proving its correctness allows the transformation to be added into the compilation chain, ultimately contributing to the compiler.

4.1.1 CakeML

The most straightforward procedure for proving code equalities in CakeML is similar to the one presented in Section 2.1: we prove that two distinct programs (represented as CakeML ASTs) when applied to the language's semantics, result in the same meaning object.

CakeML is the first and most natural alternative to PureCake. They both are general-purpose Turing-complete languages with a formal semantics for the proofs and a provenly correct language compiler. Opposed to PureCake, CakeML is not pure (see Section 3.6) and the evaluation order follows a strict semantics.

Unfortunately, the lack of purity is a deal-breaker for our task: almost all code equalities change the order of evaluation of the terms in an expression, and non-pure languages are, in general, not resilient to those changes, i.e. altering this order potentially changes the observational behavior of a program, which ultimately changes the program's meaning making the code equality false.

We could try to prove the functions and values in a particular program are in fact pure, despite being written in CakeML: as long as we are able to prove their purity, then the transformation can be applied. Proving a particular function to be pure is often feasible, for example, given:

```
map f [] = []
map f (x:xs) = f a:map f xs
```

it is enough to assume the arguments of `map` to be pure to conclude the final result is pure as well.

It is, although, much harder to conclude purity for arbitrary expressions. Various factors get in the way; referring to the example above, f could be a complex recursive function that produces side-effects only for particular arguments: under this setting, determining for which x the function f is pure could even be undecidable. This conclusion directly follows from Rice's theorem, that's states any non-trivial semantical property of a Turing-complete program is, in general, undecidable [12].

The exploitability of code rewriting in CakeML is not only limited by the lack of purity, but also by its strict semantics. As we mentioned briefly in Section 3.2, this evaluation strategy imposes and order (right to left) for the evaluation of the terms in a function application (also known as *spine* [32]).

In non-strict strategies, no order is imposed: the partial computation of terms in a program can happen freely, without altering the whole meaning. This aspect is very important, because computation in lambda calculus is performed through code rewriting (following a specific rule, β-reduction). It follows that if we restrict the computation that can be performed to a specific order (as strict semantics does), we are also limiting the amount of rewrite rules that can be applied over a program at an given moment: each rewriting rule that performs computation can potentially break the strict strategy order, and hence alter the program's semantics.

This scenario can be easily seen with transformations that discard some terms, like the `K` combinator presented in Section 3.2:

$$\mathrm{K\ x\ y} \ \simeq\ \mathrm{y}$$

This equivalence holds in any pure, non-strict language, but in CakeML substituting the left-hand-side with the right one might result in a change in the program meaning, specifically when `x` diverges:

$$[\![\mathrm{K}\ \bot\ \mathrm{y}]\!] \ =\ \bot\ \neq\ [\![\mathrm{y}]\!]$$

Another more interesting example is the following dead-code elimination transformation:

$$\texttt{x} \notin FV(A)\ \Rightarrow\ \texttt{let x = B in A} \simeq \texttt{A}$$

That says: if `x` does not occur as free variable in `A`, then the whole let-binding can be removed. Again, in case B diverges, this substitution cannot take place.

A similar arguments can be made for errors: applying a transformation that discards a term that has the potential to trigger an error alters the programs semantics, and therefore cannot be applied.

4.1.2 HOL

The HOL language, like many other proof assistants, comprise a subset that closely resembles a pure programming language. Definitions that are strictly within this programming language-like subset can hence be translated into a proper programming language (like CakeML), with the intent to be executed.

In case of HOL, this is made possible making use of the "translator" [25, 19]. The translator consists of Standard ML functions that convert HOL terms into CakeML ones, together with a proof that certifies the correctness of such transformation (proof-producing translation). This process is also referred to as *verified program extraction* in the literature.

We can write the two programs we want to show are equal in HOL, and make use of the translator to convert those in CakeML AST, that in turn can be compiled into machine code.

This approach has an advantage over PureCake and a series of drawbacks. The main advantage is that HOL can benefit from a collection of libraries with already proven code equalities. In fact, those, paired with the tactics offered in the system, constitute the main building blocks that allow programmers to construct the proofs in the logic.

For example, we can find the equality that states the correctness of a list fusion transformation [1]:

$$\vdash \texttt{map}\ f\ (\texttt{map}\ g\ l) = \texttt{map}\ (f \circ g)\ l$$

On the other side, the fact that HOL has been conceived as a logic system brings also a series of disadvantages. First of all, HOL requires each function definition to be *total*. In particular this means that most definitions of recursive functions must only recurse according to a well-founded relation. In other words, most recursive functions terminate if one views them as functions in a programming language (note there are certain classes of recursive functions that can be defined in HOL without termination proofs, e.g. tail-recursive functions or certain boolean-valued functions).

Although this is not a real limitation from the expressiveness point of view [41, 22], it forces the programmer to prove the existence of such relation for any recursive definition (or at least for the ones in which automatic methods fail). From the user point of view, this adds a small learning curve over an otherwise frictionless transition from a classic functional programming language (Standard ML, Haskell etc.) to HOL.

One more significant issue with HOL being a logic of total functions is that it makes it a bad target for a potential source-to-source compiler for common functional languages (Standard ML, Haskell, JavaScript etc.). In fact, all the main programming languages currently used allow the definition of partial functions. Turning all the functions in a program from partial to total requires an accurate refactoring by the programmer, that needs to provide a meaning to all the cases left undefined. In particular for recursive functions their totality requires a proof of termination (like in HOL), which is in general undecidable [12]. The inability to automatically port mainstream languages into the verified compilation chain limits the potential exploitability of the project as formal verification tool.

Another relevant drawback of using HOL is that we would rely on the translator to implement the verified compilation chain. Proof-producing code extraction is slightly more fragile compared with verified compilation, because it does not guarantee the successful termination of the translating function for all inputs.

4.1.3 Something in between

There is a third option for the task. We can make use of the HOL-translator to produce CakeML functions, that are latter placed into larger CakeML programs. The rationale behind it is that we can prove properties about HOL functions making

[1] from *listTheory.MAP_MAP_o*

use of all the tools and libraries already developed for the language, and latter deploy those in a full program.

We can, for example, program a sorting algorithm in HOL, and prove its correctness. Latter on, we can make use of the translator to produce a CakeML function from it and use it to implement a correct full-scale program.

While this approach turns out to be useful in some scenarios, it lacks an important aspect that we can instead find in PureCake: the properties we prove for the terms in HOL do not compose once deployed in CakeML. We can take as example the list fusion discussed above:

$$\vdash \texttt{map}\ f\ (\texttt{map}\ g\ l) = \texttt{map}\ (f \circ g)\ l$$

This equality implicitly assumes the two functions f and g to be pure and total, trivially because the proof is conducted within the HOL language. Once deployed into CakeML, those assumptions do not hold anymore.

In other words there is a forced logical separation between the two languages that is given from the fact we are translating functions from a language with a certain set of properties to another one that lacks some of those (purity, in particular). As a result, we lose all the proofs that rely on those implicit properties.

On the other side, this exact same technique might turn out to be more useful when translating HOL terms into PureCake ones. PureCake, as we said, is pure. This gives more HOL equalities the chance to hold when translated into PureCake.

A difference between the two languages (HOL and PureCake) that can limit the scope of this approach is that in PureCake there is no separation between data and co-data, while in HOL this distinction is needed to preserve the function's totality. As an example, the list fusion transformation discussed above assumes the lists to always be finite, while in PureCake this is in general not true.

4.2 Code equalities with PureCake

Informally, we define two PureCake expressions to be equivalent when one can replace the other, in any program, without altering the overall meaning.

A first naive equivalence can be defined in terms of `eval` as follows:

$$e_1 \simeq e_2 \Leftrightarrow eval\ e_1 = eval\ e_2$$

Two expressions are equivalent if the function `eval` maps them to the same value.

We soon notice that, the way it is stated, this equality trivially holds only for fully applied PureCake expressions. In fact, the evaluation of a `Lam` term results in a Closure, whose body is left uncomputed:

```
add x y = ... (addition between literals) ...
e  = Fun "n" (add (Lit 5) (Lit 6))
```

```
e' = Fun "n" (Lit 11)

eval e  = Clos "n" (add (Lit 5) (Lit 6))
eval e' = Clos "n" (Lit 11)
```

The PureCake expression `e` and `e'` can obviously be substituted with each other in any program without altering its final meaning, despite being not structurally equal.

The problem involves even expressions only consisting of application and lambda abstraction:

```
id  = Lam "x" (Var "x")
id' = Lam "y" (App id (Var "y"))

eval id = Clos "x" (Var "x") ≠ Clos "y" (App id (Var "y")) = eval id'
```

Functions are a substantial subset of all the terms we would like to substitute in an equational context, leaving those out is not an option. We need to define an equivalence relation that subsumes partially applied terms, and relates closures that can be effectively substituted without altering the meaning of a program. Luckily, this exact problem has been object of thorough studies in the past, and since then several solutions have been proposed. We refer to Ong [27] for a review of the literature.

Before discussing the formalization, a small digression over equality between functions.

4.2.1 Intensional versus Extensional equality

Two functions are said to be *extensionally equal* when they map the similar input to the similar output. In set theory, a model that is often used to describe the extensional equality of two mathematical functions is given interpreting those as a set of pairs associating each element of the input with the corresponding output element, and defining their equality in terms of equality of these sets of pairs.

Extensional equality is opposed to *intensional equality*, where functions can be understood as a set of rules in a formal language that express how to transform the input into output. This equality shows the relationship between function arguments and returning values, like a pure programming language. Under intensional equality, two lambda terms are equal if they are structurally equivalent up to renaming of the bounded variables in the expression (*alpha equivalence*).

This is exactly the issue we are facing in our equality relation for PureCake. We notice that intensional equality is not loose enough to capture the idea of contextual equivalence, and we therefore need to define a relation based upon extensional equality. The intensional equality of two lambda terms implies their extensional equality, but the other way round does not always hold. The analogy with programming languages makes the intuition clear: two observationally equivalent programs are often not structurally equal (`i++;` vs `i=i+1;`), but, obviously, two identical programs

always share the same behavior [1].

4.2.2 Contextual equivalence

The mathematical formalization for contextually equivalent expressions depends on the semantics adopted: denotational semantics states that two expressions are equivalent if those are mapped to the same denotational object:

$$e_1 \simeq e_2 \Leftrightarrow [\![e_1]\!] = [\![e_2]\!]$$

this is the definition we used to introduce the concept in Section 2.2. The simplicity of this formula relies on the ability of denotational semantics to provide a unique representation for the meaning of a program, making the comparison between the expressions ultimately trivial. In other words, all the hard work is done defining the denotational semantics in the first place.

The definition of equivalent programs under an operational semantics requires instead a more substantial approach. Suppose we have a generic program C for our abstract machine. This program C contains an "hole" in where we can place any program fragment X, obtaining a complete program $C[X]$, that can therefore be interpreted by the abstract machine.

This program C is commonly called *context*. Now, we say that two program's fragments (our PureCake expressions) are considered equivalent when for any possible context, the machine running the two expressions behaves in the exact same way.

$$\forall\, C.\ e_1 \simeq e_2 \Leftrightarrow (\forall\, v.\ C[e_1] \Downarrow v \Leftrightarrow C[e_2] \Downarrow v)$$

In other words the two executions are *observationally equivalent*.

The formalization we adopted for PureCake is based on this second approach. The implementation follows closely Pitts [33], which in turn is based on Howe [13]. The equivalence relation between two closed expressions is defined as a bi-simulation over the `eval` function (presented in Section 3.5), which in this case covers the role of transition function for our term-rewriting abstract machine. Without going into the details of the implementation, we will show the resulting main equivalence ($\simeq$) lemma[2].

For `Atom`s and `Error`s the definition of this equivalence is straightforward: we simply lift the HOL structural equality.

$$\begin{aligned}
\vdash\ & e_1 \simeq e_2 \iff \\
& \texttt{closed}\ e_1 \wedge \texttt{closed}\ e_2 \\
& \wedge\ (\forall\, a.\ \texttt{eval}\ e_1 = \texttt{Atom}\ a \Rightarrow \texttt{eval}\ e_2 = \texttt{Atom}\ a) \\
& \wedge\ (\texttt{eval}\ e_1 = \texttt{Error} \Rightarrow \texttt{eval}\ e_2 = \texttt{Error}) \\
& \ldots
\end{aligned}$$

[2] from `pure_exp_relTheory.app_bisimilarity_iff`. The definition has been slightly simplified for clarity. The relation presented lacks its second part, in where the expressions e_1 and e_2 are permuted, thus making $\simeq$ symmetric

For the other cases, which contain themselves unevaluated PureCake expressions (`Closure` and `Constructor`), the equivalence must be defined co-inductively over the semantics of such expressions:

$$\begin{array}{l} \ldots \\ \wedge\ (\forall\, x\ e1s.\ \texttt{eval}\ e_1\ =\ \texttt{Constructor}\ x\ e1s\ \Rightarrow \\ \quad \exists\, e2s.\ \texttt{eval}\ e_2\ =\ \texttt{Constructor}\ x\ e2s \\ \quad\quad \wedge\ \texttt{LIST\_REL}\ (\simeq)\ e1s\ e2s) \\ \ldots \end{array}$$

In particular for closures, the function body might contain a free variable: the relation must be proven to hold for any $\simeq$-related expressions a_1 and a_2 which have been substituted on each sided with the free variable:

$$\begin{array}{l} \ldots \\ (\forall\, x\ ce_1. \\ \quad \texttt{eval}\ e_1\ =\ \texttt{Closure}\ x\ ce_1\ \Rightarrow \\ \quad\ \exists\, y\ ce_2.\ \texttt{eval}\ e_2\ =\ \texttt{Closure}\ y\ ce_2\ \wedge \\ \quad\quad \forall\, a_1\ a_2\ .\ a_1\ \simeq\ a_2 \Rightarrow\ \texttt{subst}\ x\ a_1\ ce_1\ \simeq\ \texttt{subst}\ y\ a_2\ ce_2) \end{array}$$

From a denotational perspective, this similarity states that two closures are related when the two associated functions are *extensionally equal*.

On top of $(\simeq)$ we can then define an equivalence relation between not-closed terms:

$$\begin{array}{l} \vdash\ x\ \cong\ y\ \iff \\ \quad \forall f.\ \texttt{freevars}\ x\ \cup\ \texttt{freevars}\ y\ \subseteq\ \texttt{FDOM}\ f \\ \quad\quad \Rightarrow\ \texttt{bind}\ f\ x\ \simeq\ \texttt{bind}\ f\ y \end{array}$$

Which simply states that two terms are related when substituting their free variables with generic closed terms, we obtain two expression that are themselves $\simeq$-related.

4.3 Beta-reduction

Making use of our brand new equivalence $(\cong)$, we can prove that the beta-reduction rule preserves the programs' meaning, i.e. always relates expressions within the same $\cong$-equivalence class:

$$\vdash\ \texttt{App}\ (\texttt{Lam}\ x\ body)\ arg\ \cong\ \texttt{CA\_subst}\ x\ arg\ body$$

Where `CA_subst` is the capture-avoiding substitution of the free variables `x` in `body` with `arg`.

This proof allows us to prove that two PureCake expressions are in fact equivalent through a sequence of beta-reduction steps. For example we can now prove that:

```
id ≅ id' ⇔
Lam "x" (Var "x") ≅ (Lam "y" (App (Lam "x" (Var "x")) (Var "y"))
```

in one reduction step [3].

[3] `examples/pure_identity_equivScript.sml`

4.4 Abstracting code equalities

An important aspect that relates with referential transparency is the ability to generalize the equations we want to prove over a set of possible programs and input arguments. This is done very naturally in HOL through the use of forall quantifiers. As example, here is a variant of the list fusion transformation:

$$\forall\ f\ l.\ \texttt{closed}\ f \wedge \texttt{closed}\ l\ \Rightarrow \\ (\texttt{map}\ f\ .\ \texttt{map}\ f)\ l\ \simeq\ \texttt{map}\ (f\,.\,f)\ l$$

With f and l two generic PureCake expressions. While proving the two expressions are $\simeq$-related, we might have to partially evaluate them (through *symbolic interpretation*). The evaluation of the expressions must, in turn, go indirectly through those variables, and this might pose some challenges.

In Section 3.5, we briefly showed how the semantics for the `App` lexemes is defined in terms of a `bind` function:

$$\texttt{bind}\ m\ e\ \stackrel{\text{def}}{=} \\ \textsf{if}\ \forall n\ v.\ \texttt{FLOOKUP}\ m\ n\ =\ \texttt{Some}\ v\ \Rightarrow\ \texttt{closed}\ v\ \textsf{then}\ \texttt{subst}\ m\ e \\ \textsf{else}\ \texttt{Fail}$$

This function substitutes a series of free variables in a term with their associated expressions, and requires the latter to not contain free variables themselves (*closed*). Since some might contain forall-quantified HOL variables, `closed` must be defined over the structure of the expressions datatype:

$$\texttt{closed}\ e\ \stackrel{\text{def}}{=}\ \texttt{freevars}\ e\ =\ []$$

$$\texttt{freevars}\ (\texttt{Var}\ n)\ \stackrel{\text{def}}{=}\ [n] \\ \texttt{freevars}\ (\texttt{Prim}\ v_0\ es)\ \stackrel{\text{def}}{=}\ \texttt{FLAT}\ (\texttt{map}\ (\lambda\, a.\ \texttt{freevars}\ a)\ es) \\ \texttt{freevars}\ (\texttt{App}\ e_1\ e_2)\ \stackrel{\text{def}}{=}\ \texttt{freevars}\ e_1\ +\!\!+\ \texttt{freevars}\ e_2 \\ \texttt{freevars}\ (\texttt{Lam}\ n\ e)\ \stackrel{\text{def}}{=}\ \texttt{FILTER}\ (\lambda\, y.\ n\ \neq\ y)\ (\texttt{freevars}\ e) \\ \texttt{freevars}\ (\texttt{Letrec}\ lcs\ e)\ \stackrel{\text{def}}{=} \\ \textit{omitted...}$$

By turning the definition in this way, it is enough, during the proofs, to assume the HOL variables to be closed themselves in order to infer any term containing them would also be closed.

Another example is given by `subst` that, given a expression v, a variable identifier n and an expression e, it substitutes each occurrence of n in e with v. It might be the case, during some proofs, that `subst` is called over an HOL variable. This in theory is not a concern, since those are assumed to be closed and a substitution simply leaves the term unaltered. In practice, we have to prove a lemma that says so:

$$\vdash\ \neg\texttt{MEM}\ n\ (\texttt{freevars}\ e)\ \Rightarrow\ \texttt{subst}\ n\ v\ e\ =\ e$$

4.5 Equalities over infinite structures

To prove the semantical equivalence of recursively defined expressions is not enough to rely upon term-rewriting proof strategies, like the beta-reduction equivalence above.

We need to make use of co-induction to prove that the two expressions are equivalent. To help us with it, we proved a lemma (*progress lemma*):

$$\begin{array}{l} \vdash \texttt{progress}\ exp_1\ next\ \wedge\ \texttt{progress}\ exp_2\ next\ \wedge \\ \quad isClos\ (\texttt{eval}\ exp_1)\ \wedge\ isClos\ (\texttt{eval}\ exp_2)\ \Rightarrow \\ \qquad exp_1\ \simeq\ exp_2 \end{array}$$

The lemma saves us from explicitly construct a bi-simulation argument whenever we want to prove an equality between two closures. It states that if two expressions exp_1 and exp_2 are proved to behave equivalently (*progress*) against a certain semantical model (*next*) then those two expressions are $\simeq$-related.

The semantical model corresponds, in practice, to an HOL function that defines the expected behavior the two expressions should implement, in terms of the various shapes the input can assume. Here is, as example, the semantical model for the `map f` expression:

```
next_list f input =
  if ¬closed input then INL Fail
  else if eval input = Diverge then INL Bottom
  else case eval input of
    | Constructor "nil" [] ⇒ INL nil
    | Constructor "cons" [_;_] ⇒
        INR ("cons", App f (Proj "cons" 0 input), Proj "cons" 1 input)
    | _ ⇒ INL Fail
```

The model is parametrized over the function supposed to be mapped (f) and the *input* list, and describes the possible outcomes that can result in applying `map f` over a list. Those outcomes are divided in two classes and relate on the way the function `map` consumes its input:

```
map f [] = []
map f (x:xs) = f x:map f xs
```

at any given occasion `map` either terminates returning a value (`[]`) or produces a partial result (`f x:_`) and calls itself recursively. Hidden under the syntactic sugar, the Haskell definition above also describes the behavior of the function when a diverging term is passed as argument to `map f`, which is 'diverge' as well.

We can recognize each of those scenarios listed in the model `next_list`. The terminating cases are wrapped in an `INL` constructor ('Left' constructor for the HOL's sum type) while the recursive-call case is labelled with `INR`. Since PureCake is untyped, we also need to define the outcome for ill-typed applications (that result in an `Error`).

In turn, the `progress` relation states that an expression *exp implements* a model *next* if, for any possible input, their outputs are $\simeq$-related:

$$\begin{array}{l}\texttt{progress}\ \mathit{exp}\ \mathit{next} \iff \\ \quad \texttt{App}\ \mathit{exp}\ \mathit{input} \simeq \\ \quad \textsf{case}\ \mathit{next}\ \mathit{input}\ \textsf{of} \\ \quad\ |\ \texttt{INL}\ \mathit{final} \Rightarrow \mathit{final} \\ \quad\ |\ \texttt{INR}\ (n, x, \mathit{rec\_input}) \Rightarrow \texttt{Cons}\ n\ [x;\ \texttt{App}\ \mathit{exp}\ \mathit{rec\_input}]\end{array}$$

4.5.1 List fusion proof

We can in turn make use of the *progress* lemma to prove a simple code equality between functions over lists:

$$\begin{array}{l}\forall\ f\ g\ l.\ \texttt{closed}\ f \wedge \texttt{closed}\ g \wedge \texttt{closed}\ l \\ \quad \Rightarrow (\texttt{map}\ f\ .\ \texttt{map}\ g)\ l \simeq \texttt{map}\ (f.g)\ l\end{array}$$

This equality is an instance of a wide class of code-transformation equalities known in the literature as *list fusion* [9].

We start proving the right-hand side of the equivalence:

$$\vdash\ \texttt{closed}\, f\ \Rightarrow\ \texttt{progress}\ (\texttt{App}\, \texttt{map}\, f)\ (\texttt{next\_list}\, f)$$

The progress relation holds for any expression f, in particular the $(f\ .\ g)$ above. The proof is lengthy but straightforward. We make use of [4]:

$$\vdash\ \texttt{eval}\, x\ =\ \texttt{eval}\, y\ \wedge\ \texttt{closed}\, x\ \wedge\ \texttt{closed}\, y\ \Rightarrow\ x \simeq y$$

And proceed rolling `eval` through the expression `App map f`, showing for each `input` the result is equivalent to the expression given by the model `next_list f`.

We conclude by proving in the same fashion the `progress` relation for the left-hand side and finally make use of the progress lemma to obtain our equivalence.

4.6 Equalities and types

As we said, PureCake is untyped. This has an important implication over the amount of equalities we can prove with it. For example:

```
map id ≃ id
```

Only holds when we assume the input argument for each side of the equality to be a list. For any other value v, `map id v` returns an '`Error`' under the current semantics, while `id v` always returns `v`. Since the two expressions are not $\simeq$-related for all shapes of input, the two expressions are not equivalent.

[4] `meta-theory/pure_exp_relTheory.eval_IMP_app_bisimilarity`

In order to extend the scope of the equalities, we need to restrict the *input* to only type-correct expressions. Unfortunately, this inherently relates to the PureCake language itself rather than the equivalence relation we defined. Relaxing the equivalence condition up-to-'`Error`' is also not a viable option: two expressions would be considered contextually equivalent simply because the is no valid input that matches both type signatures, like `(\x.x+1)` and `(\x.[x]++[1])`: a complete nonsense.

The only concrete solution is to bring a type discipline in the language that prevents type-ill applications from happening, as it is case for the *Programming Computable Functions* language (PCF) and several of its variants [10].

5

Conclusions

In this report we described the main contributions and achievements of the PureCake project from its very beginning until now. We described context and motivations for its conception, the design choices and the associated challenges.

In its essence, this project shares the same spirit of CakeML: to make advanced formal methods techniques more accessible. In order to make concrete progresses over this ambitious task, we should leave no stone unturned. PureCake takes a step into non-strict functional languages, and since the conception of the project we discovered some relevant aspects that show promise.

In particular, we have identified a class of proofs (code equalities), that is currently not well supported by any other language in the CakeML project. We justified the practical importance of them in the context of code optimizations, and shown, with examples, how those can be achieved in PureCake.

5.1 Limitations and future work

The PureCake language and its semantics are continuously evolving to overcome the obstacles in the way to the implementation of the compiler, and every now and then some challenges are inevitable.

Developing provenly correct software requires great time and efforts, and there are still several components of the compiler yet to be implemented, including parsing, type-checking, strictness analysis, and the target-code generation pass itself. All these are currently under development or soon will be, hence constituting the impending "future work" for the project.

At the moment, the focus is kept over the feasibility of the compiler itself. Despite the substantial research around the topic, PureCake remains the first ever attempt to develop a provenly correct non-strict compiler. While an actual implementation seems to be within reach, it is still unclear how the language will have to change to accommodate the technical difficulties along the way.

Moving over the code rewriting optimization side of the thesis, there are many viable paths. First of all, there is the need to implement the actual rewrite engine that performs the optimization. The engine needs to pattern match the input pro-

gram over the equalities we have proven and once a match with one side of the equivalence is found, it should be replaced with the other side. This machinery is almost identical to the proof construction process adopted in HOL, which could in turn provide insightful instruments for its implementation. The correctness of such transformations should rely on the logical framework we developed (Section 4.2).

Another viable path has to do with the equivalence relation presented in Section 4.2. Currently, to prove code equalities between recursive expressions we have to rely on co-induction. The approach, although effective, requires non-trivial efforts, also computationally. This is mostly due to the fact that the current approach is not compositional: the two sides of the equivalence has to be proven correct against a model, and the only way to do so is through symbolic interpretation. The lack of compositionality should be attributed to the underlying operational semantics itself, rather than the equivalence notion we constructed on top of it [36].

An alternative approach would be to make use of denotational semantics to prove the equivalence of the two expressions (as introduced in Section 4.2.2). The adoption of a denotational semantics would solve the compositionality issue but, in turn, it would introduce another problem: how to make the complex mathematical objects returned by the denotational semantics simple enough to allow practical reasoning over their equality. Addressing this challenge, Siek [36] investigates some *elementary models* for call-by-value lambda-calculus, that show promise.

It is also worth mentioning that research over the PCF language have found a correspondence between the notions of equivalence in the two semantical frameworks (denotational and operational) [27]. This connection might help overcome the difficulties that seem to arise during the practical poof of code equalities.

Also concerning the code equivalences, in order to expand the profitability of the optimizations, it would interesting to develop a typed version of PureCake, that would allow us to prove more equalities, as we argued in Section 4.6.

5.2 Related work

Recently, formal methods have been proven to increase the reliability of large software projects, to an extent unrivaled by any other technique [46]. While this achievement marks an important milestone in the field, the profitability of those verified projects has been somehow limited by the existence in the market of unverified counterparts, that, thanks to software testing and other tools, have been nonetheless able to meet the reliability requirements of most non-critical applications.

This is the case for CakeML, for example: the existence in the market of unverified compilers for Standard ML makes the CakeML project uninteresting in contexts where reliability can be sacrificed for convenience.

Now that the efficacy of software verification for larger projects has been proven, the next step is to make use of this tool to develop classes of software that are

too complex to be approached with informal methods. In this thesis, we brought the case for compiler's optimizations, which is a class of software inherently hard to test (see Section 1.1.2). Several other are emerging, especially in the context of code analysis [24], cryptography, network communications [26] and artificial intelligence [35].

Finally, equational reasoning has always received sustained support from the research community [15], both as an optimization technique [31, 23, 6] and as an instrument for reasoning over the correctness of programs [42]. PureCake provides a logical framework for research in both aspects, and, as a plus, it give the chance to concretize the theories proven into a verified compilation chain.

PureCake also enables the formalization of more complex languages on top of it, including typed ones like Haskell, ultimately contributing with a new tool in the efforts for the formal reasoning about programs.

Bibliography

[1] Jesse Alama and Johannes Korbmacher. The Lambda Calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2019 edition, 2019.

[2] Jeremy Avigad and Richard Zach. The Epsilon Calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2020 edition, 2020.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Term Rewriting and All that. Cambridge University Press, 1999.

[4] Joachim Breitner. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2016.

[5] Joachim Breitner. The sufficiently smart compiler is a theorem prover. *arXiv e-prints*, page arXiv:1805.08106, May 2018.

[6] Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the hermit: Tool support for equational reasoning on ghc core programs. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 23–34, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Kathleen Fisher. Using formal methods to eliminate exploitable bugs. `http://chalmersfp.org`, Jun. 2020.

[8] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[9] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 223232, New York, NY, USA, 1993. Association for Computing Machinery.

[10] Andy Gordon. Bisimilarity as a theory of functional programming. In *MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1, pages 232–252. Elsevier, March 1995.

[11] Cordelia Hall and John ODonnell. *Discrete Mathematics Using a Computer*. Springer-Verlag, Berlin, Heidelberg, 2000.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[13] D. J. Howe. Equality in lazy computation systems. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 198–203, 1989.

[14] Zhenjiang Hu, John Hughes, and Meng Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 07 2015.

[15] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[16] Michael G. Hinchey PhD Jean-François Monin PhD. *Understanding Formal Methods*. Springer-Verlag London, 1 edition, 2003.

[17] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The International SAT Solver Competitions. *AI Magazine*, 33(1):89–92, Mar. 2012.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[19] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with itps should use binary code extraction to reduce the tcb. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 362–369, Cham, 2018. Springer International Publishing.

[20] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 01 1964.

[21] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107115, July 2009.

[22] Conor McBride. Turing-completeness totally free. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2015.

[23] Neil Mitchell. Rethinking supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 309320, New York, NY, USA, 2010. Association for Computing Machinery.

[24] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger sfi for the x86. *SIGPLAN Not.*, 47(6):395404, June 2012.

[25] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. *SIGPLAN Not.*, 47(9):115126, September 2012.

[26] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):6673, March 2015.

[27] C.-H. L. Ong. *Correspondence between Operational and Denotational Semantics: The Full Abstraction Problem for PCF*, page 269356. Oxford University Press, Inc., USA, 1995.

[28] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, page 589615, Berlin, Heidelberg, 2016. Springer-Verlag.

[29] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, page 9197, New York, NY, USA, 2011. Association for Computing Machinery.

[30] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.

[31] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393434, July 2002.

[32] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., USA, 1987.

[33] Andrew Pitts. *Howe's method for higher-order languages*, page 197232. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011.

[34] Dana Scott. Outline of a mathematical theory of computation. *Kiberneticheskij Sbornik. Novaya Seriya*, 14, 01 1977.

[35] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Towards Verified Artificial Intelligence. *ArXiv e-prints*, July 2016.

[36] Jeremy G. Siek. Revisiting Elementary Denotational Semantics. *arXiv e-prints*, page arXiv:1707.03762, July 2017.

[37] George Stelle and Darko Stefanovic. Verifiably lazy: Verified compilation of call-by-need. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, IFL 2018, page 4958, New York, NY, USA, 2018. Association for Computing Machinery.

[38] George Stelle, Darko Stefanovic, Stephen L. Olivier, and Stephanie Forrest. Cactus environment machine. In David Van Horn and John Hughes, editors, *Trends in Functional Programming*, pages 24–43, Cham, 2019. Springer International Publishing.

[39] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and

Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007.

[40] Andrew Tolmach. An external representation for the ghc core language. 10 2001.

[41] D. A. Turner. Total functional programming. *J. Univers. Comput. Sci.*, 10(7):751–768, 2004.

[42] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in liquid haskell (functional pearl). *SIGPLAN Not.*, 53(7):132–144, September 2018.

[43] Freek Wiedijk. A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science*, 8(1), Mar 2012.

[44] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[45] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):132, Jan 2020.

[46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 11, page 283294, New York, NY, USA, 2011. Association for Computing Machinery.