Technical Report

UCAM-CL-TR-879
ISSN 1476-2986

Number 879

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Self-compilation and self-verification

Ramana Kumar

February 2016

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 2015 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Peterhouse College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

This dissertation presents two pieces of work, one building on the other, that advance the state of the art of formal verification. The focus, in both cases, is on proving end-to-end correctness for realistic implementations of computer software. The first piece is a verified compiler for a stateful higher-order functional programming language, CakeML, which is packaged into a verified read-eval-print loop (REPL). The second piece is a verified theorem-prover kernel for higher-order logic (HOL), designed to run in the verified REPL.

Self-compilation is the key idea behind the verification of the CakeML REPL, in particular, the new technique of *proof-grounded bootstrapping* of a verified compiler. The verified compiler is bootstrapped within the theorem prover used for its verification, and then packaged into a REPL. The result is an implementation of the REPL in machine code, verified against machine-code semantics. All the end-to-end correctness theorems linking this low-level implementation to its desired semantics are proved within the logic of the theorem prover. Therefore the trusted computing base (TCB) of the final implementation is smaller than for any previously verified compiler.

Just as self-compilation is a benchmark by which to judge a compiler, I propose self-verification as a benchmark for theorem provers, and present a method by which a theorem prover could verify its own implementation. By applying proof-grounded compilation (i.e., proof-grounded bootstrapping applied to something other than a compiler) to an implementation of a theorem prover, we obtain a theorem prover whose machine-code implementation is verified. To connect this result back to the semantics of the logic of the theorem prover, we need to formalise that semantics and prove a refinement theorem. I present some advances in the techniques that can be used to formalise HOL within itself, as well as demonstrating that the theorem prover, and its correctness proof, can be pushed through the verified compiler.

My thesis is that verification of a realistic implementation can be produced mostly automatically from a verified high-level implementation, via the use of verified compilation. I present a verified compiler and explain how it was bootstrapped to achieve a small TCB, and then explain how verified compilation can be used on a larger application, in particular, a theorem prover for higher-order logic. The application has two parts, one domain-specific and the other generic. For the domain-specific part, I formalise the semantics of higher-order logic and prove its inference system sound. For the generic part, I apply proof-grounded compilation to produce the verified implementation.

Acknowledgements

Foremost, my thanks go to my supervisor, Mike Gordon. Combining extensive experience and interest in verification research, and the corresponding network of contacts, with kind approachability and helpful availability, Mike is the exemplary supervisor every candidate wants. Secondly, I thank Magnus Myreen and Scott Owens for mentoring me with care and consideration, and for providing an intensely rewarding research experience. Magnus was my second supervisor and provided helpful feedback on a draft of this dissertation. I thank the rest of the CakeML team, particularly (alphabetically) Anthony Fox, Michael Norrish, and Yong Kiam Tan, for working on this ambitious (and ongoing) project. I thank Rob Arthan for his expert advice on the semantics of HOL, for interesting discussions, and for his work as a co-author of some of my publications. Finally, I thank Freek Wiedijk for his enthusiastic interest in CakeML and for suggesting improvements (which I used) to this dissertation's title.

Contents

1 Introduction **11**

1.1 Trusted computing base 13

1.2 Verified theorem proving 14

1.3 Verified compilation 14

1.4 ML for verified compilation 16

1.5 Dissertation overview 17

1.5.1 Thesis 17

1.5.2 Contributions 17

1.5.3 Publications 18

1.5.4 Terminology and notation 20

I Self-compilation **23**

2 From verified algorithms to implementations in CakeML **25**

2.1 Evaluation in the logic 26

2.2 Translation from shallow to deep embeddings 27

2.3 CakeML 30

2.3.1 Semantics of the REPL 33

3 A verified compiler for CakeML **37**

3.1 Compiler verification 37

3.2 Data refinement 39

3.3 CakeML bytecode 41

3.4 Divergence preservation 44

3.5 Connecting the pieces 46

4 Bootstrapping the verified compiler **49**
4.1 Compilation as a verified algorithm 50
4.2 Proof-grounded bootstrapping 51
4.3 Packaging a bootstrapped compiler as a REPL 54
4.4 REPL implementation specified as a function in logic 57
4.5 Bootstrapping a function call 60
4.6 Producing verified machine code 64

II Self-verification **69**

5 Formal semantics of HOL **71**
5.1 Approach 71
5.2 Set-theory specification 74
5.2.1 Derived operations 75
5.2.2 Consistency of the specification 77
5.3 Sequents: the judgements of the logic 79
5.3.1 Terms and types 80
5.3.2 Alpha-equivalence 81
5.3.3 Substitution and instantiation 82
5.3.4 Theories 83
5.4 Semantics 84

6 A sound inference system for HOL **91**
6.1 Inference system 91
6.1.1 Inference rules 92
6.1.2 Theory extension 94
6.2 Axioms 97
6.2.1 Embedding logical operators 97
6.2.2 Statement of the axioms 100
6.3 Soundness 101
6.3.1 Inference rules 102
6.3.2 Theory extension 103
6.4 Consistency 107

7 A verified implementation of the inference kernel 113
7.1 The monadic functions 113
7.2 Producing CakeML 116
7.3 Compiling and packaging the kernel 119

8 Related work 121
8.1 Formalising compilation 122
8.1.1 Verified compilers 122
8.1.2 Verified bootstrapping 124
8.2 Formalising logic 125
8.2.1 Higher-order logic 125
8.2.2 Dependent type theory 127
8.2.3 First-order logic 127
8.2.4 Comparison to Stateless HOL 127

9 Conclusion 131
9.1 Trusted computing base 132
9.2 Verified theorem prover 134
9.3 Future work 136

Bibliography 148

Chapter 1

Introduction

How can we know that a computer system will behave correctly? One method is to produce a mathematical proof of correctness. Then the question of system correctness divides in two: is the proof valid? and did we prove the right thing? This dissertation describes progress on both fronts.

Did we prove the right thing? Often, the proof is about a high-level abstract model of an algorithm, but the running system contains a concrete implementation that was written in a programming language and compiled to machine code. Does the proof apply to the machine code that actually runs? That depends on both whether the compiler preserved the semantics of the input program, and whether the program was accurately modelled by the proof.

I present a new *verified compiler* for a realistic ML-like language called CakeML, as well as a new technique, *proof-grounded compilation*, which enables us specifically to bootstrap the compiler and, more generally, to push proofs about high-level algorithms down to the level of real implementations.

There is always a gap between a mathematical model, about which things can be proved, and the physical system to which the model is supposed to correspond. One has to look at the statement of the proved theorem, and assess whether its assumptions are reasonable and its conclusion is strong enough. But we can make the assumptions simpler and smaller by applying proof-grounded compilation, so that the final correctness result is about a more concrete model of the system.

The guiding principle here is establishing *end-to-end correctness*. By replacing trust (in the form of assumptions) with proofs, and then connecting all the

theorems up, we push the boundary of what is guaranteed by formal verification.

Is the proof valid? For proofs about realistic software, the only practical option is to use a theorem prover to produce and to check the proofs. Using a theorem prover, one is more likely to produce valid proofs than if unaided. But a theorem prover is a piece of software that is compiled to machine code before it is run, and sometimes the size and complexity of its inference kernel is comparable to the software that is being verified. Why should the theorem prover be trusted to produce valid proofs? Even if the theorem prover is trustworthy, how do we know the proof system is sound? There is a risk of infinite regress here, and a line must be drawn somewhere. But to placate the tireless sceptic, we can push it back.

I present a proof of *consistency* of higher-order logic (HOL), in particular for the entire inference system implemented by the kernel of the HOL Light theorem prover [24]. The main lemma is a proof of *soundness* against a new specification of the semantics of HOL. This formalisation extends work by Harrison [23] towards self-verification of HOL Light. Using the proof-grounded compilation technique, I show how to produce a concrete implementation of a proof checker for HOL based on the verified inference system. The result is a theorem prover with very strong guarantees of correctness, and, as I will sketch, the rare potential to verify its own concrete implementation in machine code.

Outlook Our goal is practical methodology for producing verified software with fewer of the usual caveats associated with "verified". The usual caveats come in three kinds of things one needs to trust:

- the pathway to execution (compiler, linker) and the execution environment (operating system, hardware),
- the verification methodology (soundness of the theorem prover's logic and implementation), and,
- that the properties you verified mean what you think they do.

The code comprising the first two items is known as the *trusted computing base* (TCB). My work on verified compilers and verified theorem provers is aimed at significantly reducing the TCB for verified software.

1.1 Trusted computing base

The term "trusted computing base" originates in the field of computer security. The idea is simple: what components of the system do we need to trust for the system to be secure? The point is that it may not be the whole system: we can sometimes make an argument for the whole system's security depending only on the behaviour of a trusted kernel. By reducing the size of this trusted component, we reduce opportunities for security breaches.

In the context of verified software, we can extend the trusted computing base idea by switching focus from security to correct behaviour. What do we need to trust for the system to behave correctly? Those things, for us, form the trusted computing base. In general, the TCB divides into the following categories:

- Formal models of parts of the system. We trust these to correspond to the things that are being modeled. For example, we might have a formal model of an x86 processor, and then trust that it is an accurate enough representation of real processors.

- Parts of the system about which we have no detailed models, but about which our verification makes assumptions. Strictly speaking, these could be considered under the previous category, but it is clearer to separate detailed formal models from bare or implicit assumptions. Examples in this category include: the compiler, linker, runtime, and operating system that are used to run a verified algorithm, when the verification covers the algorithm but not the concrete implementation.

- The tools we use to formally prove theorems (more precisely to check the proofs). This item represents philosophical questions about knowledge and proof, but also emerges as a practical concern about the soundness of the toolchain used to create a formal development.

My strategy for making the TCB smaller concentrates on reducing the second item. In particular, I strive to include more within the boundary of what is formally verified, so as to satisfy assumptions with proofs. In the end ideally only assumptions of the first kind are left, together with the same amount of trust in the theorem proving tools. When the software I apply the strategy to is itself a theorem prover, the third item is also reduced.

1.2 Verified theorem proving

What is the point of verifying a theorem prover and formalising the semantics of the logic it implements? One answer is that it raises our confidence in its correctness. A theorem prover implementation usually sits at the centre of the trusted code base for verification work, so effort spent verifying the theorem prover multiplies outwards. Secondly, it helps us understand our systems (logical and software), to the level of precision possible only via formalisation. Finally, a theorem prover is a non-trivial piece of software that admits a high-level specification and whose correctness is important: it is a catalyst for tools and methods aimed at developing complete verified systems, readying them for larger systems with less obvious specifications.

1.3 Verified compilation

Work on verifying a compiler also has the potential to reduce the trusted computing base for many applications. All those applications that will be compiled by the verified compiler and run on the verified runtime system stand to benefit.

Because a compiler both deals with a programming language and is written in a programming language, it can serve as a particularly good illustration of the issues involved when considering the scope of verification and what remains in the trusted computing base. Let us look now at three dimensions on which a compiler verification effort might be assessed.

The first dimension is how far the compiler and its verification go (Figure 1.1). Does the verification cover the entire compilation function that takes in source code (that is, text files) and produces executable binaries as output? Or does the verified function take and produce processed data, and therefore need to be wrapped by unverified tools for parsing, linking, assembly, or even some compilation phases? The verified compiler described in this dissertation goes further on this dimension than typical, by including a verified parser and producing machine code; however, it produces raw machine code to be run on "bare metal", and requires an unverified wrapper to create a loadable object file suitable for a traditional operating system.

The second dimension is how concrete the verified implementation is, that is, how far is it from what is actually run (Figure 1.2). Is the verification just about

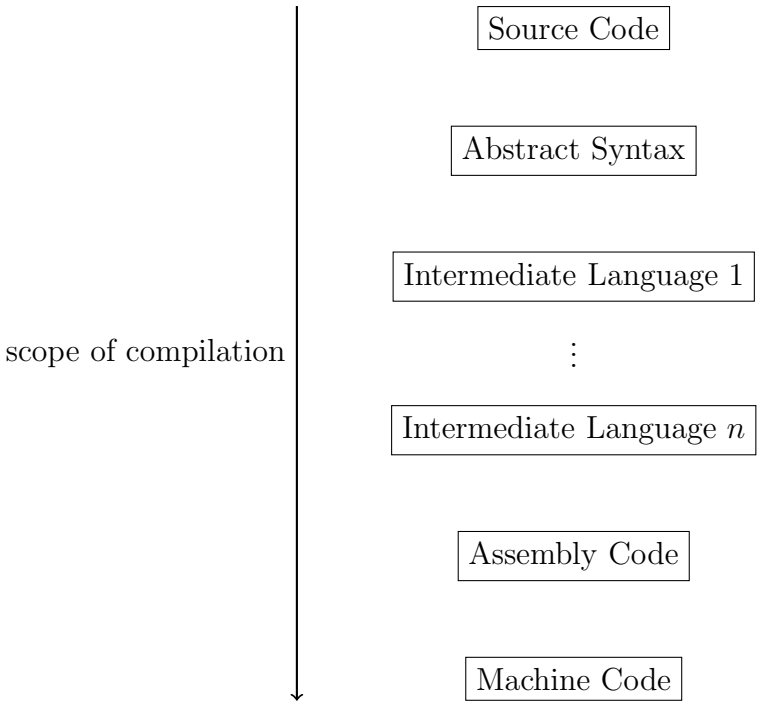

Figure 1.1: One dimension of compiler verification: the size of the gap between the source and target languages.

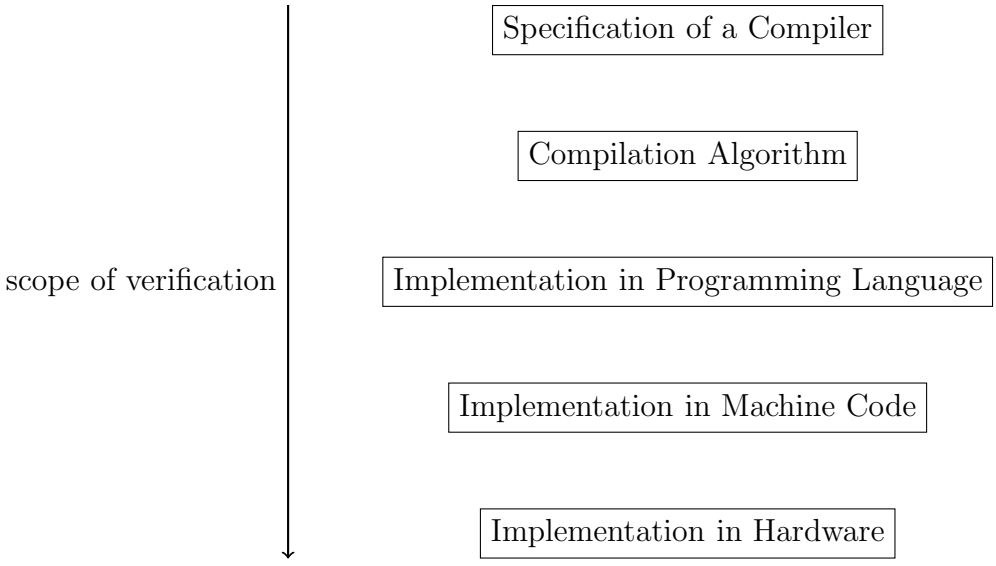

Figure 1.2: Another dimension of compiler verification: how concrete an implementation of the compiler is verified?

the compilation algorithm, or does it also cover the concrete implementation of that algorithm in hardware? Verifying a compiler only reduces trusted code if we do not need to introduce trust for the platform upon which the verified compiler runs. The compiler verification described in this dissertation is about the implementation of the compiler in machine code, which is the lowest level of abstraction we consider; I achieve this level of concreteness without a significant increase in proof effort by using the new technique of *proof-grounded bootstrapping*.

The final dimension by which to assess a verified compiler are the features supported in (and realism of) its source language. Is the compiler useful, because it accepts and produces programs in well-known existing languages, or does it only work with severely simplified or toy languages? The compiler in this dissertation accepts a large subset of an established programming language, Standard ML, and generates code for an established microprocessor architecture (x86-64). And, as we shall see, it has been tested on two reasonably large example programs: a compiler (itself in fact), and the kernel of a theorem prover.

1.4 ML for verified compilation

The choice of an ML-like language as the source language of a verified compiler, and for implementing verified algorithms including a verified theorem prover, is apt for several reasons. Theorem-prover implementation was the motivating application for the original ML [20], which stands for "meta-language" to be contrasted with the "object-language" that is the logic of the theorem prover. Standard ML is unique amongst programming languages (at least those not purely academic) in being defined by formal semantics [53, 54], and is one of few programming languages to have been given formal semantics at all (although the trend is changing, e.g., [29, 49]). (The Definition is formal, but not mechanised like the semantics used in this dissertation. However, mechanised work on ML semantics is not without precedent [30, 77, 45].) Although ML was designed for theorem provers, it is now used for a variety of general-purpose applications (including an operating system [48]), and is therefore a good target for verification. As a higher-order, functional language, ML is close enough to higher-order logic to be a good vehicle for expressing algorithms and to allow a connection between ML and HOL to be formally established (as described in Chapter 2).

1.5 Dissertation overview

This dissertation is a distillation of many of the important ideas from an ongoing substantial research project, the CakeML project, hosted at `https://cakeml.org`. The basic aim of the project is to push the boundaries of formal verification. In particular, to push in these three directions: the quality (size, scope, speed) of the artefacts that we can verify, the ease with which we can do it, and the quality (reducing trust, increasing guarantees) of the verification.

We focus on two of the results of the project: a verified read-eval-print loop (REPL), which packages a verified compiler for CakeML, and a verified theorem prover designed to run on this verified REPL. The key ideas include verified compilation of a functional language, proof-grounded compilation to push correctness claims down to the implementation level, and self-formalisation of higher-order logic. These ideas culminate in the following claim.

1.5.1 Thesis

Verification of a realistic model of a computer system does not require significantly more trust or effort than verification of a high-level abstract model. I propose the use of proof-grounded compilation from a language suitable for high-level verification to automatically produce verified implementations without increasing the trusted computing base.

1.5.2 Contributions

The specific contributions of this dissertation are as follows.

1. A verified compiler for CakeML that can bootstrap itself. The input language is a substantial subset of an established programming language (Standard ML), and the compiler produces real machine code (that is, numbers encoding instructions) for an established instruction set architecture (x86-64).

2. A new technique, *proof-grounded compilation*, for using a verified compiler to push correctness results about algorithms down to the level of the concrete implementation. We apply this technique to the CakeML compiler itself, where it becomes *proof-grounded bootstrapping*. We also explain how

a bootstrapped verified compiler can be *packaged* into a larger verified program (in particular a read-eval-print loop) with strong guarantees of correctness.

3. Formal semantics and a proof of soundness for an inference system for higher-order logic (HOL) against a new specification of set theory. Then, verification of a theorem-prover kernel for HOL, showing that it implements the semantics of the logic. Finally, a description of how proof-grounded compilation can be applied to this verified theorem-prover kernel.

All of the theorems mentioned in this dissertation have been machine-checked using the HOL4 theorem prover. At the time of writing, the CakeML repository contains roughly 100,000 lines of Standard ML including definitions, proof scripts, and proof automation, but not including the standard theories and libraries of HOL4 on which they depend.

1.5.3 Publications

Much of the work presented in this dissertation has, in earlier forms, been published in peer-reviewed conference proceedings and journals. A list of relevant publications follows.

- Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014

- Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL Light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 490–495. Springer, 2013

- Ramana Kumar. Challenges in using OpenTheory to transport Harrison's HOL model from HOL Light to HOL4. In Jasmin Christian Blanchette and Josef Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9-10, 2013*, volume 14 of *EPiC Series*, pages 110–116. EasyChair, 2013

- Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In Klein and Gamboa [37], pages 308–324

Additionally, an article based on selections of Part I (especially Chapter 4), and another based on Part II, have been submitted for review. These as-yet-unpublished papers are:

- Ramana Kumar, Magnus O. Myreen, Scott Owens, and Yong Kiam Tan. Proof-grounded bootstrapping of a verified compiler: Producing a verified read-eval-print loop for CakeML. *Journal of Automated Reasoning*, 2015. Submitted

- Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic. *Journal of Automated Reasoning*, 2015. Submitted

The papers above, and the work they describe, is the result of collaboration with the listed co-authors in the context of the CakeML project. Since this dissertation describes results in the CakeML project, it builds upon work done in collaboration. The dissertation itself, and the bulk of the research described within, is my own work.

Broadly speaking, work on CakeML was originally divided as follows: Scott Owens worked on the semantics of the language and the type system, Magnus Myreen worked on proof-producing translation into CakeML and machine-code verification of the implementation of the bytecode, Michael Norrish worked on verified parsing, and I worked on verified compilation from abstract syntax to bytecode, and the top-level compiler correctness results required for bootstrapping. For the verified HOL Light example, my focus was the semantics and soundness of HOL, while Magnus's focus was verification of the monadic kernel functions. Over the course of the project, these divisions were not held up strictly, and everyone has worked in some way on most aspects of the project. A detailed record can be found in the commit history of the code base, available via `https://github.com/CakeML/cakeml`.

1.5.4 Terminology and notation

I sometimes refer to "the logic", by which I usually mean higher-order logic (HOL) as implemented by the HOL4 theorem prover. Theorem provers are computer programs that aid in the construction of formal proofs (i.e., derivations of syntactically valid sentences) in some logic. The (programming) language in which a theorem prover itself is written is typically called the metalanguage. For the HOL4 theorem prover, the logic is HOL and the metalanguage is Standard ML.

As my work involves both refinement proofs (linking implementations to specifications) and soundness proofs (linking an inference system to the semantics of a logic), and for the latter both the meta-logic and object-logic are HOL, I often need to refer to similar but different things and some care must be paid for clarity. By "HOL" I refer to higher-order logic itself. Particular inference systems for HOL are implemented by interactive theorem-proving systems. The two theorem provers of interest are HOL Light, because I formalise its inference system and use its implementation as inspiration for the implementation in CakeML; and HOL4, because I use it to mechanise the proofs. I use "HOL4" and "HOL Light" unqualified to refer to the theorem provers, and clarify explicitly when I mean the inference system instead. HOL4 implements a different inference system from HOL Light's, but the two are inter-translatable.

The dissertation includes extracts, generated by HOL4, from the formal proofs. These include definitions. For example, here is the standard library function for checking a predicate holds for all elements of a list:

$$\begin{aligned}&\mathsf{every}\ P\ [\,] \iff \mathsf{T}\\ &\mathsf{every}\ P\ (h{::}t) \iff P\ h \wedge \mathsf{every}\ P\ t\end{aligned}$$

Since the result of $\mathsf{every}\ P\ ls$ is Boolean, we use ($\iff$) in the defining equations; at other types we simply use (=). As well as definitions we have theorems, which are shown with a turnstile, for example:

$$\vdash \neg\mathsf{every}\ P\ ls \iff \mathsf{exists}\ ((\neg) \circ P)\ ls$$

Free variables may appear in theorems; semantically, they behave as if universally quantified. Datatype definitions are shown as in the following example of the

polymorphic option type with two constructors:

$$\alpha\ \mathit{option}\ =\ \mathsf{None}\ |\ \mathsf{Some}\ \alpha$$

Terms are sometimes annotated with their types, for example: $(\mathsf{Some} : \mathit{bool} \to \mathit{bool\ option})\ \mathsf{F}$. Quantifiers are printed as binders, as in $\forall x.\ \exists y.\ x \neq \mathsf{Some}\ y$, although in HOL the quantifiers are ordinary constants (that operate on predicates, that is, functions with codomain *bool*). The existential quantifier in the previous sentence might more pedantically be printed as an application of $(\exists)$ to $\lambda y.\ x \neq \mathsf{Some}\ y$. Finally, we show the rules of inductive relations using a horizontal line to separate premises from the conclusion. Thus the rule, $\vdash R\ x\ y \wedge R^*\ y\ z \Rightarrow R^*\ x\ z$, about the reflexive transitive closure of a relation can also be written as follows:

$$\frac{R\ x\ y \qquad R^*\ y\ z}{R^*\ x\ z}$$

Part I

Self-compilation

Chapter 2

From verified algorithms to implementations in CakeML

I make a distinction between algorithms and implementations, which is not always emphasised in other work on verification. An algorithm is a formally specified procedure whose semantics is implicit and mathematical. Using terminology introduced by Boulton et. al. [12, Section 4], an algorithm is a *shallow embedding*, which might be modelled by a function that is defined in the logic of a theorem prover and inherits the semantics of the logic. Implementations, on the other hand, are *deeply embedded*: they are syntax with an explicit formal semantics, for example the operational semantics of a programming language or the next-state relation of a processor model. I make this distinction because a key theme of my work is moving from verified algorithms to verified implementations, which I see as finishing the task intended by the algorithm verification in the first place.

In this chapter, we look at two techniques that can be applied to verified algorithms that will play key roles in the proof-grounded compilation idea introduced in Chapter 4. The first technique, evaluation by rewriting in the logic [6], operates solely on shallow embeddings. The second technique, proof-producing translation from shallow to deep embeddings [66], is fundamentally concerned with deep embeddings. The programming language we use for deep embeddings is CakeML [42], which is described briefly at the end of this chapter.

Evaluation in the logic (henceforth "evaluation") and proof-producing translation to a deep embedding (henceforth "translation"[1]) are both examples of proof

[1]Although compilation is a kind of translation (from a high- to a low-level language), we

automation that can be implemented in the context of a general-purpose theorem prover such as HOL4 [74] (which I use), Isabelle [82], or Coq [10]. Theorem provers (like those three) written in the LCF style [52] produce theorems only by checking the proof steps in a small "kernel" that implements the primitive inference rules of the logic. Sophisticated proof automation, like evaluation or translation, does not demand additional trust since any theorems produced by the automation have been pushed through the theorem prover's kernel.[2]

2.1 Evaluation in the logic

Let us begin with an example of the kind of proof task I mean to be solved by evaluation. Given input $\mathsf{map}\ \mathsf{length}\ [[1;\ 1];\ [2];\ [\,]]$, we wish to produce the theorem

$$\vdash \mathsf{map}\ \mathsf{length}\ [[1;\ 1];\ [2];\ [\,]] = [2;\ 1;\ 0]$$

by evaluation using the definitions of map and length. The key characteristic is that the right-hand side of the theorem contains no more reducible expressions: it is a normal form in the rewriting system consisting of the function definitions and beta-conversion. The theorem should be produced automatically and efficiently.

The solution, introduced to HOL4 by Barras [6], is to interpret the equations characterising functions like map (shown below) as they would be by an interpreter for a functional programming language.

$$\begin{aligned}&\mathsf{map}\ f\ [\,] = [\,]\\ &\mathsf{map}\ f\ (h{::}t) = f\ h{::}\mathsf{map}\ f\ t\end{aligned}$$

Each reduction step performed by such an interpreter can be justified by a (derived) rule of inference and replayed in the inference kernel, thanks to the kernel's semantics of equality and support for beta-conversion. Logically speaking, evaluation is no more sophisticated than rewriting (or simplification) as described, for example, by Paulson [69]. The difference is in the order in which rewrite rules are applied (bottom-up versus top-down) and in the book-keeping done to make the

reserve the term "translation" for moving from a shallow to a deep embedding.

[2]The kernels of these systems vary, however, in size. Coq, for example, includes some facilities for evaluation within the kernel that would need to be implemented outside in other systems.

process more efficient. Although Barras's evaluation supports variables, for our purposes we need only consider evaluation problems, like the one above, where the input term has no free variables.

The equations characterising map above have the same status (proven theorems) as the theorem produced by evaluation. The fact that they can be viewed as *defining* equations does not distinguish them, in HOL, from any other equations. Indeed, any suitable rewrites that have been proved about a function can be used in the evaluation of that function. The resulting theorems, produced by evaluation, are proved using only the normal rules of the inference kernel, without any recourse to evaluation or compilation outside the logic, or purpose-built[3] inference rules for normalisation.

2.2 Translation from shallow to deep embeddings

The defining equations for map in the previous section are an example of a *shallow embedding* of a functional program in logic. The constant map is a function in HOL with type $(\alpha \rightarrow \beta) \rightarrow \alpha\ \mathit{list} \rightarrow \beta\ \mathit{list}$. Despite the evaluation machinery just described, the semantics of map is not operational; map is a mathematical function and has semantics according to the semantics of HOL. Indeed, there are HOL functions[4] that do not have any operational characterisation.

For functions like map which do have equations suitable for evaluation, in the sense of functional programming, there is an alternative way to model the function in logic. That alternative is to use a *deep embedding*: to model the function as a piece of syntax, animated by an explicit evaluation relation describing the operational semantics of a programming language. In our examples, the programming language for deep embeddings is always CakeML [42].

Consider the following definition of the syntax for the map function (this is

[3] One feature was added to the kernel, when evaluation was implemented, to improve the performance: the kernel datatype implementing HOL terms supports lazy substitution.

[4] For example, the existential quantifier over an uncountable type.

CakeML abstract syntax; it is pretty-printed underneath):

```
map_dec =
  Letrec
   [("map","v3",
    Fun "v4"
     (Mat (Var "v4")
       [(Pcon "nil" [],Con "nil" []);
        (Pcon "::" [Pvar "v2"; Pvar "v1"],
         Con "::"
          [App [Var "v3"; Var "v2"]; App [App [Var "map"; Var "v3"]; Var "v1"]])]))]
```

The syntax is more readable as pretty-printed concrete syntax:

```
fun map v3 =
  Fun "v4"
    (case v4
     of [] => []
     |  (v2::v1) => (v3 v2::(map v3 v1)));
```

The type of map_dec in HOL is *dec* (a CakeML declaration). Thus, it is not a HOL function and does not get its functional semantics that way. Rather, the semantics is given explicitly by an evaluation relation EvalDec env_1 *dec* env_2 that relates a declaration *dec* and an initial environment env_1 (e.g., containing the datatype declaration for lists) to a resulting environment env_2. The resulting environment for the map_dec declaration will include a binding of a new variable, called "map", to a function value (i.e., a closure).

If we want to prove something about map, working directly with the syntax and evaluation relation (operational semantics) is much more cumbersome than using the defining equations of the shallow embedding directly. However the extra machinery of the deep embedding (e.g., the environment and the explicit evaluation steps) make it a more realistic formalisation of map as a functional program. Fortunately, we can do our reasoning on the shallow embedding and carry any results over to the more realistic deep embedding automatically using a technique developed by Myreen and Owens [66], called (proof-producing) *translation*.

Translation synthesises a deep embedding following the structure of the shallow embedding's equations and simultaneously proves a *certificate theorem* about

the synthesised implementation. Synthesis happens in a bottom-up manner, using the certificate theorems for previously translated code as required. The certificate theorem is proved automatically, using the shallow embedding's induction theorem (typically proved automatically when the shallow embedding is defined) and relates the behaviour of the synthesised implementation to its shallow counterpart.

To explain certificate theorems, let us work through understanding the following one for map by taking it apart.

Example 1 (Certificate theorem for map)**.**

$$\begin{aligned}&\vdash \exists\, env\ c.\\ &\quad \mathsf{EvalDec\ InitEnv\ map\_dec}\ env \wedge \mathsf{Lookup\ "map"}\ env = \mathsf{Some}\ c \wedge\\ &\quad ((a \longrightarrow b) \longrightarrow \mathsf{ListTy}\ a \longrightarrow \mathsf{ListTy}\ b)\ \mathsf{map}\ c\end{aligned}$$

There are two important concepts contained in such a certificate theorem: refinement invariants (e.g., ListTy a) and the operational semantics (EvalDec). A refinement invariant specifies the relationship between between a shallowly-embedded value (a HOL term) and a deeply-embedded one (a CakeML value). For example, ListTy BoolTy [F] v holds when v is a CakeML value implementing the singleton list containing the HOL constant F (falsity) according to the refinement invariant ListTy BoolTy. Expanding out what this means explicitly, we have:

$$\begin{aligned}&\vdash \mathsf{ListTy\ BoolTy}\ [\mathsf{F}]\ v \iff\\ &\quad v =\\ &\quad\ \mathsf{ConV}\ (\texttt{"::"},\mathsf{TypeId}\ \texttt{"list"})\\ &\quad\quad [\mathsf{ConV}\ (\texttt{"false"},\mathsf{TypeId}\ \texttt{"bool"})\ [\,];\ \mathsf{ConV}\ (\texttt{"nil"},\mathsf{TypeId}\ \texttt{"list"})\ [\,]]\end{aligned}$$

Here, ConV *name args* represents a deeply-embedded constructor value.

Since lists are polymorphic, ListTy takes as an argument a refinement invariant to govern the type of the list elements. In the certificate theorem for map above (Example 1), there are free variables a and b standing for refinement invariants for the input and output list elements. The free variables show us that the certificate theorem applies to every instance of the polymorphic map function.

The full refinement invariant for map includes several instances of the refinement invariant, $x \longrightarrow y$, for functions (there are several instances because

map is both higher-order and curried). Given a HOL function f and refinement invariants x and y intended to describe the input and output types of f, the $(x \longrightarrow y)\ f\ c$ invariant holds when c is a CakeML closure that implements f. More specifically, whenever $x\ v_1$ holds, then application of the closure c to v_1 will, according to the CakeML operational semantics, terminate with a value v_2 that satisfies $y\ (f\ x)\ v_2$. Looking back at the refinement invariant for map in its certificate theorem, we see that map is implemented as a closure which, when given CakeML values satisfying $(a \longrightarrow b)\ f$ and ListTy $a\ l$ as inputs will terminate and produce a CakeML value satisfying ListTy b (map $f\ l$).

The certificate theorem for map is written in terms of the operational semantics, namely EvalDec. In general EvalDec env_1 dec env_2 is the assertion that the CakeML declaration dec evaluates in environment env_1 successfully and without side-effects[5] to produce the extended environment env_2. Thus for map, we see that in initial environment InitEnv, the map_dec declaration will succeed and the resulting environment, env, will bind the variable `"map"` to a closure, c, implementing map. It is not particularly important that we start in the InitEnv environment, which contains only CakeML primitives: a more general form of the certificate theorem (not shown) allows us to derive a similar result for any starting environment.

The proof-producing translation technique includes support for user-defined datatypes as well as the primitive datatypes of CakeML (Booleans, lists, etc.). The result of defining an algebraic data type in HOL provides enough information to synthesise refinement invariants (like ListTy a) for new types. There is also some support (mainly namespace management) for translation into a named CakeML module. Details on the workings of the proof automation behind translation can be found in Myreen and Owens [66]. Proof-producing translation plays a key role in bootstrapping the CakeML compiler, which is itself written in HOL but whose input language is CakeML.

2.3 CakeML

In the previous section we saw examples of CakeML being used as the target language for proof-producing translation. CakeML is a formally defined pro-

[5]Certificate theorems for programs with side-effects are more complicated, but will not concern us until Section 4.5.

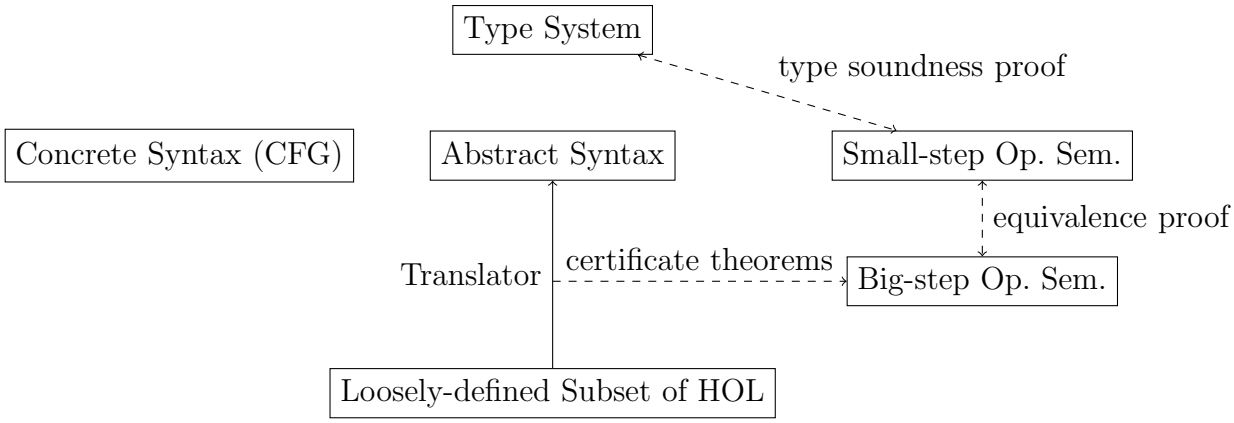

Figure 2.1: Semantics and translator for CakeML. In the centre is the abstract syntax of CakeML around which is defined concrete syntax (left), a static semantics (top) and dynamic semantics (right). At the bottom is the proof-producing translator that generates CakeML implementations of functions in HOL together with a proof of correctness (in terms of the CakeML semantics).

gramming language modelled after Standard ML. The CakeML semantics was originally formalised by Owens [65] in a style similar to his semantics for OCaml Light [68]. The pieces making up the semantics and (shallow-to-deep) translator for CakeML are illustrated in Figure 2.1.

The semantics of CakeML is an interface. It is used by two different clients: the proof-producing translator from HOL to CakeML (described in the previous section), and the verified compiler from CakeML to bytecode (described in Chapter 3). The central notion in the operational semantics is evaluation of a CakeML program to a result or an error. The translator produces certificates describing how the CakeML code it produces evaluates, whereas the compiler verification proves that the bytecode it generates evaluates the same way as the CakeML program given as input.

Apart from its role as an interface, the semantics of CakeML is an interesting artefact in its own right, since it aims to be a formal, mechanised, and useful semantics for a realistic programming language. Various metatheoretic results have been proved about the semantics. As shown in Figure 2.1, the semantics includes a type system and the operational semantics has been proved sound with respect to the type system. In other words, the language is safe: a well-typed terminating program always evaluates to a result of the same type or raises an exception.

The CakeML semantics detects type errors explicitly. The operational semantics of a program that would be ruled out by the type system is to raise a type error. Therefore, the language also has an untyped safety theorem: the semantics of a program never gets stuck. It either diverges, terminates with a result or an exception, or terminates with a type error.

Explicit type errors combine well with another property of the semantics, which together enable our strategy for proving compiler correctness for non-terminating programs as well as terminating ones (Section 3.4). The property is determinism: there is exactly one result for every terminating program. The programs with no big-step semantics, therefore, are exactly the programs that diverge. This fact has been proved using the more straightforward definition of divergence in terms of infinite traces in the small-step semantics.

A final design decision worth a small mention is to do with bound variables and function application. CakeML abstract syntax uses a first-order representation of binding with variables represented by strings. CakeML values include closures and the semantics uses environments as opposed to doing substitution for function application. This approach fits well with the goal of compiler verification, since the compiler produces closures. And making sure the abstract syntax for CakeML is straightforwardly represented as a datatype within CakeML is crucial for bootstrapping the compiler.

CakeML is a language intended for real use, not merely an academic proof-of-concept. At the time of writing, the major missing feature is support for input/output (I/O), beyond the top-level I/O of a read-eval-print loop. It is expected that I/O will be added to CakeML in the near future. The features already present include

- higher-order, polymorphic, recursive functions,
- user-defined algebraic datatypes and nested pattern matching,
- a module system with support for signatures,
- references,
- exceptions, and
- primitive types including: arbitrary precision integers, strings, characters, vectors, and arrays.

While all these features have received attention in research on programming languages, the combination within a single language with a formal, mechanised semantics is unusually ambitious.

2.3.1 Semantics of the REPL

The top-level semantics of the CakeML read-eval-print loop (REPL) is defined in terms of the specification of the syntax of the language as a context-free grammar, the type system specified as an inductive relation, and the operational semantics also specified as an inductive relation. The pieces making up the semantics are all specified using conventional techniques.

Recall the semantics of CakeML declarations: EvalDec env_1 *dec* env_2 holds when the semantics of processing the declaration *dec* in environment env_1 is to produce a new environment env_2. The semantics for a read-eval-print loop (REPL) is to read and evaluate declarations in a loop, printing the additional bindings in the new environment after each one. Since we only care about the output printed by the REPL after processing a declaration, I leave out the details of the semantics for stateful and failing declarations that are present in the full CakeML semantics.

We must, however, properly model divergence. As mentioned previously, the semantics is careful to cover *all* non-diverging declarations, marking failures with explicit errors, hence if a declaration has no semantics it must diverge. Additionally, this equivalence between failure to relate and divergence has been proved in terms of the CakeML small-step operational semantics, where divergence can be specified in the normal way (as an infinite trace).

To specify the output of the REPL, we use the following type which encodes a list of result strings ending in either termination or divergence.

repl_result = Terminate | Diverge | Result *string repl_result*

Each result is the output from a declaration: it could indicate a parse error, a type error, an exception, or some new bindings. If some declaration diverges, the REPL result ends there with Diverge; otherwise it continues until there are no more declarations and ends with Terminate.

The input of the REPL is specified as a string containing all user input. In

reality, later parts of the user input are likely to depend on the REPL's output for earlier parts. But since we do not model the user at all, apart from the input they actually produce, it is convenient to assume we have all the input up front, akin to an oracle.

The concrete syntax for CakeML requires that every declaration end with a semicolon. Consequently, the input string can be split, after lexing, into lists of tokens each representing a declaration. To specify the semantics of lexing, we have executable specifications (Lex and SplitSemicolons) of the conversion to tokens and splitting at semicolons. For the semantics of parsing, the (non-executable) function Parse checks whether there exists a parse tree for a declaration in the CakeML grammar whose fringe is the given list of tokens, and returns Some *dec* if so, otherwise None. The semantics of the entire REPL, shown below, can thus be factored through a semantics (AstReplSem) that operates on abstract syntax.

ReplSem *state input* = AstReplSem *state* (map Parse (SplitSemicolons (Lex *input*)))

Let us look now at the AstReplSem relation, of which the signature is shown below.

AstReplSem *state dec_options repl_result*

The first argument is the state of the REPL semantics, in particular that means the state of the type system (the types declared so far) and of the operational semantics (the current environment and store). As we saw above, ReplSem is parameterised by an initial state thereby allowing a basis program before user input.

With our model of what a REPL result looks like, the definition of AstReplSem is a straightforward loop down the list of input declarations. For each declaration in the list *dec_options*:

1. if it is None accumulate a parse-error result, otherwise

2. if the declaration is not well-typed according to the type system, accumulate a type-error result, otherwise

3. if the operational semantics of the declaration is to diverge, end with the Diverge result, otherwise

4. accumulate the (exceptional or normal) result of the operational semantics of the declaration, update the state (with the new results from the operational semantics and type system), and continue.

We will return to this specification of the REPL semantics in Chapter 4, where it corresponds to the top layer of Figure 4.1. Before looking at the REPL implementation, however, we look at one of its primary components: a verified compiler for individual declarations, which is the topic of the next chapter.

Chapter 3

A verified compiler for CakeML

In this chapter, I describe a verified compilation algorithm for CakeML. I focus on the function in the logic, but not the promised verified implementation in machine code; for that, see the next chapter. The purpose of splitting the material into two parts is to highlight the distinction between different levels of implementation, and to encourage the habit of asking, about verified software, what exactly is verified. Also, proof techniques by which the compilation algorithm is verified (essentially: refinement proofs by induction on the semantics) have a different flavour from the bootstrapping process described in the next chapter.

3.1 Compiler verification

A compiler is a program for translating code from a high-level language to a low-level language, and the property usually considered to constitute its correctness is *semantics-preservation*. We define the CakeML compiler as a function in the logic, since that is the natural place for carrying out verification; the compilation algorithm is defined as a shallow embedding (like `map` in the previous chapter). Such a shallow embedding, together with a correctness theorem, is what is typically meant by a "verified compiler", for example the CompCert verified compiler [46] is a verified algorithm in our terminology. CompCert is run by being extracted to OCaml (which is unverified). In the next chapter, we show how bootstrapping enables us to verify a much more concrete implementation of the CakeML compiler.

To verify the compiler, we need semantics for both the high-level and low-

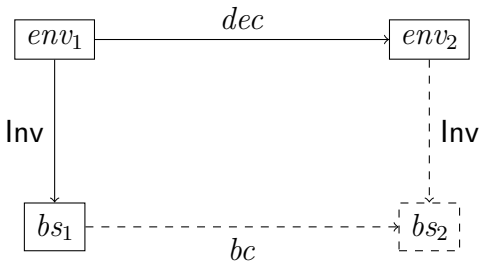

Figure 3.1: Compiler correctness (Lemma 1) illustrated as a commuting diagram. On the top is evaluation of *dec* in the CakeML operational semantics. On the bottom is evaluation of bytecode, *bc*, resulting from compiling *dec*. Lemma 1 states that the dashed lines exist whenever the solid ones do.

level languages. We have seen examples of (a simplified version of) the semantics for CakeML in the previous chapter, in particular EvalDec env_1 *dec* env_2 which specifies the evaluation of a declaration. In this chapter, our target language is CakeML bytecode (Section 3.3), which a low-level language serving as the final step before the machine-code verification in the next chapter. The semantics of CakeML bytecode is given as a state-transition system, $bs_1 \rightarrow^* bs_2$, which means bytecode-machine state bs_1 transitions to state bs_2 in zero or more steps. The bytecode-machine states (explained more thoroughly in Section 3.3) contain code and a program counter, as well as the current state of the memory.

A call to the compiler looks like this: CompileDec cs_1 *dec* = (cs_2,*bc*), where cs_1 and cs_2 are the compiler's internal state and *bc* is the generated bytecode. Because we eventually want to call the compiler multiple times in succession (for the REPL), we prove preservation not just of semantics of the input program but of an invariant, Inv *env cs bs*, between the environment *env* in the CakeML semantics, the compiler's state *cs*, and the bytecode-machine state *bs*. This is an example of *forward simulation* [16].

The compiler correctness theorem states that if the invariant holds for an environment env_1, and the semantics of *dec* in that environment produces env_2, then the compiled code for *dec* will run to completion and the invariant will hold again in env_2. The statement is illustrated in Figure 3.1, and printed formally below.

Lemma 1 (Correctness of CompileDec for successful declarations)**.**

$$\vdash \mathsf{Inv}\ env_1\ cs_1\ bs_1 \wedge \mathsf{EvalDec}\ env_1\ dec\ env_2 \wedge \mathsf{CompileDec}\ cs_1\ dec = (cs_2, bc) \Rightarrow \exists\, bs_2.\ (\mathsf{AddCode}\ bs_1\ bc) \rightarrow^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \mathsf{Inv}\ env_2\ cs_2\ bs_2$$

This form of compiler correctness theorem is only suitable for source programs that terminate successfully (as implied by the EvalDec assumption). For bootstrapping, that is the important case, since we know compilation of the compiler will terminate successfully. The CakeML compiler is, however, also verified for the cases of diverging and failing input programs. We will need the full correctness theorems (Theorems 2 and 3 in Section 3.5) when we want to verify the read-eval-print loop (Section 4.3) which runs the verified compiler on user input.

The principal method for producing our compiler correctness theorem is induction on the big-step semantics of the input program. Compilation proceeds through a series of intermediate languages, as does its verification. For each intermediate language, there are three items required to state the correctness theorem: a big-step operational semantics, a compiler into the language, and a data refinement relation. In the proceeding sections, I will explain the key design decisions made for these items, then how the theorems for all the compiler phases come together as a single comprehensive correctness theorem.

3.2 Data refinement

Each of the features of CakeML which make it a convenient language to program in but which do not have a direct implementation in bytecode need to be compiled away. The sequence of intermediate languages, each of which is aimed at removing one or more such features, is illustrated in Figure 3.2.

The shape of the correctness theorem for the compiler into each intermediate language is that of a forwards simulation theorem for the generated code. A template for the correctness theorem for compilation from intermediate language k to intermediate language $k+1$ is shown below.

$$\mathsf{EvalExp}_k\ env_k\ exp_k\ res_k \wedge \mathsf{Inv}_k\ env_k\ cs_k\ env_{k+1} \wedge \mathsf{CompileExp}_k\ cs_k\ exp_k = (cs_{k+1}, exp_{k+1}) \Rightarrow \exists\, res_{k+1}.\ \mathsf{EvalExp}_{k+1}\ env_{k+1}\ exp_{k+1}\ res_{k+1} \wedge \mathsf{ResRel}_{k+1}\ res_k\ res_{k+1}$$

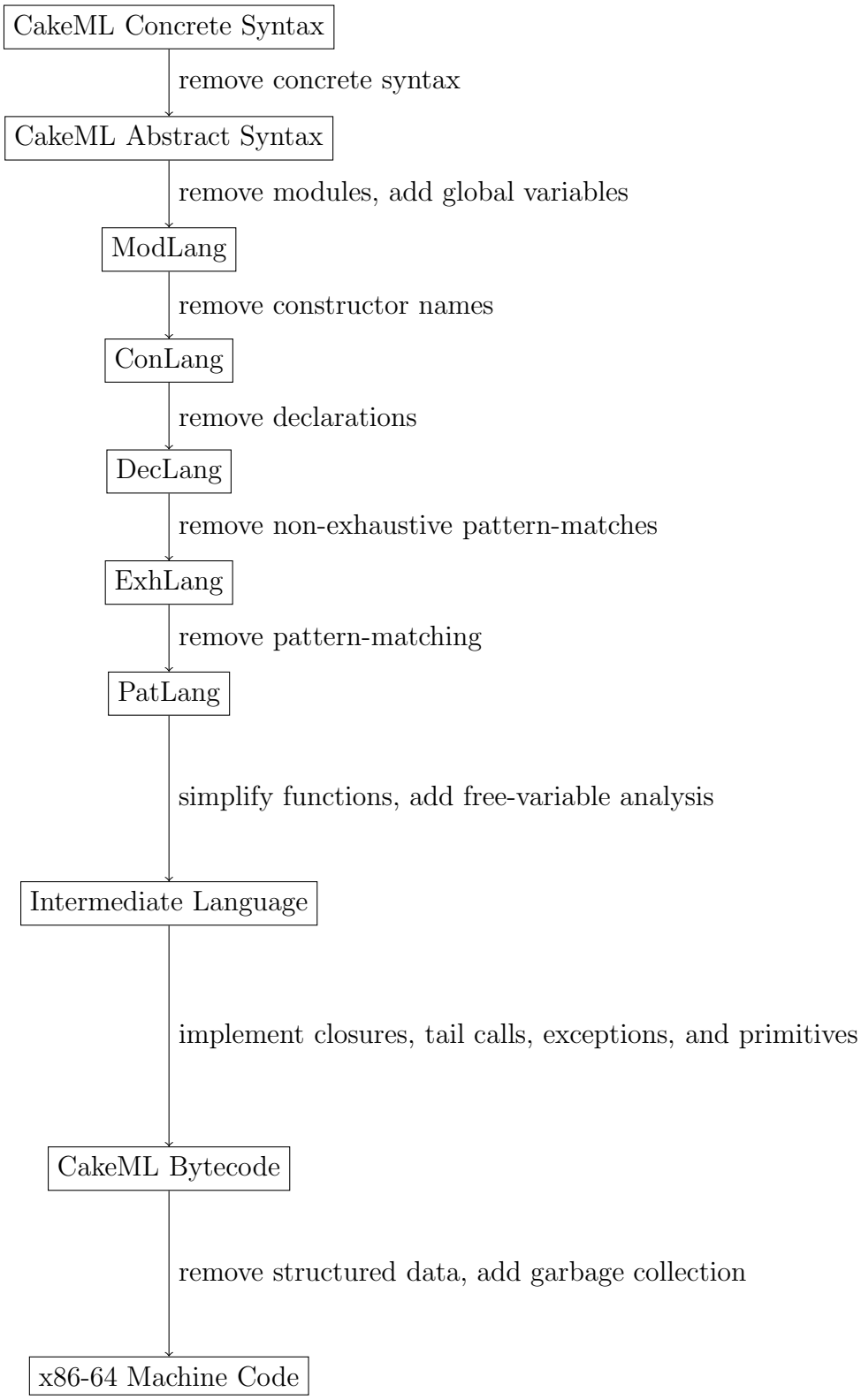

Figure 3.2: Compiler phases

Both the big-step operational semantics for language k ($\mathsf{EvalExp}_k$) and the compiler for language k ($\mathsf{CompileExp}_k$) are defined inductively over the syntax of language k, so it is natural to perform the proofs of these theorems by rule induction on the operational semantics. The part requiring ingenuity that is specific to the features in languages k and $k+1$ is the data refinement relation between values of both languages, which appears in the form of Inv_k relating semantic environments and ResRel_{k+1} relating semantic results.

CakeML is a functional programming language: its values include higher-order functions. The presence of closures in the source language make data refinement significantly more involved than for first-order languages, because closure values include code: data refinement in this setting necessarily involves program refinement too. From the semantic perspective, the best refinement relation would be something like contextual equivalence generalised to work across different languages. But establishing contextual equivalence, as would be required to use the inductive hypotheses in a compiler correctness proof, is notoriously difficult.

Fortunately, for compiler verification, semantics-laden approaches to program refinement are unnecessary. I instead take the opposite approach of a completely syntax-based relation for program refinement, typically using the compiler itself to do the syntax transformation. For data refinement of closures, this means a closure in language $k+1$ refines a closure in language k if its body is simply the result of compiling the body in language k. This kind of relation is much less permissive than contextual equivalence, but more convenient for compiler verification.

Once closures are compiled away, data refinement relations become much simpler. We turn attention next to the last intermediate language that still retains structured data.

3.3 CakeML bytecode

The purpose of CakeML bytecode[1] is to abstract over the details of data representation as machine words, and to hide the garbage collector, while being sufficiently low level for most of its instructions to map directly to small snippets of x86-64 machine code.

[1]The original bytecode instruction set was designed by Magnus Myreen, and evolved as the CakeML compiler was developed.

bc_inst	::=	Stack *bc_stack_op* | PushExc | PopExc
	|	Return | CallPtr | Call *loc*
	|	PushPtr *loc* | Jump *loc* | JumpIf *loc*
	|	Ref | Deref | Update | Print | PrintC *char*
	|	Label n | Tick | Stop | ...
bc_stack_op	::=	PushInt *int* | Pop | Pops n
	|	Load n | Store n
	|	Cons n | El | TagEq n | IsBlock | LengthBlock
	|	Equal | Less | Add | Sub | Mult | Div | Mod
loc	::=	Lab n | Addr n
n	=	*num*

Figure 3.3: CakeML bytecode instructions.

In support of data abstraction, bytecode values do not explicitly model pointers into the heap but instead provide structured data (Cons packs multiple bytecode values *vs* into Block *tag vs*) on the stack. Similarly, the bytecode provides mathematical integers (Number i) as values on the stack, abstracting over the representation as either small integers (that fit in a machine word) or pointers to heap-allocated big integers. Apart from blocks and integers, the only other bytecode values are special-purpose pointers into the heap (RefPtr p, for implementing references), into the code heap (CodePtr p, for closures and dynamic jumps), and into the stack (StackPtr p, for implementing exceptions).

The bytecode semantics is a deterministic state machine, operating over bytecode machine states, bs, that contain code (bs.code), a program counter (bs.pc), and a list of bytecode values (bs.stack). The state transition relation, $bs_1 \rightarrow bs_2$, fetches the instruction in the program counter and updates the state according to its semantics. A selection[2] of bytecode instructions are shown in Figure 3.3, and a selection of their semantics in Figure 3.4.

Since $bs_1 \rightarrow bs_2$ (and hence $bs_1 \rightarrow^* bs_2$) is deterministic, we can define a function in the logic, EvalBC bs_1, that returns the result of repeatedly stepping the semantics until no further step is possible, which occurs when there is no applicable rule for the next instruction either because the machine was mis-configured or the next instruction is Stop. If bytecode evaluation of bs_1 eventually stops, then EvalBC bs_1 = Some bs_2 for the unique final state bs_2. If, however, there is

[2]Not shown, for simplicity, are instructions supporting additional primitive types (characters and byte arrays) and global variables.

$$\frac{\mathit{fetch}\ bs = \mathsf{Stack}\ (\mathsf{Cons}\ t) \quad bs.\mathsf{stack} = \mathsf{Number}\ n :: vs\ @\ xs \quad \mathsf{length}\ vs = n}{bs \rightarrow (\mathsf{bump}\ bs)\{\mathsf{stack} = \mathsf{Block}\ t\ (\mathsf{reverse}\ vs) :: xs\}}$$

$$\frac{\mathit{fetch}\ bs = \mathsf{Return} \quad bs.\mathsf{stack} = x :: \mathsf{CodePtr}\ ptr :: xs}{bs \rightarrow bs\{\mathsf{stack} = x :: xs;\ \mathsf{pc} = ptr\}}$$

$$\frac{\mathit{fetch}\ bs = \mathsf{CallPtr} \quad bs.\mathsf{stack} = x :: \mathsf{CodePtr}\ ptr :: xs}{bs \rightarrow bs\{\mathsf{stack} = x :: \mathsf{CodePtr}\ (\mathsf{bump}\ bs).\mathsf{pc} :: xs;\ \mathsf{pc} = ptr\}}$$

$$\frac{\mathit{fetch}\ bs = \mathsf{PushExc} \quad bs.\mathsf{stack} = xs}{bs \rightarrow (\mathsf{bump}\ bs)\{\mathsf{stack} = \mathsf{StackPtr}\ bs.\mathsf{handler} :: xs;\ \mathsf{handler} = \mathsf{length}\ xs\}}$$

$$\frac{\mathit{fetch}\ bs = \mathsf{PopExc} \quad bs.\mathsf{handler} = \mathsf{length}\ ys \quad bs.\mathsf{stack} = x :: xs\ @\ \mathsf{StackPtr}\ h :: ys}{bs \rightarrow (\mathsf{bump}\ bs)\{\mathsf{stack} = x :: ys;\ \mathsf{handler} = h\}}$$

Figure 3.4: Examples of semantics of CakeML bytecode instructions. The helper function *fetch bs* fetches the next instruction according to the program counter bs.pc, and bump bs updates the program counter to the next instruction.

no final state and evaluation of bs_1 diverges, then EvalBC bs_1 = None.

Data refinement from CakeML source-level values to bytecode values must encode all CakeML values as bytecode Blocks and Numbers. The overall refinement relation decomposes into a series of relations that mirror each phase of compilation. The most complicated part of data refinement is for closures; at a high level, our strategy encodes each closure as Block closure_tag (CodePtr *ptr*::*env*), where the code pointer *ptr* points to the result of compiling the body of the closure, which must exist in the bytecode machine state's code field. For first-order values, since bytecode blocks are structured and bytecode integers are mathematical integers, data refinement is not much more complicated than assigning tags to blocks to distinguish different types of value and following a straightforward encoding scheme. Data refinement to bytecode is part of the Inv *env cs bs* invariant seen previously asserting that the semantics, the compiler, and the bytecode machine state are in correspondence.

3.4 Divergence preservation

Proof by induction on the big-step operational semantics of the source language is a natural approach for proving compiler correctness, because it means reasoning in the direction of compilation. However, such a proof structure only covers programs that are included in the big-step operational semantics; in particular, it does not cover diverging programs. We would like to also prove that the compiler maps diverging source expressions to diverging bytecode programs. Since the bytecode semantics is deterministic, this is sufficient for knowing that the compiler's output always relates to the semantics of its input whether that be divergence or termination.

Since proofs in the direction of compilation by induction on the big-step semantics are more natural, we would like to handle diverging programs in the same way. In particular, we would like to avoid resorting to an alternative semantics around which to structure the proof, such as co-inductive big-step semantics [47] or small-step traces. The solution used in CakeML is the lightweight technique of adding an optional, logical clock to the big-step semantics which enables it to talk about non-termination. Clock-preservation can be done in the same proof of semantics-preservation, and facilitates the proof of divergence-preservation.

The first step for divergence-preservation is clock-preservation. In the big-step semantics, the clock is decremented by 1 on each function call, and a timeout exception is raised if a function is called when the clock is 0. In the bytecode, the Tick instruction decrements the clock, and the semantics gets stuck if the clock is 0. The compiler emits a Tick instruction for each source function call, and we prove that if a program times out in the semantics with a certain clock, then the compiled version times out in the bytecode with the same clock.

With clock-preservation proved, it remains to show the main divergence-preservation result, which applies when the source semantics ignores the clock and the Tick instruction is implemented as a no-op (and thus produces no x86-64 instructions). I sketch the proof here with a simplified notation, omitting many of the arguments from the real semantic relations. We use $c \vdash e \Downarrow v$ for convergence in the source language with clock c to a value (or non-timeout exception), and $c \vdash e \Downarrow \emptyset$ for a timeout exception. We use a clock of ∞ to indicate the version that ignores the clock.

Lemma (Big-step clocked totality)**.** For all clocks c and expressions e, either

$c \vdash e \Downarrow \emptyset$ or $\exists v.\ c \vdash e \Downarrow v$.

Proof sketch. By well-founded induction on the lexicographic ordering of the clock and size of the expression. In all but one case, the applicable big-step rules have inductive premises that have the same or smaller clocks (because the clock is monotonically non-increasing) and smaller sub-expressions. Thus, by induction the results for the sub-expressions combine to give a result for the expression. (It is important here that all mis-applied primitives evaluate to an exceptional result.) The only case where the expression might be bigger is function application, but it decrements the clock first. □

Lemma (Big-step clock/unclock)**.** $c \vdash e \Downarrow v$ implies $\infty \vdash e \Downarrow v$ and, $\infty \vdash e \Downarrow v$ implies $\exists c.\ c \vdash e \Downarrow v$.

Proof sketch. Straightforward induction. □

The bytecode's operational semantics is small-step, so we define an evaluation relation in the standard way:

$$c \vdash bs \Downarrow_{bc} bs' \equiv bs\{\texttt{clock} = c\} \rightarrow^* bs' \wedge \forall bs''.\ \neg(bs' \rightarrow bs'')$$

We say the machine has timed out if it evaluates to a state with $\texttt{clock} = 0$ and next instruction Tick. A bytecode machine state diverges if it can always take another step.

Lemma (Bytecode clock/unclock)**.** $c \vdash bs \Downarrow_{\text{bc}} bs'$ implies $bs\{\texttt{clock} = \infty\} \rightarrow^* bs'\{\texttt{clock} = \infty\}$, and $\infty \vdash bs \Downarrow_{\text{bc}} bs'$ implies $\exists c.\ c \vdash bs \Downarrow_{\text{bc}} bs'\{\texttt{clock} = 0\}$.

Proof sketch. Straightforward induction. □

Lemma (Clocked bytecode determinism)**.** $c \vdash bs \Downarrow_{\text{bc}} bs'$ and $c \vdash bs \Downarrow_{\text{bc}} bs''$ implies $bs' = bs''$.

Proof sketch. The small-step relation is deterministic by inspection of the rules; the main result follows by induction on $\rightarrow^*$. □

Theorem 1 (Divergence preservation (sketch))**.** Evaluation of e diverges in the un-clocked semantics iff the compilation of e (loaded into a bytecode state bs) diverges in the un-clocked bytecode semantics.

Proof. For the "only if" part, we have $c \vdash e \Downarrow \emptyset$, for all clocks c, by the source language's determinism, and the totality and clock/unclock lemmas. Therefore by the compiler correctness result, we know for all clocks c there is a bs' such that $c \vdash bs \Downarrow_{\text{bc}} bs'$ and bs' is timed out. Now we must show that $bs\{\texttt{clock} = \infty\}$ diverges. Suppose, for a contradiction, there is some bs'' with $\infty \vdash bs \Downarrow_{\text{bc}} bs''$. Let c be one more than the number of Tick instructions on the trace from bs to bs'', which is unique by determinism. This contradicts the existence of a bs' above: if evaluation stops before reaching bs'', it will not have passed enough Ticks to deplete the clock, and if it reaches bs'' it stops without timing out.

The "if" part is easier, and follows directly from the clock-preservation part of compiler correctness. We prove the contrapositive: we assume evaluation of e converges and must show that bytecode evaluation of bs also converges. By compiler correctness there is a bs' such that $c \vdash bs \Downarrow_{\text{bc}} bs'$ and bs' has terminated successfully. The bytecode unclock lemma says we can unclock this evaluation to obtain $bs\{\texttt{clock} = \infty\} \rightarrow^* bs'\{\texttt{clock} = \infty\}$. Since bs' terminated successfully, this is the same as $\infty \vdash bs \Downarrow_{\text{bc}} bs'\{\texttt{clock} = \infty\}$, which means un-clocked evaluation of bs converges as required. □

3.5 Connecting the pieces

By composing the correctness theorems for the compiler of each intermediate language, and similarly composing the data refinement relations, we obtain a top-level correctness theorem for the whole compiler that executes each of compilation phases in sequence. To avoid going into too many details of the semantics of CakeML, I split this result across two theorems below, one for terminating programs and one for diverging programs.[3] The operational semantics in these theorems is represented by EvalDecTerminate and EvalDecDiverge; these extend the EvalDec semantics from Lemma 1 by taking into account stateful computations (the store is represented by s_1 and s_2) and failing computations (*res* includes a Boolean *success* indicating success and the printed output *msg* can include an error message).

The first theorem, for terminating programs, states that if the semantics of a declaration *dec* is to terminate with a result $res = (success, env_2, s_2, msg)$, which

[3] These theorems are combined again in the proof of correctness for the REPL, Theorem 4 in the next chapter.

includes the message to print and the new environment and store, and the invariant between semantics, compiler, and bytecode machine holds, then the code *bc* generated by the compiler will run to completion and print *msg*, and the invariant will continue to hold for the resulting bytecdode machine state. The compiler produces two compiler states, *cs2s* and *cs2f*, because a different compiler state will satisfy the invariant depending on whether the computation succeeded or not, since bindings introduced by a failed declaration are not kept.

Theorem 2 (Compiler correctness for terminating programs)**.**

$$
\begin{array}{l}
\vdash \mathsf{EvalDecTerminate}\ env_1\ s_1\ dec\ res \wedge \mathsf{InvFull}\ env_1\ s_1\ cs_1\ bs_1 \wedge \\
\quad \mathsf{CompileDecFull}\ cs_1\ dec = (cs2s, cs2f, bc) \Rightarrow \\
\quad\ \mathtt{case}\ res\ \mathtt{of} \\
\quad\quad (success, env_2, s_2, msg)\ \Rightarrow \\
\quad\quad\ \exists\, bs_2. \\
\quad\quad\quad (\mathsf{AddCode}\ bs_1\ bc) \rightarrow^* bs_2 \wedge \\
\quad\quad\quad \mathsf{bc\_fetch}\ bs_2 = \mathsf{Some}\ (\mathsf{Stop}\ success) \wedge \\
\quad\quad\quad bs_2.\mathsf{output} = msg \wedge \\
\quad\quad\quad \mathsf{InvFull}\ env_2\ s_2\ (\mathtt{if}\ success\ \mathtt{then}\ cs2s\ \mathtt{else}\ cs2f)\ bs_2
\end{array}
$$

The second theorem, for diverging programs, is stated using a clocked bytecode machine; the reasoning used to remove the clock for Theorem 1 is deferred until we verify the REPL. The theorem below states that if the semantics of a declaration *dec* is to diverge, and the invariant holds, then execution of the code produced by the compiler exhausts the bytecode machine's clock, and no output has been produced when the machine times out.

Theorem 3 (Compiler correctness for diverging programs)**.**

$$
\begin{array}{l}
\vdash \mathsf{EvalDecDiverge}\ env_1\ s_1\ dec \wedge \mathsf{InvFull}\ env_1\ s_1\ cs_1\ bs_1 \wedge \\
\quad \mathsf{CompileDecFull}\ cs_1\ dec = (cs2s, cs2f, bc) \Rightarrow \\
\quad\ \exists\, bs_2. \\
\quad\quad (\mathsf{AddCode}\ bs_1\ bc) \rightarrow^* bs_2 \wedge \mathsf{bc\_fetch}\ bs_2 = \mathsf{Some\ Tick} \wedge \\
\quad\quad bs_2.\mathsf{clock} = \mathsf{Some}\ 0 \wedge bs_2.\mathsf{output} = \texttt{""}
\end{array}
$$

Together, these correctness theorems for the compiler show us that the compilation of a declaration behaves the same way, according to bytecode semantics,

as the declaration itself according to CakeML's source semantics. As a result, we can execute this bytecode wherever we would otherwise appeal to the CakeML semantics; in particular, this will be our strategy for implementing the REPL. The theorems above cover compilation of abstract syntax only, but there are similar correctness theorems relating the result of parsing to CakeML's context-free grammar and the result of type inference to CakeML's type system.

Chapter 4

Bootstrapping the verified compiler

In this chapter, I describe *proof-grounded bootstrapping*, in particular how to bootstrap the verified compilation algorithm from the previous chapter by applying the techniques from Chapter 2. I also introduce the idea of a *packaged compiler*, and show how the bootstrapped compiler can be packaged into a verified runtime to produce a verified machine-code implementation of a read-eval-print loop (REPL) for CakeML.

In a nutshell, the idea of proof-grounded bootstrapping is to (automatically) derive a bootstrapping theorem, which states the result of applying the verified compilation algorithm to itself. This bootstrapping theorem includes a low-level implementation of the compiler—the output of running the compiler—in its theorem statement. Composing the bootstrapping theorem with the theorem asserting the algorithm is verified, we conclude that the low-level implementation of the compiler is also verified. Thus the TCB no longer needs to include unverified tools to compile the verified compiler: we can use the verified low-level implementation directly.

To apply proof-grounded bootstrapping, one needs a compiler that satisfies three requirements:

- the compilation algorithm is verified;
- as with ordinary compiler bootstrapping, the compiler is written in its own source language, or, more generally, something that can be translated to its source language; and,

- the compilation algorithm can be evaluated by rewriting in the logic used for its verification, which simply means that its definition can be characterised using rewrite rules.

Our focus in this chapter is on how to achieve proof-grounded bootstrapping once these requirements are satisfied, as is the case for CakeML.

4.1 Compilation as a verified algorithm

In Chapter 2 we saw two techniques, evaluation in the logic and translation to a deep embedding, which can be applied to verified algorithms. In Chapter 3 we saw a verified compilation algorithm for CakeML, in particular, CompileDec. Now, in this section, let us try some examples of applying our verified algorithm techniques to the compiler. Doing so will lead us naturally to bootstrapping.

Firstly, we can evaluate applications of the compiler to CakeML programs in the logic, for example to map_dec. Applying evaluation to the input term CompileDec InitCS map_dec, we obtain the following theorem, where MapCS stands for the concrete compiler state that results.

Example 2 (Evaluating the compilation of map)**.**

```
⊢ CompileDec InitCS map_dec =
    (MapCS,
     [Jump (Lab 12); Label 10; Stack (PushInt 0); Stack (PushInt 1); Ref;
      PushPtr (Lab 11); Stack (Load 0); Stack (Load 5); Stack (PushInt 1);
      Stack (Cons 0); Stack (... ... ); ... ... ; ... ; ... ])
```

Thus we can see that evaluation results in a theorem that produces a concrete list of bytecode for map_dec, to which the conclusion of the correctness theorem for CompileDec (Lemma 1) applies.

In addition to evaluating the compiler as a function in the logic, we can also use translation to produce an implementation of the compiler as a deep embedding. In other words, just as we produced map_dec plus its certificate theorem from the map algorithm, we can produce syntax and a certificate theorem from the compilation algorithm (the shallow embedding CompileDec). Since compilation is a rather more involved algorithm than map, it is split into 247 declarations of

auxiliary functions and datatypes. We use translation to produce a CakeML module (called "C" below) containing all these declarations (called CompileDec_decs below). Just as for map, the certificate theorem for CompileDec shows that the generated CakeML code runs successfully in the initial environment to produce an environment, abbreviated as CompEnv, containing a closure that implements CompileDec.

Lemma 2 (Certificate theorem for CompileDec)**.**

$$\begin{array}{l}\vdash \exists\, c.\\ \quad \mathsf{EvalDec\ InitEnv\ (Struct\ "C"\ CompileDec\_decs)\ CompEnv} \wedge\\ \quad \mathsf{LookupMod\ "C"\ "compiledec"\ CompEnv} = \mathsf{Some}\ c \wedge\\ \quad (\mathsf{CompStateTy} \longrightarrow \mathsf{DecTy} \longrightarrow \mathsf{PairTy\ CompStateTy\ (ListTy\ BCInstTy)})\\ \quad\ \mathsf{CompileDec}\ c\end{array}$$

The result of translating CompileDec includes CakeML syntax for the compiler, namely CompileDec_decs. A natural question is what happens if we use evaluation of CompileDec on the syntax for CompileDec produced by translation. What can we conclude about the resulting bytecode? This question is the idea behind proof-grounded bootstrapping, to which we now turn.

4.2 Proof-grounded bootstrapping

The aim of bootstrapping is to obtain a verified low-level implementation of a compiler directly from the verified compilation algorithm, and to thereby remove the need to trust the process by which the verified compilation algorithm gets compiled. Let us see how we obtain this verified low-level implementation automatically through a combination of the proof-producing-translation and evaluation-by-rewriting proof automation techniques.

Via translation we have obtained CakeML syntax for the compiler, namely, CompileDec_decs. Now, we use evaluation to calculate the application of the compiler to its syntax. This is analogous to Example 2 but instead of using map_dec as input, we use the module declaring the compiler. The result of this evaluation is what I call the *bootstrapping theorem.*

Lemma 3 (Bootstrapping theorem for CompileDec)**.**

$$\vdash \mathsf{CompileDec\ InitCS\ (Struct\ "C"\ CompileDec\_decs)} = \\ \mathsf{(CompCS,CompileDec\_bytecode)}$$

The bootstrapping theorem contains a concrete list of bytecode instructions that is the code generated by the compiler for the CompileDec_decs module, which I have abbreviated as CompileDec_bytecode.

Three theorems come together to create proof-grounded bootstrapping. Each corresponds to a different level of concreteness for the compiler, namely, the algorithm, the high-level implementation in CakeML, and the low-level implementation in bytecode. They can be described as follows:

- Correctness theorem: the output of the compiler implements the input, for all inputs. This theorem is about the compilation algorithm (shallow embedding), and corresponds to Lemma 1.

- Certificate theorem: the syntax for the compiler (CompileDec_decs) implements the compiler. This theorem is about the high-level implementation of the compiler produced by translation, and corresponds to Lemma 2.

- Bootstrapping theorem: the output of the compiler when given its syntax as input is low-level code for the compiler (CompileDec_bytecode). This theorem contains the low-level implementation of the compiler produced by evaluation, and corresponds to Lemma 3.

Instantiating the correctness theorem with the bootstrapping theorem, then composing it with the certificate theorem, we obtain the desired result that the low-level code for the compiler implements the compiler. That is the method behind proof-grounded bootstrapping.

The essence of proof-grounded bootstrapping is a consideration for the three levels of concreteness: algorithm (CompileDec), syntax (CompileDec_decs), and low-level code (CompileDec_bytecode). It is bootstrapping because the syntax happens to be syntax for the compiler. The approach can be generalised by using any other certified syntax instead. I call the general approach *proof-grounded compilation*. The generalisation of the bootstrapping theorem is a *compilation theorem* since it captures the result of a particular compilation. For the CakeML

REPL (Section 4.3), we apply proof-grounded compilation to a certificate theorem covering not just CompileDec_decs but also syntax for a verified parser and type inferencer.

In the sketch above, I used the word "implements" loosely. Let us look now at precisely what we obtain by following the bootstrapping method, and what assumptions remain undischarged. The compiler correctness theorem, repeated below, has three antecedents: the invariant, evaluation of the semantics, and an application of the compiler.

Lemma 1 (Correctness of CompileDec for successful declarations)**.**

$$\vdash \mathsf{Inv}\ env_1\ cs_1\ bs_1 \wedge \mathsf{EvalDec}\ env_1\ dec\ env_2 \wedge \mathsf{CompileDec}\ cs_1\ dec = (cs_2, bc) \Rightarrow \exists\, bs_2.\ (\mathsf{AddCode}\ bs_1\ bc) \rightarrow^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \mathsf{Inv}\ env_2\ cs_2\ bs_2$$

Following the bootstrapping method, we instantiate Lemma 1 so that the application of the compiler matches the bootstrapping theorem (Lemma 3). Evaluation of the semantics come from the certificate theorem (Lemma 2). To establish the initial invariant we can easily construct a bytecode machine state, InitBS, that only contains the primitives and satisfies the invariant:

Lemma 4 (Initial invariant)**.**

$$\vdash \mathsf{Inv}\ \mathsf{InitEnv}\ \mathsf{InitCS}\ \mathsf{InitBS}$$

After instantiating the correctness theorem and proving its hypotheses as just described, we are left with a conclusion that states that CompileDec_bytecode runs to completion and the resulting bytecode state satisfies the invariant at CompEnv, the environment containing the compiler:

Lemma 5 (Result of bootstrapping)**.**

$$\vdash \exists\, bs_2.\ (\mathsf{AddCode}\ \mathsf{InitBS}\ \mathsf{CompileDec\_bytecode}) \rightarrow^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \mathsf{Inv}\ \mathsf{CompEnv}\ \mathsf{CompCS}\ bs_2$$

In other words, according to the semantics of bytecode execution, we can produce a bytecode machine state, bs_2 above, that implements CompEnv. The certificate theorem (Lemma 2) tells us that CompEnv contains a closure (bound by

the variable "compiledec" in the "C" structure) that implements the CompileDec function according to the refinement invariants of translation. Thus, the bytecode machine state asserted to exist above (bs_2) contains a low-level implementation of the compiler, CompileDec, as promised.

The usefulness of Lemma 5 depends on the strength of the refinement invariant of translation (CompStateTy $\longrightarrow$ DecTy $\longrightarrow$ ... ...) connecting the implementation in CakeML to the shallow embedding, and the invariant (Inv) connecting the implementation in bytecode to the implementation in CakeML. What Lemma 5 provides is a closure implementing the compiler according to the refinement invariants. In fact, the invariants are strong enough for any use of the closure that depends only on its functional (i.e., input/output) behaviour, and in particular enable verification of a REPL that contains this closure and uses it for compilation. In the next section, I introduce the idea of packaging a verified compiler and then proceed to describe the implementation and verification of the CakeML REPL.

4.3 Packaging a bootstrapped compiler as a REPL

A compiler can be used as a standalone application, which does no more than take high-level code as input and produce low-level code as output. I call this kind of application a *standalone compiler*. If the compiler is verified, there will be a correctness theorem about running the low-level code under particular conditions. The correctness theorem is vacuous unless its assumptions are met. For example, Lemma 1 states that the low-level code *bc* output by CompileDec preserves the Inv invariant, which assumes the invariant holds in the first place. Lemma 4 states that InitBS satisfies the invariant, so it is sufficient to load the output of the compiler into InitBS before it is run. For a standalone compiler, it is up to the user to run the output of the compiler in such a way that satisfies the conditions of the correctness theorem if they want to leverage the verification.

With a view to reducing the trusted computing base (TCB), there is an extension to a standalone compiler that I call a *packaged compiler*, where the compiler is included within a larger verified program that always runs the compiler's output in a way that satisfies the assumptions of the compiler's correctness theorem. A packaged compiler does more than compilation: it compiles, loads, and runs

code. And because it is self-contained, a verified package has a simpler correctness theorem than a verified standalone compiler. It allows us to focus our trust in the operating system and hardware on a single point: correct execution of the machine-code implementation of the whole package.

One way to package a compiler is as a *one-shot* package, which always uses the initial compiler state (for CompileDec that is InitCS) and loads the result of compilation into a fresh initial machine state (for bytecode that is InitBS) for execution. For a one-shot package, the *wrapper* (i.e., non-compiler) code reads the input (high-level program), feeds it to the compiler, loads the output (low-level program) into an appropriate runtime environment, then jumps to the loaded program. A one-shot package is not interactive: the entire program and its input is prepared before compilation, and any further interaction is via input/output (I/O) primitives called from within the program.

By putting the wrapper into a loop, however, we obtain a *read-eval-print loop* (REPL), which is inherently interactive. A REPL intersperses execution of the compiler with execution of its output and retains state between calls to the compiler, thus later input code can depend on the results of previously input code. Since CakeML does not presently have I/O primitives, a REPL is essential for interaction; it is also a more interesting example of a packaged compiler since the compiler can be called multiple times.

To verify a machine-code implementation of a packaged compiler, it is necessary to have a machine-code implementation of the compiler itself whose correctness theorem is strong enough to support execution of the compiler at (package) runtime. A verified compilation algorithm on its own is not enough to produce a verified REPL in machine code. It is the push from verified algorithms down to a verified implementation that enables production of such machine-code programs that contain the verified compiler.

My goal now is to explain how, using proof-grounded bootstrapping, I have produced machine code that is verified to implement a REPL for CakeML. Each piece of verified machine code comprising the REPL is obtained by one of two methods. The first method is bootstrapping, which provides code for most of the compiler. The second method is *decompilation into logic* (henceforth "decompilation", as in Myreen et. al. [58, 63, 64]), which is used for the rest of the compiler and the wrapper code.

Decompilation is a tool-assisted but manual procedure for verifying programs

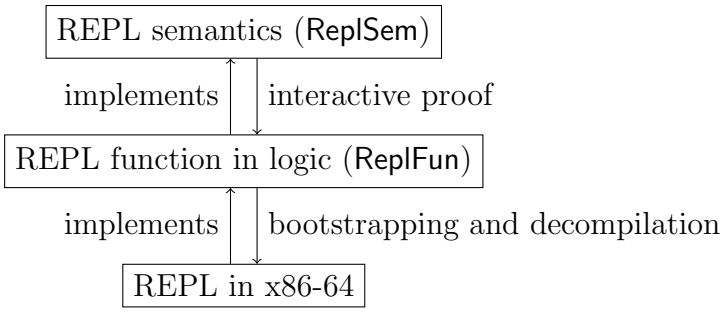

Figure 4.1: Overview of verified REPL construction.

written either directly in assembly code or as functions in the logic in a particular tail-recursive style. Because of the effort required—compared to the fully automated bootstrapping method—we use it only for those parts of the REPL that must be implemented at a low level, such as the garbage collector and the (simple) compiler from CakeML bytecode to x86-64 machine code.

The REPL and its verification comprise three layers as shown in Figure 4.1. At the top of the figure is the semantics of the REPL (ReplSem), which builds on the semantics for CakeML programs (EvalDec), and was described in Section 2.3.1. In the middle of the figure is a description of the REPL as a function in the logic (ReplFun), which replaces the CakeML operational semantics with the semantics for CakeML bytecode by packaging a verified compiler, called ParseInferCompile (Section 4.4), from concrete syntax to bytecode. This middle layer is almost an algorithm for the REPL, but deals with divergence non-algorithmically in terms of traces in the bytecode semantics. At the bottom of the figure is the implementation of the REPL package in machine code, which is produced and verified by a combination of the bootstrapping and decompilation techniques.

The REPL function in the logic acts as an intermediary between the semantics of the REPL and the machine code that is ultimately produced. It is treated like an implementation of the REPL semantics, but acts as a specification for the machine-code implementation. The specification is of the entire REPL package, that is, both the compiler and the wrapper. The compiler, ParseInferCompile, used inside ReplFun extends the CompileDec compiler we have already seen with the addition of a verified parser from concrete syntax and verified type checking. We describe the definition and verification of ReplFun in Section 4.4.

To produce the final machine-code implementation, most of ReplFun is bootstrapped and the remaining code is produced more manually using decompilation

into logic. In connecting the bootstrapped and non-bootstrapped code, we face the issue of *using* the bootstrapped function—in particular, giving input and retrieving output—touched on at the end of the previous section. We need to be able to call the closure asserted to exist after bootstrapping, and know that it will behave correctly. For this purpose, I bootstrap not just the definition of the compiler but also a declaration of a call to the compiler. I explain this small extension to the bootstrapping idea in Section 4.5.

The non-bootstrapped code comes in two categories: firstly, there is the lexer and the loop that calls the compiler on the result of lexing and jumps to its output; secondly, there is the runtime that implements a CakeML bytecode machine, which includes additional (previously verified) machine-code libraries for garbage collection [59] and arbitrary-precision integer arithmetic [60]. The main subtlety in producing a packaged compiler by bootstrapping in this way is that there are logically two distinct bytecode machine states to consider: one for running bootstrapped code, and another simulating the bytecode machine that is explicitly mentioned in ReplFun and runs user code. I describe the construction and verification of this final layer in Section 4.6.

4.4 REPL implementation specified as a function in logic

When we looked at the example of bootstrapping CompileDec, we evaluated CompileDec on syntax implementing CompileDec itself. When bootstrapping for the REPL we will still evaluate CompileDec but on syntax implementing a larger function. I combine parsing, type inference, and compilation to bytecode together as:

ParseInferCompile *tokens s*

which is called on a list of tokens, *tokens*, produced by the lexer and the state, *s*, of the REPL implementation[1]. This function returns either Failure (*msg*,*sf*) if there is a parse- or type-error, or Success (*code*,*ss*,*sf*) with bytecode *code* that executes the declaration represented by *tokens* and new REPL states to be installed if

[1]The state includes the compiler's and type inferencer's memory of previous declarations, whose results may be referred to in later declarations.

running *code* terminates normally (*ss*) or raises an exception (*sf*)[2]. In addition to the parser and type inferencer, there is also an initial program—the CakeML Basis Library—to be loaded in the REPL when it starts. The combined function representing almost[3] all the code to be bootstrapped is ReplStep, and is defined at the top of Figure 4.2. Bootstrapping affords us the ability to produce low-level implementations of the parser and type inferencer automatically after verifying their shallow embeddings: we simply include them (via ParseInferCompile) in this function to be bootstrapped.

The remainder of the REPL (the lexer, the main loop, and the runtime that executes the compiler-generated code) is not generated by bootstrapping, so requires a more manual treatment. However, we specify the entire REPL implementation, including the non-bootstrapped parts, as a function in the logic. That function is ReplFun, and its definition is shown in Figure 4.2. The majority of the code in the REPL implementation is hidden inside the ParseInferCompile algorithm inside ReplStep, but since this part is produced by bootstrapping we only need to know that the algorithm is correct and not how it is implemented. By comparison, the details of the implementation of MainLoop are important for constructing the final machine-code implementation, but there are only a handful of them.

The correctness theorem for ReplFun states that it produces exactly the same *repl_result*, *output*, for a given *input* as is specified by the semantics ReplSem (modulo an additional empty result at the front corresponding to the basis library).

Theorem 4 (Correctness of ReplFun)**.**

$$\vdash \forall\, input.\ \exists\, output.\ \mathsf{ReplFun}\ input = \mathsf{Result}\ \texttt{""}\ output \wedge \mathsf{ReplSem}\ \mathsf{Basis}\ input\ output$$

Theorem 4 is proved by complete induction on the length of the input string (which corresponds to the number of declarations made by the user), and follows the model of invariant preservation. The invariant used for the REPL extends the Inv invariant from Section 4.1 with information about type inference. It connects

[2]Different states are required since not all bindings might persist if an exception is raised, and exceptions are not statically predictable.

[3]All that is missing is an extra interface function used to make a call to ReplStep, described in Section 4.5.

```
ReplStep None = Success BasisCodeAndStates
ReplStep (Some (tokens,s)) = ParseInferCompile tokens s

MainLoop prev bs input =
 case ReplStep prev of
  Success (code,ss,sf) ⇒
   (let bs1 = AddCode bs code
    in
     case EvalBC bs1 of
      None ⇒ Diverge
    | Some bs2 ⇒
       Result bs2.output
        (case LexUntilSemicolon input of
          None ⇒ Terminate
        | Some (tokens,input2) ⇒
           (let s2 = TestException bs2 (ss,sf)
            in
              MainLoop (Some (tokens,s2)) bs2 input2)))
| Failure (msg,sf) ⇒
   Result msg
    (case LexUntilSemicolon input of
      None ⇒ Terminate
    | Some (tokens,input2) ⇒ MainLoop (Some (tokens,sf)) bs input2)

ReplFun input = MainLoop None EmptyBS input
```

Figure 4.2: REPL implementation specified as a function in the logic, ReplFun, which is partitioned into a part to be bootstrapped (ReplStep) that includes the parser, type inferencer, compiler, and initial program, and a part to be verified using decompilation (the rest of MainLoop).
The particular functions that need the manual decompilation treatment can be seen in the definition of MainLoop, they are: AddCode to install new code in the code heap, EvalBC that simulates bytecode execution, LexUntilSemicolon that reads and lexes new input, and TestException that checks whether bytecode simulation ended with success or failure and returns the corresponding new REPL state.
The main loop takes the last read declaration (*tokens*) and current state (*s*) as an argument, *prev*, so that the first thing it does is call the ReplStep function: this way of structuring the loop makes it easier to include the bootstrapped code in the final machine-code implementation.

the semantics, the compiler, and the bytecode, ensuring that: the state of the type system in the semantics is consistent with itself and with the state of the inferencer, and the state of the operational semantics is consistent with the state of both the compiler and the values in the bytecode machine. In each iteration of MainLoop, we combine the correctness theorems for the parser, type inferencer, and compiler to conclude that either the correct error message is produced or the generated bytecode, when evaluated, correctly diverges or correctly stops in a bytecode machine state that again satisfies the invariant.

ReplFun implements ReplSem, so we have reduced our task to implementing ReplFun in machine code. The function divides neatly into two parts, the part called ReplStep, and the part called MainLoop that does case-analysis on the result of ReplStep. To produce machine code for ReplStep, we put it through the proof-grounded bootstrapping process described in Section 4.2. In the next section, we look at bootstrapping ReplStep more carefully and address the question of using the bootstrapped compiler at runtime, that is, providing it input and retrieving its output. Then, in Section 4.6 we turn to verifying machine code for the rest of ReplFun and putting the two together.

4.5 Bootstrapping a function call

To bootstrap ReplStep, we follow the strategy described in Section 4.2, where we bootstrapped CompileDec by evaluating compilation of CompileDec_decs. Which declarations should we use in place of CompileDec_decs? To answer this, consider how we will use the result of bootstrapping which, analogous to Lemma 5, produces a bytecode machine state containing the declared values. Since our main loop makes a call to ReplStep, we want those values to include ReplStep, but we also want to be able to call ReplStep on input and obtain its output. To make the interface between the bootstrapped and non-bootstrapped code as simple as possible, we define one extra function, CallReplStep, that calls ReplStep and does I/O via CakeML references. Thus, the declarations we want to bootstrap, called REPL_decs, are:

```
...; fun replstep x = ...;
val input = ref NONE;
val output = ref NONE;
fun callreplstep _ =  output := (replstep (!input));
```

The first line represents 428 declarations (for the parser, type inferencer, compiler, and all dependencies) generated automatically by proof-producing translation of ReplStep, and the last three declarations are added by hand.

The important feature of CallReplStep is that its type in CakeML is *unit* $\rightarrow$ *unit*, which means it can be called multiple times uniformly. The certificate theorem for CallReplStep will be used in the same way each time around the loop of the REPL. We use references for I/O so we do not have to reason about an endless sequence of calls to CallReplStep, but instead prove a single theorem (Theorem 5 below) that is strong enough to apply to each call.

To call ReplStep, the non-bootstrapped machine code need only do three things: update the `"input"` reference, run the following declaration, called call_dec:

```
val () = REPL.callreplstep ();
```

and read the `"output"` reference. We now have two declarations serving different roles. The first is REPL_decs, which is used to declare CallReplStep and all its dependencies (including the compiler). The second is call_dec which does not declare anything (it returns *unit*), but has the side-effect of calling CallReplStep and updating the I/O references. We apply bootstrapping to both declarations, because we need verified low-level implementations of both. The first step is to produce certificate theorems.

Most of the syntax for REPL_decs is generated by proof-producing translation of ReplStep, which generates certificate theorems automatically. We use them to prove some extended certificate theorems that mention the I/O-related declarations we added. Our extended certificate theorems, shown below, say that the semantics of REPL_decs is to produce an environment, called ReplEnv, and whenever the call_dec declaration is made in ReplEnv, it has the sole effect of updating the `"output"` reference with the result of applying ReplStep to the contents of the `"input"` reference.

Theorem 5 (Certificate theorems for REPL_decs and call_dec)**.**

$$
\begin{array}{l}
\vdash \mathsf{EvalDec}\ \mathsf{InitEnv}\ (\mathsf{Struct}\ \texttt{"REPL"}\ \mathsf{REPL\_decs})\ \mathsf{ReplEnv} \\
\vdash \forall\, x\ inp\ out_1. \\
\quad \mathsf{InpTy}\ x\ inp \Rightarrow \\
\quad\ \exists\, out_2. \\
\quad\quad \mathsf{OutTy}\ (\mathsf{ReplStep}\ x)\ out_2 \wedge \\
\quad\quad \mathsf{EvalDec}\ (\mathsf{UpdRefs}\ inp\ out_1\ \mathsf{ReplEnv})\ \mathsf{call\_dec}\ (\mathsf{UpdRefs}\ inp\ out_2\ \mathsf{ReplEnv})
\end{array}
$$

As usual, there are refinement invariants (in this case InpTy and OutTy) mediating the connection between HOL values (x and ReplStep x) and CakeML values (inp and out_2). The helper function above, UpdRefs inp out ReplEnv, denotes an instance of ReplEnv where nothing has changed except for the contents of the two references which are now inp and out.

Now for the bootstrapping theorems. We use the same compiler as before (CompileDec), and apply evaluation in the logic to our two declarations to obtain bytecode (REPL_bytecode and Call_bytecode) that implements them. The compiler needs to know how to compile the variable lookup for `"REPL.callreplstep"` when compiling call_dec, so we use the compiler state (ReplCS) that resulted from compiling the REPL declarations when compiling call_dec.

Theorem 6 (Bootstrapping theorems for the REPL)**.**

$$
\begin{array}{l}
\vdash \mathsf{CompileDec}\ \mathsf{InitCS}\ (\mathsf{Struct}\ \texttt{"REPL"}\ \mathsf{REPL\_decs}) = (\mathsf{ReplCS}, \mathsf{REPL\_bytecode}) \\
\vdash \mathsf{CompileDec}\ \mathsf{ReplCS}\ \mathsf{call\_dec} = (\mathsf{CallCS}, \mathsf{Call\_bytecode})
\end{array}
$$

Let us review the three theorems used for bootstrapping, and what results from following the method.

- Correctness theorem: since we are still using CompileDec as our compilation algorithm, we continue to use its correctness theorem, Lemma 1.

- Certificate theorem: Theorem 5 states that the semantics of call_dec is to make a call to ReplStep via I/O references.

- Bootstrapping theorem: Theorem 6 contains the bootstrapped bytecode, REPL_bytecode and Call_bytecode, that comes from evaluating the compiler.

Instantiate the correctness theorem with the bootstrapping theorem, then apply the certificate theorem. For REPL_decs, we get a result stating that we can produce a bytecode machine state, ReplBS, implementing ReplEnv.

Theorem 7 (Result of bootstrapping REPL_decs)**.**

$$\vdash (\mathsf{AddCode\ InitBS\ REPL\_bytecode}) \rightarrow^* \mathsf{ReplBS} \wedge \mathsf{Halted\ ReplBS} \wedge \mathsf{Inv\ ReplEnv\ ReplCS\ ReplBS}$$

The first thing the non-bootstrapped machine code for the REPL does is load REPL_bytecode into InitBS and run it. By Theorem 7, this produces the ReplBS bytecode machine state, which will be used for all subsequent calls to ReplStep. The invariant governing these calls to ReplStep is a specialised version of the Inv invariant, which fixes everything except the I/O references, so that it can be re-established after each call. Specifically, the specialised invariant is InvIO *inp out bs*. This invariant means that *bs* is AddCode ReplBS Call_bytecode modulo I/O references, and Inv (UpdRefs *inp out* ReplEnv) ReplCS holds for *bs* before Call_bytecode is added.

If we write the result of bootstrapping the call using this InvIO invariant, it is clear that if the non-bootstrapped code sets the input reference correctly, it can run Call_bytecode after which the output reference will be set to the result of calling ReplStep. The function ResetPC *bs* sets the program counter back to the beginning of Call_bytecode, in preparation for the next iteration of the REPL.[4]

Theorem 8 (Result of bootstrapping call_dec)**.**

$$\begin{aligned}\vdash\ & \mathsf{InvIO}\ inp\ out_1\ bs_1 \wedge \mathsf{InpTy}\ x\ inp \Rightarrow \\ & \exists\, out_2\ bs_2. \\ & \quad \mathsf{OutTy}\ (\mathsf{ReplStep}\ x)\ out_2 \wedge bs_1 \rightarrow^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \\ & \quad \mathsf{InvIO}\ inp\ out_2\ (\mathsf{ResetPC}\ bs_2)\end{aligned}$$

Theorem 8 lets the non-bootstrapped part of the REPL implementation call the bootstrapped compiler. This result is simply the bytecode-level version of Theorem 5 (the certificate theorem describing this process at the level of the operational semantics). Together, Theorems 7 and 8 represent the results of

[4]The conclusion of Theorem 8 is not used immediately again as its hypothesis. First the *inp* parameter is changed by the non-bootstrapped code.

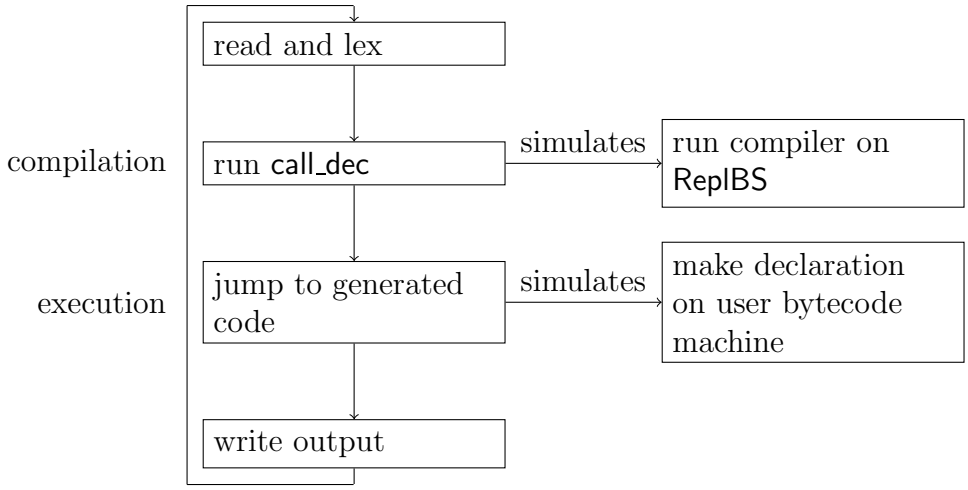

Figure 4.3: The two bytecode machines simulated by the final REPL implementation. The left half of the figure is specified by MainLoop (Figure 4.2). The call to EvalBC within MainLoop happens via simulation of the main bytecode machine (for user code). The call to ReplStep within MainLoop happens via simulation of another bytecode machine, which stays in ReplBS (modulo references), and runs the bootstrapped compiler.

bootstrapping for the REPL. We turn now to the non-bootstrapped parts of the REPL, and putting the whole package together.

4.6 Producing verified machine code

The machine-code implementation of the REPL for CakeML does the following steps in a loop: read and lex the next declaration (LexUntilSemicolon), compile the declaration to bytecode (ReplStep), evaluate the compiled bytecode (EvalBC) by first compiling to x86-64 then jumping to the new code, print the result and continue. These steps can be seen in the specification of the REPL main loop in Figure 4.2. The most involved part of each iteration is compilation to bytecode, but we have verified bootstrapped code for ReplStep to do that part. The next most complicated part is compilation and evaluation of bytecode (EvalBC).

Because it includes bootstrapped code, the final REPL implementation depends on the two separate sessions of the REPL semantics, and simulates two bytecode machines. The first session, for the compiler, is the one that is initialised with REPL_decs and thereafter stays in ReplBS (with input/output references updated each iteration). The second session, for the user, is the one that runs the

user's input on the bytecode machine state *bs* passed around in the definition of MainLoop. Figure 4.3 illustrates how these fit together.

To simulate each bytecode machine, we write a simple compiler from bytecode instructions (Section 3.3) to snippets of x86-64 machine code. For the semantics of x86-64 machine code, we use the model developed by Sarkar et. al. [73] and updated for the verified Lisp runtime, Jitawa [61]. We verify the compiler using the technique of decompilation into logic [64]. The most difficult part of this verification is devising the invariant that holds between a bytecode machine state and an x86-64 machine state, which also includes data refinement from bytecode values to immediate values or pointers into the x86-64 heap. I do not delve into the details of this invariant here, since they are not especially relevant to packaging bootstrapped code.

There are a handful of bytecode instructions (e.g., structural equality) and helpers (e.g., lexing) that are implemented by machine-code routines that are larger than the snippets used for most instructions. Also, instructions which do allocation or arithmetic make use of separately verified machine-code routines for garbage collection [59] and arbitrary-precision integer arithmetic [60]. In each case, the larger routine is verified using decompilation and plugged into the overall correctness proof. Producing the non-bootstrapped parts of MainLoop, including the runtime for simulating bytecode execution, is an example of machine-code verification as used in previous work [61] verifying the Jitawa runtime for Lisp.

To use the bootstrapped code, it is sufficient to establish the InvIO invariant, since we can then apply Theorem 8. We prove that the InvIO invariant holds after REPL_bytecode runs when the REPL starts, and then continues to hold when the input reference is updated with the result of lexing. Theorem 8 lets us preserve the invariant across calls to the compiler, and therefore throughout execution of the main loop.

Our interface to the x86-64 machine semantics is via predicates that apply to sequences of steps (traces) made by the x86-64 state machine. The kinds of predicates we use are inspired by temporal logic. The assertion TemporalX64 *code A* states that if *code* is loaded in memory then the temporal predicate A is satisfied by all runs of the machine. Satisfaction of a temporal predicate by a run, s, is defined as follows:

- Now P is satisfied by s if $P\ (s\ 0)$.

- Holds p is satisfied by s if p is true. (p does not depend on the machine state).
- $\Diamond\ A$ is satisfied by s if A is satisfied by $\lambda\, n.\ s\ (n + k)$ for some k.
- $\Box\ A$ is satisfied by s if A is satisfied by $\lambda\, n.\ s\ (n + k)$ for all k.
- $A \wedge B$ is satisfied by s if A is satisfied by s and B is satisfied by s. Similarly for $A \vee B$, $A \Rightarrow B$, and $\exists\, x.\ A\ x$.

The final correctness theorem we obtain is about a single machine-code program (a list of bytes), which I abbreviate as ReplX64, and is phrased as a temporal assertion about running that program. It states that: if at some time the machine state is appropriately initialised, then either it will eventually run out of memory, or it will eventually diverge or terminate with output according to the CakeML REPL semantics.

Theorem 9 (Correctness of REPL implementation in x86-64)**.**

$$
\begin{array}{l}
\vdash \mathsf{TemporalX64\ ReplX64} \\
\quad (\mathsf{Now\ (InitialisedX64}\ ms) \Rightarrow \\
\quad\ \Diamond\ \mathsf{Now\ (OutOfMemX64}\ ms) \vee \\
\quad\ \exists\, output. \\
\quad\quad \mathsf{Holds\ (ReplSem\ Basis}\ ms.\mathsf{input}\ output) \wedge \\
\quad\quad \texttt{if}\ \mathsf{Diverges}\ output\ \texttt{then}\ \Box\ \Diamond\ \mathsf{Now\ (RunningX64}\ output\ ms) \\
\quad\quad \texttt{else}\ \Diamond\ \mathsf{Now\ (TerminatedX64}\ output\ ms))
\end{array}
$$

The helper function Diverges *repl_result* tests whether *repl_result* ends in termination or divergence (the ***repl_result*** type is described in Section 2.3.1). There are four predicates on machine states ms that encode our invariants and conventions concerning the x86-64 machine as it simulates a bytecode machine.

- InitialisedX64 ms states that the machine is initialised. The heap invariant is satisfied, there is a return pointer on the stack, and the machine's output stream is empty.
- OutOfMemX64 ms states that the machine has aborted execution and is out of memory.

- RunningX64 *output ms* states that the heap invariant is satisfied and the output stream is equal to the concatenation of results in *output*.
- TerminatedX64 *output ms* states that the machine is about to jump to the return pointer and the output stream is equal to the concatenation of results in *output*.

Theorem 9 thus connects execution of an x86-64 machine loaded with the verified code produced by bootstrapping and decompilation back to the CakeML REPL semantics, completing the picture shown in Figure 4.1.

Part II

Self-verification

Chapter 5

Formal semantics of HOL

In this part, we shift attention away from CakeML and delve instead into the syntax and semantics of the logical system we have been using to conduct our proofs. We look at the semantics of HOL and a formalisation of that semantics in HOL itself (this chapter), including a proof of its soundness (next chapter). In Chapter 7 we return to CakeML, producing a verified implementation in CakeML of a proof checker for HOL.

5.1 Approach

The steps I have taken to formalise HOL within itself and produce a verified implementation of a theorem prover are as follows:

1. Specify the set-theoretic notions needed. (§5.2)
2. Define the syntax of HOL types, terms, and sequents. (§5.3)
3. Define semantic functions assigning appropriate sets to HOL types and terms, and use these to specify validity of a sequent. (§5.4)
4. Define the inference system: how to construct sequents-in-context (the rules of inference), how to extend a context (the rules of definition) (§6.1). Also, define the initial context (§6.2).
5. Verify the inference system: prove that every derivable sequent is valid (§6.3). Deduce that the inference system is consistent: it does not derive a contradiction. (§6.4)

6. Write an implementation of the inference system as recursive functions in HOL, and verify it against the relational specification of the inference rules. (§7.1)

7. Use the proof-grounded compilation technique from Chapter 4 to synthesise a verified implementation of the inference rules in CakeML. (§7.2)

All the specifications, definitions, and proofs mentioned above are constructed in HOL itself (using the HOL4 theorem prover), in the style of Harrison's work [23] towards self-verification of HOL Light. Compared to Harrison's work, items 6–7 are new, items 2–5 are extended and reworked to support a context of definitions, and item 1 uses an improved specification.

The bulk of the work is concerned with soundness: an inference rule is sound if whenever its antecedents hold in all models[1] then so does its succedent — a statement which does not depend on the existence of models. To validate our formalisation of the semantics it is useful to also carry out some proofs that give evidence of consistency. Assuming HOL is consistent, Gödel's second incompleteness theorem [18] prevents us from proving HOL's consistency in HOL. However, we can prove two relative consistency results: consistency, in HOL, of HOL with its axiom of infinity removed, and consistency, in HOL with a large-cardinal assumption added, of HOL. This gives good evidence that we have correctly captured the semantics.

My approach to the semantics, which is largely inspired by Arthan [4], avoids making any axiomatic extensions to HOL. I isolate results that are dependent on the set-theoretic axiom of infinity, so that as much as possible is proved without any undischarged assumptions. It is possible to use assumptions on theorems rather than asserting new axioms in the logic because I formalise a *specification* of set theory rather than defining a particular instance of a set theory as Harrison [23] did.

The results of following the plan above fit together as shown in Figure 5.1. The overall theorems we obtain are about evaluating the CakeML implementations of the HOL Light kernel functions in CakeML's operational semantics. For each kernel function, I prove that if the function is run in a good state on good

[1] We are concerned only with standard models of HOL, that is, where the Boolean and function types and the equality constant are interpreted in the standard way, and function spaces are full (unlike in Henkin semantics [25]). See the paragraph on standard interpretations in Section 5.4.

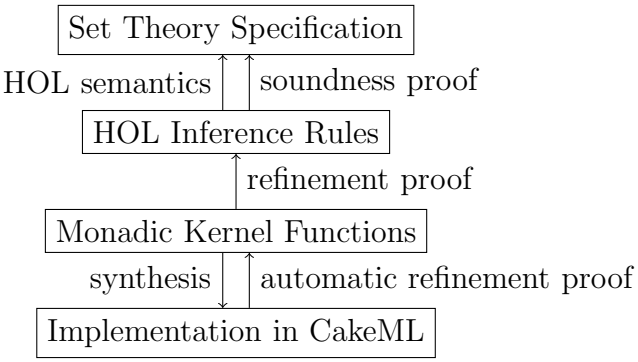

Figure 5.1: Producing a Sound Implementation by Refinement

arguments, it terminates in a good state and produces good results. Here "good" refers to refinement invariants. In particular, a good value of type *thm* must be a HOL sequent that is valid according to the set-theoretic semantics.

I prove these results by composing the three proof layers in the diagram. The top layer is the result of steps 1–5. The HOL semantics gives meaning to HOL sequents, from which we obtain definitions of validity and consistency. Validity concerns the truth of a sequent within a fixed context of definitions, whereas consistency is about whether the context itself has a model. The soundness proof says that each of the HOL inference rules preserves validity of sequents, and each of the HOL principles of definition preserves consistency of the context.

The middle layer corresponds to step 6. I define shallowly-embedded HOL functions, using a state-exception monad, for each of the HOL Light kernel functions (a version of this method was described in previous work [67, 40]). These "monadic kernel functions" are a hand-crafted implementation, but are written following the original OCaml code for HOL Light closely, and I prove that they implement the inference rules. Specifically, if one of these functions is applied to good arguments, it terminates with a good result; any theorem result must refine a sequent that is provable in the inference system.

Finally, for step 7 I use proof-grounded compilation (as in Chapter 4), starting with the proof-producing translation technique described in Chapter 2 that generates CakeML code for the inference kernel. The certificate theorems for the generated kernel complete the refinement proof that links theorem values produced by the implementation to sequents that are semantically valid. We can give the generated code to the CakeML compiler to produce a verified low-level implementation of the inference kernel.

The next steps of future work (i.e., not covered in this dissertation) include: a) proving, against CakeML's semantics, that the kernel implementation can be wrapped in a module to protect the key property, provability, of values of type *thm*; and b) embedding the compiled kernel in an interactive read-eval-print loop that is verified to never print a false theorem.

5.2 Set-theory specification

Axiomatic set theory can be specified in terms of a single binary relation, the membership relation. In HOL, we can give a quite straightforward development of the basic concepts of set theory as may be found in any standard text (e.g., Vaught [79]) thus achieving clarity through familiarity and making it easy to compare our formalisation with Pitts' informal account [70]. Since our specification is in HOL, we can write the membership relation and its axioms within the logic without resorting to the metavariables and schemata required in the first-order setting[2].

The most common set theory in textbook accounts is Zermelo-Fraenkel set theory ZF. However, ZF's axiom of replacement plays no rôle in giving semantics to HOL, so all we need are the axioms of Zermelo's original system: extensionality, separation (a.k.a. comprehension or specification), power set, union, (unordered) pairing, and infinity. It will be convenient to deal with the axiom of infinity separately. So we begin by defining a predicate on membership relations, is_set_theory $(mem : \mathcal{U} \rightarrow \mathcal{U} \rightarrow bool)$, that asserts that the membership relation satisfies each of the Zermelo axioms apart from the axiom of infinity. By formalising the set-theoretic universe as a type variable, $\mathcal{U}$, we can specify what it means to be a model of Zermelo set theory, while deferring the problem of whether such a model can be constructed.

[2] In our statement of the separation axiom, if the set x is infinite then P ranges over an uncountable set corresponding to all subsets of x. Technically, this is a significant strengthening of the axiom of separation, since it is not restricted to the countably many subsets of x that can be specified in the language of first-order set theory. However, this is irrelevant to our purposes: it would simply complicate the description of the semantics to impose this restriction (although our proofs in fact do not need instances of the axiom that could not be expressed in first-order set theory). Similarly, we find it convenient to use the metalanguage choice function and the metalanguage notion of finiteness rather than trying to give a first-order description of these notions in a model.

The specification of the set-theoretic axioms is as follows:

$$\mathsf{is\_set\_theory}\ mem \iff \\ \mathsf{extensional}\ mem \wedge (\exists\, sub.\ \mathsf{is\_separation}\ mem\ sub) \wedge \\ (\exists\, power.\ \mathsf{is\_power}\ mem\ power) \wedge (\exists\, union.\ \mathsf{is\_union}\ mem\ union) \wedge \\ \exists\, upair.\ \mathsf{is\_upair}\ mem\ upair$$

$$\mathsf{extensional}\ mem \iff \\ \forall\, x\ y.\ x = y \iff \forall\, a.\ mem\ a\ x \iff mem\ a\ y$$

$$\mathsf{is\_separation}\ mem\ sub \iff \\ \forall\, x\ P\ a.\ mem\ a\ (sub\ x\ P) \iff mem\ a\ x \wedge P\ a$$

$$\mathsf{is\_power}\ mem\ power \iff \\ \forall\, x\ a.\ mem\ a\ (power\ x) \iff \forall\, b.\ mem\ b\ a \Rightarrow mem\ b\ x$$

$$\mathsf{is\_union}\ mem\ union \iff \\ \forall\, x\ a.\ mem\ a\ (union\ x) \iff \exists\, b.\ mem\ a\ b \wedge mem\ b\ x$$

$$\mathsf{is\_upair}\ mem\ upair \iff \\ \forall\, x\ y\ a.\ mem\ a\ (upair\ x\ y) \iff a = x \vee a = y$$

To state the (set-theoretic) axiom of infinity, we define what it means for an element of $\mathcal{U}$ to be infinite: $\mathsf{is\_infinite}\ mem\ s \iff \neg\mathsf{FINITE}\ \{\ a \mid mem\ a\ s\ \}$. Here FINITE is inductively defined (in HOL) for sets-as-predicates, so we are saying a set is infinite if it does not have finitely many members. The (set-theoretic) axiom of infinity asserts that such a set exists.

5.2.1 Derived operations

Using the axioms above, it is straightforward to define standard set-theoretic constructions that will support our specification of the semantics of HOL. In this subsection, I introduce some notation for such derived operations. Since all the semantic functions are parametrised by the membership relation, ($mem : \mathcal{U} \to \mathcal{U} \to$ *bool*), I often elide this argument with a pretty-printing abbreviation,

for example writing Funspace x y instead of funspace mem x y.[3] When mem is used as a binary operator, I will from now on write it infix as $x \prec y$ instead of $mem\ x\ y$. Also, most of the theorems require the assumption is_set_theory mem, and I usually elide this assumption in the theorem statement. These notational ellipses are akin to working in an Isabelle [82] locale in which mem is fixed and is_set_theory mem is assumed.

Using the axiom of separation we define the empty set and prove it has no elements, then using pairing we define sets containing exactly one and exactly two elements. The latter serves as our representation of the set of Booleans. We have the following, shown with and without abbreviations for clarity:

(full notation)
$\vdash$ is_set_theory $mem \Rightarrow$
$\quad \forall x.\ mem\ x\ (\mathsf{two}\ mem) \iff x = \mathsf{true}\ mem \vee x = \mathsf{false}\ mem$

(abbreviated notation)
$\vdash x \prec \mathsf{Boolset} \iff x = \mathsf{True} \vee x = \mathsf{False}$

We define Kuratowski pairs (defn. V in [44]) as well as the cross product of two sets so that the following properties hold.

$$\vdash (a,b) = (c,d) \iff a = c \wedge b = d$$
$$\vdash a \prec x \times y \iff \exists b\ c.\ a = (b,c) \wedge b \prec x \wedge c \prec y$$

From cross products, we can define relations, and then functions (graphs) as functional relations. Abstract s t f is our notation for the subset of $s \times t$ that is the graph of $(f : \mathcal{U} \to \mathcal{U})$, and $a\ \prime\ x$ denotes application of such a set-theoretic function a to an argument x. The main theorem about application in set theory is that it acts like application in HOL:

$$\vdash x \prec s \wedge f\ x \prec t \Rightarrow \mathsf{Abstract}\ s\ t\ f\ \prime\ x = f\ x$$

[3] I follow a loose convention that capitalised functions have the hidden mem argument. Be aware that datatype constructors, which are also capitalised, are amongst the exceptions.

Furthermore, we know functions obey extensional equality:

$$\vdash (\forall x.\ x \prec s \Rightarrow f_1\ x \prec t_1 \wedge f_2\ x \prec t_2 \wedge f_1\ x = f_2\ x) \Rightarrow \\ \mathsf{Abstract}\ s\ t_1\ f_1 = \mathsf{Abstract}\ s\ t_2\ f_2$$

We define the set of functions between two sets, and prove that its elements are precisely those made using Abstract:

$$\vdash (\forall x.\ x \prec s \Rightarrow f\ x \prec t) \Rightarrow \mathsf{Abstract}\ s\ t\ f \prec \mathsf{Funspace}\ s\ t$$
$$\vdash a \prec \mathsf{Funspace}\ s\ t \Rightarrow \\ \exists f.\ a = \mathsf{Abstract}\ s\ t\ f \wedge \forall x.\ x \prec s \Rightarrow f\ x \prec t$$

The derived operations in our formalisation (a selection of which were shown in this subsection) may be considered as an alternative description (compared to the Zermelo axioms) of the interface required for giving semantics to HOL. In other words, any structure supporting such constructions as pairs and functions is suitable.

It is worth noting that a relation ($mem : \mathcal{U} \to \mathcal{U} \to \mathit{bool}$) satisfying is_set_theory mem will automatically satisfy the set-theoretic axiom of choice (AC), that is, we can prove the following:

$$\vdash \forall x.\ (\forall a.\ a \prec x \Rightarrow \mathsf{inhabited}\ a) \Rightarrow \exists f.\ \forall a.\ a \prec x \Rightarrow f\ `\ a \prec a$$

We prove this by using the axiom of choice in HOL (i.e., the language we are using to formalise set theory) to provide a HOL function ($g : \mathcal{U} \to \mathcal{U}$) such that for every non-empty ($a : \mathcal{U}$), we have $g\ a \prec a$. Then, given a set ($x : \mathcal{U}$) whose members are all non-empty, we use Abstract (ultimately depending on our strong form of the axiom of separation) to define the graph of g restricted to the members of x (as a member of the set-theoretic universe $\mathcal{U}$) and hence conclude that AC holds in our set theory.

5.2.2 Consistency of the specification

So far we have specified a predicate on a membership relation asserting that it represents a set theory with our desired structure (without the axiom of infinity). As a sanity check to convince us that this part of the specification is consistent,

we can construct a membership relation that satisfies the predicate.

The hereditarily finite sets provide a simple model of set theory without the axiom of infinity. This model can be represented concretely by taking $\mathcal{U}$ to be the type *num* of natural numbers and defining $mem\ n\ m$ to hold whenever the nth bit in the binary representation of m is not zero. Of course, the axiom of infinity fails in this model since every set in the model is finite. Nevertheless, we can prove that it satisfies the other axioms (V_mem is essentially the membership relation just described):

$$\vdash \mathsf{is\_set\_theory}\ \mathsf{V\_mem}$$

This shows that the notion of a set theory that we have formalised is not vacuous. One might argue that we could just use the monomorphic type *num* in place of the type variable $\mathcal{U}$. Or a little more abstractly, we could introduce a new type witnessed by the above construction on *num*. However, we wish to state some properties that are conditional on the set-theoretic axiom of infinity. Unfortunately, the axiom of infinity is provably false in a model subtyped from the countable set *num* and so results that assumed the axiom of infinity would be trivially true.

Instead, if we identify the universe of the hereditarily finite sets construction with the right-hand summand of the polymorphic type $\alpha + \mathit{num}$, we can define a subtype $\alpha\ V$ of $\alpha + \mathit{num}$ whose defining property is the existence of a membership relation satisfying the Zermelo axioms other than infinity. Hence we can introduce a constant V_mem of type $\alpha\ V \to \alpha\ V \to \mathit{bool}$ with $\vdash$ is_set_theory V_mem as its defining property. Thus if we work with V_mem the axiom of infinity is not provably false and we can meaningfully take it is as an assumption when necessary.

In the remainder of our development, we leave *mem* as a free variable and add one or both of the assumptions, is_set_theory *mem* and $\exists\, inf.$ is_infinite *mem* *inf*, whenever they are required. We provide V_mem in this section as a possible non-contradictory instantiation for the free variable *mem* in our theorems. Any instantiation that satisfies both assumptions would do, but we know we cannot exhibit one within HOL itself, so we prefer to leave the theorems uninstantiated. The decision to leave *mem* loosely specified (i.e., as a free variable) throughout the development is made easier by HOL4's parsing/printing support for hiding the free variable. In a theorem prover without such syntactic abbreviations, the

notational clutter might lend some encouragement to picking an instantiation up front.

With the specification of set theory in place, we now turn to the task of specifying the syntax and semantics of HOL. At the level of terms and types, my specification of the syntax is almost the same as Harrison's [23]. My terms are simplified by not needing to bake in any primitive constants since we support a general mechanism for introducing new constants, and my approach to substitution and instantiation of bound variables is improved. Also, my abstraction terms match the implementation better by using a term (rather than just a name and type) for the bound variable.

At the level of sequents, I introduce the notion of a *theory* describing the defined constants, which is implemented in the inference system as a *context* of theory-extending updates. An earlier version of this work [40] used the Stateless HOL [83] approach, where information about defined constants is carried on the terms and types themselves. Using a separate context of definitions makes the inference system clearer and allows us to easily quantify over all interpretations of the constants.

I present the HOL specification concisely, but give the important definitions in full so that it might serve as a reference.

5.3 Sequents: the judgements of the logic

Formally, derivations in HOL produce judgements of the following form[4]:

$$(thy,h) \texttt{ |- } c$$

This judgement is known as a *sequent*. It has a conclusion, c, a set of hypotheses (represented by a list of terms), h, and is interpreted in a theory, *thy* consisting of axioms and a signature. The meaning of a sequent is that the conclusion is true whenever all the hypotheses are true, all the axioms are true, and all the terms are well-formed with respect to the signature. We begin the specification at the bottom of this structure, starting with terms and types.

[4]I use the symbol (|-) for the sequents defined in our specification of HOL, reserving ($\vdash$) for theorems proved in the meta-logic (that of HOL4).

5.3.1 Terms and types

The syntax of HOL is the syntax of the (polymorphic) simply-typed lambda-calculus. Types are either variables or applied type operators.

$$\textit{type} \;=\; \mathsf{Tyvar}\ \textit{string} \mid \mathsf{Tyapp}\ \textit{string}\ (\textit{type list})$$

Primitive type operators include Booleans and function spaces. I abbreviate Tyapp `"bool"` [] by Bool and Tyapp `"fun"` $[x;\ y]$ by Fun $x\ y$.

A term is either a variable, a constant, a combination (application), or an abstraction.

$$\begin{aligned}\textit{term} \;=\;& \\ &\mathsf{Var}\ \textit{string type} \\ \mid\ &\mathsf{Const}\ \textit{string type} \\ \mid\ &\mathsf{Comb}\ \textit{term term} \\ \mid\ &\mathsf{Abs}\ \textit{term term}\end{aligned}$$

Variables carry their types: two variables with the same name but different types are distinct. We expect the first argument to Abs to be a variable, but use a *term* so the implementation can avoid destructing and reconstructing variables whenever it manipulates an abstraction.

Constants also carry a type, but are identified by their name: the type is there only to indicate the instantiation of polymorphic constants. (Different constants with the same name are disallowed, as we will see when we describe the signature of a theory and how it is updated.) The sole primitive constant is equality; I abbreviate Const `"="` (Fun ty (Fun ty Bool)) by Equal ty.

Well-typed terms The datatype above might better be called "pre-terms", because the only terms of interest are those that are well-typed. Every well-typed term has a unique type, which is specified by the following relation.

$$\frac{}{(\mathsf{Var}\ n\ ty)\ \mathsf{has\_type}\ ty} \qquad \frac{}{(\mathsf{Const}\ n\ ty)\ \mathsf{has\_type}\ ty}$$

$$\frac{s\ \mathsf{has\_type}\ (\mathsf{Fun}\ dty\ rty) \qquad t\ \mathsf{has\_type}\ dty}{(\mathsf{Comb}\ s\ t)\ \mathsf{has\_type}\ rty} \qquad \frac{t\ \mathsf{has\_type}\ rty}{(\mathsf{Abs}\ (\mathsf{Var}\ n\ dty)\ t)\ \mathsf{has\_type}\ (\mathsf{Fun}\ dty\ rty)}$$

Well-typed terms differ from pre-terms only in that the first argument of every combination has a function type, where the domain matches the second argument's type, and the first argument of every abstraction is a variable. I define $\mathsf{welltyped}\ tm \iff \exists\, ty.\ tm\ \mathsf{has\_type}\ ty$ and also define a function, typeof, to calculate the type of a term if it exists, thereby obtaining this characterisation: $\mathsf{welltyped}\ tm \iff tm\ \mathsf{has\_type}\ (\mathsf{typeof}\ tm)$.

Two operations over terms and types remain to be described, namely alpha-equivalence and substitution. Both are complicated by the need to correctly implement the concept of variable binding.

5.3.2 Alpha-equivalence

Terms are alpha-equivalent when they are equal up to a renaming of bound variables. The key idea of Harrison's original approach to alpha-equivalence is to formalise when two variables are equivalent in a context of pairs of equivalent bound variables.

$$\begin{aligned}&\mathsf{avars}\ [\,]\ (v_1,v_2) \iff v_1 = v_2\\ &\mathsf{avars}\ ((b_1,b_2)::bvs)\ (v_1,v_2) \iff\\ &\quad v_1 = b_1 \wedge v_2 = b_2 \vee v_1 \neq b_1 \wedge v_2 \neq b_2 \wedge \mathsf{avars}\ bvs\ (v_1,v_2)\end{aligned}$$

The variables must be equal to some pair of bound variables (or to themselves) without either of them being equal to (captured by) any of the bound variables above. I lift this relation up to terms, for example:

$$\frac{\mathsf{avars}\ bvs\ (\mathsf{Var}\ x_1\ ty_1,\mathsf{Var}\ x_2\ ty_2)}{\mathsf{aterms}\ bvs\ (\mathsf{Var}\ x_1\ ty_1,\mathsf{Var}\ x_2\ ty_2)}$$

$$\frac{\mathsf{typeof}\ v_1 = \mathsf{typeof}\ v_2 \qquad \mathsf{aterms}\ ((v_1,v_2)::bvs)\ (t_1,t_2)}{\mathsf{aterms}\ bvs\ (\mathsf{Abs}\ v_1\ t_1,\mathsf{Abs}\ v_2\ t_2)}$$

Finally, I define $\mathsf{aconv}\ t_1\ t_2 \iff \mathsf{aterms}\ [\,]\ (t_1,t_2)$ It is straightforward to show that this is an equivalence relation.

5.3.3 Substitution and instantiation

Now on substitution, let us first deal with types. Since there are no type variable binders, type variables can simply be replaced uniformly throughout a type, given a type substitution mapping variable names to types. I define tysubst i ty as the type obtained by instantiating ty according to the type substitution i. I say is_instance ty_0 ty if $\exists\, i.\ ty =$ tysubst i ty_0.

Substitution of terms for variables and of types for type variables in terms are the most complex operations we need to deal with. Naïve substitution for variables in a term may introduce unwanted binding, for example when substituting Comb v_1 t_1 for v_2 in Abs v_1 v_2 the variable v_1 ought to remain free. The algorithm for term substitution (subst) therefore renames bound variables as required to avoid unintended capture.

The algorithm for type instantiation (inst) in terms is also complicated by this kind of problem. Consider, with $x_1 =$ Var "x" (Tyvar "A") and $x_2 =$ Var "x" Bool, substitution of Bool for Tyvar "A" in Abs x_1 (Abs x_2 x_1). The inner x_1 refers to the outer binder, but after a naïve substitution (which makes $x_1 = x_2$) it would incorrectly refer to the inner binder. The solution is for the type instantiation algorithm to keep track of potential shadowing as it traverses the term, and if any occurs to backtrack and rename the shadowing bound variable.

In Harrison's original formulation of HOL in HOL, the main lemma about type instantiation takes 377 lines of proof script and mixes reasoning about name clashes with the semantics of instantiation itself. To clarify our formalisation, I formalise a small theory of nameless terms using de Bruijn indices, where substitution and instantiation are relatively straightforward, and shift the required effort to the task of translating to and from de Bruijn terms, which is somewhat easier than tackling capture-avoiding substitution directly. The analogous lemma about type instantiation in my formalisation is only 47 lines: the bulk of the work about name clashes appears in two lemmas totalling 166 lines about how instantiation can just as well be done on de Bruijn terms.[5]

I have proved that two terms are alpha-equivalent if and only if their de Bruijn representations are equal. Using this fact, the main theorems we obtain about

[5]Harrison's lemma is called `SEMANTICS_INST_CORE`, and mine are `INST_CORE_dbINST`, `INST_CORE_simple_inst`, and `termsem_simple_inst`.

substitution and instantiation are that they both respect alpha-equivalence:

$$\begin{array}{l}\vdash \mathsf{welltyped}\ t_1 \wedge \mathsf{welltyped}\ t_2 \wedge \mathsf{subst\_ok}\ \mathit{ilist} \wedge \mathsf{aconv}\ t_1\ t_2 \Rightarrow \\ \quad \mathsf{aconv}\ (\mathsf{subst}\ \mathit{ilist}\ t_1)\ (\mathsf{subst}\ \mathit{ilist}\ t_2) \\ \vdash \mathsf{welltyped}\ t_1 \wedge \mathsf{welltyped}\ t_2 \wedge \mathsf{aconv}\ t_1\ t_2 \Rightarrow \\ \quad \mathsf{aconv}\ (\mathsf{inst}\ \mathit{tyin}\ t_1)\ (\mathsf{inst}\ \mathit{tyin}\ t_2)\end{array}$$

Here subst_ok *ilist* means *ilist* is a substitution mapping variables to well-typed terms of the same type. Since I also prove that alpha-equivalent terms have the same semantics, these theorems allow us to prove soundness of the inference rules that do substitution and instantiation.

5.3.4 Theories

In my specification of HOL, every sequent carries a *theory*, which embodies information about constants and type operators and thereby allows us to support principles of definition. Formally, a theory (*thy*) consists of a signature (sigof *thy*) together with a set of axioms (axsof *thy*). The signature restricts the constants and type operators that may appear in a sequent, and the axioms provide sequents that may be derived immediately. The principles of definition introduce axioms to characterise the things that are defined.

A *signature* is specified as a pair of maps, (tysof *sig*,tmsof *sig*), assigning the defined type operator names to their arities and the defined term constant names to their types. Well-formed types obey the type signature:

$$\frac{}{\mathsf{type\_ok}\ \mathit{tysig}\ (\mathsf{Tyvar}\ x)} \qquad \frac{\mathsf{lookup}\ \mathit{tysig}\ \mathit{name} = \mathsf{Some}\ (\mathsf{length}\ \mathit{args}) \quad \mathsf{every}\ (\mathsf{type\_ok}\ \mathit{tysig})\ \mathit{args}}{\mathsf{type\_ok}\ \mathit{tysig}\ (\mathsf{Tyapp}\ \mathit{name}\ \mathit{args})}$$

And well-formed terms obey both signatures:

$$\frac{\mathsf{type\_ok}\ (\mathsf{tysof}\ sig)\ ty}{\mathsf{term\_ok}\ sig\ (\mathsf{Var}\ x\ ty)} \qquad \frac{\begin{array}{c}\mathsf{type\_ok}\ (\mathsf{tysof}\ sig)\ ty \\ \mathsf{lookup}\ (\mathsf{tmsof}\ sig)\ name = \mathsf{Some}\ ty_0 \\ \mathsf{is\_instance}\ ty_0\ ty\end{array}}{\mathsf{term\_ok}\ sig\ (\mathsf{Const}\ name\ ty)}$$

$$\frac{\begin{array}{c}\mathsf{term\_ok}\ sig\ tm_1 \\ \mathsf{term\_ok}\ sig\ tm_2 \\ \mathsf{welltyped}\ (\mathsf{Comb}\ tm_1\ tm_2)\end{array}}{\mathsf{term\_ok}\ sig\ (\mathsf{Comb}\ tm_1\ tm_2)} \qquad \frac{\begin{array}{c}\mathsf{type\_ok}\ (\mathsf{tysof}\ sig)\ ty \\ \mathsf{term\_ok}\ sig\ tm\end{array}}{\mathsf{term\_ok}\ sig\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ tm)}$$

Thanks to the conditions above that combinations are well-typed and abstractions must be of variables, we have $\vdash \mathsf{term\_ok}\ sig\ t \Rightarrow \mathsf{welltyped}\ t$.

A signature is *standard* if it maps the primitive type operators—function spaces and Booleans—and the primitive constant—equality—in the standard way:

$$\begin{array}{l}\mathsf{is\_std\_sig}\ sig \iff \\ \quad \mathsf{lookup}\ (\mathsf{tysof}\ sig)\ \texttt{"fun"} = \mathsf{Some}\ 2 \wedge \\ \quad \mathsf{lookup}\ (\mathsf{tysof}\ sig)\ \texttt{"bool"} = \mathsf{Some}\ 0 \wedge \\ \quad \mathsf{lookup}\ (\mathsf{tmsof}\ sig)\ \texttt{"="} = \mathsf{Some}\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ \mathsf{Bool}))\end{array}$$

We have a straightforward condition for a theory to be well-formed: all its components are well-formed and the axioms are Boolean terms.

$$\begin{array}{l}\mathsf{theory\_ok}\ thy \iff \\ \quad (\forall\, ty.\ ty \in \mathsf{range}\ (\mathsf{tmsof}\ thy) \Rightarrow \mathsf{type\_ok}\ (\mathsf{tysof}\ thy)\ ty) \wedge \\ \quad (\forall\, p.\ p \in \mathsf{axsof}\ thy \Rightarrow \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ p \wedge p\ \mathsf{has\_type}\ \mathsf{Bool}) \wedge \\ \quad \mathsf{is\_std\_sig}\ (\mathsf{sigof}\ thy)\end{array}$$

(Here $\mathsf{tmsof}\ thy$ is shorthand for $\mathsf{tmsof}\ (\mathsf{sigof}\ thy)$, and similarly for the types.)

5.4 Semantics

The idea behind the standard (e.g., Pitts [70]) semantics for HOL is to interpret types as non-empty sets and terms as their elements. Equality and function

application and abstraction are interpreted as in set theory, and a sequent is considered true if its interpretation is the true element of the set of Booleans. Semantics for HOL in this style are a mostly straightforward example of model theory.

The most fiddly parts of the semantics arise when dealing with polymorphic constants and type operators with arguments, followed closely by issues arising from substitution and type instantiation (which we covered in Section 5.3.3). Polymorphism is especially relevant to being able to support defined constants. The approach I have taken is to keep the treatment of constants and type operators separate from the semantics of the lambda-calculus terms, by parameterising the semantics by an interpretation, so that both pieces remain simple.

My goal is to show how to give semantics to sequents (and their component parts) in a theory. The ultimate notion we are aiming for is validity, $(thy,h) \models c$, which says that the semantics of c is true whenever the semantics of all the h are true and the axioms of *thy* are satisfied. Validity quantifies over, and hence does not need to mention the semantic parameters that give meaning to constants and variables. But these parameters, called interpretations and valuations, are required for building the definition of validity out of the semantics for the component parts of a sequent.

The details of the semantic apparatus are new, compared to Harrison's work [23] on HOL semantics in a fixed context without definitions, and are inspired by Arthan's specification [4] of ProofPower HOL's logic.

Semantics of types The meaning of a HOL type is a non-empty set. Thus, we require type valuations (τ) to assign type variables to non-empty sets.

$$\mathsf{is\_type\_valuation}\ \tau \iff \forall x.\ \mathsf{inhabited}\ (\tau\ x)$$

The type signature (*tysig* below) says what the type operators are and how many arguments they each expect. A type assignment (δ) gives semantics to type operators; we require it to assign correct applications of type operators to non-

empty sets.

$$\begin{array}{l}\mathsf{is\_type\_assignment}\ \mathit{tysig}\ \delta\ \Longleftrightarrow \\ \quad \mathsf{every} \\ \quad\quad (\lambda\,(\mathit{name},\mathit{arity}). \\ \quad\quad\quad \forall\, \mathit{ls}.\ \mathsf{length}\ \mathit{ls} = \mathit{arity} \wedge \mathsf{every}\ \mathsf{inhabited}\ \mathit{ls} \Rightarrow \mathsf{inhabited}\ (\delta\ \mathit{name}\ \mathit{ls})) \\ \quad\quad \mathit{tysig}\end{array}$$

The semantics of types simply maps the type valuation and type assignment through the type, as follows:

$$\begin{array}{l}\mathsf{typesem}\ \delta\ \tau\ (\mathsf{Tyvar}\ s) = \tau\ s \\ \mathsf{typesem}\ \delta\ \tau\ (\mathsf{Tyapp}\ \mathit{name}\ \mathit{args}) = \delta\ \mathit{name}\ (\mathsf{map}\ (\mathsf{typesem}\ \delta\ \tau)\ \mathit{args})\end{array}$$

Observe that since the type assignment (δ) is a function in HOL, there are not necessarily any set-theoretic functions involved in the semantics of type operators.

Semantics of terms The meaning of a HOL term is an element of the meaning of its type. Thus, a term valuation (σ) must assign each variable to an element of the meaning of its type. To speak of valid types and their meanings requires a type signature and type assignment, so the notion of a term valuation depends on them.

$$\begin{array}{l}\mathsf{is\_term\_valuation}\ \mathit{tysig}\ \delta\ \tau\ \sigma\ \Longleftrightarrow \\ \quad \forall\, v\ \mathit{ty}.\ \mathsf{type\_ok}\ \mathit{tysig}\ \mathit{ty} \Rightarrow \sigma\ (v,\mathit{ty}) \lessdot \mathsf{typesem}\ \delta\ \tau\ \mathit{ty}\end{array}$$

The constant signature (*tmsig* below) gives the names and types of the constants, and a term assignment (γ) must assign each constant to an element of the meaning of the appropriate type. This picture is complicated by the fact that constants may be polymorphic (that is, their types may contain type variables), so a term assignment takes not only the name of the constant but a list of meanings for the type variables, and the condition it must satisfy quantifies over type valuations. For any type valuation, the term assignment must assign the constant under that

type valuation to an element of the meaning of the constant's type.

$$\begin{array}{l}\mathsf{is\_term\_assignment}\ tmsig\ \delta\ \gamma\ \Longleftrightarrow \\ \quad \mathsf{every} \\ \quad\quad (\lambda\,(name,ty). \\ \quad\quad\quad \forall\,\tau. \\ \quad\quad\quad\quad \mathsf{is\_type\_valuation}\ \tau \Rightarrow \\ \quad\quad\quad\quad\quad \gamma\ name\ (\mathsf{map}\ \tau\ (\mathsf{sorted\_tyvars}\ ty)) \lessdot \mathsf{typesem}\ \delta\ \tau\ ty)\ tmsig\end{array}$$

The semantics of terms is defined recursively as follows. For variables, we simply apply the valuation.

$$\mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Var}\ x\ ty) = \sigma\ (x,ty)$$

For constants, we apply the interpretation but need to match the instantiated type of the constant against its generic type, that is, the type given for the constant in the signature. This is done using the instance function, explained in the next paragraph.

$$\begin{array}{l}\mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Const}\ name\ ty) = \\ \quad \mathsf{instance}\ tmsig\ (\delta,\gamma)\ name\ ty\ \tau\end{array}$$

Assuming[6] the given type is an instance of the generic type under some type substitution i, instance applies the term assignment for the constant passing the meanings of the types to which the type variables are bound under i.

$$\begin{array}{l}\mathsf{lookup}\ tmsig\ name = \mathsf{Some}\ ty_0 \Rightarrow \\ \quad \mathsf{instance}\ tmsig\ (\delta,\gamma)\ name\ (\mathsf{tysubst}\ i\ ty_0)\ \tau = \\ \quad\quad \gamma\ name\ (\mathsf{map}\ (\mathsf{typesem}\ \delta\ \tau \circ \mathsf{tysubst}\ i \circ \mathsf{Tyvar})\ (\mathsf{sorted\_tyvars}\ ty_0))\end{array}$$

For applications, we simply use function application at the set-theoretic level.

$$\begin{array}{l}\mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Comb}\ t_1\ t_2) = \\ \quad \mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ t_1\ \prime\ (\mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ t_2)\end{array}$$

[6] We leave unspecified the semantics of constants that are not in the signature or whose types do not match the type in the signature.

Similarly, for abstractions we create a set-theoretic function that takes an element, m, of the meaning of the type of the abstracted variable to the meaning of the body under the appropriately extended valuation.

$$\begin{array}{l}\mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ b) = \\ \quad \mathsf{Abstract}\ (\mathsf{typesem}\ \delta\ \tau\ ty)\ (\mathsf{typesem}\ \delta\ \tau\ (\mathsf{typeof}\ b)) \\ \quad\quad (\lambda\, m.\ \mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,((x,ty) \mapsto m)\ \sigma)\ b)\end{array}$$

Above, $((x,ty) \mapsto m)\ \sigma$ means the valuation that returns m when applied to (x,ty) but otherwise acts like σ.

Standard interpretations The semantics so far makes no special treatment of HOL's primitive types and constants; indeed, we can neatly factor out the special treatment as a condition on interpretations. First, we collect the parameters for terms and types together. A pair of a type valuation and a term valuation is called a *valuation*. Similarly, a pair of a type assignment and a term assignment is called an *interpretation*.

$$\begin{array}{l}\mathsf{is\_valuation}\ tysig\ \delta\ (\tau,\sigma) \iff \\ \quad \mathsf{is\_type\_valuation}\ \tau \wedge \mathsf{is\_term\_valuation}\ tysig\ \delta\ \tau\ \sigma \\ \mathsf{is\_interpretation}\ (tysig,tmsig)\ (\delta,\gamma) \iff \\ \quad \mathsf{is\_type\_assignment}\ tysig\ \delta \wedge \mathsf{is\_term\_assignment}\ tmsig\ \delta\ \gamma\end{array}$$

An interpretation is *standard* if it interprets the primitive constants in the standard way; namely, function types as set-theoretic function spaces, Booleans as the set of Booleans, and equality as set-theoretic equality (which is inherited from the meta-logic).

$$\begin{array}{l}\mathsf{is\_std\_type\_assignment}\ \delta \iff \\ \quad (\forall\, dom\ rng.\ \delta\ \texttt{"fun"}\ [dom;\ rng] = \mathsf{Funspace}\ dom\ rng) \wedge \\ \quad \delta\ \texttt{"bool"}\ [\,] = \mathsf{Boolset}\end{array}$$

$$
\begin{array}{l}
\mathsf{is\_std\_interpretation}\ (\delta,\gamma)\ \Longleftrightarrow \\
\quad \mathsf{is\_std\_type\_assignment}\ \delta\ \wedge \\
\quad \gamma\ \mathsf{interprets}\ \texttt{"="}\ \mathsf{on}\ [\texttt{"A"}]\ \mathsf{as} \\
\quad\quad (\lambda\, l. \\
\quad\quad\quad \mathsf{Abstract}\ (\mathsf{head}\ l)\ (\mathsf{Funspace}\ (\mathsf{head}\ l)\ \mathsf{Boolset}) \\
\quad\quad\quad\quad (\lambda\, x.\ \mathsf{Abstract}\ (\mathsf{head}\ l)\ \mathsf{Boolset}\ (\lambda\, y.\ \mathsf{Boolean}\ (x = y))))
\end{array}
$$

The notation used above is defined as follows:

$$
\begin{array}{l}
\gamma\ \mathsf{interprets}\ \mathit{name}\ \mathsf{on}\ \mathit{args}\ \mathsf{as}\ f\ \Longleftrightarrow \\
\quad \forall\, \tau.\ \mathsf{is\_type\_valuation}\ \tau \Rightarrow \gamma\ \mathit{name}\ (\mathsf{map}\ \tau\ \mathit{args}) = f\ (\mathsf{map}\ \tau\ \mathit{args})
\end{array}
$$

We will only be concerned with standard interpretations.

Satisfaction We now turn to packaging the basic semantics of types and terms up and lifting it to the level of sequents. A sequent, containing both hypotheses and a conclusion, represents an implication. An interpretation *satisfies* a sequent if the conclusion of the sequent is true whenever the hypotheses are (for all valuations). Precisely,

$$
\begin{array}{l}
(\delta,\gamma)\ \mathsf{satisfies}\ ((\mathit{tysig},\mathit{tmsig}),h,c)\ \Longleftrightarrow \\
\quad \forall\, v. \\
\quad\quad \mathsf{is\_valuation}\ \mathit{tysig}\ \delta\ v\ \wedge\ \mathsf{every}\ (\lambda\, t.\ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ v\ t = \mathsf{True})\ h \Rightarrow \\
\quad\quad\quad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ v\ c = \mathsf{True}
\end{array}
$$

We defer checking syntactic well-formedness of the sequent (for example, that c has type Bool) until the definition of validity below.

Modeling An interpretation *models* a theory if it is standard and satisfies the theory's axioms.

$$
\begin{array}{l}
i\ \mathsf{models}\ \mathit{thy}\ \Longleftrightarrow \\
\quad \mathsf{is\_interpretation}\ (\mathsf{sigof}\ \mathit{thy})\ i\ \wedge\ \mathsf{is\_std\_interpretation}\ i\ \wedge \\
\quad \forall\, p.\ p \in \mathsf{axsof}\ \mathit{thy} \Rightarrow i\ \mathsf{satisfies}\ (\mathsf{sigof}\ \mathit{thy},[\,],p)
\end{array}
$$

Validity Finally, a sequent is *valid* if every model of the sequent's theory also satisfies the sequent itself.

$$
\begin{array}{l}
(thy,h) \models c \iff \\
\quad \mathsf{theory\_ok}\ thy \wedge \mathsf{every}\ (\mathsf{term\_ok}\ (\mathsf{sigof}\ thy))\ (c::h) \wedge \\
\quad \mathsf{every}\ (\lambda\, p.\ p\ \mathsf{has\_type}\ \mathsf{Bool})\ (c::h) \wedge \mathsf{hypset\_ok}\ h \wedge \\
\quad \forall\, i.\ i\ \mathsf{models}\ thy \Rightarrow i\ \mathsf{satisfies}\ (\mathsf{sigof}\ thy, h, c)
\end{array}
$$

Chapter 6

A sound inference system for HOL

In the previous chapter, we saw what HOL sequents look like and how they are to be interpreted in set theory. In this chapter, we turn to the inference system used to construct derivations of sequents, and the initial axioms of HOL. After specifying the inference system, I prove that it is sound (that it preserves truth according to the semantics, and that the axioms are true), and therefore that HOL is consistent. For this consistency result to include all of HOL's axioms (including the axiom of infinity), I need to assume that the set theory used for the semantics also supports infinity.

6.1 Inference system

Whereas the notion of a derivable sequent in a particular theory depends only on the abstract formulation (signature plus axioms) of theories, when it comes to extending the theory with new definitions (and other extensions) I introduce the more concrete notion of a *context*. A context is a linear sequence of theory-extending[1] updates. This formulation corresponds nicely to the actual behaviour of an implementation of the inference system (that is, a theorem prover).

We first look at the (within-theory) inference rules, then turn to the rules for theory extension (definitions and non-definitional updates).

[1] Our updates have a finer granularity than HOL4 theory segments or Isabelle/HOL theories, which usually include multiple updates in our sense.

6.1.1 Inference rules

Recall that a sequent has the form (thy,h) |- c where thy is a theory, h is a set of hypotheses (Boolean terms) and c is the conclusion (another Boolean term). For simplicity, I represent the hypothesis set as a list but endeavour to ensure its elements are distinct (up to alpha-equivalence); we write the union of two such lists as $h_1 \uplus h_2$, removal of an element c from h as $h \setminus c$, and the image of h under f as map_set f h.

In the HOL Light kernel, there are ten inference rules. Like Harrison, I define an abbreviation for equations since they appear frequently:

$$s == t = \mathsf{Comb}\ (\mathsf{Comb}\ (\mathsf{Equal}\ (\mathsf{typeof}\ s))\ s)\ t$$

I also use the following helper functions: vfree_in v tm means v occurs free in tm, and subst_ok sig $ilist$ ensures only well-formed terms are substituted and only for variables of the same type. The rules are as follows:

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy \\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ t\end{array}}{(thy,[\,])\ \vdash\ t == t}\ \mathsf{REFL} \qquad \frac{\begin{array}{c}(thy,h_1)\ \vdash\ l == m_1 \\ (thy,h_2)\ \vdash\ m_2 == r \\ \mathsf{aconv}\ m_1\ m_2\end{array}}{(thy,h_1 \uplus h_2)\ \vdash\ l == r}\ \mathsf{TRANS}$$

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy \\ p\ \mathsf{has\_type\ Bool} \\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ p\end{array}}{(thy,[p])\ \vdash\ p}\ \mathsf{ASSUME} \qquad \frac{\begin{array}{c}(thy,h_1)\ \vdash\ p == q \\ (thy,h_2)\ \vdash\ p' \\ \mathsf{aconv}\ p\ p'\end{array}}{(thy,h_1 \uplus h_2)\ \vdash\ q}\ \mathsf{EQ\_MP}$$

$$\frac{\begin{array}{c}(thy,h_1)\ \vdash\ c_1 \\ (thy,h_2)\ \vdash\ c_2\end{array}}{(thy,h_1 \setminus c_2 \uplus h_2 \setminus c_1)\ \vdash\ c_1 == c_2}\ \mathsf{DEDUCT\_ANTISYM}$$

$$\frac{(thy,h_1) \vdash l_1 == r_1 \quad (thy,h_2) \vdash l_2 == r_2 \quad \mathsf{welltyped}\ (\mathsf{Comb}\ l_1\ l_2)}{(thy,h_1 \uplus h_2) \vdash \mathsf{Comb}\ l_1\ l_2 == \mathsf{Comb}\ r_1\ r_2}\ \mathsf{MK\_COMB}$$

$$\frac{\neg\mathsf{exists}\ (\mathsf{vfree\_in}\ (\mathsf{Var}\ x\ ty))\ h \quad \mathsf{type\_ok}\ (\mathsf{tysof}\ thy)\ ty \quad (thy,h) \vdash l == r}{(thy,h) \vdash \mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ l == \mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ r}\ \mathsf{ABS}$$

$$\frac{\mathsf{theory\_ok}\ thy \quad \mathsf{type\_ok}\ (\mathsf{tysof}\ thy)\ ty \quad \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ t}{(thy,[\,]) \vdash \mathsf{Comb}\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ t)\ (\mathsf{Var}\ x\ ty) == t}\ \mathsf{BETA}$$

$$\frac{\mathsf{subst\_ok}\ (\mathsf{sigof}\ thy)\ ilist \quad (thy,h) \vdash c}{(thy,\mathsf{map\_set}\ (\mathsf{subst}\ ilist)\ h) \vdash \mathsf{subst}\ ilist\ c}\ \mathsf{INST}$$

$$\frac{\mathsf{every}\ (\mathsf{type\_ok}\ (\mathsf{tysof}\ thy))\ (\mathsf{map}\ \mathsf{fst}\ tyin) \quad (thy,h) \vdash c}{(thy,\mathsf{map\_set}\ (\mathsf{inst}\ tyin)\ h) \vdash \mathsf{inst}\ tyin\ c}\ \mathsf{INST\_TYPE}$$

There is one additional way for a sequent to be provable, namely, if it is an axiom of the theory:

$$\frac{\mathsf{theory\_ok}\ thy \quad c \in \mathsf{axsof}\ thy}{(thy,[\,]) \vdash c}$$

Thus the new piece of the sequent syntax, the theory, interacts with the inference system (which remains essentially as formalised by Harrison) only via the axioms

of the theory and the checks that all types and terms respect the signature of the theory.

6.1.2 Theory extension

In the previous section, I defined provability within a fixed theory. To complete the inference system, we also need mechanisms for changing the theory. At this point, we take a more concrete view of theories, called contexts, by considering the specific changes that can be made. For simplicity, we restrict ourselves to a linear sequence of extensions and do not allow redefinition or branching or merging of theories. This linear view is sufficient for HOL Light; a more complicated model might be necessary for theorem provers like HOL4, which supports redefinition, or Isabelle/HOL [82], which supports both redefinition and context merging.

In the linear view, each change is an update and updates come in two kinds: definitional extensions (the first two) and postulates (a.k.a. axiomatic extensions, the last three).

$$
\begin{aligned}
&\mathit{update} = \\
&\quad \mathsf{ConstSpec}\ ((\mathit{string} \times \mathit{term})\ \mathit{list})\ \mathit{term} \\
&\ \mid \mathsf{TypeDefn}\ \mathit{string}\ \mathit{term}\ \mathit{string}\ \mathit{string} \\
&\ \mid \mathsf{NewType}\ \mathit{string}\ \mathit{num} \\
&\ \mid \mathsf{NewConst}\ \mathit{string}\ \mathit{type} \\
&\ \mid \mathsf{NewAxiom}\ \mathit{term}
\end{aligned}
$$

I call a list of such updates a *context*. From a context ($ctxt$) we can recover a theory ($\mathsf{thyof}\ ctxt$) by calculating the constants and axioms introduced by each kind of update. Postulates simply add new constants or axioms to the theory. I will specify exactly how the definitional updates extend a theory shortly.

Some basic well-formedness conditions are required. To specify the conditions under which an update is allowed to be made, I define a relation $upd\ \mathsf{updates}\ ctxt$ specifying when upd is a valid extension of $ctxt$. For example, the conditions for the postulates, which simply ensure names remain distinct and the each piece of the postulate is well-formed, are shown below.

$$
\frac{name \notin \mathsf{domain}\ (\mathsf{tysof}\ ctxt)}{\mathsf{NewType}\ name\ arity\ \mathsf{updates}\ ctxt}
\qquad
\frac{name \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt) \quad \mathsf{type\_ok}\ (\mathsf{tysof}\ ctxt)\ ty}{\mathsf{NewConst}\ name\ ty\ \mathsf{updates}\ ctxt}
$$

$$\frac{\textit{prop}\ \mathsf{has\_type}\ \mathsf{Bool} \qquad \mathsf{term\_ok}\ (\mathsf{sigof}\ \textit{ctxt})\ \textit{prop}}{\mathsf{NewAxiom}\ \textit{prop}\ \mathsf{updates}\ \textit{ctxt}}$$

A context *extends* another if it is a series of valid updates applied to the other. The initial context, supporting only equality, can be specified as an extension of the empty context:

```
init_ctxt =
 [NewConst "=" (Fun (Tyvar "A") (Fun (Tyvar "A") Bool));
  NewType "bool" 0; NewType "fun" 2]
```

We turn now to the changes introduced by definitional extensions and the conditions on making them. Let us start with the definition of new types, represented by the update TypeDefn *name pred abs rep*. Here *name* is the name of the new type and *pred* is a predicate on an existing type called the *representing type*. The intuition behind the principle of type definition is to make the new type isomorphic to the subset of the representing type carved out by *pred*, which is required to be inhabited. A type definition introduces the new type and also two constants between the new type and the representing type, asserting a bijection via the following two axioms.

```
axioms_of_upd (TypeDefn name pred abs_name rep_name) =
 (let abs_type = Tyapp name (sorted_tyvars pred) in
  let rep_type = domain (typeof pred) in
  let abs = Const abs_name (Fun rep_type abs_type) in
  let rep = Const rep_name (Fun abs_type rep_type) in
  let a = Var "a" abs_type in
  let r = Var "r" rep_type
  in
   [Comb abs (Comb rep a) == a;
    Comb pred r == (Comb rep (Comb abs r) == r)])
```

As can be seen in the construction of *abs_type* above, the new type has a type argument for each of the type variables appearing in *pred*. The type variables are sorted (according to their name) to ensure a canonical order for the new type's

arguments. The two introduced axioms assert that the introduced constants, *abs* and *rep*, are inverses (when restricted to elements of the representing type that satisfy *pred*).

The main condition on making a type definition is that the new type is non-empty. This is ensured by requiring a sequent asserting that the predicate holds of some witness. Additionally the predicate itself must not contain free variables, and the new names must not already appear in the context.

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt}, [\,])\ \mathsf{|-}\ \mathsf{Comb}\ \mathit{pred}\ \mathit{witness} \\ \mathsf{closed}\ \mathit{pred} \\ \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \\ \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \\ \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \\ \mathit{abs} \neq \mathit{rep}\end{array}}{\mathsf{TypeDefn}\ \mathit{name}\ \mathit{pred}\ \mathit{abs}\ \mathit{rep}\ \mathsf{updates}\ \mathit{ctxt}}$$

I will prove that context extension by type definition is sound, that is, the axioms it introduces are not contradictory, in Section 6.3.2.

Finally, let us look at the definition of new term constants via our new generalised rule for constant specification. The update ConstSpec *eqs prop*, where *eqs* are equations with variables (varsof *eqs*) on the left, signifies introduction of a new constant for each of the variables which together share the defining specification *prop*. Thus *prop* (after substituting the new constants for the variables) is the sole new axiom:

$$\begin{array}{l}\mathsf{axioms\_of\_upd}\ (\mathsf{ConstSpec}\ \mathit{eqs}\ \mathit{prop}) = \\ \quad (\mathtt{let}\ \mathit{ilist}\ =\ \mathsf{consts\_for\_vars}\ \mathit{eqs}\ \mathtt{in}\ [\mathsf{subst}\ \mathit{ilist}\ \mathit{prop}])\end{array}$$

The purpose of the equations is to provide witnesses that *prop* is satisfiable, so the rule takes as input a theorem concluding *prop* assuming the equations. The

complete list of conditions can be seen below:

$$\frac{\begin{array}{c}(\mathsf{thyof}\ ctxt,\mathsf{map}\ (\lambda\,(s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t)\ ==\ t)\ eqs)\ \mathsf{|-}\ prop \\ \mathsf{every}\ (\lambda\,t.\ \mathsf{closed}\ t \wedge \mathsf{closed\_tyvars}\ t)\ (\mathsf{map\ snd}\ eqs) \\ \forall\, x\ ty.\ \mathsf{vfree\_in}\ (\mathsf{Var}\ x\ ty)\ prop \Rightarrow \mathsf{member}\ (x,ty)\ (\mathsf{varsof}\ eqs) \\ \forall\, s.\ \mathsf{member}\ s\ (\mathsf{map\ fst}\ eqs) \Rightarrow s \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt) \\ \mathsf{all\_distinct}\ (\mathsf{map\ fst}\ eqs)\end{array}}{\mathsf{ConstSpec}\ eqs\ prop\ \mathsf{updates}\ ctxt}$$

Here closed_tyvars t means that type variables appearing in t also appear in typeof t. The design of this principle of constant specification is explained in detail by Arthan [5]. I prove its soundness in Section 6.3.2. The rule of constant definition, which defines a new constant x to be equal to a term t, can be recovered as an instance of constant specification:

$$\mathsf{ConstDef}\ x\ t = \mathsf{ConstSpec}\ [(x,t)]\ (\mathsf{Var}\ x\ (\mathsf{typeof}\ t)\ ==\ t).$$

6.2 Axioms

We need some logical connectives and quantifiers to state two of the axioms asserted in HOL Light. Since they are generally useful, it is convenient to define them first before asserting the axioms.

6.2.1 Embedding logical operators

The connectives of propositional logic and universal and existential quantifiers (ranging over HOL types) can be defined as constants[2] in HOL. I define a list of updates each of which defines a connective or quantifier as it is defined in HOL Light, and show, by calculating out the semantics, that they all behave as intended.

Each of the connectives and quantifiers can be defined by an equation, so we use the simple ConstDef *name term* version of the rule for constant specification. The following function extends a context with definitions of the Boolean

[2]In HOL theorem prover parlance these are sometimes collectively known as the theory of Booleans.

constants[3].

```
mk_bool_ctxt ctxt =
 ConstDef "~" NotDef::ConstDef "F" FalseDef::
   ConstDef "\\/" OrDef::ConstDef "?" ExistsDef::
   ConstDef "!" ForallDef::ConstDef "==>" ImpliesDef::
   ConstDef "/\\" AndDef::ConstDef "T" TrueDef::ctxt
```

Here the definition terms are as in HOL Light, for example,

```
ForallDef =
 Abs (Var "P" (Fun (Tyvar "A") Bool))
  (Var "P" (Fun (Tyvar "A") Bool) ==
   Abs (Var "x" (Tyvar "A")) (Const "T" Bool))

FalseDef =
 Comb (Const "!" (Fun (Fun Bool Bool) Bool))
  (Abs (Var "p" Bool) (Var "p" Bool))
```

I also specify the expected signature for constants with these names, for example,

```
is_forall_sig tmsig ⟺
 lookup tmsig "!" = Some (Fun (Fun (Tyvar "A") Bool) Bool)
```

and show, by simple calculation, that sigof (mk_bool_ctxt *ctxt*) has the right signatures.

For the desired semantics of the Boolean constants, we refer to the connectives and quantifiers in the meta-logic (that is, for us, the logic of HOL4). For example,

[3] The names, "\\/" and "/\\", associated with OrDef and AndDef may appear to include extra backslashes, because backslashes must be escaped in strings in HOL4. The names are intended to be textual representations of $\vee$ and $\wedge$.

for implication and universal quantification, we have

```
is_implies_interpretation γ ⟺ γ interprets "==>" on [] as (λ y. Boolrel (⇒))
is_forall_interpretation γ ⟺
 γ interprets "!" on ["A"] as
  (λ l.
    Abstract (Funspace (head l) Boolset) Boolset
     (λ P. Boolean (∀ x. x ≺ head l ⇒ P ' x = True)))
```

where the following helper functions interpret meta-level Booleans and relations on Booleans in our set theory:

```
Boolean b = if b then True else False
Boolrel R =
 Abstract Boolset (Funspace Boolset Boolset)
  (λ p. Abstract Boolset Boolset (λ q. Boolean (R (p = True) (q = True))))
```

The desired interpretations for all the Boolean constants are collected together as follows.

```
is_bool_interpretation (δ,γ) ⟺
  is_std_interpretation (δ,γ) ∧ is_true_interpretation γ ∧
  is_and_interpretation γ ∧ is_implies_interpretation γ ∧
  is_forall_interpretation γ ∧ is_exists_interpretation γ ∧
  is_or_interpretation γ ∧ is_false_interpretation γ ∧
  is_not_interpretation γ
```

The theorem I prove about the definitions of the Boolean constants says they have the desired semantics, that is, any interpretation that models a theory containing the definitions interprets the constants as specified by is_bool_interpretation.

```
⊢ theory_ok (thyof (mk_bool_ctxt ctxt)) ∧
   i models (thyof (mk_bool_ctxt ctxt)) ⇒
    is_bool_interpretation i
```

The semantics of a constant defined by an equation is uniquely specified, since that equation must be satisfied by any model of the definition. So, proving the

theorem above is simply a matter of calculating out the semantics of the definitions of each of the constants and observing that they match the specification.

6.2.2 Statement of the axioms

The standard library of HOL Light appeals to NewAxiom exactly three times, to assert the basic axioms of HOL that make it a classical logic and allow it to define the natural numbers. The axioms are: functional extensionality, choice, and infinity. Since the deeply-embedded syntax for the statements of the axioms is somewhat verbose, let us first look at their statements at the meta level:

- extensionality: $(\lambda\, x.\, f\ x) = f$
- choice: $P\ x \Rightarrow P\ ((\varepsilon)\ P)$
- infinity: $\exists (f : \mathit{ind} \to \mathit{ind}).\ \mathsf{ONE\_ONE}\ f \wedge \mathsf{ONTO}\ f$

While extensionality can be asserted in the initial context, the other two need additional constants to be added to the signature. For choice, we need to define implication, and to introduce the choice operator (ϵ above, but named `"@"` in the deep embedding.) For infinity, we need to introduce the type *ind* of individuals, and to define the existential quantifier[4], conjunction, and the ONE_ONE and ONTO functions.

I define context-updating functions for each of the axioms, asserting the axiom with NewAxiom after introducing new constants if necessary. These are defined below, with some of the deeply-embedded syntax abbreviated (SelectAx,

[4] Axioms do not need to universally quantify their variables: free variables act as if universally quantified because of the INST rule of inference.

InfinityAx, OntoDef, and OneOneDef).

```
mk_eta_ctxt ctxt =
 NewAxiom
  (Abs (Var "x" (Tyvar "A"))
    (Comb (Var "f" (Fun (Tyvar "A") (Tyvar "B"))) (Var "x" (Tyvar "A"))) ==
   Var "f" (Fun (Tyvar "A") (Tyvar "B")))::ctxt

mk_select_ctxt ctxt =
 NewAxiom SelectAx::NewConst "@" (Fun (Fun (Tyvar "A") Bool) (Tyvar "A"))::ctxt

mk_infinity_ctxt ctxt =
 NewAxiom InfinityAx::NewType "ind" 0::ConstDef "ONTO" OntoDef::
   ConstDef "ONE_ONE" OneOneDef::ctxt
```

6.3 Soundness

We have now seen HOL's inference system, which provides rules for proving sequents within a theory and updating that theory, and we have seen a specification of the meaning of such sequents: in particular, when they are considered valid. The main results of this chapter next are that every sequent proved by the inference system is valid (soundness), and its corollary that some sequents cannot be proved (consistency).

Soundness holds for both the inference rules and the rules for theory extension, with the exception of NewAxiom. For an extension rule to be sound, it must put the inference system in a state whereby it continues to produce only valid sequents. I ground this idea by proving the continued existence of a model of the theory. Since one cannot prove NewAxiom sound in general, I also need to prove the three axioms used in HOL Light to set up the initial HOL context sound on a case-by-case basis.

I prove consistency for any non-axiomatic extensions of the following contexts:

```
fhol_ctxt = mk_select_ctxt (mk_eta_ctxt (mk_bool_ctxt init_ctxt))
hol_ctxt = mk_infinity_ctxt fhol_ctxt
```

The name of the first context above stands for "finitary HOL" since it omits the

axiom of infinity. I name it separately because the consistency theorem we can prove of it has no assumptions apart from is_set_theory *mem*, which we saw in Section 5.2.2 can be discharged. The consistency theorem for the full hol_ctxt requires the set-theoretic axiom of infinity as an additional assumption.

6.3.1 Inference rules

The main soundness result for a fixed theory context is that every provable sequent is valid:

$$\vdash (thy,h) \mid\!- c \Rightarrow (thy,h) \models c$$

My proof of this does not differ substantially from Harrison's, apart from my indirect treatment of substitution and instantiation via de Bruijn terms. Recall that by convention we elide on the theorem above the assumption is_set_theory *mem* and the *mem* argument passed to the validity relation ($\models$).

The result above is proved by induction on the provability relation ($\mid\!-$). Thus we have a case for each of HOL's inference rules, for example for EQ_MP:

$$\vdash (thy,h_1) \models p == q \wedge (thy,h_2) \models p' \wedge \mathsf{aconv}\ p\ p' \Rightarrow (thy,h_1 \uplus h_2) \models q$$

The proof for each case typically expands out the semantics of the sequents involved then invokes properties of the set theory. The case for the rule allowing an axiom to be proved is trivial by the definition of validity which assumes the theory is modeled.

The main work in proving soundness of the inference rules is establishing properties of the semantics of the operations used by the inference rules in constructing their conclusions. For example, for instantiation of type variables in terms, I show that instead of instantiating the term we can instantiate the valuations:

$$\begin{array}{l}\vdash \mathsf{term\_ok}\ (tysig,tmsig)\ tm \Rightarrow \\ \quad \mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{inst}\ tyin\ tm) = \\ \quad\ \mathsf{termsem}\ tmsig\ (\delta,\gamma) \\ \quad\quad ((\lambda\, x.\ \mathsf{typesem}\ \delta\ \tau\ (\mathsf{tysubst}\ tyin\ (\mathsf{Tyvar}\ x))), \\ \quad\quad\ (\lambda\, (x,ty).\ \sigma\ (x,\mathsf{tysubst}\ tyin\ ty)))\ tm\end{array}$$

This lemma is the main support for the INST_TYPE case of the soundness theorem

$$\vdash \mathsf{every}\ (\mathsf{type\_ok}\ (\mathsf{tysof}\ thy))\ (\mathsf{map}\ \mathsf{fst}\ tyin) \wedge (thy,h) \models c \Rightarrow (thy,\mathsf{map\_set}\ (\mathsf{inst}\ tyin)\ h) \models \mathsf{inst}\ tyin\ c$$

since we need to establish conclusions about instantiations of terms from hypotheses about the terms themselves.

6.3.2 Theory extension

The definition of new types and constants extends the context in which sequents may be proved, in particular it changes the signature of the theory and introduces new axioms depending on the kind of definition. Intuitively, we do not want such extensions to invalidate previously proved sequents, nor do we want the definitions to introduce an inconsistency.

The first property, preserving existing sequents, is easy to prove because the only dependence of a term's semantics on the theory is via the signature of constants and type operators that appear in the term. Thus, as shown below, satisfaction is preserved as long as the context grows monotonically, that is, without changing the signature of existing constants and type operators (the syntax $f \sqsubseteq f'$ means that f' agrees with f on everything in $\mathsf{domain}\ f$).

$$\vdash tmsig \sqsubseteq tmsig' \wedge tysig \sqsubseteq tysig' \wedge \mathsf{every}\ (\mathsf{term\_ok}\ (tysig,tmsig))\ (c::h) \wedge i\ \mathsf{satisfies}\ ((tysig,tmsig),h,c) \Rightarrow i\ \mathsf{satisfies}\ ((tysig',tmsig'),h,c)$$

All of the context-updating rules are monotonic, since we do not allow redefinition.

The second desired property of an update, not introducing an inconsistency, is what we shall designate as making the update *sound*. To be precise, I call an update sound if any model of a theory before the update can be extended to a

model of the theory with the update:

$$\begin{array}{l}\mathsf{sound\_update}\ ctxt\ upd \iff \\ \quad \forall\, i. \\ \quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \Rightarrow \\ \quad\quad\quad \exists\, i'.\ \mathsf{equal\_on}\ (\mathsf{sigof}\ ctxt)\ i\ i' \wedge i'\ \mathsf{models}\ (\mathsf{thyof}\ (upd{::}ctxt))\end{array}$$

The constant equal_on helps formalise what we mean by one interpretation being an extension of another: they must be equal on terms and types in the previous context.

$$\begin{array}{l}\mathsf{equal\_on}\ sig\ i\ i' \iff \\ \quad (\forall\, name.\ name \in \mathsf{domain}\ (\mathsf{tysof}\ sig) \Rightarrow \mathsf{tyaof}\ i'\ name = \mathsf{tyaof}\ i\ name) \wedge \\ \quad \forall\, name.\ name \in \mathsf{domain}\ (\mathsf{tmsof}\ sig) \Rightarrow \mathsf{tmaof}\ i'\ name = \mathsf{tmaof}\ i\ name\end{array}$$

It is now simply a matter of showing that each of our rules for updating the context are sound when their side conditions are met.

It is straightforward to show that NewType and NewConst are sound, because they do not introduce any new axioms. We simply need to extend the interpretation with some plausible interpretation of the data. The extended interpretation cannot be completely arbitrary, because to be a model of a theory an interpretation must be well-formed (that is, must satisfy is_interpretation). But a well-formed extension is always possible: for example mapping each new type to the set of Booleans and each new constant to an arbitrary member of the interpretation of its type (which is non-empty since the original theory is modelled). I thereby prove the following theorems.

$$\begin{array}{l}\vdash \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge name \notin \mathsf{domain}\ (\mathsf{tysof}\ ctxt) \Rightarrow \\ \quad \mathsf{sound\_update}\ ctxt\ (\mathsf{NewType}\ name\ arity) \\ \vdash \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge name \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt) \wedge \mathsf{type\_ok}\ (\mathsf{tysof}\ ctxt)\ ty \Rightarrow \\ \quad \mathsf{sound\_update}\ ctxt\ (\mathsf{NewConst}\ name\ ty)\end{array}$$

Soundness of type definition A type definition, TypeDefn *name pred abs rep*, is sound if the two axioms it introduces (asserting the *abs* and *rep* constants form a bijection between the new type and the range of *pred*) can be made true by extending the original model with well-formed interpretations for the new type

and two new constants. Such an extension is always possible, thus we can prove the following:

$$\begin{aligned}\vdash\ & (\mathsf{thyof}\ \mathit{ctxt}, [\,])\ \mathsf{|-}\ \mathsf{Comb}\ \mathit{pred}\ \mathit{witness} \wedge \mathsf{closed}\ \mathit{pred} \wedge \\ & \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \wedge \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \\ & \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \mathit{abs} \neq \mathit{rep} \Rightarrow \\ & \quad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{TypeDefn}\ \mathit{name}\ \mathit{pred}\ \mathit{abs}\ \mathit{rep})\end{aligned}$$

The idea behind the proof is to interpret the new type as the subset of the representing type delineated by the semantics of *pred*, and to interpret the new constants as inclusion maps. When the *abs* constant is applied to a member of the representing type that is not in the new type, it simply picks an arbitrary element of the new type. The new type is guaranteed not to be empty by the theorem saying *pred* holds for some witness, which is required to make the type definition.

The proof of soundness of type definitions is the longest of the proofs about the rules for extension, taking around 400 lines of proof script compared to around 200 for constant specifications below and 40 for each of the other (non-definitional) updates. The reason is not that the soundness argument is significantly more complicated; rather, it is because the rule introduces many things (two axioms, two constants, and a type operator), where by contrast constant specification only introduces one axiom and introduces its constants uniformly; some work is required to calculate out the semantics of the equations in the axioms introduced by a type definition, and to ensure that each piece of the extension is well-formed.

Soundness of constant specification Specification of new constants, via ConstSpec *eqs prop*, introduces a single axiom, namely *prop* with its term variables replaced by the new constants, and is sound if the new constants are interpreted so as to make this axiom true. Such an interpretation is always possible

when the side-conditions of the rule are met, thus we have the following:

$$\begin{array}{l}
\vdash \mathsf{theory_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \wedge \\
\quad (\mathsf{thyof}\ \mathit{ctxt}, \mathsf{map}\ (\lambda\,(s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t) == t)\ \mathit{eqs})\ \mathsf{|-}\ \mathit{prop} \wedge \\
\quad \mathsf{every}\ (\lambda\, t.\ \mathsf{closed}\ t \wedge \mathsf{closed_tyvars}\ t)\ (\mathsf{map\ snd}\ \mathit{eqs}) \wedge \\
\quad (\forall\, x\ \mathit{ty}.\ \mathsf{vfree_in}\ (\mathsf{Var}\ x\ \mathit{ty})\ \mathit{prop} \Rightarrow \mathsf{member}\ (x,\mathit{ty})\ (\mathsf{varsof}\ \mathit{eqs})) \wedge \\
\quad (\forall\, s.\ \mathsf{member}\ s\ (\mathsf{map\ fst}\ \mathit{eqs}) \Rightarrow s \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})) \wedge \\
\quad \mathsf{all_distinct}\ (\mathsf{map\ fst}\ \mathit{eqs}) \Rightarrow \\
\qquad \mathsf{sound_update}\ \mathit{ctxt}\ (\mathsf{ConstSpec}\ \mathit{eqs}\ \mathit{prop})
\end{array}$$

The idea behind the proof is to interpret the new constants as the semantics of the witness terms (that is, map snd *eqs*) given in the input theorem that concludes *prop*. This works because then substitution of the new constants for the variables in *prop* has the same effect, semantically, as discharging the hypotheses of the input theorem.

The key lemmas required are about how the term semantics interacts with the interpretation and valuation. In particular, term substitution can be moved into the valuation; and, we can ignore extensions made to the interpretation when considering the semantics of a term that does not mention the new constants, since the semantics only cares about the interpretation of things in the signature.

$$\begin{array}{l}
\vdash \mathsf{welltyped}\ \mathit{tm} \wedge \mathsf{subst_ok}\ \mathit{ilist} \Rightarrow \\
\quad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{subst}\ \mathit{ilist}\ \mathit{tm}) = \\
\qquad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma \uplus \mathsf{map_subst}\ (\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma))\ \mathit{ilist})\ \mathit{tm}
\end{array}$$

$$\begin{array}{l}
\vdash \mathsf{is_std_sig}\ (\mathit{tysig},\mathit{tmsig}) \wedge \mathsf{term_ok}\ (\mathit{tysig},\mathit{tmsig})\ \mathit{tm} \wedge \mathsf{equal_on}\ (\mathit{tysig},\mathit{tmsig})\ i\ i' \Rightarrow \\
\quad \mathsf{termsem}\ \mathit{tmsig}\ i'\ v\ \mathit{tm} = \mathsf{termsem}\ \mathit{tmsig}\ i\ v\ \mathit{tm}
\end{array}$$

Above, $f \uplus \mathit{ls}$ means the function that maps according to a binding in *ls* if it exists else defaults to applying f; and map_subst g *ilist* modifies the substitution *ilist*, which binds variables to terms, by applying g to all the terms.

Using these lemmas, we can reduce showing that the new axiom is satisfied to showing that *prop* is true under a valuation assigning the variables to the interpretations of the new constants. Since we interpreted the new constants as the witness terms corresponding to each variable, this then follows directly from the input theorem.

Sequences of definitions Combining the results in this subsection, which cover soundness of all the update types except for NewAxiom, I prove that well-formed updates are sound.

$$\vdash \mathit{upd}\ \mathsf{updates}\ \mathit{ctxt} \wedge \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \wedge (\forall\, p.\ \mathit{upd} \neq \mathsf{NewAxiom}\ p) \Rightarrow \mathsf{sound\_update}\ \mathit{ctxt}\ \mathit{upd}$$

It is then a straightforward induction to show that a sequence of updates that do not introduce any axioms except via definitions preserve the existence of a model.

$$\vdash \mathit{ctxt}_2\ \mathsf{extends}\ \mathit{ctxt}_1 \wedge \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}_1) \wedge i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}_1) \wedge (\forall\, p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt}_2 \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt}_1) \Rightarrow \exists\, i'.\ \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt}_1)\ i\ i' \wedge i'\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}_2)$$

6.4 Consistency

We have seen that the inference system for HOL is sound in that every sequent it derives is semantically valid (provided all appeals to NewAxiom assert consistent axioms). As a corollary, we can show that there are some sequents which cannot be derived (since some sequents are not valid). My strategy for proving this syntactic notion of consistency is to use the fact, sometimes called semantic consistency, that every theory produced by the inference system has a model (as proved in the previous section).

I define a consistent theory as one for which there are sequents one of which can be derived and the other which cannot. In fact, I choose particular sequents for this purpose, an equation of equal variables and an equation of potentially different variables:

$$\begin{aligned}&\mathsf{consistent\_theory}\ \mathit{thy} \iff \\ &\quad (\mathit{thy},[\,])\ \mathsf{|-\ Var\ "x"\ Bool == Var\ "x"\ Bool} \wedge \\ &\quad \neg((\mathit{thy},[\,])\ \mathsf{|-\ Var\ "x"\ Bool == Var\ "y"\ Bool})\end{aligned}$$

Any theory with a model is consistent, as the following lemma demonstrates.

$$\vdash \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \forall\, \mathit{thy}.\ \mathsf{theory\_ok}\ \mathit{thy} \wedge (\exists\, i.\ i\ \mathsf{models}\ \mathit{thy}) \Rightarrow \mathsf{consistent\_theory}\ \mathit{thy}$$

We can prove the lemma by appeal to soundness: if the sequent equating two different variables were derivable, it would be valid (by soundness), and since the theory has a model it would be true in that model under every valuation. But it is not true under the valuation that sends Var "x" Bool to True and Var "y" Bool to False, so it cannot be derivable. As for the sequent equating equal variables, it is derivable as an instance of the REFL rule.

Now to show that hol_ctxt and all its non-axiomatic extensions are consistent theories, we have only to prove that the axioms asserted in hol_ctxt have a model.

I show that each of the axioms is consistent by proving: if the axiom is asserted in a theory that has a model, there is an extended interpretation that models the resulting theory. (This is the same idea as was formalised for sound_update, which I do not reuse since it only applies to a single update).

At this point, let us drop the convention of eliding the is_set_theory *mem* assumption from our theorems, to make clear which of the axioms depend on which facts about the set theory.

The semantics of the axiom of extensionality is true because set-theoretic functions are extensional, and HOL functions are interpreted as set-theoretic functions. No constants are introduced, so the interpretation does not need extending.

$$\begin{array}{l}\vdash \mathsf{is\_set\_theory}\ mem \Rightarrow \\ \quad \mathsf{is\_std\_sig}\ (\mathsf{sigof}\ ctxt) \Rightarrow \\ \quad\quad \forall\, i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \Rightarrow i\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_eta\_ctxt}\ ctxt))\end{array}$$

For the axiom of choice, the soundness theorem asserts existence of a model of the context extension produced by mk_select_ctxt, presuming the original context has a model, does not already define "@", and correctly interprets implication.

The theorem is as follows

$$
\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ mem \Rightarrow \\
\quad \texttt{"@"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt) \wedge \mathsf{is\_implies\_sig}\ (\mathsf{tmsof}\ ctxt) \wedge \\
\quad \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \Rightarrow \\
\quad\ \forall i. \\
\quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \wedge \mathsf{is\_implies\_interpretation}\ (\mathsf{tmaof}\ i) \Rightarrow \\
\quad\quad\ \exists i'. \\
\quad\quad\quad \mathsf{equal\_on}\ (\mathsf{sigof}\ ctxt)\ i\ i' \wedge \\
\quad\quad\quad i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_select\_ctxt}\ ctxt))
\end{array}
$$

To prove this theorem, we need to provide an interpretation of the Hilbert choice constant, `"@"`, that satisfies the axiom: given a predicate on some type it should return an element of the type satisfying the predicate if one exists, or else an arbitrary element of the type. A suitable interpretation can be constructed using the choice operator in the meta-logic, that is, the logic of HOL4 (whose properties imply the set-theoretic axiom of choice, as shown at the end of Section 5.2.1).

For the axiom of infinity, the statement of the soundness theorem follows essentially the same structure as for the axiom of choice, except it uses mk_infinity_ctxt instead of mk_select_ctxt and assumes the set-theoretic axiom of infinity. Additionally, there are more assumptions about the context—that it contains certain constants, and does not already contain others—so we can define ONE_ONE and

ONTO correctly. The theorem is as follows:

$$
\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ mem \wedge (\exists\, inf.\ \mathsf{is\_infinite}\ mem\ inf) \Rightarrow \\
\quad \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge \texttt{"ONTO"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt) \wedge \\
\quad \texttt{"ONE\_ONE"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt) \wedge \texttt{"ind"} \notin \mathsf{domain}\ (\mathsf{tysof}\ ctxt) \wedge \\
\quad \mathsf{is\_implies\_sig}\ (\mathsf{tmsof}\ ctxt) \wedge \mathsf{is\_and\_sig}\ (\mathsf{tmsof}\ ctxt) \wedge \\
\quad \mathsf{is\_forall\_sig}\ (\mathsf{tmsof}\ ctxt) \wedge \mathsf{is\_exists\_sig}\ (\mathsf{tmsof}\ ctxt) \wedge \\
\quad \mathsf{is\_not\_sig}\ (\mathsf{tmsof}\ ctxt) \Rightarrow \\
\quad\ \forall\, i. \\
\quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \wedge i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \wedge \\
\quad\quad \mathsf{is\_implies\_interpretation}\ (\mathsf{tmaof}\ i) \wedge \mathsf{is\_and\_interpretation}\ (\mathsf{tmaof}\ i) \wedge \\
\quad\quad \mathsf{is\_forall\_interpretation}\ (\mathsf{tmaof}\ i) \wedge \\
\quad\quad \mathsf{is\_exists\_interpretation}\ (\mathsf{tmaof}\ i) \wedge \mathsf{is\_not\_interpretation}\ (\mathsf{tmaof}\ i) \Rightarrow \\
\quad\quad\ \exists\, i'. \\
\quad\quad\quad \mathsf{equal\_on}\ (\mathsf{sigof}\ ctxt)\ i\ i' \wedge \\
\quad\quad\quad i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_infinity\_ctxt}\ ctxt))
\end{array}
$$

To prove this theorem, we need to provide an interpretation of the type of individuals in such a way that the axiom of infinity is satisfied. We can pick the infinite set *inf* whose existence is assumed. Then proving the theorem is simply a matter of calculating out the semantics and observing that the axiom holds because the set is infinite.

Having proved the soundness of each axiom separately, we can put them together within a single context and prove soundness for it and all its extensions (as long as they do not introduce further axioms). Recall the definitions of the contexts that assert the axioms:

$$
\begin{array}{l}
\mathsf{fhol\_ctxt} = \mathsf{mk\_select\_ctxt}\ (\mathsf{mk\_eta\_ctxt}\ (\mathsf{mk\_bool\_ctxt}\ \mathsf{init\_ctxt})) \\
\mathsf{hol\_ctxt} = \mathsf{mk\_infinity\_ctxt}\ \mathsf{fhol\_ctxt}
\end{array}
$$

We obtain the following results by combining the soundness theorems for the three axioms presented in this section with the result from Section 6.3.2 about

theory extensions that do not add any further new axioms.

$$\begin{array}{l}
\vdash \mathsf{is_set_theory}\ mem \Rightarrow \\
\quad \forall\, ctxt. \\
\quad\quad ctxt\ \mathsf{extends}\ \mathsf{fhol_ctxt} \wedge \\
\quad\quad (\forall\, p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{fhol_ctxt}) \Rightarrow \\
\quad\quad\quad \mathsf{theory_ok}\ (\mathsf{thyof}\ ctxt) \wedge \exists\, i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt)
\end{array}$$

$$\begin{array}{l}
\vdash \mathsf{is_set_theory}\ mem \wedge (\exists\, inf.\ \mathsf{is_infinite}\ mem\ inf) \Rightarrow \\
\quad \forall\, ctxt. \\
\quad\quad ctxt\ \mathsf{extends}\ \mathsf{hol_ctxt} \wedge \\
\quad\quad (\forall\, p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{hol_ctxt}) \Rightarrow \\
\quad\quad\quad \mathsf{theory_ok}\ (\mathsf{thyof}\ ctxt) \wedge \exists\, i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt)
\end{array}$$

The order in which the extensions are made ensure that the signature and interpretation assumptions of each of the soundness theorems for the axioms is satisfied.

Combining the lemma above with the results from the beginning of this section, the desired consistency theorems follow immediately.

$$\begin{array}{l}
\vdash \mathsf{is_set_theory}\ mem \Rightarrow \\
\quad \forall\, ctxt. \\
\quad\quad ctxt\ \mathsf{extends}\ \mathsf{fhol_ctxt} \wedge \\
\quad\quad (\forall\, p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{fhol_ctxt}) \Rightarrow \\
\quad\quad\quad \mathsf{consistent_theory}\ (\mathsf{thyof}\ ctxt)
\end{array}$$

$$\begin{array}{l}
\vdash \mathsf{is_set_theory}\ mem \wedge (\exists\, inf.\ \mathsf{is_infinite}\ mem\ inf) \Rightarrow \\
\quad \forall\, ctxt. \\
\quad\quad ctxt\ \mathsf{extends}\ \mathsf{hol_ctxt} \wedge \\
\quad\quad (\forall\, p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{hol_ctxt}) \Rightarrow \\
\quad\quad\quad \mathsf{consistent_theory}\ (\mathsf{thyof}\ ctxt)
\end{array}$$

The free variable *mem* in these theorems only appears in the assumptions, but those assumptions are of course necessary since we appealed to soundness, which depends on *mem* via the i models *thy* relation (and ultimately the semantics of terms and types).

Chapter 7

A verified implementation of the inference kernel

We have now seen that the HOL inference system, as specified by the provability relation (|-) and the rules for updating the context, is sound and consistent. Next, we turn our attention to producing a verified theorem prover implementing this sound inference system. Recall that our strategy is to produce the implementation in two steps: first, we define a theorem-prover kernel as recursive functions in a state-exception monad within the logic of HOL4, then we use an automated proof-producing technique to translate these recursive functions into code in the CakeML programming language.

7.1 The monadic functions

In implementations of HOL theorem provers, including the original OCaml implementation of HOL Light, the kernel module defines a datatype of theorems whose values correspond to the provable sequents of the HOL inference system. Our theorem datatype is defined with a single constructor as follows.

$$\mathit{thm} \;=\; \mathsf{Sequent}\ (\mathit{term\ list})\ \mathit{term}$$

In the implementation, the theory part of a sequent is not included on the theorem values, being instead embodied by the state of the theorem prover and the history of computations that led it into that state. The state of the theorem prover

consists of the following four values, which will be implemented as references in CakeML.

$$
\begin{array}{l}
\mathit{state} = \\
\quad \langle| \ \mathsf{the\_type\_constants} : ((\mathit{string} \times \mathit{num})\ \mathit{list}); \\
\quad\quad \mathsf{the\_term\_constants} : ((\mathit{string} \times \mathit{type})\ \mathit{list}); \\
\quad\quad \mathsf{the\_axioms} : (\mathit{thm}\ \mathit{list}); \\
\quad\quad \mathsf{the\_context} : (\mathit{update}\ \mathit{list})\ |\rangle
\end{array}
$$

The first three fields of the state correspond to references found in the original OCaml implementation of HOL Light. The fourth field represents the current context. As we saw when describing the inference system, the type constants, term constants, and axioms can all be calculated from the context, so it is redundant to include them all in the state. For efficiency, and faithfulness to the original, we do not discard the other three references in favour of the context; rather, we think of the context as a "ghost" variable, which we will prove is always consistent with the rest of the state but which is not actually required for the implementation. For clarity, we leave it in the implementation rather than as an existentially quantified variable on the correctness theorems.

The monadic functions only raise two kinds of exceptions: failure with an error message, and, in the implementation of instantiation of type variables within a term, a "clash" exception for backtracking when unintended variable capture is detected.

$$\mathit{exn} \ = \ \mathsf{Fail}\ \mathit{string} \mid \mathsf{Clash}\ \mathit{term}$$

With the models of state and exceptions in place, we define the state-exception monad ($\alpha\ M$) as follows.

$$
\begin{array}{l}
\alpha\ \mathit{result} \ = \ \mathsf{HolRes}\ \alpha \mid \mathsf{HolErr}\ \mathit{exn} \\
\alpha\ M = \mathit{state} \ \rightarrow \ \alpha\ \mathit{result} \times \mathit{state}
\end{array}
$$

We define monadic bind as would be expected (that is, either compute with the result or propagate the exception, and propagate the state in both cases), and make use of HOL4's support for `do` notation (as found also in Haskell) for composition of monadic binds.

Let us look now at how the monadic functions are defined. For example, here

is the function implementing the ASSUME rule of inference.

```
ASSUME tm =
 do
  ty ← type_of tm;
  bty ← mk_type ("bool",[]);
  if ty = bty then return (Sequent [tm] tm)
  else failwith "ASSUME: not a proposition"
 od
```

This definition uses auxiliary functions: type_of *tm* computes the type of *tm* (failing on ill-typed terms), mk_type (*name*,*args*) constructs a type operator (failing if the number of arguments does not match the current signature in the state's type-constants reference), and failwith *msg* raises the Fail *msg* exception. We define a function like the one above for each of the rules of inference and of definition, as well as for all the auxiliary functions, following the original OCaml implementation closely.

The monadic functions operate over the ***thm*** datatype, and re-use the underlying terms and types from the inference system. What we prove about them is that every computation preserves invariants on the values being computed. Importantly, the invariant on theorem values states that they are provable within the HOL inference system. The full list of invariants used, each of which is parametrised by the current context, is given below.

$$\mathsf{TYPE}\ ctxt\ ty \iff \mathsf{type\_ok}\ (\mathsf{tysof}\ ctxt)\ ty$$
$$\mathsf{TERM}\ ctxt\ tm \iff \mathsf{term\_ok}\ (\mathsf{sigof}\ ctxt)\ tm$$

$$\mathsf{THM}\ ctxt\ (\mathsf{Sequent}\ h\ c) \iff (\mathsf{thyof}\ ctxt,h)\ \mathsf{|-}\ c$$

$$\mathsf{STATE}\ ctxt\ state \iff$$
$$ctxt = state.\mathsf{the\_context} \wedge ctxt\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge$$
$$state.\mathsf{the\_type\_constants} = \mathsf{type\_list}\ ctxt \wedge$$
$$state.\mathsf{the\_term\_constants} = \mathsf{const\_list}\ ctxt$$

The STATE invariant requires the current context to be a valid extension (of init_ctxt). Thus preserving the STATE invariant entails only making valid updates

to the context.

For each monadic function, we prove that good inputs produce good output. For example, for the ASSUME function, we prove that, if the input is a good term and the state is good, then the state will be unchanged on exit and if the function returned successfully, the return value is a good theorem:

$$\begin{aligned}&\vdash \mathsf{TERM}\ ctxt\ tm \wedge \mathsf{STATE}\ ctxt\ s \wedge \mathsf{ASSUME}\ tm\ s = (res,s') \Rightarrow \\ &\quad s' = s \wedge \forall\, th.\ res = \mathsf{HolRes}\ th \Rightarrow \mathsf{THM}\ ctxt\ th\end{aligned}$$

This theorem is proved by stepping through the definition of ASSUME, and, at the crucial point where a Sequent value is created, observing that the assumptions for the ASSUME clause of the provability (|-) relation are satisfied, so the THM invariant holds.

We prove a similar theorem for each function in the kernel, showing that they implement the HOL inference system correctly. As another example, consider the rule for constant specification, which may update the state. We prove that the new state still satisfies our invariants, as does the returned theorem.

$$\begin{aligned}&\vdash \mathsf{THM}\ ctxt\ th \wedge \mathsf{STATE}\ ctxt\ s \Rightarrow \\ &\quad \texttt{case new\_specification}\ th\ s\ \texttt{of} \\ &\quad\ (\mathsf{HolRes}\ th,s') \ \Rightarrow\ \exists\, upd.\ \mathsf{THM}\ (upd{::}ctxt)\ th \wedge \mathsf{STATE}\ (upd{::}ctxt)\ s' \\ &\quad \mid (\mathsf{HolErr}\ exn,s') \ \Rightarrow\ s' = s\end{aligned}$$

7.2 Producing CakeML

The monadic functions constitute a *shallow embedding* of a theorem-prover-kernel implementation, because they are functions whose semantics is given implicitly by HOL (as implemented by HOL4): consider the fact that the ASSUME function has type *term → thm M*. In this section, we turn to production of a *deep embedding* of the same functions, with semantics given explicitly as the operational semantics of the CakeML programming language. In the deep embedding, the ASSUME function is a piece of syntax; its type is *dec*, that is, a CakeML declaration. Furthermore, since CakeML supports references and exceptions directly, the functions no longer need to be monadic.

We produce the deep embeddings from our shallow embeddings automatically, using the proof-producing translation technique described in Chapter 2. The

result of translation is syntax and a certificate theorem. For example, for the monadic ASSUME function, we obtain the following syntax (shown as abbreviated CakeML abstract syntax):

```
⊢ nth_element 99 ml_hol_kernel_decls =
    Dlet (Pvar "assume")
     (Fun "v3"
       (Let (Some "v2") (App [Var "type_of"; Var "v3"])
         (Let (Some "v1") (App [Var "mk_type"; Con None [... ... ; ... ]])
           (If (App Equality [Var "v2"; ... ... ])
             (Con "Sequent" [Con "::" [... ... ; ... ]; Var "v3"])
             (Raise
               (Con "Fail"
                 [Lit (StrLit "ASSUME: not a proposition")])))))))
```

The same code pretty-printed in CakeML concrete syntax:

```
fun assume v3 =
  let val v2 = type_of v3
      val v1 = mk_type ("bool",[])
  in
    if (v2 = v1)
    then (Sequent([v3],v3))
    else (raise Fail("ASSUME: not a proposition"))
  end;
```

The meaning of the declaration above is specified by the operational semantics of CakeML. The certificate theorem produced by translation connects evaluation of the declaration to the monadic function ASSUME:

```
⊢ DeclAssum (Some "Kernel") ml_hol_kernel_decls env tys ⇒
    EvalM env (Var "assume")
     ((PURE TERM_TYPE —M→ HOL_MONAD THM_TYPE) ASSUME)
```

Here, DeclAssum *mn decls env tys* means that (*env*,*tys*) is the environment (of

declared values and types) obtained by evaluating the list *decls* of declarations within a module *mn*; and, EvalM *env exp P* means that evaluation of the expression *exp* in environment *env* terminates and produces a result satisfying the refinement invariant P. In the theorem above, the refinement invariant takes the form $(A \xrightarrow{\text{M}} B)\ f$, which specifies a closure that, when applied to an input value satisfying A, terminates and produces an output value, which will satisfy B, according to the monadic function f.

To understand the guarantee provided by the certificate theorem, let us unpack the refinement invariant a little further (as we did in Chapter 2). The thing to remember is that the refinement invariants specify the relationship between certain HOL terms (values in the shallow embedding) and deeply-embedded CakeML values. For example, the following fact demonstrates how the THM_TYPE invariant relates values of type *thm* to CakeML values (ConV *name args* denotes a CakeML value made from application of a CakeML constructor):

$$\begin{array}{l} \vdash \mathsf{ListTy}\ \mathsf{TERM\_TYPE}\ [\,]\ v_1 \wedge \mathsf{TERM\_TYPE}\ tm\ v_2 \Rightarrow \\ \quad \mathsf{THM\_TYPE}\ (\mathsf{Sequent}\ [\,]\ tm) \\ \qquad (\mathsf{ConV}\ (\texttt{"Sequent"}, \mathsf{TypeId}\ (\mathsf{Long}\ \texttt{"Kernel"}\ \texttt{"thm"}))\ [v_1;\ v_2]) \end{array}$$

Here, we have CakeML values, v_1 and v_2, that are related by the refinement invariants for terms (and lists of terms) to the empty list and a term *tm*, and they are used to put together a CakeML value that is related to the theorem Sequent [] *tm*. The operators PURE and HOL_MONAD extend these refinement invariants to also relate the CakeML store (that is, the contents of references) and result (normal termination or raised exception) to the corresponding parts of the state-exception monad. (PURE lifts non-monadic values into the monad while HOL_MONAD works directly on a monadic value.)

Finally, $(A \xrightarrow{\text{M}} B)\ f$ is the refinement invariant for monadic functions as explained earlier. Thus using the certificate theorem for ASSUME we can prove in CakeML's operational semantics that the return value of any successful application of the deeply-embedded `assume` function will be related by the THM_TYPE invariant to the corresponding application of the monadic ASSUME function. And, as we saw in the previous section, the result of applying the monadic ASSUME function is related by the THM invariant to a sequent in the sound inference system (|-).

There are certificate theorems like this for every function in the CakeML implementation of the HOL Light kernel. It is on that basis that I make the claim that the kernel only produces theorem values that correspond to true sequents according to the semantics of HOL.

7.3 Compiling and packaging the kernel

By producing certificate theorems for the kernel functions, we have applied the first step of proof-grounded compilation to the kernel. The result of the first step is an implementation of the kernel in CakeML. The remaining two steps are to use evaluation in the logic to produce a compilation theorem, and then to use the compiler correctness theorem to verify the result of compilation. The result will be a verified implementation of the kernel in machine code.

The story for the kernel is slightly complicated by the fact that we only have certificate theorems for each kernel function, but ultimately want to package and compile the whole kernel as a single CakeML module. The correctness theorem for this module will need to capture the idea that any user code that runs after the kernel has been declared cannot produce a false theorem value, since the only way to produce theorem values is via the kernel. The precise formulation, and proof, of this correctness theorem is not complete, but is on the roadmap for the CakeML project over the next year.

Finally, to achieve self-verification, we need to replay the final overall proof of correctness for the machine-code implementation of the theorem prover in the verified theorem prover itself. Doing so is also future work, but the strategy is to extract the proof from HOL4 using OpenTheory and write a simple OpenTheory reader above the verified kernel. Although the technology to follow this strategy exists, the challenge will be scaling it up to the task. I anticipate that the OpenTheory proof of the implementation's correctness will be several hundreds of gigabytes large, and very slow both to extract and to replay.

Chapter 8

Related work

This dissertation describes work in two parts. The first concerns a verified compiler that can bootstrap, and the second concerns formalising a logic and, using the verified compiler, producing a verified theorem prover. In this chapter, I give an overview of the related work first to verification of compilers and bootstrapping and second to verification of theorem provers.

Our focus has been on end-to-end correctness, and, taking the two parts together, the aim is end-to-end verification of a complete application (the theorem prover). End-to-end correctness is a longstanding target in the area of formal verification. A substantial effort, with a similar focus, was made in the late 1980's by Computational Logic, Incorporated [11, 56], to produce a verified stack from applications down to hardware (i.e., below the machine code that has been our lowest level so far). Moore [57] writes that this project was very ambitious for its time, and the results fall short on realism and usefulness; however, he also notes that "the CLI stack was a *technology driver*", and indeed the now industrial-strength theorem prover, ACL2 [34], was one of its products. I hope that CakeML will form the basis of a more satisfying verified stack, and that it continues to also be a technology driver for the HOL theorem proving community. Hales [22] gives a nice overview, from the perspective of trusting a theorem prover, of the verified stack provided by CakeML.

8.1 Formalising compilation

In this section I review work relevant to Part I of this dissertation. I start with verified compilation, especially with a view to end-to-end correctness, and then look at related work on bootstrapping in this context.

8.1.1 Verified compilers

CompCert [46] is foremost amongst verified realistic compilers, being a compiler for C that includes verified optimisations and is deployed in the real world. Recent improvements to CompCert include validated parsing [32] and there are versions with the ability to do (verified) separate compilation [75]. CompCert is an algorithm that is verified in Coq, and the implementation of the compiler is extracted from Coq as an OCaml program before it is compiled and run. The correctness theorem covers the compilation algorithm for compiling whole programs down to assembly code. The trusted computing base for running the compiler includes the OCaml compiler and runtime and other build tools.

Unfortunately, it is not possible to immediately apply proof-grounded bootstrapping to CompCert to obtain a correctness theorem about its implementation. The reason is that the source and implementation languages of the compiler are very different, so it does not satisfy the second prerequisite for bootstrapping: that the compiler is written in its own source language. It may be possible to create a kind of proof-producing translation from Coq to C to fill this gap and enable proof-grounded bootstrapping of CompCert, but such a tool would be bridging a much larger gap (from Coq to C) than is bridged by the proof-producing translator from HOL to CakeML.

Considering verified compilers now for higher-order functional languages, the Lambda Tamer project [13] precedes our work on CakeML. Lambda Tamer includes a compilation algorithm, verified in Coq, from an ML-like language to an idealised assembly language. The emphasis is on clever choices of representations for formalisation that lead to highly automatic proofs: additional language features were added to the core proof of correctness with little manual intervention. The definition of the compiler uses dependent types, which are not present in its source language. To bootstrap this compiler one would need a more sophisticated proof-producing translator that can translate away dependent types.

Another example of verified compilation for a high-level language came out of the VerifiCard project [9], aimed at formalising a subset of Java as used on smartcards. This work predates CompCert, and highlighted the approach of generating executable code from a verified algorithm by using the "code extraction" facility of the theorem prover (in their case, Isabelle/HOL). While it provides a convenient route for producing executable code, which can further be integrated with code developed separately, this approach requires trust in the code extraction facility and in the compiler used on the extracted output. By contrast, with proof-grounded bootstrapping one need only trust "extraction" (really just printing) of machine code, where preservation of semantics is a simpler claim.

Returning to the theme of end-to-end verification, there have been several impressive projects developed recently with broadly similar goals to the original CLI stack endeavour. Chlipala's Bedrock [14] framework emphasises building end-to-end proofs from high level languages down to assembly code using modular interfaces, and comes with a great deal of proof automation. The specifications, proofs, and automation are all implemented in the Coq theorem prover. A recent example of the use of Bedrock covers end-to-end verification of web applications [15]. Gu et. al. [21] describe another modularity-focused approach to end-to-end correctness, also in Coq; their main application example is an operating-system kernel.

The Verified Software Toolchain [2, 3] is also geared towards end-to-end correctness. The particular approach is to build a program logic, in Coq, above the subset of C accepted by CompCert, enabling verification of source-level C programs that can then be compiled by CompCert. Since the program logic, Verifiable C, is proved sound with respect to CompCert C's semantics, properties proved about the source programs carry down to the generated assembly code.

Turning now to techniques for compiler verification, I mentioned in Chapter 3 that in verifying the CakeML compiler a choice was made to stick to highly syntactic relations to encode the invariants and correctness theorems for compilation between intermediate languages. In other words, I use a different relation for each compiler phase and it generally uses the specific translation made by the compiler in its definition without resorting to a more semantic correspondence. On the other end of the spectrum, there has been some substantial work on developing logical relations for compiler verification based on semantics, often highly type-directed, which promise a more compositional approach to compiler verifi-

cation and better support for things like separate compilation, linking, and the inclusion of manually verified code. Examples include work [8, 26, 27] by Benton, Hur, Dreyer, and others.

The closest work to the CakeML compiler and theorem prover is the verified Milawa theorem prover that runs on a verified Lisp compiler on the verified Jitawa runtime [61]. As with CakeML, the correctness theorems for Milawa and Jitawa are about implementations in machine code. However, the Lisp compiler used in Jitawa is not bootstrapped; rather, the whole compiler is verified using the decompilation techniques that in CakeML were used only for smaller libraries (garbage collector, lexer, etc.). The machine-code verification techniques (decompilation in particular) used in verifying the implementation of CakeML bytecode were also used recently in binary validation of the seL4 verified operating system microkernel [36]. Other work on machine code verification includes Jensen, Benton, and Kennedy's [31, 35] work that makes heavy use of dependent types for modelling machine code, and develops a higher-order separation logic above the model which is in some ways similar to Myreen's machine-code Hoare logic [58] used in decompilation.

8.1.2 Verified bootstrapping

Bootstrapping of verified compilers is less common in the literature than verified compilation in general. However, it is not without precedent.

An early reference can be found in work on the Verifix project (e.g., Goerigk and Hoffman [19]), which describes a bootstrapping process that is closer to traditional compiler bootstrapping than the proof-grounded bootstrapping method described in this thesis: the bootstrapping is used to introduce implementations of new language features into the compiler and thus there are many phases of bootstrapping. Goerigk and Hoffman's approach includes manual review of the output of bootstrapping, and considers trusting execution inside the theorem prover risky. I take the opposite view, and consider execution of the compiler inside the logic a much more trustworthy process than execution outside the logic followed by manual review. The Verifix view may have been influenced by the state of the art of theorem provers at the time, which may not have supported efficient execution that nevertheless produces theorems checked by a small kernel.

More recently, Strub et. al. [76] have proposed self-certification of type check-

ers, and demonstrated it with a bootstrapping type checker for F* (itself implemented in Coq). The idea is to write a type checker in the type system whose specifications it checks, and to give the type checker the specification that it correctly implements the type system. Then, run the type checker on itself to produce a certificate and check that certificate in the theorem prover. The point is to only check a single application of the type checker (application to itself) in the theorem prover and thereafter use the verified typechecker without need of the theorem prover. This is analogous to proof-grounded bootstrapping (of a compiler): we execute the compiler once (on itself) in the logic and thereafter can run the verified machine-code implementation outside of the theorem prover.

8.2 Formalising logic

The themes of the work described in Part II of this disseration are formalising (and mechanising) the syntax and semantics of logic, and verifying (or producing verified) theorem-prover implementations. In this section, I review prior work in these areas, factored by the particular logic under consideration.

8.2.1 Higher-order logic

There has been prior work on producing a formal (mechanised) specification of the semantics of HOL. The documentation for HOL4 includes a description, originally due to Pitts [70], of the semantics of HOL. However, this description is given in the traditional semiformal style of the mathematical logic literature. In the early 1990s, the development of the ProofPower logical kernel was informed by a formal specification in ProofPower-HOL of the proof development system, including a formalisation of the HOL language, logic and semantics. However, no formal proofs were carried out. The present work found several errors in the ProofPower formalisation of the semantics (all now corrected [4]). Pioneering work by von Wright [80] includes a mechanised formalisation of the syntax of HOL and its inference system (though no semantics). As mentioned in Section 5.1, Harrison's work [23] on a proof in HOL Light of the consistency of the HOL logical kernel without definitions formed the starting point for the present work (initially, [67, 40]).

Rather than formalising a specification of axiomatic set theory (which can

then be instantiated), Harrison [23] constructs his model of HOL in HOL at the same time as proving its requisite properties. In fact, his proof scripts allow one to choose (by rearranging comments) between an axiomatic formalisation of the set theory with an axiom of infinity or a conservative definition of the set theory without that axiom. More specifically, with the first option, he declares a new type, intended to be the universe of sets and asserts an axiom about its size, while in the second option he defines the type to be countably infinite. He then (in both options) uses what amounts to a type system (Harrison calls the types "levels") to define a coherent notion of membership in terms of injections into the universe. As a result, his sets are not extensional since there are empty sets at every level; because of this technicality his construction does not satisfy our is_set_theory.

Under the conservative option, Harrison still achieves a model without axiomatic extensions for HOL without the axiom of infinity, since he can prove his size axiom on a construction similar to our V_mem above. The disadvantage is that the set-theoretic axiom of infinity is provably false when one chooses the conservative option. In our approach, the polymorphism means that the set-theoretic axiom of infinity is unprovable rather than false, and so it is meaningful to prove theorems with that axiom as an assumption.

Harrison's use of levels is motivated by the desire to assert just one axiom: the cardinality property of the universe. With only this property, levels are required to distinguish different embeddings into the universe (e.g., to distinguish power-sets from cross products). Our approach with an explicit membership relation gives us a specification where these distinctions are explicit. We do not need to appeal to a theory of cardinalities in the meta-logic, since the assumptions we make (is_set_theory $mem \wedge \exists inf.$ is_infinite $mem\ inf$) mirror the standard axioms of Zermelo set theory.

Krauss and Schropp [38] have formalised a translation from HOL to set theory, automatically producing proofs in Isabelle/ZF [82]. Their motivation was to revive Isabelle/ZF by importing Isabelle/HOL proofs into it, but this task necessitates formalising an interpretation of HOL in set theory for which they use the standard approach (as we did) sending types to non-empty sets and terms to elements of their types. Although the Isabelle/HOL logic is slightly more complicated than the HOL we described, due to type classes and overloading, they remove the extra features in a preprocessing phase. They handle type defini-

tions and (equational) constant definitions by making equivalent definitions in Isabelle/ZF, which supports Isabelle's general facility for definitions.

8.2.2 Dependent type theory

Barras [7] has formalised a reduced version of the calculus of inductive constructions, the logic used by the Coq proof assistant [10], giving it a semantics in set theory and formalising a soundness proof in Coq itself. The approach is modular, and Wang and Barras [81] have extended the framework and applied it to the calculus of constructions plus an abstract equational theory.

Anand and Rahli [1] have formalised the semantics of NuPRL's type theory and proved soundness for its sequent calculus. The mechanisation is carried out within Coq. The semantics of NuPRL is rather more complex than of HOL, so its formalisation is impressive; on the other hand, they do not yet go so far as producing a verified implementation, but allude to the interesting possibility of producing it directly from the proof term for the soundness of the inference system.

8.2.3 First-order logic

Myreen and Davis [62] formalised Milawa's ACL2-like first-order logic and proved it sound using HOL4. This soundness proof for Milawa produced a top-level theorem which states that the machine-code which runs the prover will only print theorems that are true according to the semantics of the Milawa logic. Since Milawa's logic is weaker than HOL, it fits naturally inside HOL without encountering of the delicate foundational territory necessitating our $\mathsf{is\_set\_theory}\ mem$ and $\exists\, inf.\ \mathsf{is\_infinite}\ mem\ inf$ assumptions.

Other noteworthy prover verifications include a simple first-order tableau prover by Ridge and Margetson [72] and a SAT solver algorithm with many modern optimizations by Marić [50].

8.2.4 Comparison to Stateless HOL

The semantics (and inference system) described in Chapters 5 and 6 cleanly separates the semantics of types from the semantics of terms. It also uses an explicit theory, with an interpretation, to track which constants are defined, what

their semantics are, and the axioms. By contrast, Stateless HOL [83] puts types and terms in mutual recursion and has no separate theory parameter. Stateless HOL constants carry their definitions as syntactic tags (rather than in a separate signature), and the semantics interprets those tags directly (instead of using a separate interpretation parameter). My initial work (with others) on formalising HOL started with Stateless HOL, and is described in an ITP paper [40]. By keeping the theory and its interpretation separate in the present work, we gain the following advantages:

- The semantics of types and terms are no longer in mutual recursion, and are simpler to understand individually.

- We can more naturally use functions (typesem and termsem) for the semantics, instead of mutually recursive relations.

- Specific parameters to support the axioms of choice and infinity are no longer required within the semantics. Instead, they are handled generically by the type and term interpretations, applied to `"ind"` and `"@"`.

- We can support new axioms, beyond the initial three axioms asserted in HOL Light, in the same manner as the initial ones; the initial axioms are not baked into the semantics.

- The semantics of constants defined by new specification now properly captures the abstraction intended to be provided by that rule. The semantics are not tied to the specific witnesses given when the definition is made.

  In the Stateless HOL semantics, the semantics of a defined constant needs to be given in terms of the tag on the constant which provides the witnesses. By contrast in the current setup, the witnesses are only used in proving that the rule is sound (see Section 6.3.2). Since we now have an explicit interpretation of the constants, it can vary over many possible interpretations, constrained only by the axiom produced by the definition.

The primary motivation for Stateless HOL is the ability to "undo" definitions (this is achieved by soundly allowing simultaneous distinct definitions of constants with the same name). We did not take advantage of this ability in the verified implementation built under a Stateless HOL semantics [40], since we

first translated to a stateful implementation. If one wanted to support undoing definitions, Stateless HOL is still an option worth consideration, but based on the experience formalising it, I would first consider adding undo support to the context-based approach.

Additional advantages of Stateless HOL are that its kernel is purely functional, and therefore, one might think, would be easier to understand theoretically. After formalising Stateless HOL, I would now claim that the difficulty of verifying a stateful implementation (as in Section 7.2) is smaller than the difficulty of giving semantics to the mutually recursive datatypes of Stateless HOL especially when the rules of definition are included.

As an alternative approach to purely functional kernels, the OpenTheory [28] kernel achieves purity by a careful redesign of the interface to the kernel while maintaining the traditional idea of a context-extending mechanism for making definitions. The semantics of an OpenTheory article is specified via a stateful virtual machine, but the higher-level operations on the resulting OpenTheory packages are pure, and names are carefully managed, so definitions never accidentally collide or go out of date. I expect to be able to verify an OpenTheory proof checker against the HOL semantics of Chapter 5.

Chapter 9

Conclusion

In this chapter, we return to the notion of trusted computing base (TCB) and I examine how well the verified compiler for CakeML from Part I does from this perspective. Then, I summarise the results of Part II on the verified theorem prover for HOL implemented using CakeML. Finally, I describe planned future work for the CakeML project.

Unverified compilers usually contain bugs [84]. To reduce dependence on unverified tools, I suggest bootstrapping a verified compiler—compiling it with itself—in a proof-grounded way so that the correctness theorem applies to the final implementation that runs without further compilation. The proof-grounded bootstrapping method is mostly automatic. It uses proof automation techniques (proof-producing translation from shallow to deep embeddings, and evaluation in the logic) to push hard-won results about the correctness of compilation algorithms down to the level of real implementations.

The theorems that result from proof-grounded bootstrapping let us package a verified compiler implementation inside a larger machine-code program and prove a correctness theorem about the combined system. We used bootstrapping to eliminate compilation from the trusted computing base (TCB) of a read-eval-print loop (REPL) for CakeML, a machine-code program that contains the verified compiler for CakeML and calls it repeatedly at runtime.

9.1 Trusted computing base

What is in the trusted computing base for the CakeML REPL? The correctness theorem for the final implementation, Theorem 9, is written in terms of the semantics of x86-64. It has an assumption that the x86-64 machine starts in a correctly initialised state, and concludes that its behaviour implements the semantics of the CakeML REPL. To run the REPL implementation, we need to create the initial state, then we simply run the verified machine code. The TCB, therefore, consists of three things:

1. Verification: the software that checked the proof of Theorem 9, and our method for extracting the verified code, ReplX64, from the theorem statement.

2. Initialisation: the code used to create an initial machine state that satisfies the assumption of Theorem 9.

3. Execution: the hardware and operating system that runs the verified implementation. Our x86-64 semantics needs to capture the execution environment accurately.

What have we removed from the TCB by bootstrapping and packaging the compiler? Without bootstrapping, there would have been an additional item, after initialisation, about compilation from a verified algorithm to an executable, and the execution item would additionally include the language runtime. Without packaging, if we had merely verified a standalone compiler, there would have been additional initialisation and execution steps for running the output of the compiler. Thus, we have succeeded in removing trust in the compiler and runtime for running CakeML applications.

Now let us look more closely at what is left in the TCB, starting at the bottom with the execution environment. The x86-64 semantics we use is naive in two ways:

1. The semantics only covers user-mode instructions, and only a subset of them. This is particularly important for I/O: we simply assume it is possible to make system calls to read and write characters.

2. The semantics has a flat view of memory. We do not model virtual memory.

Trust in hardware is unavoidable, but it can be reduced with more accurate models. The hardware model can be made more accurate independently of the bootstrapping technique, which sits above it.

What we require of the operating system (if any) and memory subsystem is transparency: we leave them out of formal assumptions and thereby trust them to keep up the illusion of running on the hardware directly and without virtual memory. These items, together with initialisation code, represent realistic opportunities for more accurate modelling.

The initialisation code represents work traditionally done by a boot loader, or by a linker and loader. In theory, we could produce a boot loader to initialise a machine with the CakeML REPL implementation, which would then run "on bare metal". In practice, we write our initialisation code in a small (30 SLOC) C wrapper program, which includes the CakeML REPL machine code as inline assembly. We compile this C program with standard (unverified) tools. In this setup, the initialisation part of the TCB includes the C compiler and linker, and the operating system's loader. While we have theoretically avoided trusting a C compiler, we would need to formalise and verify linking and loading to produce a practical alternative to using a C compiler for initialisation.

Finally, we trust the theorem prover, and its execution environment (compiler, runtime, etc.), that we use to produce our verified implementation and to check its correctness theorem. Trust in the theorem prover is a methodological hazard of formal verification. However, it is not as bad as it sounds, because the real products of verification are *proofs* that can be checked independently. We must trust a theorem prover, but we are not constrained to a single one.

The question is again of practicality. In theory, we can export our proofs from HOL4 using OpenTheory [28] or similar technology, to be checked independently by OpenTheory itself or another theorem prover such as HOL Light, or indeed the verified theorem prover described in Part II. Such proofs are the sequences of primitive inferences that pass through the LCF-style kernel of HOL4. In practice, the proofs generated by automation like evaluation in the logic and translation from shallow to deep are extremely large and would require improved infrastructure to export. Possible directions for making independent checking more practical include compressing proofs as or after they are exported, or exporting proofs at intermediate levels rather than expanding everything out as primitive inferences. The latter demands greater sophistication from the inde-

pendent checker.

Proofs about realistic software are too large to be checked by hand. But, as we have seen in Part II, the required machine assistance can itself be verified. Specifically, we have considered verifying an implementation of a theorem prover and proving that it only produces theorems that are true according to the semantics of the logic. We have also considered self-verification, that is, a theorem prover that can verify its own implementation. Although such a self-verifying prover is an interesting and impressive achievement it does not eliminate our need to trust something altogether. There always remains the possibility that a self-verifying theorem prover is unsound in a way that causes it to incorrectly verify itself.

Ken Thompson, in his Turing award lecture, Reflections on Trusting Trust [78], describes a method by which a Trojan horse—deliberate mis-compilation of certain programs—can be inserted into a bootstrapped compiler while leaving no trace in the compiler's source code. The trick is to make the compiler introduce the Trojan horse, and code for introducing the Trojan horse, whenever it recognises that it has been given its own source code as input. Can a similar trick be used to introduce a Trojan horse into a compiler produced by proof-grounded bootstrapping? The crux of Thompson's example is that the compiler executable used to re-compile the compiler is already contaminated; in proof-grounded bootstrapping we do not use a compiler executable to compile the compiler, rather, we use evaluation in the logic of the theorem prover. Thus, to insert a Trojan horse we would need to contaminate the theorem prover to recognise when it is being asked to evaluate our compiler in the logic, or, perhaps simpler, when it is being asked to export the result of bootstrapping at which point it could substitute malicious machine code instead. Thompson's example simply reinforces the need to trust the verification tools we use, and is mitigated as explained above by independent checking of proofs.

9.2 Verified theorem prover

A theorem prover is a computer program whose correctness can be understood at many levels. At the highest level, we focus on the logic and its semantics, and on the particular inference system, which should be sound with respect to the semantics and therefore consistent. At the next level down, we consider whether

the inference system is implemented correctly, that is, whether the (abstract) computations performed by the theorem prover correspond to construction of derivations in the inference system. The remaining levels all concern correct implementation of those computations at more concrete levels of abstraction, from a high-level programming language down to hardware. In Part II we have dealt with correctness of a theorem prover for higher-order logic (HOL) spanning all the levels between consistency of the logic itself and implementation in a high-level programming language (CakeML), within a single mechanically-checked formalisation.

Part II goes further than previous work in this vein in two directions: the coverage of the logic formalised and the concreteness of the theorem-prover implementation verified. My formalisation, with full support for making extensions to the context, now covers all of HOL as it is implemented by real theorem provers. My implementation is a deeply-embedded program verified against the operational semantics of a realistic programming language. On both fronts, however, the end of the line has not been reached: one might like to verify a more sophisticated approach to contexts (such as the one implemented by Isabelle [82]), and a more concrete implementation (for example, covering more of the execution environment). Additionally, I have so far only verified the kernel of a theorem prover, and would like to extend the result to a complete theorem prover, which means formally validating the LCF design [52] by reasoning about the guarantees provided by a protected (abstract) type in CakeML.

In constructing a formal specification of the semantics of HOL that is suitable both for proofs about the logic and inference system, and for proofs about implementing that inference system, I faced several design decisions. The main theme of the lessons learned is to value explicitness and separation of concerns. Using an explicit theory context gives a simpler semantics than that of Stateless HOL, as was discussed in Section 8.2.4. Similarly, specifying the axioms of set theory with an explicit membership relation yields a development that is easier to work with than the theoretically equivalent approach based solely on a cardinality assumption. And by specifying the set theory separately from defining an instance of it, we obtain a conservative approach using isolated assumptions about free variables rather than global axiomatic extensions. On a smaller scale, the choice to factor reasoning about substitution and instantiation, which is complicated with name-carrying terms, through a separate small theory about de Bruijn terms led

to simplifications.

Continuing the self-verification project initiated by Harrison [23] for HOL Light, my formalisation of HOL is conducted within HOL itself. It is common to cite Gödel's incompleteness theorems as making it meaningless or impossible for a logical system to be used to prove its own properties. The risk with such proofs is that they degenerate into an uninteresting tautology. In the present work, however, what we have done is analogous to proving the soundness of first-order logic within a first-order formalisation of ZF set theory, which as standard logic textbooks (e.g., Mendelson [51]) show is well-known and uncontroversial.

The theorem prover I use (HOL4) is distinct from the verified implementation produced (in CakeML, of a kernel based on HOL Light's kernel). The implementation of HOL4 is not itself verified; one might wonder whether we gain anything by trusting one theorem prover to verify another of a similar (in fact lesser) complexity. While I acknowledge this objection, my reply is that HOL4 can be seen merely as a tool to help us organise the development if we consider the fact that the proofs can be exported from HOL4 (for example, via OpenTheory [28]) for independent checking. Thus although something ultimately needs to be trusted, I do not require it to be HOL4. A second reply is that the exercise of developing the formalisation leads us to clarify our thinking about the systems under consideration, and, on the implementation side, uncovers the kinds of bugs that are likely to occur in theorem provers in practice [23].

9.3 Future work

As mentioned in Section 9.1, a promising line for future work is to integrate the correctness theorem for a packaged compiler implementation with verified tools for linking and loading. Such an integration would let us replace the initialisation code in the trusted computing base with a formal semantic model of linking and loading, and our compiler would produce an executable (e.g., an ELF) verified against this semantics rather than raw machine code. Similarly, we are considering what it would take to run the CakeML REPL as a verified user application on the seL4 verified operating system [36] and thereby remove the operating system from the execution part of the TCB.

Because CakeML does not support I/O primitives directly, we had to resort

in Section 4.5 to tricks using references to give the REPL I/O at the top level. More seriously, to mix bootstrapped and non-bootstrapped code we had to use the subtle method of simulating two different bytecode machine states. We are currently investigating a more straightforward approach to producing a packaged compiler that adds both I/O and dynamic installation of new code as primitives to the source language, thereby allowing a REPL implementation to be written entirely in the source language.

Finally, I would like to follow through on the plan to create a self-verifying theorem prover. Concretely, one would export the theorem asserting that the kernel is verified and replay that proof in the verified kernel itself. Checking a correctness proof about its own concrete implementation would be closer to self-verification than any theorem prover has yet achieved. Of course, such a check does not rule out the possibility that the theorem prover is not sound, because it might be broken in such a way that it fails to detect an incorrect correctness proof. But we would have high confidence in a theorem prover with such an ability (alongside other evidence for soundness, like a readable, concise implementation) and would expect the practical facility required for self-verification to be useful for tackling more ambitious software verification challenges.

Bibliography

[1] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Klein and Gamboa [37], pages 27–44.

[2] Andrew W. Appel. Verified software toolchain - (invited talk). In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

[3] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 5th ed. edition, 2014.

[4] Rob Arthan. HOL formalised: Semantics. `http://www.lemma-one.com/ProofPower/specs/spc002.pdf`.

[5] Rob Arthan. HOL constant definition done right. In Klein and Gamboa [37], pages 531–536.

[6] Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2000.

[7] Bruno Barras. Sets in Coq, Coq in sets. *J. Formalized Reasoning*, 3(1):29–48, 2010.

[8] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 97–108. ACM, 2009.

[9] Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. *Electr. Notes Theor. Comput. Sci.*, 82(2):377–394, 2003.

[10] Yves Bertot. A short presentation of Coq. In Mohamed et al. [55], pages 12–16.

[11] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.

[12] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

[13] Adam Chlipala. A verified compiler for an impure functional language. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 93–106. ACM, 2010.

[14] Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402. ACM, 2013.

[15] Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In Rajamani and Walker [71], pages 609–622.

[16] Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[17] John Field and Michael Hicks, editors. *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM, 2012.

[18] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[19] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous compiler implementation correctness: How to prove the real thing correct. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods - FM-Trends 98, International Workshop on Current Trends in Applied Formal Method, Boppard, Germany, October 7-9, 1998, Proceedings*, volume 1641 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 1998.

[20] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Principles of Programming Languages (POPL)*, pages 119–130. ACM Press, 1978.

[21] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Rajamani and Walker [71], pages 595–608.

[22] Thomas C. Hales. Developments in formal proofs. *CoRR*, abs/1408.6474, 2014.

[23] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[24] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.

[25] Leon Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15:81–91, 1950.

[26] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 133–146. ACM, 2011.

[27] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and Kripke logical relations. In Field and Hicks [17], pages 59–72.

[28] Joe Hurd. The OpenTheory standard theory library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.

[29] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[30] Myra Van Inwegen and Elsa L. Gunter. HOL-ML. In Joyce and Seger [33], pages 61–74.

[31] Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 301–314. ACM, 2013.

[32] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.

[33] Jeffrey J. Joyce and Carl-Johan H. Seger, editors. *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93,*

Vancouver, BC, Canada, August 11-13, 1993, Proceedings, volume 780 of *Lecture Notes in Computer Science*. Springer, 1994.

[34] Matt Kaufmann and J Strother Moore. An ACL2 tutorial. In Mohamed et al. [55], pages 17–21.

[35] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 13–24. ACM, 2013.

[36] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.

[37] Gerwin Klein and Ruben Gamboa, editors. *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.

[38] Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 323–338. Springer, 2010.

[39] Ramana Kumar. Challenges in using OpenTheory to transport Harrison's HOL model from HOL Light to HOL4. In Jasmin Christian Blanchette and Josef Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9-10, 2013*, volume 14 of *EPiC Series*, pages 110–116. EasyChair, 2013.

[40] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In Klein and Gamboa [37], pages 308–324.

[41] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic. *Journal of Automated Reasoning*, 2015. Submitted.

[42] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014.

[43] Ramana Kumar, Magnus O. Myreen, Scott Owens, and Yong Kiam Tan. Proof-grounded bootstrapping of a verified compiler: Producing a verified read-eval-print loop for CakeML. *Journal of Automated Reasoning*, 2015. Submitted.

[44] Casimir Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1):161–171, 1921.

[45] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 173–184. ACM, 2007.

[46] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[47] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[48] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 461–472. ACM, 2013.

[49] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008.

[50] Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.

[51] Elliott Mendelson. *Introduction to mathematical logic. 5th ed.* Boca Raton, FL: CRC Press, 5th ed. edition, 2009.

[52] Robin Milner. LCF: A way of doing proofs with a machine. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 146–159. Springer, 1979.

[53] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML.* MIT Press, 1990.

[54] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: revised*. MIT press, 1997.

[55] Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.

[56] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[57] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2002.

[58] Magnus O. Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, 2008.

[59] Magnus O. Myreen. Reusable verification of a copying collector. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2010.

[60] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael

Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2013.

[61] Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *ITP*, volume 6898 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2011.

[62] Magnus O. Myreen and Jared Davis. The reflective Milawa theorem prover is sound - (down to the machine code that runs it). In Klein and Gamboa [37], pages 421–436.

[63] Magnus O. Myreen and Michael J. C. Gordon. Function extraction. *Sci. Comput. Program.*, 77(4):505–517, 2012.

[64] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic - improved. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 78–81. IEEE, 2012.

[65] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *ICFP*, pages 115–126. ACM, 2012.

[66] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.

[67] Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL Light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 490–495. Springer, 2013.

[68] Scott Owens. A sound semantics for OCamllight. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[69] Lawrence C. Paulson. A higher-order implementation of rewriting. *Sci. Comput. Program.*, 3(2):119–149, 1983.

[70] Andrew M. Pitts. *The HOL System: Logic*, 3rd edition. `http://hol.sourceforge.net/documentation.html`.

[71] Sriram K. Rajamani and David Walker, editors. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 2015.

[72] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first-order logic. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2005.

[73] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 379–391. ACM, 2009.

[74] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [55], pages 28–32.

[75] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In Rajamani and Walker [71], pages 275–287.

[76] Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. In Field and Hicks [17], pages 571–584.

[77] Don Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [33], pages 43–60.

[78] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.

[79] Robert L. Vaught. *Set theory: An introduction*. Basel: Birkhäuser, 2nd edition, 1994.

[80] Joakim von Wright. Representing higher-order logic proofs in HOL. *Comput. J.*, 38(2):171–179, 1995.

[81] Qian Wang and Bruno Barras. Semantics of intensional type theory extended with decidable equational theories. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 653–667. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[82] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [55], pages 33–38.

[83] Freek Wiedijk. Stateless HOL. In Tom Hirschowitz, editor, *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009.*, volume 53 of *EPTCS*, pages 47–61, 2009.

[84] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 283–294. ACM, 2011.