Fakultät für Informatik
Technische Universität München

Verified Code Generation from Isabelle/HOL

Lars Richard Hupel

Vollständiger Abdruck der von der Fakultät der Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 12.03.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.06.2019 angenommen.

Abstract

In this thesis, I develop a verified compilation toolchain from executable specifications in Isabelle/HOL to CakeML abstract syntax trees. This improves over the state-of-the-art in Isabelle by providing a trustworthy procedure for code generation. The work consists of three major contributions.

First, I have implemented a certifying routine to eliminate type classes and instances in Isabelle specifications. Based on defining equations of constants, it derives new definitions that do not use type classes. This can be used to bypass an unverified step in the current code generator.

Second, I formalized an algebra for higher-order λ-terms that generalizes the notions of free variables, matching, and substitution. Terms can be thought of as consisting of a generic (free variables, constants, application) and a specific part (abstraction, bound variables). With this algebra, it becomes possible to reason abstractly over a variety of different types.

These two parts are independent from each other and can also be used for other purposes. For example, I have successfully instantiated the term algebra for other term types in the Isabelle universe.

Third, a compiler that works similarly to the existing code generator, but produces a CakeML abstract syntax tree together with a correctness theorem. More precisely, I have combined a simple proof producing translation of recursion equations in Isabelle into a deeply embedded term language with a fully verified compilation chain to the target language CakeML.

Acknowledgements

Firstly, I would like to thank my advisor Tobias Nipkow. His guidance and the close collaboration was invaluable. I had not even finished my Master's thesis when Tobias suggested to work on this topic as a *Doktorand.* I thoroughly enjoyed my time in his research group, including the freedoms, the archive, the teaching, and the colleagues. I had the pleasure of working with Mohammad Abdulaziz, Jasmin Blanchette, Julian Brunner, Manuel Eberl, Florian Haftmann, Maximilian Haslbeck (the elder), Johannes Hölzl, Fabian Immler, Ondřej Kunčar, Peter Lammich, Lars Noschinski, Andrei Popescu, Dimitriy Traytel, Simon Wimmer and Bohua Zhan.

I am grateful to the numerous people who have read and commented on early versions of this thesis. They are, in alphabetical order: Mohammad Abdulaziz, Jasmin Blanchette, Adelbert Chang, Manuel Eberl, Peter Lammich, Maximilian Haslbeck (the elder), Marek Kubica, Magnus Myreen, Tobias Pflug, and George Pirlea. This also applies to the anonymous reviewers at POPL, ESOP, and Fundamenta Informaticae. Additionally, I had long and fruitful discussions about some of the subjects with Konrad Slind and Freek Wiedijk.

I would also like to thank everyone involved in the CakeML project with whom I had the pleasure to collaborate with, both in person at Chalmers and remotely. Those are, in no particular order, Ramana Kumar, Magnus Myreen, Scott Owens, Yong Kiam Tan, Johannes Åman Pohjola, Oskar Abrahamsson, Alejandro Gómez Londoño, Andreas Lööw, Michael Sproul, and Michael Norrish. Thanks to Magnus for hosting me twice in Chalmers and encouraging me to continue my work in this direction.

My intern for two months, Yu Zhang, has contributed some of the proofs that are relevant for the CakeML semantics in Isabelle and the compiler. That was some great work!

Last but not least, my parents always encouraged me to aim high. Their support was what made my entire academic path possible, starting back in high school.

Contents

1 Introduction

It's called CakeML 'cause it's f**king sweet, mate!

(Raf Kolanski)

The first part of this thesis is concerned with the generation of executable code in a functional programming language from specifications in an interactive theorem prover. I picked the *CakeML* language (§1.2) and the *Isabelle* prover (§1.1). This chapter will give an introduction to those two systems and give a motivation why a connection between them is desirable (§1.3).

1.1 Isabelle

Isabelle is a system in the category of *interactive theorem provers* [97]. These systems provide a facility for *machine-checked proofs:* a user may present a proof phrased in a formal language to the system which will then check its correctness according to the logic implemented in the system.

Working with *interactive* theorem provers has often been compared to playing a video game, in which the user's objective is to navigate some rules (the logics) to achieve their goal (the proof), while constantly facing adversities (flaws). Interactivity here means that feedback is immediate and the user can change their reasoning steps incrementally, repeated until both user and machine are satisfied. To that end, Isabelle ships with an integrated development environment providing similar amenities to those of industry-strength programming languages.

The immediate question is the trustworthiness of Isabelle itself. As is common in the *LCF* family, its implementation strategy is to clearly delineate a *kernel* providing a set of primitive inference rules and bookkeeping of definitions and other logical content. All reasoning goes through this kernel, which – being only a small part of the full system – can be inspected by critical readers for flaws. The kernel is implemented in the programming language Standard ML whose strong abstraction and type soundness guarantees make it particularly suitable for proof assistants. In fact, Standard ML's predecessor, *Meta Language* (*ML* for short) has been designed by Robin Milner as the implementation language for the LCF proof assistant [107, §6]. Furthermore, there is research on verification of proof kernels of similar systems themselves [72, 75].

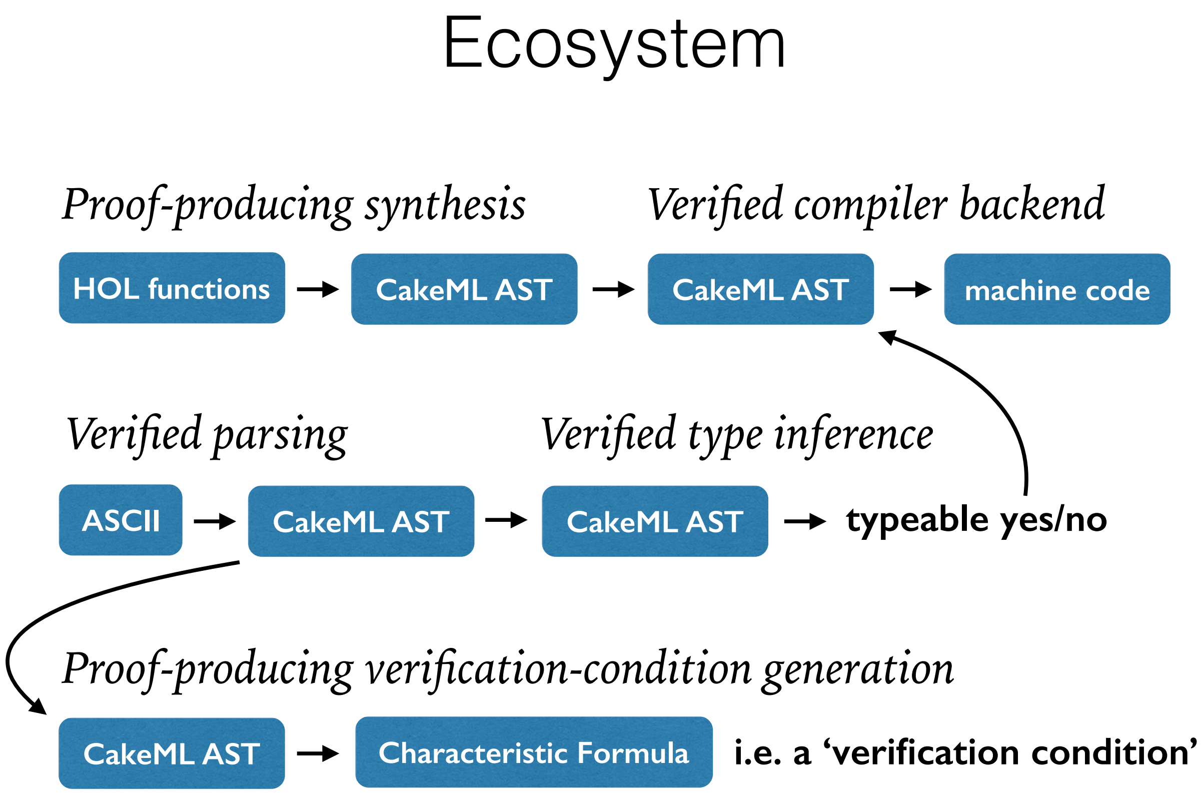


Figure 1.1: The CakeML ecosystem[1]

1.2 CakeML

CakeML is a verified implementation of a subset of Standard ML [74, 121]. To quote the website, it is supplemented by "an ecosystem of proofs and tools built around the language" with the "ecosystem [including] a proven-correct compiler that can bootstrap itself".[2] At time of writing, the project sports over forty developers and contributors.

Figure 1.1 gives an overview over the ecosystem. The verified part comprises a parser, type checker, formal semantics and backend for machine code. The correctness proofs are carried out in the *HOL4* system [122]. HOL4 is a proof assistant in the LCF family, similar to Isabelle.

CakeML is an integral part of this work. My compiler produces CakeML abstract syntax trees and the correctness theorems are justified against its semantics.

1.2.1 Semantics

CakeML's semantics has been specified in *Lem* [93], "a tool for lightweight executable mathematics".[3] It provides a formal specification language that has features comparable to Isabelle/HOL; notably, definition of recursive types, functions and (inductive) predicates. Lem is

[1]Image source: `https://cakeml.org/ecosystem.png`, used with permission
[2]`https://cakeml.org/`
[3]`https://www.cl.cam.ac.uk/~pes20/lem/`

```
type lit =                       datatype lit =
  | IntLit of integer                IntLit int
  | Char of char                   | Char char
  | StrLit of string               | StrLit string
  | Word8 of word8                 | Word8 (8 word)
  | Word64 of word64               | Word64 (64 word)
```

Listing 1.1: Lem specification of CakeML's type of literals and the resulting Isabelle text

capable of compiling these specifications to other specification languages, including Isabelle, HOL4, and Coq.

The remainder of the CakeML ecosystem is implemented in HOL4. For that, the developers maintain the CakeML semantics in Lem and compile it to HOL4. Consequently, assuming trust in Lem, the version of the semantics that I use in Isabelle can be considered identical to the one in HOL4. The Isabelle theories are available in the Archive of Formal Proofs [66].

The CakeML formalization in Lem consists of multiple parts:

Foundation libraries Lem provides a standard library, notably for machine words.

Abstract syntax As is usual in formalizations of programming languages, datatypes for expressions and values are provided.

Semantics There are three flavours of semantics: (relational) big step, small step, and functional big step [102]. For this thesis, only the big-step semantics is relevant.

Note the absence of parsing and printing (i.e., concrete syntax). This has been developed in HOL4 without the help of Lem, so a translation to Isabelle would be a significant undertaking.

1.2.2 Compiler

The simplest way for a user to interact with CakeML is to write some source code as they would for any other language. The CakeML project provides a compiler that can be executed on x86-64 systems, producing a binary that may run on a variety of hardware platforms. That compiler has itself been extracted from a formal specification. Assuming the correctness of HOL4, every compilation guarantees that the resulting binary works correctly according to the semantics of the source program; in other words, compilation preserves semantics.

Another way to produce a binary program is to use the code extraction facility from HOL4 that goes directly to CakeML abstract syntax trees. This thesis provides a similar tool, but implemented based on a different approach and for Isabelle/HOL specifications.

1.3 Motivation

The purpose of using an interactive theorem prover – or, more generally, any kind of prover – is to provide high assurance and trust in the produced artifacts. The range of applications is vast: program and hardware verification, automata and formal languages, security and confidentiality guarantees, analysis and probability theory, topology, and even general relativity, to name just a few [4]. The Isabelle community also runs a peer-reviewed repository of formal proof developments, the *Archive of Formal Proofs*.[4]

As outlined in §1.1, Isabelle is designed in such a way that proofs of statements that are accepted by the system can generally be considered to hold. However, a proof is just a proof; in absence of other mechanisms, they "do nothing". This is why the *extraction* of (or equivalently, transformation into) executable code is not just an afterthought, but rather an important area of research.

While Isabelle offers a rich toolkit for functional programming and mathematical specifications, so far, there is no trustworthy translation to executable code. This means that the guarantees provided by the system end at its boundaries. The generated code bears – apart from an unverified algorithm having produced it – no connection to the original specification. This thesis aims to bridge the gap by developing a verified compiler from Isabelle/HOL to CakeML.

The compiler operates in multiple stages that can be roughly characterized as *preprocessing, deep embedding,* and *compilation phases.* All stages are either *certifying* or *verified*, i.e., an error in the implementation of the compiler would lead to an Isabelle error that is displayed to the user. Under no circumstances will the system emit executable code that once compiled produces an erroneous result.

1.4 Contributions

Many theorem provers – including Isabelle – have the ability to generate executable code in some (typically functional) programming language from definitions, lemmas and proofs [9, 19, 20, 31, 45, 84, 112]. This makes code generation part of the trusted kernel of the system. Myreen and Owens [94] closed this gap for the HOL4 system: they have implemented a tool that translates specifications from HOL4 into *CakeML*, a subset of SML, and proves a theorem stating that a result produced by the CakeML code is correct with respect to the HOL functions. They also have a verified implementation of CakeML [74, 121].

Here, I go one step further and provide a once-and-for-all verified compiler from (deeply embedded) function definitions in Isabelle/HOL [97] into CakeML proving partial correctness of the generated CakeML code with respect to the original functions. This is comparable to the step from "dynamic" to "static" type checking. It also means that preconditions on the input to the compiler are explicitly given in the correctness theorem rather than implicitly by a failing translation.

My compiler is in principle applicable to other languages than Isabelle/HOL or even HOL: Types are erased right away. Hence, the type system of the source language is irrelevant. I merely assume that the source language has a semantics based on *equational logic.*

All topics discussed in this thesis have been formalized in Isabelle; definitions and proofs are machine-checked.

[4]`https://www.isa-afp.org/`

Publications The following publications stem from work in this thesis:

- Lars Hupel. "Lazifying case constants". In: *Archive of Formal Proofs* (Apr. 2017). `http://isa-afp.org/entries/Lazy_Case.shtml`, Formal proof development. ISSN: 2150-914x
- Lars Hupel. "Constructor Functions". In: *Archive of Formal Proofs* (Apr. 2017). `http://isa-afp.org/entries/Constructor_Funs.shtml`, Formal proof development. ISSN: 2150-914x
- Lars Hupel. "Dictionary Construction". In: *Archive of Formal Proofs* (May 2017). `http://isa-afp.org/entries/Dict_Construction.html`, Formal proof development. ISSN: 2150-914x
- Lars Hupel. "Certifying Dictionary Construction in Isabelle/HOL". Preprint. 2018. URL: `https://lars.hupel.info/pub/dict.pdf`
- Lars Hupel and Tobias Nipkow. "A Verified Compiler from Isabelle/HOL to CakeML". in: *Programming Languages and Systems.* Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 999–1026. ISBN: 978-3-319-89884-1
- Lars Hupel and Yu Zhang. "CakeML". in: *Archive of Formal Proofs* (Mar. 2018). `http://isa-afp.org/entries/CakeML.html`, Formal proof development. ISSN: 2150-914x
- Lars Hupel. "An Algebra for Higher-Order Terms". In: *Archive of Formal Proofs* (Jan. 2019). `http://isa-afp.org/entries/Higher_Order_Terms.html`, Formal proof development. ISSN: 2150-914x

1.5 Structure of this thesis

After this introduction, a chapter on technical preliminaries will follow (§2). Subsequently, the thesis is roughly structured according to the phases of the compiler (Figure 1.2). The diagram shows the source object as specified by the user and its transformation into the target object. Each phase is driven by an implementation, which can either be a certifying one in ML, or a verified one in Isabelle/HOL (the difference is explained in more detail in §2.2).

The preprocessing phase statically eliminates features that are not supported by the compiler (§3). Most importantly, the *dictionary construction* (§3.1) eliminates uses of an advanced type system feature.

There are various term types that are required for later stages in the compiler. §4 introduces a term algebra and discusses the differences between those types.

The *deep embedding* phase (§5) lifts Isabelle terms into an internal model. This allows reasoning about Isabelle terms in Isabelle itself.

There are multiple *compiler phases* that process defining equations until a CakeML expression is reached. These phases are described in §6.

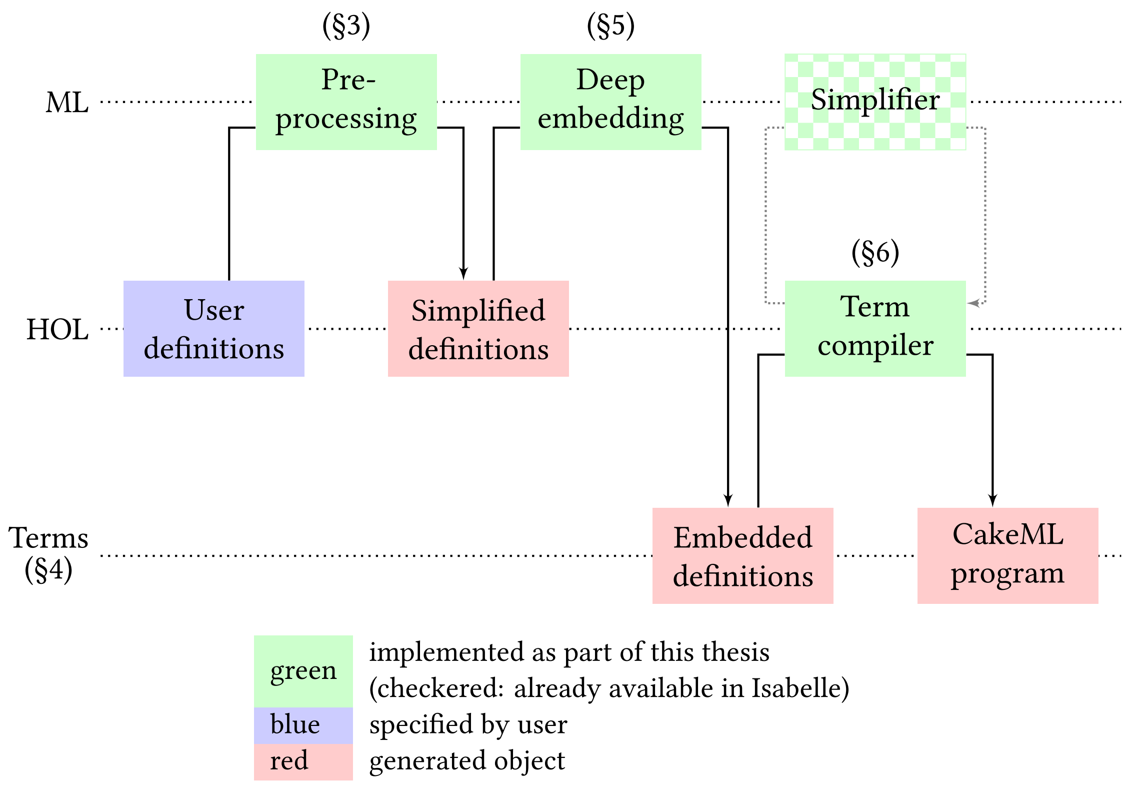


Figure 1.2: Stages of the compiler from Isabelle to CakeML

1.6 How to read this thesis

A paragraph with a pencil indicates that the following section – or portions thereof – has appeared previously in another publication (§1.4). Unless stated otherwise, in publications with a coauthor, I have contributed the majority of the content, including implementation.

Particularly thorny issues, or complicated design decisions stemming from restrictions of Isabelle or CakeML, are described in a box decorated with Bourbaki's dangerous bend symbol. They are not crucial for understanding and can be safely skipped.

Ast

When describing a formalization, I frequently refer to theories. Such references (for example to the Archive of Formal Proofs) appear in the margin. For other non-bibliographic references, for example to external tools, I use footnotes.

Some sections in this thesis consist of large quantities of definitions, lemmas, and proofs. They have been simplified and streamlined for better presentation. Their actual Isabelle representation can be found in the formalization.

2 Background

A proof is a proof.
What kind of a proof?
It's a proof.
A proof is a proof,
and when you have a good proof,
it's because it's proven.

(Jean Chrétien)

This chapter describes in detail necessary technical background on Isabelle that is relevant for understanding the remainder of this thesis.

2.1 Isabelle design

One of the defining features of Isabelle as compared to other proof assistants in the same family is its modular logic design [104]: the kernel provides the minimal logic *Pure,* on top of which other logics can be implemented by users. Pure, in its essence, is a framework for natural deduction proofs. Proved statements are internally represented as values of the abstract type `thm`. Standard ML's typing discipline ensures that such values can only be constructed through a finite set of primitives that represent rules or axiom schemas of constructive logic.

The most basic rule is *modus ponens:* two statements $P \implies Q$ and P can be combined to deduce Q. As is common in literature, this thesis will frequently express such inferences using the following notation:

$$\frac{P \implies Q \qquad P}{Q}$$

Based on such primitives, a variety of automated *tactics* are provided. Tactics in their most basic form are ML programs that transform theorems into theorems. This can be used for user interaction when proving theorems:

1. The user indicates that they would like to prove the statement P.
2. Because the system does not know yet that P holds, it generates the *goal state* $P \implies P$. A goal state is an implication whose premises are called *subgoals* and the conclusion is the statement that the user wants proved. Observe that $P \implies P$ holds for all P: it is an axiom schema.

```
datatype α seq = Empty | Seq α (α seq)

fun conc :: α seq ⇒ α seq ⇒ α seq where
conc Empty ys = ys
conc (Seq x xs) ys = Seq x (conc xs ys)
```

Listing 2.1: A simple functional program in Isabelle/HOL

3. The user can apply tactics that manipulate some (or all) subgoals. For example, a goal state $Q_1 \wedge Q_2 \implies P$ can be transformed into $Q_1 \implies Q_2 \implies P$; an instance of conjunction introduction, where a conjunction is split into two separate subgoals.
4. Eventually, if all subgoals disappear (have been *discharged*) and the goal state is P, the system will accept this as a proved statement. Because the goal state is a value of type `thm` at all times, P is directly usable as a theorem.

Isabelle comes equipped with a set of standard tactics, for example the simplifier, which is able to rewrite terms according to (possibly conditional) rewrite rules $t \equiv u$, and a classical reasoner based on a tableau calculus [103].

System interaction can happen on two "layers": the raw ML programming environment, or the high-level language Isar [127]. For most purposes, users do not manipulate low-level ML values, but can instead use the abstract Isabelle/Isar syntax.

The most commonly used logic of Isabelle is *Higher-Order Logic* (*HOL* for short) based on work by Gordon [44]. Besides standard features of classical higher-order logic (definitions, quantifiers, connectives) it provides tools for functional programming, e.g. recursive datatypes and functions with pattern matching.

2.2 Terminology

The term *theory* has two meanings: on a physical level, files containing Isabelle/Isar sources; on a theoretical level, a collection of definitions, constants, theorems, and other logical content. A theory file is a sequence of Isar *commands* that alter the logical theory. I will frequently refer to the actual Isabelle source files that accompany this thesis as *the formalization*.

Higher-level tools in Isabelle are usually referred to as *packages*. For example, the two facilities that enable functional programming are the **function** and the **datatype** packages [18, 69, 70]. A simple example of their interplay is given in Listing 2.1. Because in the architecture of Isabelle, all of these tools need to justify their constructions against the Isabelle kernel. An implementation error would not produce an unsound theory; instead, the kernel would print an error that some construction failed.

Besides theorems, the second foundational ingredient of an Isabelle theory are *constants*. Contrary to what the name suggests, the type of a constant can also be a function type. The logical distinction between constants and *variables* in Isabelle is that constants may have definition (or multiple, see §2.3), whereas variables may not. Datatype constructors and functions, as in Listing 2.1, are also constants, as far as the kernel is concerned; even though they have no user-accessible definition. Still, they are internally constructed and defined by their respective packages.

Internally, Isabelle keeps track of a special kind of variable: *schematic variables*. Schematic variables can be instantiated with arbitrary terms. This is an implementation trick to avoid quantifiers in many situations.

A theory can be augmented with arbitrary auxiliary data. A particular extension is referred to as a *proof context*, or *context* for short. Contexts, for example, keep track of fixed variables and their types, and local assumptions that are not valid on the global theory level. Most frequently, Isabelle users encounter contexts when they write a structured proof. Furthermore, contexts enable modular reasoning (§4.1.3).

Certifying and verified routines Consider a routine that takes a value x, transforms it according to a function f, and ensures that the result $y = f\,x$ satisfies a predicate P. In proof assistants, there are two ways to implement this:

1. A block of ML code analyses the value x, defines a new object y and carries out a proof that $P\,y$. This proof may fail if the ML code has an error, or if some precondition is not satisfied.
2. The function f is implemented inside the logic, together with a proof that $P\,(f\,x)$ holds for all x. This is less flexible, but the routine has been shown to be correct once and for all.

The first strategy is known as *certifying*, because after each run, it produces a single certificate that the generated object is valid. Contrary to that, the second one is called as *verified*, because the implementer has given a full correctness proof. Both approaches have their own (dis)advantages, which means they are often mixed, such as in this formalization.

Notational conventions As can be observed in Listing 2.1, Isar notation is similar to that of programming languages like Haskell or ML. For easier readability, code samples are slightly modified from their actual representation that can be processed by Isabelle.

By convention, types and constants are set in `typewriter` font. Term variables are, as is usual in mathematics, set in *italics*.

An actual Isabelle source file (a *theory*) is composed of a sequence of commands. Commands are set in **sans-serif**. Special syntactic constructs in terms, like branching, are written as **if** P **then** x **else** y.

2.3 Type system

Isabelle implements an ML-style *simple type system* with *schematic polymorphism*. Types can be formed by type constructors (e.g. `list`) and type variables ($\alpha, \beta, \ldots$). Composite types are written in postfix notation, i.e., a list of integers is written as `int list`. All types in HOL are non-empty, that is, they have at least one inhabitant.

```
consts size :: α ⇒ nat

overloading size_list ≡ size :: α list ⇒ nat
begin

  definition size_list where
  size_list = List.length

end
```

Listing 2.2: Declaration of a constant and overloaded definitions

Polymorphism Isabelle supports *type schemes* (or *polytypes* in recent literature) [55, 90]: types can be quantified at the outermost level. For example, $\forall\tau.\ \tau$ `list` is a valid type scheme for the empty list []. In contrast, ($\forall\tau.\ \tau$ `list`) `list` is not admissible as a type scheme for the list containing the empty list [[]], because the quantifier is nested inside a type constructor; i.e., no second-rank polymorphism is allowed [82].

To implement this, Pure does not provide an explicit type quantifier; instead, it uses schematic polymorphism. In addition to type variables, there are also *schematic type variables* that can be instantiated. Those are prefixed with a question mark: $?\alpha$. This distinction becomes important later for technical reasons (§3.1.3). Users can largely ignore schematic type variables, as the system automatically introduces them.

Overloading Isabelle supports *overloading* of constants based on their type. It is possible to declare a constant with a polymorphic type and then give definitions for a specific instantiation [78]. As an example, consider Listing 2.2 that declares a function that should represent the "size" of a value. As defined there, it works for lists and can be extended for other types.

This mechanism is very flexible, but is hardly employed directly by users. Instead, the system implements *type classes* based on overloading. Type classes are widely used in the Isabelle community and require special treatment for the purpose of this thesis (§3.1).

2.4 Executability

Specifications formalized in Isabelle/HOL – a classical logic – are not necessarily *executable*. This affects proofs who may use classical constructs like law of excluded middle and Hilbert choice. However, this thesis is concerned with generating a functional program that can be compiled to machine code from Isabelle specifications.

This mismatch can be reconciled by identifying an executable subset of types and definitions that can be translated into a functional program. Importantly, this is not a new concept and can be traced back to previous work in this area, most recently by Haftmann [48, 49]. In general, that subset can be characterized as "functional program in, functional program out".

However, there are a few notable exceptions: HOL is more expressive than programming languages; for example, it is possible to quantify over an infinite set, which is naturally not executable. By default, specifications that are created by **datatype** and **function** and that only use other executable functions are themselves executable: these packages have been designed with executability in mind.

Apart from **function**, which enables definition of recursive functions, there is also the **definition** command. It only supports non-recursive, non-pattern-matching definitions. For the purposes of this thesis, their internal implementation differences are not relevant: both allow introducing constants into a theory based on *defining equations*.

Isabelle's code generator also supports post-hoc introduction of executability of a specification. Sometimes it can be more convenient to specify a constant using the non-executable fragment, for example using an unconstrained quantifier. Later on, a *code equation* can be added to the theory, which will then be used instead of the original defining equation (§3.1.2.1). Code equations can override any existing definition. In a nutshell, a specification is executable if its transitive set of code equations only use recursion and pattern matching. In this thesis, the term *defining equation* is preferred, because the point at which they are introduced into the theory does not matter to the compiler.

Even though the term "executability" suggests some form of efficient program on hardware, it is not necessary that executable code has to be executed outside of Isabelle. Most notably, the simplifier can be set up to only use the defining equations, which could be used to emulate a graph-reduction style [57] evaluation inside the Isabelle kernel.

Lazy_Eval

When designing a specification, it is advantageous to stay within the executable fragment of HOL. Many Isabelle tools aimed at developer productivity work better (or only work) in such cases. The prime example is **quickcheck** [24], which is able to uncover flaws in theorem statements by running an automated counterexample finder. This saves a lot of time by preventing one from trying to prove false statements. Consequently, in many parts of this thesis, I made design decisions supporting these tools (for example in §2.7), even though at times it complicates the specifications.

2.5 Inductive predicates

Both in formal and informal mathematics, the notion of *inductively defined*, or *inductive predicates* is pervasive. Winskel [131] gives a comprehensive introduction into their mathematical background. Parts of this will be revisited in later chapters, notably well-founded induction.

Listing 2.3 presents a simple inductive characterization of even numbers. The two rules declare that 0 is even, and if n is even, so is $n + 2$. The result of this abstract specification is the smallest predicate `even` :: `nat` ⟹ `bool` satisfying the given rules, i.e., the *least fixed point*. Literature commonly uses inductive sets instead of predicates; however, the types α `set` and α ⟹ `bool` are isomorphic to each other.

In Isabelle post-2012 versions, sets and predicates are distinct types and many libraries are duplicated for both. In this formalization, predicates are preferred, unless finitary constraints need to be enforced.

$$\frac{}{\mathsf{even}\ 0} \qquad \frac{\mathsf{even}\ n}{\mathsf{even}\ (n + 2)}$$

(2.3.a) Mathematical characterization

```
inductive even :: nat ⟹ bool where
even 0 |
even n  ⟹  even (n + 2)
```

(2.3.b) Isabelle notation

$$\frac{\mathsf{even}\ x \qquad P\,0 \qquad (\forall n.\ \mathsf{even}\ n \implies P\,n \implies P\,(\mathsf{Suc}\ n))}{P\,x}$$

(2.3.c) Induction principle

Listing 2.3: A simple inductively defined predicate

In Isabelle, the **inductive** package can be used to introduce inductive predicates. The **inductive** command automates the internal construction of a least fixed-point based on the given rules, which are referred to as *introduction rules*. Conversely, there are also *elimination rules*. They can be used to prove properties like $\mathsf{even}\ (n + 2) \implies \mathsf{even}\ n$.

The command also generates an induction schema. In the even example, it can be used to prove properties of the form $\mathsf{even}\ n \implies P\,n$ for arbitrary P. This is usually referred to as *rule induction*.

Inductively defined predicates are the most important ingredient for defining semantics. Nipkow and Klein [97, §4.5] give further explanations of how those work in Isabelle. In fact, the example in Listing 2.3 is taken from their book.

Semantics for non-trivial languages, including CakeML, are not purely specified as inductive predicates. Frequently, they are based on *semantic functions* which are used to carry out substitution, matching, and other fundamental operations.

Inductive predicates are convenient to use because contrary to function definitions, users need not care about termination. For example, one can define a big-step semantics for a programming language that admits non-terminating programs. It is not possible to define this in an equational way without additional tricks [102, 115].

There are still constraints on the kinds of rules that can constitute a least-fixed point, but these can often be automatically discharged by the **inductive** package. When higher-order predicates are involved (for example $\mathsf{pred}_{\mathtt{list}} :: (\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha\ \mathtt{list} \Rightarrow \mathsf{bool}$), monotonicity has to be proved by the user first.

The obvious downside resulting from that convenience is that inductive predicates are by default not executable (§2.4). Berghofer et al. [8] have introduced a *predicate compiler* which can transform a large fragment of inductive specifications into executable code. While it works mostly automatically, it is tricky to use and has several edge cases; as such, I consider it to be a feature of last resort and prefer equational definitions where possible.

```
datatype term =
  Const string |
  Free string |
  Abs term |
  Bound nat |
  App term term
```

(2.4.a) Abstract syntax of de Bruijn terms

$$\textsc{Step}\ \frac{(\mathit{lhs}, \mathit{rhs}) \in R \qquad \mathtt{match}\ \mathit{lhs}\ t = \mathtt{Some}\ \sigma}{R \vdash t \longrightarrow \mathtt{subst}\ \sigma\ \mathit{rhs}}$$

$$\textsc{Beta}\ \frac{}{R \vdash (\Lambda t)\ \$\ t' \longrightarrow t[t']} \qquad \textsc{Fun}\ \frac{R \vdash t \longrightarrow t'}{R \vdash t\ \$\ u \longrightarrow t'\ \$\ u}$$

$$\textsc{Arg}\ \frac{R \vdash u \longrightarrow u'}{R \vdash t\ \$\ u \longrightarrow t\ \$\ u'}$$

(2.4.b) Small-step semantics

Listing 2.4: Overview of the deeply embedded de Bruijn `term` type

In the even example, the predicate compiler produces code equations that terminate for even numbers – correctly answering `True` – but fail to terminate for odd numbers. The reason is that the compiler does not know that the search for an n' such that $n = n' + 2$ is bounded.

The categorical dual, *coinductive* predicates, are only needed at single point in the formalization (Listing 6.11). Because of their only brief appearance, an explanation of coinduction is out of the scope for this thesis. I refer the reader to the literature for more details [42, 110].

2.6 Term rewriting

Based on the internal definition of terms in Isabelle/Pure, one can model terms as a datatype in HOL: the *deeply embedded term language* is depicted in Listing 2.4.a. Similarly to the Pure type, it uses *de Bruijn indices* [23], but omits types and schematic variables.

The embedded HOL `term` type uses the same conventions as its ML counterpart. I write `App` $t\ u$ as $t\ \$\ u$ and `Abs` t as $\Lambda\ t$. The notation $t[t']$ represents β-reduction, that is, substitution of the innermost bound variable (i.e. with index zero) in t with t' (the implementation of β-reduction will be revisited in §4.3.1). Throughout this thesis, I will use an upper-case Λ to refer to the concrete syntax of abstraction in the embedded term language, whereas lower-case λ is used for Pure abstractions. More details on this and other term types can be found in §4.

Observe that types are not preserved in this embedded language. In Pure, the `Const`, `Free`, and `Abs` constructors carry type information, each for a different purpose: constants can be polymorphic (the type specifies the instantiation of the type scheme), free variables are identified by their name and their type (there may be multiple variables with the same name but different types), and finally, abstractions specify the type of the bound variable.

While type checking admits multiple variables with the same name and differing types, it rarely happens in practice because type inference rejects such terms. If a user was to feed, for example, the term x # x into Isabelle (where # is list cons), type inference would report a unification failure, because α and α `list` cannot be unified. Consequently, it is no real restriction that the model presented here does not handle such odd terms.

For abstractions, type erasure is unproblematic, because they are assumed to be parametric; i.e., for a polymorphic parameter, they may not behave differently depending on the concrete instantiation. However, it becomes impossible to distinguish between different instantiations of constants. This is problematic because of overloading (§2.3). I employ the so-called dictionary construction (§3.1) to avoid this problem.

Listing 2.4.b specifies the small-step semantics for terms. It is reminiscent of *higher-order term rewriting*, and modelled closely after equality in HOL. The basic idea is that if the proposition $t = u$ can be proved equationally in HOL (without symmetry), then $R \vdash \langle t \rangle \longrightarrow^* \langle u \rangle$ holds, where R contains all defining equations. The angle brackets denote the deep-embedding operator that will be explained in more detail in §5.

In the semantic for terms with de Bruijn indices, substitution under binders, i.e., rewriting below an abstraction, can be easily implemented: there are no bound variable names that could capture free variables of the term that is substituted. Still, the semantics in Listing 2.4.b does not recurse below binders; only below application. The reason for that is twofold:

1. during the process of the compiler, new term types are introduced that carry explicit bound variable names (§4.3), and
2. in the CakeML semantics, substitution does not happen below binders (§6.8).

This model of term rewriting in HOL coincides by design with the notion of executability. All defining equations can be lifted into the set R, which will then be transformed in later stages of the compiler.

2.7 CakeML/Isabelle integration

Because the Lem specification of CakeML assumes some special features of HOL4, some adaptations were necessary:

- Fabian Immler and Johannes Åman Pohjola have contributed to the adaptation of machine words and floating-point arithmetic.
- Pohjola has contributed the s-expression printer that converts CakeML programs into string form, which can subsequently be consumed by the CakeML compiler.
- The functional big step semantics is implemented using a special function to enforce monotonically decreasing clocks according to Kumar and Myreen [73, §3.2]. This makes the termination proof simpler, but complicates reasoning about the semantics. Proofs from HOL4 that rephrase the semantics without that function had to be ported to Isabelle. Additionally, the equivalence proofs between big step and functional big step semantics were ported.

Evaluate_Clock

Big_Step_Fun_Equiv

- In general, Isabelle tools and automation work better with nested recursive definitions instead of mutually recursive definitions. Additionally, my formalization ignores certain features of CakeML; notably mutable cells, modules, and literals. Consequently, I have derived a smaller, executable version of the original CakeML semantics, together with an equivalence proof, called *CupCakeML* (§6.8). Portions of this have been implemented by Yu Zhang.
- Where necessary, I have set up code equations and **quickcheck** for more rapid development of theories. In general, the functional big step semantics is better suited for executability than the relational variant, because it is defined as a **function**.

🗎 `Evaluate_Single`

🗎 `CupCake_Semantics`

🗎 `CakeML_Quickcheck`

In order to obtain a full toolchain from Isabelle definitions to machine code, I have implemented a small pretty-printer of the CupCakeML fragment to concrete CakeML syntax, which is part of the trusted code base. The resulting source text can then be fed into the CakeML compiler.

🗎 `CakeML_Compiler`

3 Preprocessing definitions

Axiomatic type classes are definitional.
Type definitions are axiomatic.

(Old Isabelle proverb)

Contents

Before definitions can be processed by the compiler, a *preprocessing* phase needs to modify them to remove unsupported features. This chapter explains these steps, starting with the elimination of type classes (§3.1), followed by lazy evaluation of case combinators (§3.2) and partially applied constructors (§3.3). Finally, it deals with restrictions on pattern matching (§3.4) and other limitations (§3.5).

3.1 Dictionary construction

Portions of this section appear in the AFP entry "Dictionary Construction" (Hupel [63]), and the preprint "Certifying Dictionary Construction in Isabelle/HOL" (Hupel [61]). Users are recommended to refer to that AFP entry for latest documentation.

Isabelle/Pure features *type classes* [50, 128]. These are built into the kernel and are used extensively in theory developments. The code generator, when targeting Standard ML, performs the well-known dictionary construction or *dictionary translation* [49]. This works by replacing type classes with records, instances with values, and occurrences with explicit parameters.

Similarly to Standard ML, CakeML does not support type classes. To avoid complicating the correctness proofs, I decided to also not support them in the embedded term language (§2.6). Instead, a *dictionary construction* eliminates classes and instances before embedding into the term language. This section deals with the chosen encoding of type classes and instances, the certifying translation, and treatment of partial functions.

3.1.1 Preliminaries

In Isabelle parlance, the term *class* refers to a type class; a concept known from Haskell [50, 126]. A class can fix a type α and some constants whose type contains α. Those constants are officially referred to as *class parameters*. Classes and parameters live in different name spaces. In the example in Listing 3.1, `plus` is a parameter of the `plus` class. In this section, I will use the term *class constant* instead of class parameter, to avoid the ambiguity with function parameters.

A set of classes is called a *sort*. Type variables may carry *sort constraints*, which are preceded by double colons: $\alpha :: \{\texttt{plus}, \texttt{times}\}$. Formally, a sort is an intersection of classes: the set of types that satisfy the sort $\{\texttt{plus}, \texttt{times}\}$ is the set of the types have both a `plus` and a `times` *instance*. Constants are said to have sort constrains if their types contain type variables with sort constraints. Sort constraints can be omitted by the user and will be inferred. If they are provided by the user, it is sufficient to specify them once per type variable.

As an example, consider the function $\texttt{f}\ x = x + x$. In HOL, the + operator is defined in the type class `plus`. This means that the type of `f` is $(\alpha :: \texttt{plus}) \Rightarrow \alpha$.

Classes can extend other classes, with the inheritance relationship forming a directed acyclic graph (and with it, a partial order). Sorts can be normalized according to this partial order. Assuming that the class `group` extends both `zero` and `plus`, the sort $\{\texttt{group}, \texttt{zero}\}$ can equivalently be written as $\{\texttt{group}\}$. I always assume that sorts are *normal*.

```
class plus =
  fixes plus :: α ⇒ α ⇒ α (infixl + 65)

definition f :: α::plus ⇒ α where
f x = x + x
```

(3.1.a) Source program

```
type 'a plus = {plus : 'a -> 'a -> 'a};
val plus = #plus : 'a plus -> 'a -> 'a -> 'a;

fun f dict x = plus dict x x;
```

(3.1.b) Target program (Standard ML)

Listing 3.1: Dictionary construction in Isabelle (current state)

Largely, classes work similarly in Haskell – Haskell98, to be specific – and Isabelle (see also §3.1.7). Isabelle's type system does not admit type constructor classes, nor other extensions like multi-parameter classes or undecidable instances. However, Isabelle offers major enhancements over Haskell:

- Isabelle being a proof assistant allows *class axioms* that must be proved for every type that wishes to instantiate the class. The precise axioms are irrelevant for the dictionary construction (§3.1.2.3); they are abstracted as a predicate.
- It is possible to add edges in the inheritance relationship after the definition of the classes, as long as the user is able to produce a proof that the subclass axioms imply the superclass axioms.

While parts of the dictionary construction are implemented incrementally, modifications in the class graph after their records have already been generated are not supported.

3.1.2 Elimination of classes

The basic idea is to replace classes by *dictionaries* containing all class constants and to replace instances by values. Constants with sort constraints are rewritten in a way that they require additional dictionary parameters.

This transformation is an integral part of Isabelle's code generator. It is described in detail by Haftmann and Nipkow [49, §4], together with an informal correctness proof.

A complete example for f $x = x + x$ is reproduced in Listing 3.1. Note that this translation is only required for target languages that do not support type classes (OCaml, Standard ML). For other languages (Haskell, Scala), type classes are preserved with only minor syntactic changes.

```
datatype α dict_plus = mk_plus (const_plus: α ⇒ α ⇒ α)

definition cert_plus :: α::plus dict_plus ⇒ bool where
cert_plus dict = (const_plus dict = plus)

fun f' :: α dict_plus ⇒ α ⇒ α where
f' dict x = const_plus dict x x

lemma f'_eq: cert_plus dict ⟹ f' dict = f
(* proof omitted *)
```

Listing 3.2: Source program after dictionary construction in HOL (certifying translation)

3.1.2.1 Current state

In the code generator, the dictionary construction happens outside the logic. It starts with a set of defining equations that represent the program to be exported. These equations are proper theorems and are generated automatically by various commands for datatype and function definitions. To improve efficiency, the user may provide alternative (verified) equations, for example, to replace a naive recursive implementation of a function by a more stack-efficient tail-recursive definition.

Then, these equations are internalized into an intermediate language. The dictionary construction then proceeds in this internal language, following the approach outlined by Hall et al. [52].

3.1.2.2 Certifying translation

In this work, dictionary translation is performed before internalizing the defining equations into the deeply embedded term language. It is a procedure implemented in ML which takes existing HOL definitions and produces new, derived HOL definitions, coupled with theorems certifying their equivalence.

To continue with the above example: My mechanism introduces a derived constant `f'` with an additional dictionary parameter *dict* :: *α* `dict_plus`. Then, it proves a theorem stating that for any *valid* dictionary *dict*, `f'` is equivalent to `f`:

$$\texttt{cert\_plus}\ \mathit{dict} \implies \texttt{f'}\ \mathit{dict} = \texttt{f}$$

Validity of a dictionary is captured by the `cert_plus` predicate. Intuitively, `cert_`*c* *dict* means that *dict* represents a known and lawful instance of class *c*. The precise notion of "validity" is mainly dictated by technical considerations and discussed in the following section.

Additionally, for each type class instance $\kappa :: (s_1, \ldots, s_k)\ c$, where κ is an k-ary type constructor and $s_i = \{c_{i,1}, \ldots, c_{i,n_i}\}$ are sorts, a new constant `inst_`c`_`κ is defined. Given the dictionaries for the s_i, it computes the dictionary for $\kappa :: c$. Its correctness theorem is of the form

$$\left(\bigwedge_{i=1}^{k}\bigwedge_{j=1}^{n_i} \texttt{cert\_}c_{i,j}\; dict_{1,1}\right) \Longrightarrow \texttt{cert\_}c\,(\texttt{inst\_}c\texttt{\_}\kappa\; dict_{1,1}\;\ldots\; dict_{1,n_1}\;\ldots\; dict_{k,1}\;\ldots\; dict_{k,n_k})$$

For both instances and constants, each constituent class of each type variable's sort constraints gets assigned a dictionary argument and a premise certifying its validity.

The resulting program (as it would have been written by a user) is reproduced in Listing 3.2. My procedure defines the types and constants through the ML interfaces of various Isabelle packages, that is, users never see its results directly. Instead, users would write **declassify** `f`, which is a command that has the same effect as the hand-written definitions in Listing 3.2.

3.1.2.3 Possible encodings

The choice of the representation of dictionaries is straightforward: I can model it as a datatype, along with functions returning values of that type. The alternative here would have been to use Isabelle's extensible records [95]. The obvious advantage of records is that I could easily model subclass relationships through record inheritance. However, records do not support multiple inheritance. Consequently, records offer no advantage over datatypes. Instead, I opted for the more modern **datatype** command [18]. As of Isabelle2018, I have also introduced a **datatype_record** command that provides a subset of the syntax of records, but internally constructs a datatype.

`Datatype_Records`

A more controversial design question is how to represent dictionary certificates. For example, given a value of type `nat dict_plus`, how can one know that this is a faithful representation of the `plus` instance for `nat`?

1. Florian Haftmann, in private communication, proposed a shallow encoding. It works by exploiting the internal treatment of constants with sort constraints in the Isabelle kernel. Constants themselves do not carry sort constraints, only their defining equations. The fact that a constant only appears with these constraints on the surface of the system is a feature of type inference.

   Instead, I can instruct the system to ignore these constraints. Isabelle's logic supports definitions of *subtypes:* a type copy of an existing type that imposes additional constraints on values. For example, non-empty lists can be defined as a copy of lists with the constraint $xs \neq []$. Because HOL is a total logic, i.e., all types are non-empty, the system demands a witness satisfying the constraint.

   Applied to this situation, the key idea is to introduce a new type with a parameter α and the constraint that α implements a type class. However, this is ultimately futile: The nonemptiness proof requires a witness of a valid dictionary for an arbitrary, but fixed type α, which is of course not possible, because type classes in general cannot be instantiated for all types.

`Impossibility`

2. The certificates contain the class axioms directly. For example, the `semigroup_add` class requires $(a + b) + c = a + (b + c)$. Such certificates are already defined for each class by Isabelle. Transferred to this setting, they would like this:

```
definition cert_plus :: α dict_plus ⇒ bool where
cert_plus dict = (∀x y z. const_plus dict (const_plus dict x y) z =
                              const_plus dict x (const_plus dict y z))
```

Proving that instances satisfy this certificate is trivial. However, the equality proof of a constant before and after the construction is impossible: they are simply not equal in general. Nothing would prevent someone from defining an alternative dictionary using multiplication instead of addition and the certificate would still hold; but obviously functions using `plus` on numbers would expect addition. Intuitively, this makes sense: the above notion of certificate establishes no connection between original instantiation and newly generated dictionaries.

Instead of proving equality, one would need to lift all existing theorems over the old constants to the new constants. This requires proof terms and replaying all proofs accordingly, which would be prohibitively expensive.

3. In order for equality between new and old constants to hold, the certificate needs to capture that the dictionary corresponds exactly to the class constants. This is achieved by the representation in Listing 3.2. It literally states that the fields of the dictionary are equal to the class constants. The condition of the resulting equation can only be instantiated with dictionaries corresponding to existing class instances. This constitutes a *closed world* assumption, i.e., callers of generated code may not invent own instantiations.

My choice of representation is the third of these possibilities: I expect dictionaries to be identical to the class constants. For the user, that means that the conditions of the equivalence theorems (`f'` *dict* = `f`) can only be instantiated with existing class instantiations. Unconditional equivalences can be achieved by monomorphizing constants. Applied to the example in Listing 3.1, that would mean defining a constant $\mathtt{f}_{\mathtt{nat}}$ :: `nat` ⇒ `nat`. Its correctness theorem is unconditional, because no sort constraints occur in the type of `f`.

3.1.3 Implementation

The mechanism that transforms defining equations is similar to the one described by Haftmann and Nipkow [49, §4], which is presently used by the code generator to target OCaml and Standard ML.

Translating constants is the top-level operation in the dictionary construction. The user invokes it with a set of constants. Internally, the procedure uses existing mechanisms in Isabelle to obtain the *code graph* of that set. That graph contains all defining equations of the set and of all its transitive dependencies (i.e., other constants). Each of these dependencies has to be re-defined as a new constant in some way, depending on whether or not it is a class constant.

```
class plus =
  fixes plus :: α ⇒ α ⇒ α

instantiation nat :: plus
begin

fun plus_nat where
0 + n = (n::nat)
Suc m + n = Suc (m + n)

instance ..

end

definition f :: α::plus ⇒ α where
f x = x + x

(* f specialized to nat *)
definition g :: nat ⇒ nat where
g x = f x
```

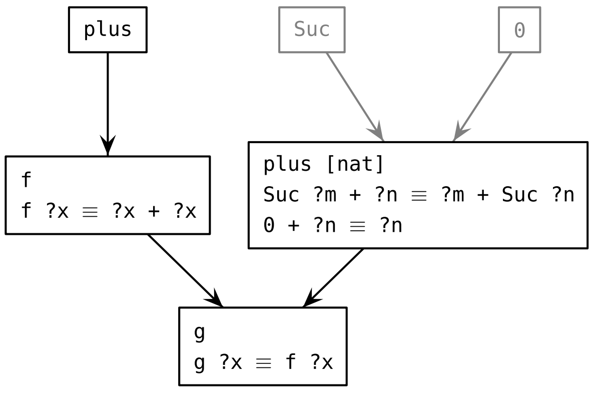


Figure 3.1: A slightly extended (from Listing 3.1) source program and its code graph

---

Strictly speaking, data constructors are also constants that may have class constraints. The dictionary construction does not support those in general. Additionally, the underlying type definition based on bounded natural functors largely ignores sort constraints.[a] Consequently, they do not participate in the dictionary construction and are not relevant for this section.

[a] `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2018-May/msg00065.html`

Along the way, auxiliary objects must be defined, for example the dictionary types for classes. Unlike with the existing code generator, all of these steps need to be carried out inside the logic and are hence bound by its constraints. Most notably, all definitions must be sequentialized to avoid forward references. This means the implementation comprises mutually recursive, state-updating functions.

The code graph of a small program is given in Figure 3.1. As can be seen, the constant g depends on the constant f and the instance `nat :: plus`. The data constructors Suc and `0` are greyed out. The graph has to be traversed in topological order.

Readers familiar with Isabelle's internals will notice that the code graph has been slightly redacted: The zero constructor for `nat` is actually the overloaded constant `zero` from the type class `zero`. This introduces technical complications, but does not in principle affect the dictionary construction.

```
datatype α dict_c = mk_c
  (super_c₁: α dict_c₁) (super_c₂: α dict_c₂) … (super_cₙ: α dict_cₙ)
  (const_f₁: ⟦τ₁⟧) (const_f₂: ⟦τ₂⟧) … (const_fₘ: ⟦τₘ⟧)

definition cert_c :: α::c dict_c ⟹ bool where
cert_c dict =
  (cert_c₁ (super_c₁ dict) ∧ cert_c₂ (super_c₂ dict) ∧ … cert_cₙ (super_cₙ dict) ∧
   const_f₁ dict = f₁ ∧ const_f₂ dict = f₂ ∧ … ∧ const_fₘ dict = fₘ)
```

Listing 3.3: Dictionary datatype and certificate predicate

Throughout this section, the overloaded notations $[\![\cdot]\!]$ and $(\!|\cdot|\!)$ are used to describe the translation of various kinds of objects. I will first explain how types and classes themselves are processed. Then, assuming a translation for terms exists, I will give a translation for type schemes and constants. Lastly, the knot is tied by explaining how terms are processed. In the actual implementation, all of these steps are intertwined.

3.1.3.1 Types

Recall that Isabelle distinguishes between type variables and schematic type variables (§2.3). Simple types, i.e., types that contain no schematic type variables, can be translated very easily: $[\![\tau]\!]$ forgets all sort constraints. This is possible because those cannot have intrinsic sort constraints; those are imposed from the context and will be introduced accordingly when dealing with type schemes, which will be explained later.

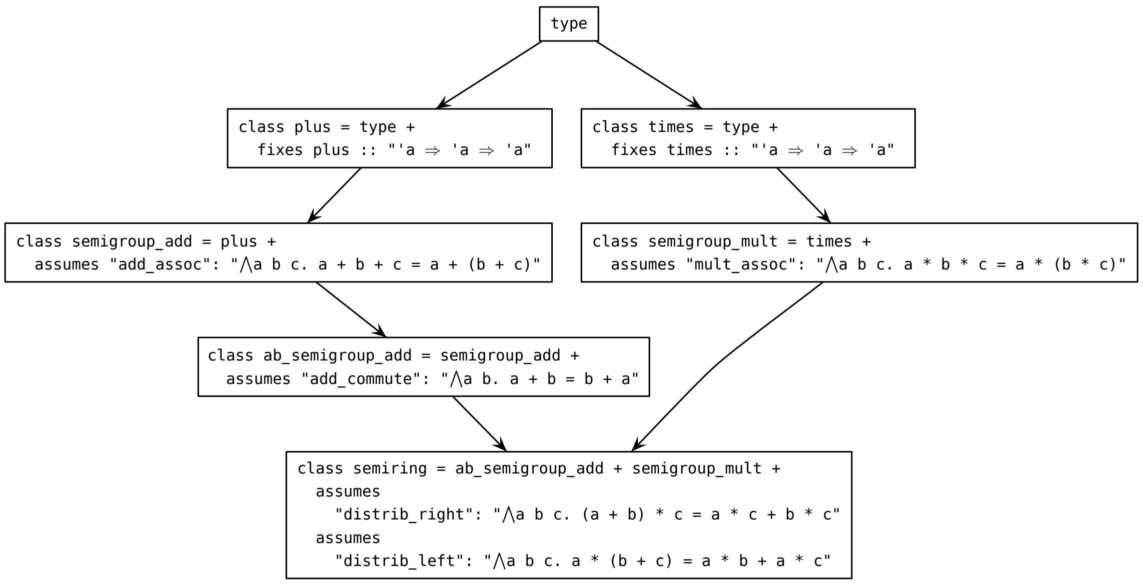
3.1.3.2 Classes

A class c over a type variable α may have superclasses $c_1, c_2, \ldots, c_n$ and constants $f_1 :: \tau_1, \ldots, f_m :: \tau_m$. Assuming the set $\{c_1, c_2, \ldots, c_n\}$ is normal, this generates the definitions in Listing 3.3. Note that the only type variable that may occur in the τ_i is α itself, which is an Isabelle restriction. Consequently, it is not necessary to perform a recursive dictionary translation on the class constants, and I can get away with using the translation for simple types.

This newly introduced constructor and its fields have the following types:

$$\begin{aligned}
\texttt{mk\_}c &:: \alpha\,\texttt{dict\_}c_1 \Longrightarrow \ldots \Longrightarrow \alpha\,\texttt{dict\_}c_n \Longrightarrow \alpha\,\texttt{dict\_}c \\
\texttt{const\_}f_i &:: \alpha\,\texttt{dict\_}c \Longrightarrow [\![\tau_i]\!] \\
\texttt{super\_}c_i &:: \alpha\,\texttt{dict\_}c \Longrightarrow \alpha\,\texttt{dict\_}c_i \\
\texttt{cert\_}c &:: \alpha :: c \Longrightarrow \texttt{bool}
\end{aligned}$$

Apart from the certificate definition (which is only required for the correctness proofs), no sort constraints are left.



```
type
```

```
class plus = type +
  fixes plus :: "'a ⇒ 'a ⇒ 'a"
```

```
class times = type +
  fixes times :: "'a ⇒ 'a ⇒ 'a"
```

```
class semigroup_add = plus +
  assumes "add_assoc": "⋀a b c. a + b + c = a + (b + c)"
```

```
class semigroup_mult = times +
  assumes "mult_assoc": "⋀a b c. a * b * c = a * (b * c)"
```

```
class ab_semigroup_add = semigroup_add +
  assumes "add_commute": "⋀a b. a + b = b + a"
```

```
class semiring = ab_semigroup_add + semigroup_mult +
  assumes
    "distrib_right": "⋀a b c. (a + b) * c = a * c + b * c"
  assumes
    "distrib_left": "⋀a b c. a * (b + c) = a * b + a * c"
```

Figure 3.2: Class hierarchy for the `comm_semigroup_add` class

For any class constant f of a class c, let $(\!| f |\!)$ denote the corresponding constant field. If d is a direct superclass of c, I use $(\!| c \rightsquigarrow d |\!)$ to denote the corresponding superclass field. In other words, $(\!| f |\!)$ = `c.const_`f and $(\!| c \rightsquigarrow d |\!)$ = `c.super_`d.

3.1.3.3 Superclass paths

The class hierarchy in HOL is rather complex. An excerpt, relating to the running example, is reproduced in Figure 3.2. For example, to obtain the `plus` operation from a `semiring` constraint, one has to follow three subclass–superclasses edges.

In general, for any two classes c and d, there may be multiple different paths from the subclass c and the (possibly indirect) superclass d. It is not obvious that the choice of path is irrelevant for the semantics of the generated program, i.e., that the system is *coherent* according to Jones [68]. Isabelle's type system guarantees coherence [98, 99], which the dictionary construction assumes. If that assumption were violated, the equivalence proof (§3.1.6) would fail. Coherence is consequently a meta-theorem and not internalised in the logic.

For the purpose of this presentation, it is sufficient to assume that the implementation uses the "first" path according to the kernel-defined order of superclasses. It is straightforward to extend the notation $(\!| c \rightsquigarrow d |\!) :: \alpha$ `dict_`$c \Longrightarrow \alpha$ `dict_`d for an indirect superclass d of c, where the edges are conjoined using the function composition operator $\circ$.

3.1.3.4 Non-class constants

Class constants can be easily distinguished from non-class ones: The former have no defining equations. They are only given meaning by an instance of a class.

In the example in Figure 3.1, the constants `f`, `g` and `plus_nat` are non-class constants, whereas `plus` is a class constant. This is reflected in the graph: the box for `plus` has no defining equations. Note that while `plus_nat` participates in the instantiation of a type class, it is itself not considered to be a class constant.

Assume a transformation of a non-class constant f with a set of defining equations eq_i. Each of the eq_i is of the form $f\ p_{i,1}\ p_{i,2}\ \cdots\ p_{i,n_i} \equiv rhs_i$, with the $p_{i,j}$ being constructor patterns. Furthermore, the type of f is a type scheme, i.e., it is of the form $\forall\alpha_1 :: s_1 \ldots \forall\alpha_k :: s_k.\ \tau$. Each of the schematic type variables α_i may carry a sort constraint $s_i = \{c_{i,1}, \ldots, c_{i,m_i}\}$ that is assumed to be normal.

Let $[\![t]\!]_\Gamma$ denote the translation of terms in a context Γ (to be defined later). Also, let $(\!|f|\!)$ mean a fresh name, e.g. f' to refer to the newly defined constant.

I can now explain the translation of the defining equations. Each equation eq_i gives rise to a new equation $[\![eq_i]\!]$ as follows:

- For every class constraint of every type variable, a new parameter is introduced.
- The existing parameters stay unchanged, because data constructors do not participate in the dictionary construction.
- The right-hand side is translated with all new parameters as context.

Formally:

$$\begin{aligned}
[\![eq_i]\!] &= (lhs'_i \equiv [\![rhs_i]\!]_\Gamma) \\
\Gamma &= [dict\_c_{1,1}, \ldots, dict\_c_{k,m_k}] \\
lhs'_i &= (\!|f|\!)\ (dict\_c_{1,1} :: \alpha_1\ \texttt{dict\_}c_{1,1})\ \ldots\ (dict\_c_{k,m_k} :: \alpha_k\ \texttt{dict\_}c_{k,m_k})\ p_{i,1}\ p_{i,2}\ \cdots\ p_{i,n_i}
\end{aligned}$$

Note that the translation for left-hand and right-hand sides differs: left-hand sides, consisting only of patterns, need no context.

All resulting equations are considered as defining equations for $(\!|f|\!)$. Subsequently, they are fed into the internal interface of the **function** command to produce a new logical constant. The additional technical challenges of this are documented in the following sections.

3.1.3.5 Instance definitions and composition

An instance $\kappa :: (s_1, \ldots, s_k)\ c$ is treated as if it is a (non-class) constant with no arguments, returning a dictionary containing instantiations of all class constants. Consequently, each instance gives rise to a new definition that I refer to as $(\!|\kappa :: c|\!)$.

However, it is also necessary to compose instances from contexts. This might mean a combination of following along superclass paths and applying instance definitions to arguments. I use $[\![\tau :: c]\!]_\Gamma$ as notation for this, where τ is a simple type and Γ a context.

I will first describe the (deterministic) algorithm to obtain $[\![\tau :: c]\!]_\Gamma$.

1. If τ is a type variable, find an instance *dict* for $\tau :: c'$ in Γ where c' is a subclass of c. Then, $[\![\tau :: c]\!]_\Gamma = (\!| c' \leadsto c |\!)\ \mathit{dict}$.
2. Otherwise, τ is of the form $(\tau_1, \ldots, \tau_k)\ \kappa$, i.e., a k-ary type constructor κ applied to k types. Find an instance definition $\kappa :: (s_1, \ldots, s_k)\ c'$ where:
   - c' is a subclass of c and
   - for each constraint $\tau_i :: c_{i,j}$ stemming from the s_i, $r_{i,j} = [\![\tau_i :: c_{i,j}]\!]_\Gamma$ is defined

   Then, $[\![\tau :: c]\!]_\Gamma = (\!| c' \leadsto c |\!)\ ((\!| \kappa :: c' |\!)\ r_{1,1}\ \ldots\ r_{k,m_k})$.
3. If no suitable instance exists, fail.

For any well-sorted judgement $\tau :: c$, this algorithm is guaranteed to find at least one composed instance. Similar to finding superclass paths, the choice of instance is irrelevant. This is a meta-theorem based on the *coregularity* property that is guaranteed by Isabelle's type system [98, 99].

It remains to treat instance definitions $(\!| \kappa :: c |\!)$. Assuming the same naming conventions as above, the generated definition is of the following form:

$$(\!| \kappa :: c |\!)\ \mathit{dict}_{1,1}\ \ldots = \mathsf{mk\_}c\ [\![(\alpha_1, \ldots, \alpha_k)\ \kappa :: c_1]\!]_\Gamma\ \ldots\ [\![(\alpha_1, \ldots, \alpha_k)\ \kappa :: c_n]\!]_\Gamma\ [\![f_1]\!]_\Gamma\ \ldots\ [\![f_m]\!]_\Gamma$$

3.1.3.6 Terms

I define the translation of terms $[\![t]\!]$ that are not constants recursively as follows:

$$\begin{aligned} [\![x]\!]_\Gamma &= x && \text{(where } x \text{ is a variable)} \\ [\![t\ u]\!]_\Gamma &= [\![t]\!]_\Gamma\ [\![u]\!]_\Gamma \\ [\![\lambda x.\, t]\!]_\Gamma &= \lambda x.\, [\![t]\!]_\Gamma \end{aligned}$$

The rule for constants is a bit more involved. Let f be a constant with k type parameters, i.e., of type scheme $\forall \alpha_1 :: s_1 \ldots \forall \alpha_k :: s_k$. In any occurrence of f in a term, these type parameters are instantiated with simple types $\tau_1, \ldots, \tau_k$.

$$[\![f]\!]_\Gamma = (\!| f |\!)\ [\![\tau_1 :: c_{1,1}]\!]_\Gamma\ \ldots\ [\![\tau_k :: c_{k,m_k}]\!]_\Gamma$$

3.1.3.7 Challenges

In the standard case, where the user has not performed a custom code setup, the resulting function looks similar to its original definition. But the user may have also changed the implementation of a function significantly afterwards. This poses some challenges:

- The new constants need to be proven terminating. The routine heuristically transfers the original termination proof to the new definitions (§3.1.4). This only works when the termination condition does not rely on class axioms.
- The domain of functions must be tracked, because even though HOL is a total logic, functions may be under-specified. Congruence rules are used to construct an inductive predicate (§2.5) representing the *side condition* of a function (§3.1.5).

```
fun f :: nat ⟹ nat where
f 0 = 0
f (Suc n) = f n

lemma [code]: f x = f x by simp
```

Listing 3.4: Pathological example of a non-terminating defining equation

- In order to fine-tune executable code, the code generator allows users to specify different constructors of a datatype than those it has been defined with, or even to introduce constructors for non-datatypes. However, the **function** command does not support that in general (§3.5). But even if it did, the equivalence of old and new definition may become conditional on invariants, which is conceptually not supported (§3.5).
- The set of defining equations must be non-overlapping to ensure determinism. Additionally, to accommodate for later phases in the compiler (§6.3), some pattern variables need to be renamed (§3.4).

3.1.4 Preservation of termination

As indicated above, the newly defined functions must be proven terminating. In general, I cannot reuse the original termination proof, as the example in Listing 3.4 illustrates. While the original function is primitively recursive, and hence trivially proved to be terminating, the user has added a defining equation that characterizes a non-terminating implementation. My construction cannot deal with such pathological cases, but fortunately they are rare in practice. The invocation of the dictionary construction would just fail for this example.

Instead, based on my experience, the most common cases are that users either

- do not adapt the defining equations at all,
- adapt them without changing the termination scheme, or
- adapt them to use different recursive calls, while still being terminating.

For the last case, it is impossible to port the existing termination proof, because it is not applicable any more. Hence, the construction falls back to use the same automated proof method as the **function** package.

Termination

However, the other cases are more interesting. In the remainder of this section, I will illustrate the first case, which is a specialization of the second one. The original termination proof should intuitively be still applicable.

The running example will be a function that sums up values in a list.

```
fun sum_list :: α::{plus,zero} list ⟹ α where
sum_list [] = 0
sum_list (x # xs) = x + sum_list xs
```

This function carries two distinct class constraints – arising from the use of addition and zero, both of which are provided by a class in Isabelle – which are translated into two dictionary parameters:

```
sum_list' dict_plus dict_zero [] =
    const_zero dict_zero
sum_list' dict_plus dict_zero (x # xs) =
    const_plus dict_plus x (sum_list' dict_plus dict_zero xs)
```

Here, the termination argument has not changed: While two additional parameters have been introduced, they remain unchanged in between recursive calls. Observe that whenever sort constraints are present, the dictionary construction always introduces new arguments, but keeps the termination scheme.

Now, the termination of `sum_list'` must be proved. The **function** package analyses the structure of recursive calls and collects them into a set of constraints.

As a notation for constraints, I will use $\bar{p} \rightsquigarrow \bar{x}$. $\bar{p}$ stands for the (tupled) patterns on the left-hand side of an equation and $\bar{x}$ for the (also tupled) actual parameters passed to a recursive invocation. Only variables bound in $\bar{p}$ may appear in $\bar{x}$.

The **function** command not only tracks the parameters passed to a recursive call, but also, under which conditions such a call appears. For example, a recursive call may appear in a **then** or **else** branch. To properly represent that, the notation needs to be extended to allow for arbitrary predicates. For explaining the termination heuristics, this generality is not needed; but it will be revisited for another purpose in §3.1.5.

For the above example, this looks as follows:

$$\{(x \mathbin{\#} xs) \rightsquigarrow xs\} \tag{sum\_list}$$

$$\{(dict\_plus, dict\_zero, x \mathbin{\#} xs) \rightsquigarrow (dict\_plus, dict\_zero, xs)\} \tag{sum\_list'}$$

Internally, for every function $f :: \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau$, the package defines an inductive relation f`_rel` $:: (\sigma_1, \sigma_2, \ldots, \sigma_n) \Rightarrow (\sigma_1, \sigma_2, \ldots, \sigma_n) \Rightarrow$ `bool` with one introduction rule per constraint. Note that the arguments are tupled, i.e. all function arguments participate in the definition of this *termination relation.*

In the example, the predicate `sum_list_rel` is defined by the following introduction rule:

$$\frac{}{\texttt{sum\_list\_rel}\ xs\ (x \mathbin{\#} xs)}$$

For details on how the **function** package assembles the termination relation based on the constraints, in particular for more complicated recursion schemes, refer to Krauss' thesis [69].

To prove that a function terminates, it is sufficient to show that its termination relation is *well-founded.* In the majority of cases, this happens by supplying a suitable *measure function* that maps the arguments to natural numbers and decreases for each recursive call. The **function** package is able to try out various measure functions automatically.

In this setting however, the termination of f has already been proved, either automatically or by the user. The construction tries to re-use that proof, i.e., the well-foundedness theorem

$$\begin{tikzcd}
P & x_1 \arrow[r] & x_2 \arrow[r] & x_3 \arrow[r] & \cdots \\
P' & x'_1 \arrow[r] \arrow[u, dashed, "g"'] & x'_2 \arrow[r] \arrow[u, dashed, "g"'] & x'_3 \arrow[r] \arrow[u, dashed, "g"'] & \cdots
\end{tikzcd}$$

Figure 3.3: Well-founded simulation

of *f*_`rel`, for the proof of well-foundedness of *f′*_`rel`, where *f′* is the result of applying the dictionary construction to *f*. Except for the additional (unchanging) dictionary arguments, these relations are more or less equivalent to each other.

Lemma 3.1 (Well-founded simulation). *Let* $P :: \tau \Rightarrow \tau \Rightarrow$ `bool` *be a well-founded relation and* $g :: \sigma \Rightarrow \tau$ *a function such that*

$$\forall x\, y.\ P'\ x\ y \implies P\,(g\ x)\,(g\ y)$$

Then, $P' :: \sigma \Rightarrow \sigma \Rightarrow$ `bool` *is also a well-founded relation.*

This theorem allows to *simulate* the structure of the recursive calls of *f′* with those of *f* (depicted in Figure 3.3). There is an important difference, though: *f*_`rel` may have sort constraints, *f′*_`rel` does not.

Instantiating the above lemma with the two termination relations entails choosing a suitable function *g* that maps arguments of `sum_list'` to arguments of `sum_list`, i.e., a function of type

$$(\alpha\ \texttt{dict\_plus} \times \alpha\ \texttt{dict\_zero} \times \alpha\ \texttt{list}) \Rightarrow (\beta :: \{\texttt{plus}, \texttt{zero}\})\ \texttt{list}$$

for an arbitrary type *β*. Obviously, *g* can drop the first two elements of the tuple. The challenge arises when trying to map a list with element type *α* to one with element type *β*. I cannot instantiate $\beta = \alpha$, because *β* carries a sort constraint.

In a parametric setting, this would be the end of it, because it is impossible to write such a function [41, 85, 109]. Isabelle however offers an escape hatch: recall that all types are non-empty. The polymorphic constant `undefined` :: *α* can serve as a witness for an arbitrary type. Assuming that there is at least one concrete type *τ* that satisfies the sort constraints of *β*, I can instantiate $\beta = \tau$. The desired mapping function can now be specified as follows:

$$g\,(\_, \_, xs) = \texttt{map}\ (\lambda\_.\ \texttt{undefined} :: \tau)\ xs$$

In case there is no such concrete *τ*, the above expressions fails to type check, causing the heuristic to fail.

It remains to show how the premise of the well-founded simulation theorem is proved in this particular case:

$$\forall x\, y.\ \texttt{sum\_list'\_rel}\ x\ y \implies \texttt{sum\_list\_rel}\ (g\ x)\ (g\ y)$$

The proof proceeds by induction using the induction principle of the `sum_list'_rel` inductive predicate, which gives one case per introduction rule, that is:

$$\forall d\_plus\ d\_zero\ x\ xs.\ \texttt{sum\_list\_rel}\ (g\ (d\_plus, d\_zero, xs))\ (g\ (d\_plus, d\_zero, x \mathbin{\#} xs))$$

After unfolding the definition of g:

$$\forall xs.\ \texttt{sum\_list\_rel}\ (\texttt{map}\ (\lambda\_.\ \texttt{undefined})\ xs)\ (\texttt{undefined} \mathbin{\#} \texttt{map}\ (\lambda\_.\ \texttt{undefined})\ xs)$$

This can now trivially be proved by using the introduction rule of `sum_list_rel`.

This construction critically depends on the non-emptiness of types and that there is at least one type satisfying the sort constraints of a function. In that sense, it only works because HOL admits non-parametric definitions [41, 85, 109].

More generally, this construction allows the proof of well-foundedness of any relation

$$R' :: (\beta_1 \times \ldots \times \beta_n \times (\alpha_1, \ldots, \alpha_k)\ \tau) \Rightarrow (\beta_1 \times \ldots \times \beta_n \times (\alpha_1, \ldots, \alpha_k)\ \tau) \Rightarrow \texttt{bool}$$

given a well-founded relation

$$R :: (\alpha_1, \ldots, \alpha_k)\ \tau' \Rightarrow (\alpha_1, \ldots, \alpha_k)\ \tau' \Rightarrow \texttt{bool}$$

where τ is a suitable type constructor equipped with a functorial $\texttt{map}_\tau$, τ and τ' differ only in sort constraints, R and R' are structurally equivalent and parametric in all α_i.

The precise nature of "suitability" is not relevant for the discussion. In the implementation, bounded natural functors as introduced by Blanchette et al. [16, 18] without dead variables are considered suitable.

The mapping function g is defined as follows:

$$\begin{aligned} &g :: (\beta_1 \times \ldots \times \beta_n \times (\alpha_1, \ldots, \alpha_k)\ \tau) \Rightarrow (\alpha_1, \ldots, \alpha_k)\ \tau' \\ &g\ (\_, \ldots, \_, t) = \texttt{map}_\tau\ \underbrace{(\lambda\_.\ \texttt{undefined})\ \ldots\ (\lambda\_.\ \texttt{undefined})}_{\text{one for each } \alpha_i}\ t \end{aligned}$$

3.1.5 Partially specified functions

HOL is a total logic, that is, it is always possible to assign a value to a function $f :: \alpha \Rightarrow \beta$ applied to any argument $x :: \alpha$. This immediately raises the question how to represent *partially specified* (or *under-specified*) functions, e.g. to obtain the head of a list:

```
fun hd :: α list ⇒ α where
hd (x # xs) = x
```

Obviously, in this function definition, the case for the empty list is omitted. Note that under-specification and non-termination are different kinds of partiality; the latter of which is not supported by this work.

It is possible to define and reason about non-terminating functions in Isabelle. However, both the dictionary construction and the deep embedding steps (§5) are unable to process such definitions. Other steps of the pipeline, in particular the verified part (§6), do not presuppose termination of the original equations.

There are various ways to deal with under-specification (a more detailed survey is given in §3.1.7):

1. Lift the result type into `option`, i.e. $\alpha \Longrightarrow \beta$ turns into $\alpha \Longrightarrow \beta$ `option`. Arguments for which the function is not specified get assigned a None value. The domain of the function dom_f can conveniently be expressed as a set $\{x \mid f\,x \neq \mathsf{None}\}$.
2. Function definitions are "artificially" completed to be always specified. In HOL, `undefined` is an unspecified constant of arbitrary type. The meta-theorem about this unspecified constant is that for all predicates P, P `undefined` is provable if and only if $P\,x$ is provable for all x.
3. Based on the function equations, derive a set carrying the *side condition* of the function and allow reasoning over a function application $f\,x$ only if $x \in \mathtt{side}_f$ holds. This is comparable to refinement types [43], but where the constraints are external and not part of the type.

I will examine the specification of the hd function for each of the different approaches.

Lifting This approach is preferred in many functional programming languages, like Haskell, where types may be non-empty (ignoring exceptions). The major problem is that it may lead to complicated proof statements when reasoning about such functions.

```
fun hd :: α list ⟹ α option where
hd (x # xs) = Some x
hd [] = None
```

In this setting, this would require non-trivial transformation of existing defining equations. Wimmer et al. [130] solve a similar problem in the context of memoization: they lift functions into the state monad. The main weakness is that higher-order functions need to be lifted manually. Because many existing Isabelle formalizations make use of custom combinators, their approach is not feasible here.

Completion Isabelle's **function** package uses this approach by default. For any given function definition, a catch-all clause is added (a process called *completion*):

```
fun hd :: α list ⟹ α where
hd (x # xs) = x
hd _ = undefined
```

As far as the **function** package is concerned, this function is now specified for all input values.

To avoid leaking this implementation detail to users, Isabelle's simplifier will not rewrite the term hd [] to `undefined`. But the completion is visible in the generated induction principle `hd.induct`:

$$\frac{\forall x\ xs.\ P\,(x \mathbin{\#} xs) \qquad P\,[]}{P\,a}$$

The premise $P\,[]$ is necessary for this theorem to hold. Otherwise, $P\,xs = (xs \neq [])$ would be a counterexample.

While this approach is conceptually simple, it poses a significant challenge for the dictionary construction and associated proof tactics. The reason is as profound as it is technical: Applications of functions to values on which they are not specified are practically "opaque"; in the sense that it is difficult to rewrite or prove anything about them. To make matters worse, identities like $\texttt{undefined}\ x = \texttt{undefined}$ are unprovable, meaning `undefined` behaves differently than e.g. $\bot$ in Haskell (where $\bot\ x = \bot$ holds). It is hence an insufficient approximation of under-specification for the purposes of code generation.

Side conditions To avoid modifying the internal mechanics of the **function** package, and consistent with Myreen and Owens' approach [94], I have chosen to track side conditions of functions. They are represented as inductive predicates. In the case of the head function, the predicate is specified as follows:

$$\frac{}{\texttt{hd\_side}\ (x \mathbin{\#} xs)}$$

When producing a certificate for the dictionary translation (§3.1.2.2) for an under-specified function `f`, the routine introduces a new premise:

$$\texttt{cert\_plus}\ dict \implies \texttt{f\_side}\ x \implies \texttt{f'}\ dict\ x = \texttt{f}\ x$$

A more subtle change is that the theorem now has to be stated in η-expanded form. This may limit its applicability in higher-order position, e.g. `map f`.

Before describing the construction of the inductive predicates for side conditions, I will rehash the concept of *congruence rules.*

3.1.5.1 Congruence rules

The notion of *congruence rules* goes back to the literature on term rewriting [36, 37]. Later, they have become instrumental in the context of admitting recursive definitions in higher-order logics [69, 70, 113].

Definition 3.2 (Congruence rule). *A* congruence rule *for the function* `c` *is a theorem of the form*

$$\frac{P_1 \qquad \cdots \qquad P_n}{\texttt{c}\ x_1\ \ldots\ x_n = \texttt{c}\ y_1\ \ldots\ y_n}$$

where the P_i may refer to arbitrary x_i and y_i.

Usually, the P_i takes either of these two forms:

- $Q_i \implies x_i = y_i$, when $x_i :: \tau$ and τ is not a function type
- $\forall \bar{z}.\ Q_i\ \bar{z} \implies x_i\ \bar{z} = y_i\ \bar{z}$, otherwise

Slind [113, §2.7.1] calls a congruence rule where no functions are passed as arguments *simple*. Two examples of those are given below:

$$\frac{C_1 = C_2 \qquad C_1 \implies x_1 = x_2 \qquad \neg C_1 \implies y_1 = y_2}{\textbf{if } C_1 \textbf{ then } x_1 \textbf{ then } y_1 = \textbf{if } C_2 \textbf{ then } x_2 \textbf{ then } y_2}$$

$$\frac{A_1 = A_2 \qquad A_1 \implies B_1 = B_2}{A_1 \wedge B_1 = A_2 \wedge B_2}$$

The purpose of these rules is to track *evaluation context*. This is important because HOL itself has no notion of evaluation order, but target languages do. In particular, both in Slind's **recdef** and Krauss' **function** package, congruence rules are used to determine the termination relation of a function. Both take the Q_i of the rules into account to guard recursive invocations, like in the following example:

```
fun fac :: nat ⇒ nat where
fac n = (if n = 0 then 1 else n * fac (n - 1))
```

A naive termination analysis would complain that `fac` never terminates, because there is always a recursive call. The **function** package however derives the following termination relation:

$$\frac{n \neq 0}{\texttt{fac\_rel}\ (n-1)\ n}$$

This is clearly well-founded, i.e. `fac` terminates on all inputs, because there is no n' such that `fac_rel` n' 0.

Extending the notation from §3.1.4 with arbitrary conditions, the above relation can be more succinctly expressed as:

$$\{n \overset{n \neq 0}{\rightsquigarrow} (n-1)\}$$

More complex cases arise when higher-order recursion is present. Consider this datatype and function:

```
datatype α tree = Fork (α tree list) | Leaf α

fun map_tree where
map_tree f (Fork ts) = Fork (map (map_tree f) ts)
map_tree f (Leaf x) = Leaf (f x)
```

It takes more work to understand this *nested* recursion principle. It is not directly obvious on which values `map_tree` is called recursively, because it only appears in partially applied form in the function body. The **function** package uses the higher-order congruence rule for `map` to deduce the termination relation:

$$\frac{\forall x.\ x \in \mathtt{set}\ xs \implies f\ x = g\ x \qquad xs = ys}{\mathtt{map}\ f\ xs = \mathtt{map}\ g\ ys}$$

Intuitively speaking, this represents the fact that the function passed to `map` is applied to each element in the set of *xs*, where `set` is a function that turns a list into a set. Consequently, the termination relation states exactly that:

$$\frac{t \in \mathtt{set}\ ts}{\mathtt{map\_tree\_rel}\ (f, t)\ (f, \mathtt{Fork}\ ts)}$$

The $\leadsto$ notation starts to break down for this example, because the recursive call depends on a variable (here: *t*) that is not bound by the patterns of the defining equation (here: *ts*). Consequently, in the remainder of this section, I will only use the rule-based notation to represent inductive predicates.

Well-foundedness can be proved by appealing to the size of the arguments. The **datatype** package provides a $\mathtt{size}_\tau$ function for each type constructor τ that counts the number of data constructors in the value. Consequently, if *t* is an element of *ts*, then the size of *t* is smaller than the size of `Fork` *ts*.

All of the necessary infrastructure for this is fully automated in Isabelle:

- generation of $\mathtt{map\_}\tau$ and $\mathtt{set\_}\tau$ functions for datatypes and bounded natural functors τ,
- proof of a suitable higher-order congruence rule,
- generation of $\mathtt{size\_}\tau$ functions for datatypes τ,
- setup of the **function** package.

The only occasion when a user has to adjust the setup is when they introduce a custom higher-order recursion combinator, or when a function definition uses a more complicated termination measure than the size of the inputs.

3.1.5.2 Specifiedness

Congruence rules can also be used to determine on which inputs functions are specified. A similar routine as in the **function** package can be employed to analyse function definitions. Here, the goal is to construct an inductive predicate capturing the set of arguments for which a function is specified.

For example, consider the following (contrived) function definition:

```
fun hd_tl :: α list list ⇒ α list
hd_tl [] = []
hd_tl (x # xs) = map hd xs
```

The function itself has no obvious unspecified behaviour, because all possible inputs are covered by pattern matching. However, the function `hd` is unspecified for empty lists. The desired side condition is:

```
type rule = {rule: thm, concl: term, prems: term list, proper: bool}

type ctx = (string * typ) list * term list

datatype ctx_tree = Tree of (term * (rule * (ctx * ctx_tree) list) option)
```

Listing 3.5: Type of context trees

$$\frac{}{\mathtt{hd\_tl\_side}\ []} \qquad \frac{\forall x.\ x \in \mathsf{set}\ xs \implies \mathtt{hd\_side}\ x}{\mathtt{hd\_tl\_side}\ (x \# xs)}$$

This can further be simplified by noting that `hd_side` $x \Longleftrightarrow x \neq []$. This inductive definition can be obtained by performing a recursive analysis on the defining equations of a constant. Each equation of `f` gives rise to a rule in `f_side`.

Note that as far as Isabelle's total logic is concerned, this function is total: The `hd` function just returns `undefined` on empty lists. Because of this, any notion of *specifiedness* cannot be fully formalized and has to be – to some extent – a heuristic. A good intuition is that I want to characterize all inputs x to a function `f` such that evaluation after code generation to a target language does not yield a run-time exception.

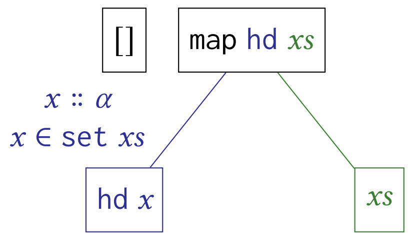
Transformation to forest The routine starts by importing the (unstructured) set of congruence rules into a dedicated data structure (Listing 3.5). Then, the right-hand sides of all defining equations are converted into a *context tree.* Each node of the tree is labelled with a term and optionally, a congruence rule, and may have arbitrarily many children. An edge from a node to a child is labelled with a *context:* a list of variables and of assumptions.

In usual Isabelle parlance, a "context" would refer to a `Proof.context` that may contain arbitrary data; in this case, it is just fixed variables and assumptions.

An example based on the `hd_tl` function is given in Figure 3.4. It illustrates how the congruence rule of `map` participates in the transformation.

The transformation algorithm itself can be summarized as follows. For a term t, a node is generated. Then, it adds children to the node by case distinction on the shape of t:

- If t is atomic, it becomes a leaf node.
- If t is a function application `f` $x_1 \ \dots \ x_n$, it tries to find a congruence rule that matches the term. k children are added to the node according to the k premises of the rule, tracking their variables and assumptions as context of the respective child. If there is no matching rule, the function and its arguments are considered separately, hence creating $n + 1$ children.
- Otherwise, t is an abstraction $\lambda x.\ u$. One child for u is added with x as context.

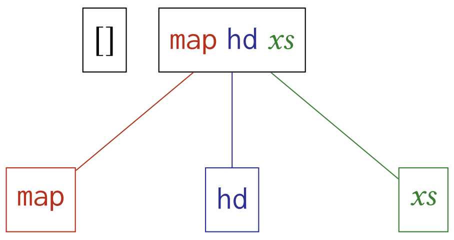


(a) Forest with congruence rule for map

$$\frac{}{\texttt{hd\_tl\_side}\ []}$$

$$\frac{\texttt{map\_side hd}\ xs \qquad \forall x.\ x \in \texttt{set}\ xs \Longrightarrow \texttt{hd\_side}\ x}{\texttt{hd\_tl\_side}\ (x\ \#\ xs)}$$

(b) Side condition with congruence rule for map



(c) Forest without congruence rule for map

$$\frac{}{\texttt{hd\_tl\_side}\ []}$$

$$\frac{\texttt{map\_side hd}\ xs \qquad \forall x.\ \texttt{hd\_side}\ x}{\texttt{hd\_tl\_side}\ (x\ \#\ xs)}$$

(d) Side condition without congruence rule for map

Figure 3.4: Context forests and resulting side conditions of hd_tl

In Figure 3.4a, there are two trees.

- `hd_tl [] = []` gives rise to the tree with just a leaf node [], because [] is an atomic constant.
- `hd_tl (`*x* `#` *xs*`) = map hd` *xs* produces a node with two children, after applying the congruence rule for map. The left child has a context enriched with a variable and assumption, whereas the right child is the atomic *xs*.

Ignoring the congruence rule results in the forest in Figure 3.4c, where three children are generated for the binary call to map.

Transformation to predicate Finally, the tree is transformed into a set of introduction rules for the inductive predicate representing the specifiedness. Figure 3.4b illustrates the result of the transformation. In a tree, each path from root to the side condition generates one assumption in the side condition based on the pre-existing side conditions, unless:

- it is the first child of a function application with no matching congruence rule (nothing is known about that function call), or
- it is a free variable (always specified), or
- no side condition is known about the term (assumed total).

Crucially, each layer of the tree still contributes to the side condition. In the running example, this means that the root node contributes the assumption `map_side hd` *xs*.

A special case arises in Figure 3.4d, where an assumption is generated for the node hd. Because the forest has been created without a suitable congruence rule, there is no variable in the context of the node. The transformation hence introduces a synthetic variable and universally quantifies over it (marked brown in the figure). It can be proved that $\forall x.$ `hd_side` x is false, because there is a list that violates `hd_side`: the empty list. In general, absence of congruence rules or congruence rules that are too weak may lead to vacuous side conditions.

The side condition for `undefined` is a prime example for being vacuous by definition: there are no defining equations, hence no context trees, hence the inductive predicate is the empty least-fixed point. The **inductive** package admits such empty predicates. They are logically vacuous, i.e., `undefined_side` $\Longleftrightarrow$ `False`.

Simplification To avoid overly complicated side conditions, there are two strategies to simplify them. The routine tries to:

1. prove totality, i.e. $\forall x_1 \ldots x_n.$ `f_side` $x_1 \ \ldots \ x_n$, and
2. discharge auxiliary side conditions, e.g. `hd_side` $(x$ `#` $xs)$ = `True`.

Both work by suitable preprocessing of the goal, then running Isabelle's full simplifier. While this can make the result unpredictable, I have found that this prevents many redundant assumptions.

For the example in Figure 3.4, this removes the assumption `map_side hd` xs, because `map_side` can be proved to be total.

It is important to note that the generated side conditions are *shallow*, that is, they only characterize the specifiedness of one function, but not any other non-constant functions, i.e. functions that are passed in as parameters, that are called along the way. This is nicely illustrated by this example: While the `map` function is obviously fully specified, it can be used in a partially specified way; namely, when the mapping function is only partially specified. The challenges to fully capture specifiedness are described in §3.5.

3.1.5.3 Differences to Krauss' routine

As indicated above, the **function** package employs a similar routine. The differences are mainly technical in nature, but are significant enough to prevent code reuse.

- The internally produced congruence tree is not exported as a data structure.
- Traversal happens on an intermediate constant that represents all functions in a mutually recursive bundle with tupled arguments. For example, simultaneous recursive definitions of `odd` and `even` functions would internally be presented as a single function of type (`nat` + `nat`) $\Rightarrow$ (`bool` + `bool`), where $\alpha + \beta$ denotes the sum type of α and β.
- Side conditions of auxiliary constants (in the running example: `map` and `hd`) are not considered: after defining and proving a function to be terminating, it is "total" by virtue of completion.

Notably, the extraction of a termination relation – just like specifiedness – critically depends on the presence of appropriate congruence rules. Similarly to the special case described above, absence of congruence rules may lead to unprovable termination relations. Consequently, it is reasonable to assume that a user of the **function** package is aware of this required setup; hence, it is not an extra burden to require the same setup for the dictionary construction.

3.1.6 Correctness proofs

There are two kinds of propositions that need to be proved in the routine: dictionary certificates and equivalence theorems (§3.1.2.3). In general, they are of the form:

$$\begin{aligned} \ldots &\Longrightarrow \texttt{cert\_}c\,(\texttt{inst\_}c\texttt{\_}\kappa\ dict_1\ \ldots) \\ \ldots &\Longrightarrow \texttt{f'}\ dict_1\ \ldots\ x_1\ \ldots = \texttt{f}\ x_1\ \ldots \end{aligned}$$

Both can carry preconditions for auxiliary dictionary certificates, and in the case of the equivalence theorems, also side conditions of the arguments x_i (§3.1.5). The proof strategies for both kinds of theorems differ, so I will discuss them separately. Both strategies have in common that they require the proofs to happen in exactly the same (topological) order as the dictionary construction itself (Figure 3.1). At any point during a sequence of proofs, the previous correctness theorems are referred to as *base theorems.*

Dictionary certificates Recall the definition of $\texttt{cert\_}c$ and $\llparenthesis \kappa :: c \rrparenthesis$. The latter is a plain constructor application ($\texttt{mk\_}c$); the former inspects each field. In other words, $\llparenthesis \kappa :: c \rrparenthesis$ "bundles" existing constants into a dictionary. Consequently, the proof proceeds by simple application of the base theorems.

Equivalence theorems In general, these theorems need to be proved by induction using the induction scheme generated by the side condition, or if the side condition is trivial, the termination relation of the function. Both are similar, so I focus on the latter.

Applying the induction principle creates one proof obligation per defining equation.Recall the `sum_list` function from §3.1.4. The proof obligations after induction are (dictionary certificates omitted):

$$\begin{aligned} &\texttt{sum\_list'}\ d_1\ d_2\ [] = \texttt{sum\_list}\ [] \\ \texttt{sum\_list'}\ d_1\ d_2\ xs = \texttt{sum\_list}\ xs \Longrightarrow\ &\texttt{sum\_list'}\ d_1\ d_2\ (x \# xs) = \texttt{sum\_list}\ (x \# xs) \end{aligned}$$

These can be discharged by first unfolding the defining equations of `sum_list'` and `sum_list`. Then, base theorems and induction hypotheses are applied by walking the congruence tree. Note that the base theorems include equivalences for class constants and the corresponding dictionary fields.

The reality is a bit more complicated: one equation may create multiple defining equations, because the **function** command disambiguates equations. For example, consider the definition `single` $[x]$ = `True` and `single` xs = `False`. The package instantiates the second equation ($xs = y \# ys$ and $xs = []$) to avoid ambiguities.

3.1.7 Related work

Dictionary construction Type classes have been pioneered by the Haskell programming language [92, 126]. There, a very similar construction to here is used, replacing classes by records and instances by functions [3, 26, 105]. Additional complications arise because Haskell admits cyclic dependencies between class instances [80], which are prohibited in Isabelle and hence pose no problem for this work. A further simplification compared with Haskell is that Isabelle only features an ML-style simply typed polymorphic lambda calculus without constructor or multi-parameter classes.

Idris, a dependently typed functional language, generalizes the concept of type classes to classes that can be parametrized by any value. In more recent versions of Idris, they are called *interfaces*. Still, the elaboration of expressions with class constraints and class and instance declarations is surprisingly similar to here [21].

In Scala, type classes are represented in an object-oriented style with objects and implicits [101]. Type classes are just regular classes that are subject to the same compilation process to the Java Virtual Machine (and other backends). Instances are also regular functions and constraints regular function arguments. However, programmers do not need to pass instances manually; the compiler fills them in automatically. The end result again is similar to that of Haskell and Idris, albeit it is mostly visible in the surface syntax instead of hidden in some intermediate compiler representation.

For Isabelle, Haftmann and Nipkow give a pen-and-paper correctness proof of the dictionary construction construction [49, §4.1], based on a notion of *higher-order rewrite systems*. The resulting theorem states that any well-typed term is reduction-equivalent before and after class elimination. In this work, the dictionary construction is performed in a certified fashion, that is, the equivalence is a theorem inside the logic.

Partially specified functions The notion of functions that are not defined universally for all possible (i.e. type-correct inputs) is pervasive in programming languages. For example, major functional languages, like Haskell, Scala, and OCaml admit introduction of unspecified behaviour by allowing incomplete pattern matches. The respective run-time environments throw an exception which can be caught by the caller of an underspecified function. This is not generally how it works in total logics, where it is impossible to carry out non-trivial proofs about unspecified values.

However, much work on modelling partiality in proof assistants is centred around partiality induced by non-termination, instead of under-specification. Finn et al. [38] introduce an approach for the *LAMBDA* system that tackles both issues in a similar way as Isabelle/HOL. The undefined or arbitrary constant can be defined in terms of Hilbert's choice operator: under the assumption that τ is inhabited, $\epsilon x :: \tau.$ False delivers a value of type τ that is otherwise unspecified. The authors use a similar example (head of a list) and point out that the usual axioms of classical logic still apply; for example, hd [] = 5 $\vee$ hd [] $\neq$ 5 is provable regardless of the unspecified result of retrieving the head of an empty list. They also give a routine that, given a set of recursive equations, defines a domain predicate and a function that is qualified on that predicate. My approach (§3.1.5.2) is similar to the one by Finn et al., with the notable improvement that I support higher-order recursion.

Cheng and Jones [27] give a survey of the treatment of partial functions, with the focus on usability. They examine a total of eight different approaches: restricting functions to a set while defining them, treating functions as relations, employing a different notion of equality, extending logical operators to handle undefined values, three-valued logics, introducing a distinction between strict and total and non-strict and non-commutative connectives, introducing a distinction between computation and reasoning, and finally, adding a separate judgement for definedness. My approach is most closely related to the last of these, which has initially been described by Plotkin as *Partial Function Logic* [27, §9]. The notable difference is that the definedness judgement is represented as “normal” boolean predicates in the logics.

3.2 Lazy evaluation

Portions of this section appear in the AFP entry “Lazifying case constants”, authored by Hupel [64]. Users are recommended to refer to that AFP entry for the latest documentation.

HOL has no built-in notion of pattern matching. Since my initial `term` type (§2.6) is modelled after it, this restriction applies there too.

Instead, what looks like pattern matching in the input syntax, is in fact syntax sugar for **case** functions. For example, the expression

$$\textbf{case}\ xs\ \textbf{of}\ [] \Longrightarrow a \mid y \# ys \Longrightarrow b\ y\ ys$$

is internally represented as

$$\texttt{case\_list}\ a\ b\ xs$$

Every datatype τ comes equipped with a corresponding **case** function $\texttt{case\_}\tau$. Nested pattern matches are translated to nested case function applications. **if-then-else** expressions are a special case of pattern matching, namely it can be directly represented as a `case_bool`.

A naive translation of nested function applications would yield strict evaluation of matches and **if-then-else**. This is in conflict to the semantics of these constructs in CakeML. Therefore, I introduce `unit`-abstraction to defer evaluation of the cases.

Lazy_Case

For a datatype τ, the type of $\texttt{case\_}\tau$ is $(\kappa_{1,1} \Longrightarrow \ldots \Longrightarrow \kappa_{1,n_1} \Longrightarrow \alpha) \Longrightarrow \ldots \Longrightarrow (\kappa_{k,1} \Longrightarrow \ldots \Longrightarrow \kappa_{k,n_1} \Longrightarrow \alpha) \Longrightarrow \tau \Longrightarrow \alpha$. That is, it is a higher-order function, where the first k arguments are functions from n_i constructor parameters to α. I insert a “dummy” unit argument for each constructor that has no parameters.

Consider the factorial function:

$$\texttt{fact}\ n = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \texttt{fact}\ (n - 1)$$

It can be rewritten to use pattern matching:

$$\texttt{fact}\ n = (\textbf{case}\ n = 0\ \textbf{of}\ \texttt{True} \Longrightarrow 1 \mid \texttt{False} \Longrightarrow\ n * \texttt{fact}\ (n - 1))$$

After preprocessing:

$$\texttt{fact}\ n = \texttt{case\_bool'}\ (\lambda(u :: \texttt{unit}).\ 1)\ (\lambda(u :: \texttt{unit}).\ n * \texttt{fact}\ (n - 1))\ (n = 0)$$

The definition of `case_bool'` itself is straightforward:

$$\begin{aligned}
\texttt{case\_bool' True}\ t\ f &= t\,() \\
\texttt{case\_bool' False}\ t\ f &= f\,()
\end{aligned}$$

This scheme means that CakeML's conditional expressions will never occur in generated code. However, pattern matching will be used, because **case** functions are regular functions and as such subjects to the pattern compilation phase (§6.3).

Another way to avoid the problem of strict evaluation would be to add a pattern-matching construction inside the `term` type. However, this would instead have meant special treatment of the **case** functions in the deep embedding (§5), because Isabelle's `term` type also does not have pattern matching.

3.3 Constructor functions

Portions of this section appear in the AFP entry "Constructor Functions", authored by Hupel [62]. Users are recommended to refer to that AFP entry for the latest documentation.

For each data constructor $\mathtt{C}_i$ of datatype τ (as exemplified above), Isabelle generates a function $\mathtt{C}_i :: \kappa_{i,1} \Longrightarrow \ldots \Longrightarrow \kappa_{i,n_i} \Longrightarrow \tau$. It behaves like a regular function, in that it can be passed to a higher-order function, partially applied, and generally be treated as a value.

In implementations of programming languages, data constructors behave differently: An application of $\mathtt{C}_i$ to n_i arguments cannot be reduced further, but instead creates an "atomic" value. Hence, the question arises on how to represent partially applied constructors.

Different functional languages answer this question differently. OCaml disallows partial application and forces users to explicitly write an anonymous function, e.g. $\lambda x.\ \mathtt{C}_i\ t_1\ \ldots\ t_{n_i-1}\ x$, as does CakeML. Standard ML however allows that and internally constructs such a function. The naive approach of plain eta-expansion does not work in Isabelle, because many routines, including code preprocessing, spontaneously eta-contract terms.

To bridge the gap, preprocessing introduces synthetic *constructor functions*. For each data constructor $\mathtt{C}_i$, it defines an additional function $\mathtt{C}_i'$ as $\mathtt{C}_i'\ t_1\ \ldots\ t_{n_i} = \mathtt{C}_i\ t_1\ \ldots\ t_{n_i}$, having the same type as $\mathtt{C}_i$. Every occurrences of $\mathtt{C}_i$ is replaced by $\mathtt{C}_i'$, except in the definition of $\mathtt{C}_i'$. Thus, it is statically guaranteed that each constructor call is fully applied.

For example, the term `map ((#) 1)` is rewritten to `map ((#′) 1)`. Unfolding the synthetic definition of (#′) yields the expression `map` $(\lambda x.\ 1\ \#\ x)$. There, # appears in fully-applied form.

3.4 Pattern compatibility

The pattern elimination phase of the compiler (§6.3) demands a stronger guarantee than non-overlapping patterns. The reason for that is deferred until that section, whereas here, I will explain how this requirement is ensured during preprocessing.

Definition 3.3 (Overlapping patterns). *Two patterns t_1 and t_2* overlap *if there is a term u that is matched by both t_1 and t_2.*

The definition for overlapping patterns given above differs from the one used in the formalization. A refined version based on different primitives is presented in §4.2.4.4.

Definition 3.4 (Pattern compatibility)**.** *Two patterns t and u are compatible:*

- *if both are applications, i.e., there are t_1, t_2, u_1, and u_2 such that $t = t_1\ t_2$ and $u = u_1\ u_2$, then*
  1. *t_1 and u_1 must be compatible, and*
  2. *if $t_1 = u_1$, then t_2 and u_2 must be compatible,*
- *otherwise,*
  1. *$t = u$, or*
  2. *t and u do not overlap.*

For example, the following function definition does not satisfy pattern compatibility, because the first parameter is named differently in the two equations:

```
fun map where
map f [] = []
map g (x # xs) = (* ... *)
```

Lemma 3.5 (Reflexivity)**.** *t is compatible to itself.*

Lemma 3.6. *For two compatible patterns t and u, they are either equal or do not overlap.*

Proof. I first prove the weaker result that t and u are equal or do not overlap by induction on the definition of pattern compatibility. The actual lemma follows together with reflexivity.

Consider the case where both t and u are not applications. Then, by definition, $t = u$ or t and u do not overlap.

The remaining case is where both are applications, i.e., $t = t_1\ t_2$ and $u = u_1\ u_2$. By definition, t_1 and u_1 must be compatible. I perform a case distinction on $t_1 = u_1 \wedge u_1 = u_2$.

- If $t_1 = u_1 \wedge u_1 = u_2$, then $t = u$.
- If $t_1 = u_1$, then $t_2 \neq u_2$. By definition, t_2 and u_2 are also compatible. Applying the induction hypothesis yields that $t_2 = u_2$ (contradiction) or t_2 and u_2 do not overlap. Hence, $t_1\ t_2$ and $u_1\ u_2$ do not overlap.
- Otherwise, I know that $t_1 \neq u_1$. Applying the induction hypothesis on the fact that t_1 and u_1 are compatible yields that $t_1 = u_1$ (contradiction) or t_1 and u_1 do not overlap. Hence, $t_1\ t_2$ and $u_1\ u_2$ do not overlap. □

The preprocessing will rename g to f to ensure that matching sub-patterns are also equal. This does not work in all cases (§3.5).

The algorithm itself is rather simple: starting out with an empty set of defining equations, each defining equation gets added to the set. Before adding, it is compared with all existing equations in the set and variables are renamed if necessary and possible. The resulting set are the new defining equations.

A more formal definition of pattern compatibility will be given in §4.2.4.5.

3.5 Limitations

This section discusses the limitations of the preprocessing phase. Here, *limitation* can either mean that a particular step in preprocessing fails to work on a particular set of definitions, or that it produces a result that is less general than expected. Most limitations can be avoided with specific workarounds.

3.5.1 Specifiedness

A particularly thorny issue is presented by functions that return other functions. While *currying* itself is a common idiom in functional programming, manipulation of partially applied functions would require a non-trivial data flow analysis. As a synthetic example, consider the expression `map (div)` xs :: `(nat` ⇒ `nat) list`, with `div` being the division operator on natural numbers. Clearly, the expression is fully specified for all xs, but its resulting list of functions is not: Passing in 0 to any of the resulting functions yields unspecified behaviour, or at run-time in a target language, throw an exception.

The underlying problem is that the congruence rule for `map` can only be used to extract side conditions when the return type of the function that is passed to it is not itself a function type. More formally, the heuristic requires that no type variables are instantiated with a function type in order to work correctly.

A similar situation arises in practice in the commonly used `show` derivation framework by Sternagel and Thiemann [119]. They employ Hughes' difference list representation of strings [59]. Luckily, all these functions are fully specified, i.e. the side condition is always true.

3.5.2 Patterns in function definitions

The **function** package, by default, allows only function definitions where the left-hand side matches on constructors. Consider as an example treating list as "snoc lists" instead of "cons lists", i.e., a pair of *init* and *last* instead of *head* and *tail*. A complete example is given in Listing 3.6. It first introduces the datatype for cons lists and defines the `append` and `Snoc` functions. Then, it instructs the code generator to use `Nil` and `Snoc` in the target language representation.

However, the code generator cannot export code for the `append` function, because it is defined in terms of `Cons`, and aborts with an error message that `Cons` is not a constructor on the left-hand side of the defining equation.

To fix this, Listing 3.6 demonstrates how to add defining equations for `append` that are defined in terms of `Snoc`. But now the **function** package would not accept such a definition of `append`, because while `Snoc` is a constructor as far as the code generator is concerned, it is still not a constructor as far as the **datatype** package is concerned. The key problem is that both subsystems may have diverging notions of exactly which constructors a datatype is comprised of.

The workaround for this problem is that it is possible to use the **function** package in a mode which allows to use arbitrary patterns on the left-hand side of a defining equation. It is

```
datatype α list = Cons α (α list) | Nil

fun append :: α list ⇒ α list ⇒ α list where
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

fun Snoc :: α list ⇒ α ⇒ α list where
Snoc xs x = append xs [x]

code_datatype Nil Snoc
```

(3.6.a) Full definition of a "cons list"

```
datatype 'a list = Snoc of 'a list * 'a | Nil;
```

(3.6.b) Generated datatype definition in Standard ML

```
lemma [code]:
  append xs Nil = xs
  append xs (Snoc ys y) = Snoc (append xs ys) y
(* proof *)
```

(3.6.c) Defining equations for append

Listing 3.6: Adapting a datatype to a different representation

---

only a workaround because the package demands some additional proofs (exhaustiveness, well-definedness) that are tedious to do by hand and impossible to automate in general.

For that reason, any type adaptations, including data refinement [51], are not supported by this work.

3.5.3 Pattern compatibility

The classic example for a set of equations with patterns that are difficult to deal with is the `diagonal` function [106, 113]:

```
fun diagonal :: bool ⇒ bool ⇒ bool ⇒ nat where
diagonal x True False = 1
diagonal False y True = 2
diagonal True False z = 3
```

The difficulty arises from the overlapping patterns. There is no possible renaming of variables such that this set of equations satisfies the pattern compatibility constraint. Consequently, the preprocessing step will produce an error message.

The workaround is to instantiate the first two equations with {`True`, `False`}, which can be achieved with modest effort in Isabelle:

```
diagonal True True False = 1
diagonal False True False = 1
diagonal False True True = 2
diagonal False False True = 2
diagonal True False z = 3
```

The process could also be automated, but because it is potentially an expensive operation, the user has the choice how to deal with the situation.

3.5.4 Congruence rules with constraints

Consider a variant of the `sum_list` function that first applies a function before summing up the values:

```
fun sum_by :: (α ⇒ β::{plus,zero}) ⇒ α by ⇒ β where
sum_by _ [] = 0
sum_by f (x # xs) = f x + sum_by f xs
```

In order to use this function in a higher-order recursion, a congruence rule needs to be proved. The resulting rule is very similar to the congruence rule of `map`:

$$\frac{\forall x.\; x \in \mathtt{set}\; xs \implies f\,x = g\,x \qquad xs = ys}{\mathtt{sum\_by}\; f\; xs = \mathtt{sum\_by}\; g\; ys}$$

Since `sum_by` has sort constraints, the dictionary construction will introduce additional parameters.

Now suppose that another function `h` uses `sum_by`. Dictionary construction replaces occurrences of `sum_by` in `h` by `sum_by'`. But `sum_by'` has no congruence rule, which means that the termination proof of `h` must fail.

Besides the termination heuristic, I have also implemented a heuristic method to transfer congruence rules to the new constants. The **function** package only accepts unconditional congruence rules; consequently, this only works when a function equipped with a congruence rule is fully specified and has no visible sort constraints.

For that reason, a function like `sum_by` cannot be used as a higher-order recursion combinator. Instead, it needs to be split into two parts: mapping and summing. This can be registered in the code generator as follows:

```
lemma sum_by[code_unfold]: sum_by f = sum_list ∘ sum_by f
(* proof *)
```

3.5.5 Preservation of termination

The following pathological example exhibits the problem that some functions cannot be proved to terminate after elimination of sort constraints:

```
function sum_set :: α::{finite,comm_monoid_add,linorder} set ⇒ α where
sum_set S = (if S = {} then 0 else Min S + sum_set (S - {Min S}))
```

This function, analogously to `sum_list`, should compute the sum of a set. It can only be proved terminating because of the sort constraints: otherwise, there may be infinitely many elements in *S*, no well-defined minimum element, or the result may be depending on the order. With the constraint, the termination relation is the cardinality of the set which decreases with each recursive call.

However, the termination heuristics cannot cope with this example. The dictionary construction removes all sort constraints from the type variables; instead introducing value parameters. The dictionary type for `finite` would be isomorphic to `unit`, because the class does not have any parameters. But now, the new termination relation cannot be simulated by the old one: it has to deal with arbitrary, possibly infinite sets.

Fortunately, function definitions like this are rare. If necessary, they can be replaced by a recursion on lists as follows:

```
lemma sum_set_set[code_unfold]: sum_set (set xs) = sum_list (remdups xs)
(* proof *)
```

This replaces all occurrences of `sum_set` applied to a finite set of elements *xs* by an application of `sum_list` (after removing duplicate elements). After the automatic preprocessing phase of the code generator, no traces of recursion through sets will be left. This pattern of replacing logical recursion on sets by executable recursion on lists is common in Isabelle's standard libraries.

Furthermore, the heuristic cannot prove all function definitions that are still terminating after class elimination to be terminating. The following circumstances prevent a termination proof to be ported:

- Recursive calls that receive the result of another recursive call as an argument. A classic example is *McCarthy's 91 function* [86], which is defined as follows:

  $$\mathtt{f91}\ n = (\textbf{if}\ 100 < n\ \textbf{then}\ n - 10\ \textbf{else}\ \mathtt{f91}\ (\mathtt{f91}\ (n + 11)))$$

  The termination proof requires an additional lemma that gives an estimate on the return value. Krauss' **function** package can deal with this specification [69, §2.7.2]. My heuristic cannot, because the generated termination condition mentions the function itself.

- Functions with at least one polymorphic parameter that is not a suitable type constructor. The termination heuristic will ask the **datatype** package to deliver a map function from the new relation to the old relation, which is impossible in some cases.

When the heuristic fails, the system falls back to an automatic termination proof using the lexicographic order method. Presently, for technical reasons, it is impossible to give a manual proof when the automatic proof also fails.

4 Terms, patterns, and values

Terms and conditions apply.

(Every business, ever.)

Contents

The compiler transforms the initial type of terms through various stages. This section introduces necessary terminology (§4.1) and gives an overview of the intermediate types (§4.3).

All of them support similar notions such as "free variables", "matching" and "substitution". This is abstracted in a *term algebra* (§4.2).

4.1 Preliminaries

Throughout this chapter, the concept of *mappings* is pervasive. I use the type notation $\alpha \rightharpoonup \beta$ to denote a function $\alpha \Rightarrow \beta$ `option` with a finite domain. In certain contexts, a mapping may also be called an *environment*.

Mapping literals are written using brackets: $[a \mapsto x, b \mapsto y, \ldots]$. If it is clear from the context that σ is defined on a, the lookup $\sigma\ a$ can be treated as returning a value of type β. The pair (a, b) is also called an *entry* of the mapping. A mapping can be constructed from a list or set of entries with the function `map_of`.

The functions `dom` $:: (\alpha \rightharpoonup \beta) \Rightarrow \alpha$ `set` and `range` $:: (\alpha \rightharpoonup \beta) \Rightarrow \beta$ `set` return the *domain* and *range* of a mapping, respectively. In this context, *domain* refers to the set of all keys in a mapping and is not to be confused with the domain of a function (§3.1.5).

Dropping entries from a mapping is denoted by $\sigma - k$, where σ is a mapping and k is either a single key or a set of keys. I use $\sigma' \subseteq \sigma$ to denote that σ' is a sub-mapping of σ, that is, the set of entries of σ' is a subset of the set of entries of σ; formally: `dom` $\sigma' \subseteq$ `dom` σ and $\forall a \in$ `dom` $\sigma'.\ \sigma'\ a = \sigma\ a$.

Adding two mappings σ and ρ is denoted with $\sigma +\!\!+\ \rho$. It constructs a new mapping with the union domain of σ and ρ. Entries from ρ override entries from σ. That is, $\rho \subseteq \sigma +\!\!+\ \rho$ holds, but not necessarily $\sigma \subseteq \sigma +\!\!+\ \rho$.

Finite_Map

All mappings and sets are assumed to be finite. In the formalization, this is enforced by using subtypes of $\rightharpoonup$ and `set`. In fact, nesting recursion through $\rightharpoonup$ and `set` would be unsound [10, 11, 46]. I leverage facilities of Blanchette et al.'s **datatype** command to construct the various term types [18].

The term *abstraction*, when used in the context of terms, refers to a λ-abstraction. An abstraction – in general of the form $\lambda x.\ t$ – *binds* a variable x in t. Variables that are not *bound* are *free*.

4.1.1 Names

Constant and variable names are modelled as `strings`. The full flexibility of that type (i.e. string manipulations) is only required where fresh names are being produced (§6.2.1).

Otherwise, only a linear order on terms is needed, which in turn needs a linear order on names. Conveniently, Sternagel and Thiemann [117] provide tooling to automatically generate such a lexicographic order on types generated by the **datatype** package. It requires existing orderings for all constituent types of a datatype.

In Isabelle, `string` is a synonym for `char list`; i.e., a list of characters. However, there is no default order on lists, as there could be multiple reasonable implementations: e.g. lexicographic and point-wise. For both choices, users can import the corresponding instantiation. In Isabelle, only at most one implementation of a given type class for a given type may be present in the same theory. Consequently, I avoided importing a list ordering from the library, because it may cause conflicts with users who use another ordering.

The general approach for these situations is to introduce a type copy. I have created a `name` type that wraps `strings` and requires no other instances to work. For simplicity, I will use just `string` in this thesis. The ordering on names is a copy of the lexicographic order on lists.

4.1.2 Common functions

All type constructors used in this formalization ($\rightharpoonup$, `set`, `list`, `option`, ...) support the common operations `map` and `rel`. For a type constructor τ with a single type variables α, these two functions have the following types:

$$\begin{aligned}&\mathtt{map}_\tau :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\,\tau \Rightarrow \beta\,\tau\\&\mathtt{rel}_\tau :: (\alpha \Rightarrow \beta \Rightarrow \mathtt{bool}) \Rightarrow \alpha\,\tau \Rightarrow \beta\,\tau \Rightarrow \mathtt{bool}\end{aligned}$$

If the type constructor is obvious from the context, it is omitted.

For parametrized datatypes like lists, $\mathtt{map}_\tau$ is the usual functorial map. The intuition behind $\mathtt{rel}_\tau$ is to lift a binary predicate $P :: \alpha \Rightarrow \beta \Rightarrow \mathtt{bool}$ to the type constructor τ. This lifted relation is the *relator* for a particular type. Its definition is structural, based on the constructors:

$$\frac{}{\mathtt{rel}_{\mathtt{list}}\ P\ []\ []} \qquad \frac{\mathtt{rel}_{\mathtt{list}}\ P\ xs\ ys \qquad P\ x\ y}{\mathtt{rel}_{\mathtt{list}}\ P\ (x\ \#\ xs)\ (y\ \#\ ys)}$$

For other types, the situation is a bit more complicated. The types of $\mathtt{map}_{\mathtt{set}}$ and $\mathtt{rel}_{\mathtt{set}}$ are as expected. Mapping a set applies a function to all elements; an operation usually known as image of a function under a set.

Definition 4.1 (Set relator). *For each element $a \in A$, there must be a corresponding element $b \in B$ such that $P\ a\ b$, and vice versa. Formally:*

$$\mathtt{rel}_{\mathtt{set}}\ P\ A\ B \Longleftrightarrow (\forall x \in A.\ \exists y \in B.\ P\ x\ y) \wedge (\forall y \in B.\ \exists x \in A.\ P\ x\ y)$$

Finally, consider mappings. The map function has the type $(\beta \Rightarrow \gamma) \Rightarrow (\alpha \rightharpoonup \beta) \Rightarrow (\alpha \rightharpoonup \gamma)$. It leaves the domain unchanged, but applies a function to the range of the mapping. In other words, the keys do not participate in this operation.

Definition 4.2 (Mapping relator). *Two maps m_1 and m_2 are related with respect to P if for all values k:*

$$\mathtt{rel}_{\mathtt{option}}\ P\ (m_1\ k)\ (m_2\ k)$$

Corollary 4.3. *Related maps share a common domain. Formally:*

$$\mathtt{rel}_{\rightharpoonup}\ m\ n \Longrightarrow \mathsf{dom}\ m = \mathsf{dom}\ n$$

```
locale simple_syntactic_and =
  fixes P :: α::term ⇒ bool
  assumes P_app: P (app t u) = P t ∧ P u
begin

  lemma list_comb: P (list_comb f xs) = P f ∧ list_all P xs
  (* proof *)

end

interpretation no_abs: simple_syntactic_and no_abs
(* proof *)
```

Listing 4.1: Example locale

4.1.3 Modularity

The formalization relies on two of Isabelle's modularization concepts: *classes* and *locales* [5]. Both of them represent named contexts: they can be used to bundle one or more types together with operations and their properties.

Classes are briefly introduced in §3.1.1, in the context of the dictionary construction. Here, I will focus on them as a mechanism to organize a formalization abstractly.

Locales are a generalization of that concept. They allow arbitrarily many fixed types and support more complex inheritance relationships. In the formalization, this is used extensively to avoid duplication.

An example locale is shown in Listing 4.1. It introduces the locale `simple_syntactic_and` (§4.2.4.3) that fixes a variable and assumes a property. In the body (Isabelle parlance: *locale context*), a lemma is stated; the proof may refer to the locale assumptions. Finally, the locale is *interpreted*, i.e., the variables instantiated and assumptions proved.

Interpretations may occur at various places in Isabelle theories. Notably, they can appear in other locale contexts; then, they are called *sublocale* interpretations. Sublocales do not suffer from the same restriction as subclasses: parameters can be instantiated freely and need not be shared.

4.1.4 Monad syntax

Many operations on terms are partial, i.e., they are not specified on some inputs. To avoid carrying around too many constraints, I have decided to model some as returning `option`s. As a notational convenience, Isabelle provides *monad syntax* that is similar to Haskell's **do** notation [87, §3.14].

Monad_Syntax

In the case of the `option` type constructor, the expression **do** $\{x \longleftarrow f;\ g\ x\}$ is desugared to $\texttt{bind}_{\texttt{option}}\ f\,(\lambda x.\ g\ x)$, where `bind` is defined as follows:

```
fun bind :: α option ⇒ (α ⇒ β option) ⇒ β option
bind None f = None
bind (Some x) f = f x
```

The desugaring process is iterated until all statements in the **do** block are replaced by `bind`.

In other words, a **do** block can be interpreted as a series of **let** expressions that "short-circuit" the expression to `None` if any of the statements evaluates to `None`.

The monad syntax also allows monads other than `option`. In this chapter, only `option` is used; this choice will be revisited in §6.2.

4.2 An algebra for terms

Portions of this section appear in the AFP entry "An Algebra for Higher-Order Terms", authored by Hupel [60].

Terms can be thought of as consisting of a *generic* (free variables, constants, application) and a *specific* part. The generic part is shared, whereas the specific part varies across different term types. This opens the opportunity to use a type class to abstract over these types. In this section, I introduce the `term` class, explain its axioms, and describe the resulting theory.

Term_Class

In Isabelle, classes and types live in different namespaces. The `term` type and the `term` class are separate entities.

4.2.1 Basic term operations

Definition 4.4. *A term type* τ *supports the operations*

$$\begin{aligned} \mathtt{const} &:: \mathtt{string} \Rightarrow \tau & \mathtt{frees} &:: \tau \Rightarrow \mathtt{string\ set} \\ \mathtt{free} &:: \mathtt{string} \Rightarrow \tau & \mathtt{consts} &:: \tau \Rightarrow \mathtt{string\ set} \\ \mathtt{app} &:: \tau \Rightarrow \tau \Rightarrow \tau & \mathtt{subst} &:: (\mathtt{string} \rightharpoonup \tau) \Rightarrow \tau \Rightarrow \tau \\ \mathtt{abs\_pred} &:: (\tau \Rightarrow \mathtt{bool}) \Rightarrow (\tau \Rightarrow \mathtt{bool}) & \mathtt{size} &:: \tau \Rightarrow \mathtt{nat} \end{aligned}$$

The `const`, `free`, and `app` functions are the *generic constructors*, that is, they behave like regular **datatype** constructors, but are polymorphic in the constructed type. I abbreviate `app` t_1 t_2 as $t_1 \,\$\, t_2$. Conversely, there are also three corresponding destructors that can be defined in terms of Hilbert's choice operator:

$$\mathtt{unconst}\ t = (\textbf{if}\ \exists x. t = \mathtt{const}\ x\ \textbf{then}\ \mathtt{Some}\ (\textbf{SOME}\ x'.\ t = \mathtt{const}\ x)\ \textbf{else}\ \mathtt{None})$$

In this formalization, I have instead opted to let instances define destructors directly, which is simpler for execution purposes (see §2.4 for the advantages).

Consequently, the basic class axioms mandate that the three pairs of constructors and destructors behave – except for exhaustiveness – like a *freely constructed* datatype (Listing 4.2)

$$\begin{aligned} \mathsf{app}\ t_1\ t_2 &\neq \mathtt{const}\ x \\ \mathsf{app}\ t_1\ t_2 &\neq \mathtt{free}\ x \\ \mathtt{const}\ y &\neq \mathtt{free}\ x \end{aligned}$$

$$\begin{aligned} \mathtt{free}\ x_1 = \mathtt{free}\ x_2 &\implies x_1 = x_2 \\ \mathtt{const}\ x_1 = \mathtt{const}\ x_2 &\implies x_1 = x_2 \\ \mathsf{app}\ t_1\ t_2 = \mathsf{app}\ u_1\ u_2 &\implies t_1 = u_1 \wedge t_2 = u_2 \end{aligned}$$

(4.2.a) Free constructors

$$\begin{aligned} \mathtt{unfree}\ (\mathtt{free}\ x) &= \mathsf{Some}\ x \\ \mathtt{unconst}\ (\mathtt{const}\ x) &= \mathsf{Some}\ x \\ \mathtt{unapp}\ (\mathsf{app}\ t_1\ t_2) &= \mathsf{Some}\ (t_1, t_2) \end{aligned}$$

$$\begin{aligned} \mathtt{unfree}\ t = \mathsf{Some}\ x &\implies t = \mathtt{free}\ x \\ \mathtt{unconst}\ t = \mathsf{Some}\ x &\implies t = \mathtt{const}\ x \\ \mathtt{unapp}\ t = \mathsf{Some}\ (t_1, t_2) &\implies t = \mathsf{app}\ t_1\ t_2 \end{aligned}$$

(4.2.b) Destructors

$$\begin{aligned} \mathtt{frees}\ (\mathtt{free}\ x) &= \{x\} \\ \mathtt{frees}\ (\mathtt{const}\ x) &= \emptyset \\ \mathtt{frees}\ (\mathsf{app}\ t_1\ t_2) &= \mathtt{frees}\ t_1 \cup \mathtt{frees}\ t_2 \end{aligned} \qquad \begin{aligned} \mathtt{consts}\ (\mathtt{free}\ x) &= \emptyset \\ \mathtt{consts}\ (\mathtt{const}\ x) &= \{x\} \\ \mathtt{consts}\ (\mathsf{app}\ t_1\ t_2) &= \mathtt{consts}\ t_1 \cup \mathtt{consts}\ t_2 \end{aligned}$$

(4.2.c) Partial definitions

$$\mathtt{size}\ (\mathsf{app}\ t_1\ t_2) = 1 + \mathtt{size}\ t_1 + \mathtt{size}\ t_2$$

(4.2.d) Wellfoundedness

Listing 4.2: Axioms for the basic operations of the `term` class

[18, §3]: they are distinct and injective. The *wellfoundedness* axiom is required so that functions over abstract terms that recurse into applications can be proved to be terminating. Most other properties that would otherwise be generated by the **datatype** package follow from these axioms. Also, discriminators can be defined easily by checking if the corresponding destructor returns Some.

The `frees` and `consts` functions select the set of free variables and constants in a term. For constants, abstractions need not be considered, because they only bind variables. The basic axioms strictly prescribe the behaviour of both functions for the generic constructors. For convenience, `closed` t abbreviates `frees` $t = \emptyset$.

Types defined through **datatype** are equipped with a `size` function that counts the number of constructors in the value. `size` itself is a class constant from the `size` class. Consequently, the `term` class extends `size` and assumes that the `size` constant behaves exactly like expected for app.

The `subst` function substitutes free variables in a term. As arguments, it takes a mapping from names to terms and the term to be substituted.

Substituting terms with free variables into a term is underspecified: some axioms (Listing 4.4) require terms to be closed as a precondition. Term types are not required to implement α-renaming to prevent capturing of variables. This is consistent with the assumptions on term rewriting outlined in §2.6.

$$
\begin{aligned}
\texttt{const} &= \texttt{Const} & \texttt{frees}\ (\mathsf{Abs}\ t) &= \texttt{frees}\ t \\
\texttt{free} &= \texttt{Free} & \texttt{frees}\ (\mathsf{Bound}\ n) &= \emptyset \\
\texttt{app} &= \texttt{App} & \texttt{subst}\ \sigma\ (\mathsf{Abs}\ t) &= \mathsf{Abs}\ (\texttt{subst}\ \sigma\ t) \\
\texttt{consts}\ (\mathsf{Abs}\ t) &= \texttt{consts}\ t & \texttt{subst}\ \sigma\ (\mathsf{Bound}\ n) &= \mathsf{Bound}\ n \\
\texttt{consts}\ (\mathsf{Bound}\ n) &= \emptyset
\end{aligned}
$$

$$\texttt{abs\_pred}\ P\ t = ((\forall n.\ t = \mathsf{Bound}\ n \implies P\ t) \land (\forall t'.\ t = \mathsf{Abs}\ t' \implies P\ t' \implies P\ t))$$

Listing 4.3: Implementation of the `term` operations for the `term` type

Finally, `abs_pred` is a partial induction predicate, covering the remaining cases not covered by `const`, `free` and `app`. The reasoning behind this and the necessary axiom is explained in §4.2.3.

A sample implementation for the initial term type `term` (Listing 2.4.a) is given in Listing 4.3. For brevity, the cases that are fully specified by the axioms are omitted.

4.2.2 Matching

Taking only the above definitions already admits a generic definition of *matching* a *pattern* with a term. Importantly, the type of patterns is neither generic, nor a dedicated pattern type; instead, it is `term` itself. I chose this representation for two reasons:

1. Patterns are a proper subset of terms, with the restriction that no abstractions may occur and there must be at most a single occurrence of any variable (usually known as *linearity*). The first restriction can be modelled in a datatype, the second cannot. Hence, I define a predicate `linear :: term ⇒ bool` that captures both properties.
2. The logical requirements of patterns do not change throughout – including the CakeML backend (§6.8) – the formalization. In particular, it is never allowed to match against an abstraction. Consequently, it is not necessary to change types midway through the compilation pipeline.

The above reasoning notwithstanding, a dedicated `pat` type is present in the formalization and will be motivated and described in §4.3.6.

The linearity constraint is represented as a recursive function on `term`s:

```
fun linear :: term ⇒ bool where
linear (Free _) ⟺ True
linear (Const _) ⟺ True
linear (t₁ $ t₂) ⟺ linear t₁ ∧ linear t₂ ∧ disjoint (frees t₁) (frees t₂) ∧ ¬ is_free t₁
linear _ ⟺ False
```

Equipped with this, it becomes possible to define a generic matching function using monad syntax (§4.1.4):

```
fun match :: term ⇒ α ⇒ (string ⇀ α) option where
match (Const x) t = do {
  y ← unconst t
  if x = y then Some [] else None
}
match (t1 $ t2) u = do {
  (u1, u2) ← unapp u
  σ1 ← match t1 u1
  σ2 ← match t2 u2
  Some (σ1 ++ σ2)
}
match (Free s) t = Some [s ↦ t]
match _ _ = None
```

The first argument denotes the pattern, the second the *object*. Matching yields an optional mapping of type `string` ⇀ α from free variable names to terms. The object is traversed according to the destructors as defined by a concrete implementation of the `term` type. In particular, it is only necessary to distinguish between constants, applications, and "everything else". Taking free variables into account would enable defining a generic unification algorithm, which is not needed here, but would be an opportunity for future work.

There are two notable deviations from term rewriting literature:

1. The result of matching is not directly a substitution $\sigma :: \alpha \Longrightarrow \alpha$, but rather a mapping that needs to be applied by a substitution function.
2. When joining the sub-results of matching a function application, there is no check that they are consistent. Luckily, consistency is a consequence of linearity. The result of matching any object against a non-linear pattern is unspecified; but no target language of the code generator even supports it.

Corollary 4.5 (Domain of matching). *If the object matches a pattern t with environment σ, then* dom σ = `frees` *t.*

Naturally, the definitions of `linear` and `match` can be lifted to a list of terms:

```
definition freess :: α list ⇒ string set where
freess ts = ⋃ (set (map frees ts))

fun linears :: term list ⇒ bool where
linears [] ⟺ True
linears (t # ts) ⟺ linear t ∧ fdisjoint (frees t) (freess ts) ∧ linears ts

fun matchs :: term list ⇒ α list ⇒ (string ⇀ α) option where
matchs [] [] = Some []
matchs (t # ts) (u # us) = do {
  σ1 ← match t u
  σ2 ← matchs ts us
```

$$
\begin{aligned}
\texttt{subst}\ \sigma\,(\texttt{const}\ x) &= \texttt{const}\ x \\
\texttt{subst}\ \sigma\,(\texttt{free}\ x) &= (\textbf{if}\ x \in \mathsf{dom}\ \sigma\ \textbf{then}\ \sigma\, x\ \textbf{else}\ \texttt{free}\ x) \\
\texttt{subst}\ \sigma\,(t_1\ \$\ t_2) &= \texttt{subst}\ \sigma\, t_1\ \$\ \texttt{subst}\ \sigma\, t_2
\end{aligned}
$$

(4.4.a) Partial definition

$$\texttt{id\_env}\ \sigma \implies \texttt{subst}\ \sigma\, t = t \tag{4.6}$$

$$x \notin \texttt{frees}\ t \implies \texttt{subst}\ (\sigma - x)\ t = \texttt{subst}\ \sigma\, t \tag{4.7}$$

$$\texttt{closed}\ \sigma_2 \wedge \texttt{disjoint}\ \sigma_1\ \sigma_2 \implies \texttt{subst}\ (\sigma_1 +\!\!+\ \sigma_2)\ t = \texttt{subst}\ \sigma_1\ (\texttt{subst}\ \sigma_2\ t) \tag{4.8}$$

$$\texttt{closed}\ \sigma \implies \texttt{frees}\ (\texttt{subst}\ \sigma\, t) = \texttt{frees}\ t - \mathsf{dom}\ \sigma \tag{4.9}$$

$$\texttt{consts}\ (\texttt{subst}\ \sigma\, t) = \texttt{consts}\ t \cup \bigcup_{\substack{x \in \texttt{frees}\ t \\ x \in \mathsf{dom}\ \sigma}} \texttt{consts}\ (\sigma\, x) \tag{4.10}$$

where:

$$
\begin{aligned}
\texttt{id\_env}\ \sigma &= (\forall x\, t.\ \sigma\, x = \mathsf{Some}\ t \implies t = \texttt{free}\ x) \\
\texttt{closed}\ \sigma &= (\forall t \in \mathsf{range}\ \sigma.\ \texttt{frees}\ t = \emptyset) \\
\texttt{disjoint}\ \sigma_1\ \sigma_2 &= \texttt{disjoint}\ (\mathsf{dom}\ \sigma_1)\ (\mathsf{dom}\ \sigma_2)
\end{aligned}
$$

(4.4.b) Non-trivial properties

Listing 4.4: Axioms for the `subst` operation

```
  Some (σ1 ++ σ2)
}
matchs _ _ = None
```

An obvious consequence is that successful matching of two lists requires that both lists have the same length.

4.2.3 Substitution axioms

In contrast to matching, substitution cannot solely be defined in terms of the generic constructors of a term type, because the `term` class provides no combinators to work with abstractions. Instead, Listing 4.4 provides two sets of axioms: partial definition for the constructor cases, and additional, non-trivial properties that any implementation must adhere to.

To make proofs of the latter simpler, it is only required to prove them for the non-generic cases. This works by requiring instantiations to provide the higher-order predicate `abs_pred`, together with the following axiom:

$$\frac{\forall x.\ P\,(\mathtt{free}\ x) \qquad \forall t_1\ t_2.\ P\,t_1 \implies P\,t_2 \implies P\,(t_1\ \$\ t_2) \qquad \forall x.\ P\,(\mathtt{const}\ x) \qquad \forall t.\ \mathtt{abs\_pred}\ P\,t}{P\,t} \tag{4.11}$$

This axiom resembles an induction schema. It also explains why abs_pred was introduced earlier as a partial induction predicate: it is used to establish exhaustiveness of the generic and specific parts of the concrete term type. Compare this to the induction principle of term:

$$\frac{\forall t_1\ t_2.\ P\,t_1 \implies P\,t_2 \implies P\,(\mathtt{App}\ t_1\ t_2) \qquad \forall x.\ P\,(\mathtt{Const}\ x) \qquad \forall t.\ P\,t \implies P\,(\mathtt{Abs}\ t) \qquad \forall x.\ P\,(\mathtt{Free}\ x) \qquad \forall x.\ P\,(\mathtt{Bound}\ x)}{P\,t}$$

The abs_pred predicate covers exactly the cases for Abs and Bound.

Interestingly, this axiom enables fully generic induction proofs on abstract terms. The principle can be illustrated with an example property from Listing 4.4.b. The actual class axiom is not Axiom 4.6:

$$\mathtt{id\_env}\ \sigma \implies \mathtt{subst}\ \sigma\,t = t$$

but rather the more technical

$$\mathtt{abs\_pred}\ (\lambda t.\ \forall\sigma.\ \mathtt{id\_env}\ \sigma \implies \mathtt{subst}\ t\,\sigma = t)\ t$$

where the desired property is explicitly quantified and wrapped into abs_pred. This is similar for the other axioms: all of them are phrased in terms of abs_pred. The desired properties as given in Listing 4.4.b can then be derived by induction using the above schema (Axiom 4.11), together with the partial definitions (Listing 4.4.a). The notable exception is Axiom 4.8, which requires more background theory and will be revisited later (§4.2.4.4).

It remains to explain how the technical abs_pred axioms are proved in the instantiations. In the case of term :: term, they can be easily discharged by simplification, because there, substitution needs not deal with bound variable names (Listing 4.3). For other term types, these properties require more complicated proofs, which will be explained in §4.3.

4.2.4 Derived operations & theory

Based on the operations specified in the term class, a wide range of polymorphic functions can be defined.

4.2.4.1 Combinators

To simplify definition of functions performing a case analysis on terms, a **case** combinator is useful. Its structure follows **case** combinators generated by the **datatype** package; however, it has to ensure exhaustiveness by taking an additional default:

```
definition term_cases ::
  (string ⟹ β) ⟹ (string ⟹ β) ⟹ (α ⟹ α ⟹ β) ⟹ β ⟹ α ⟹ β where
term_cases if_const if_free if_app otherwise t =
  (case unconst t of
    Some name ⟹ if_const name |
    None ⟹ (case unfree t of
      Some name ⟹ if_free name |
      None ⟹
        (case unapp t of
          Some (t, u) ⟹ if_app t u
        | None ⟹ otherwise)))
```

Equipped with this function and corresponding congruence rule (§3.1.5.1), it becomes possible to easily define a conversion functions between arbitrary term types:

```
fun convert_term :: α ⟹ β where
convert_term t = term_cases const free (λt u. app (to_term t) (to_term u)) undefined t
```

This function is partially specified (in the sense of §3.1.5). Its domain can be characterized by the following function:

```
fun no_abs :: α ⟹ bool where
no_abs t = term_cases (λ_. True) (λ_. True) (λt u. no_abs t ∧ no_abs u) False t
```

Thanks to the wellfoundedness axiom (Listing 4.2.d), termination of such functions can be proved automatically.

The no_abs function is a misnomer: judging by the name, it should be true for a term *t* that does not contain an abstraction. But for the term instantiation, it is also false if the term contains a bound variable.

Lemma 4.12. *The image of* convert_term *over its domain is its domain.*

Lemma 4.13. *Assuming the arguments are in its domain,* convert_term *is*

- *idempotent,*
- *injective, and*
- *for all type instantiations $\alpha = \beta$, the identity function.*

Proof. Injectivity follows from the injectivity of the generic constructors (Listing 4.2.a). □

Lemma 4.14 (Preservation of matches)**.**

$$\mathtt{match}\ t\ u = \mathtt{Some}\ \sigma \implies \mathtt{match}\ t\ (\mathtt{convert\_term}\ u) = \mathtt{Some}\ (\mathtt{map}\ \mathtt{convert\_term}\ \sigma)$$

Term conversions that are restricted to terms not containing abstractions becomes useful in parts of the compiler with complicated correspondence relations between terms. It can be used to show that on terms without abstractions, they collapse to convert_term (or equality, e.g. §6.3.3).

4.2.4.2 Applicative term structure

Function applications with multiple arguments can be expressed as repeated application with a single argument. Operations to create and destructure those are named according to Isabelle terminology:

```
fun list_comb :: α ⇒ α list ⇒ α where
list_comb f [] = f
list_comb f (t # ts) = list_comb (app f t) ts

fun strip_comb :: α ⇒ α × α list where
strip_comb t =
  (case unapp t of
    Some (t, u) ⇒ (let (f, args) = strip_comb t in (f, args @ [u]))
  | None ⇒ (t, []))
```

The most common variant of list_comb – a constant is used as function – is abbreviated with $$, i.e., *f* $$ *xs* = list_comb (const *f*) *xs*. Left-hand sides of defining equations are structured in such a way, where *xs* are patterns.

Lemma 4.15. strip_comb *and* list_comb *are inverses:*

$$\texttt{list\_comb}\,(\texttt{fst}\,(\texttt{strip\_comb}\,t))\,(\texttt{snd}\,(\texttt{strip\_comb}\,t)) = t$$
$$\texttt{strip\_comb}\,(\texttt{list\_comb}\,f\,ys) = (\texttt{fst}\,(\texttt{strip\_comb}\,f), \texttt{snd}\,(\texttt{strip\_comb}\,f)\,@\,ys)$$
$$\texttt{strip\_comb}\,(f\,\$\$\,ys) = (\texttt{const}\,f, ys)$$

Observe that list_comb produces applications that are nested to the left. With the following definition, it is possible to prove that list_comb is injective in its second argument:

Definition 4.16 (Left nesting). *A term has a* left nesting *of n if* None *is reached after n recursive applications of* unapp, *proceeding with the left branch. Formally:*

$$\texttt{left\_nesting}\,t = \texttt{term\_cases}\,(\lambda\_.\,0)\,(\lambda\_.\,0)\,(\lambda t\,u.\,1 + \texttt{left\_nesting}\,t)\,0\,t$$

Lemma 4.17 (Conditional injectivity). *Assuming that* left_nesting *f* = left_nesting *g and* list_comb *f xs* = list_comb *g ys, then f* = *g and xs* = *ys.*

Proof. By induction on *xs*, generalizing all other variables. Injectivity follows from Listing 4.2.a. ☐

Corollary 4.18 (Injectivity). *The (partially applied) function* list_comb *f is injective.*

Corollary 4.19. *If xs and ys have the same length and* list_comb *f xs* = list_comb *g ys, then f* = *g and xs* = *ys.*

Of course, it is possible to establish a connection between match and matchs using list_comb:

Lemma 4.20.

$$\texttt{match}\,(name\,\$\$\,xs)\,(name\,\$\$\,ys) = \texttt{matchs}\,xs\,ys$$

This lemma is a consequence of a more general statement for arbitrary functions, which is too technical to be reproduced here. Observe that this lemma requires the instantiation term :: term, because it uses list_comb as a pattern. But like the other results on list_comb, it is polymorphic (here: in *ys* :: *α* :: term).

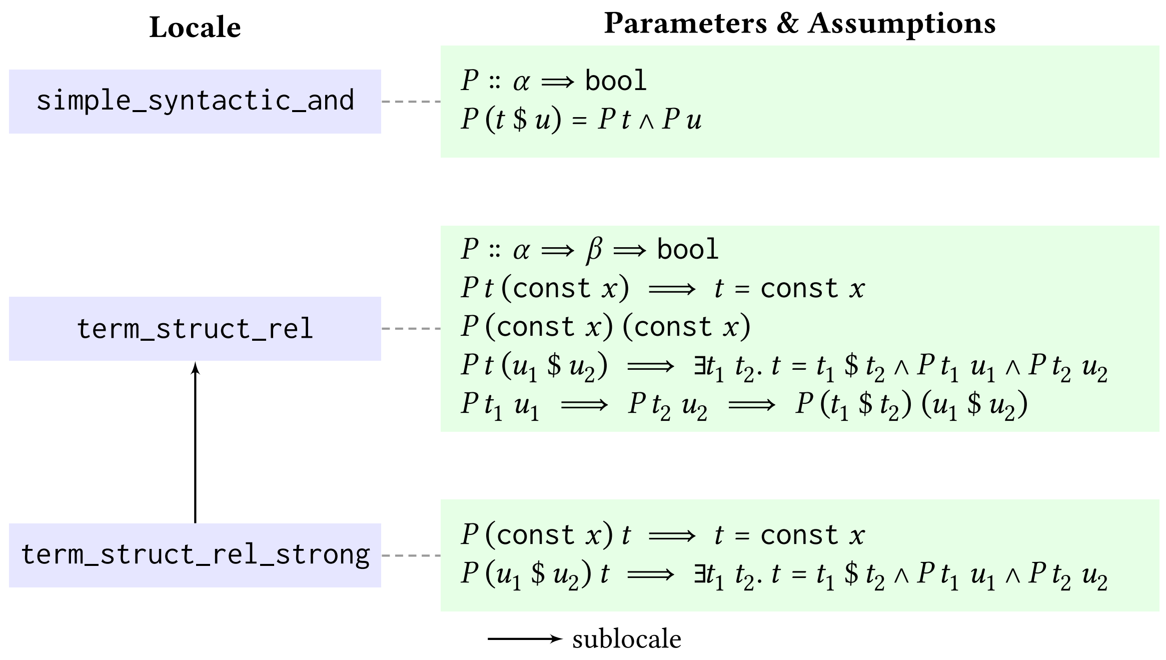

Locale
Parameters & Assumptions
simple_syntactic_and
$P :: \alpha \Longrightarrow \mathsf{bool}$
$P\,(t\ \$\ u) = P\,t \wedge P\,u$
term_struct_rel
$P :: \alpha \Longrightarrow \beta \Longrightarrow \mathsf{bool}$
$P\,t\,(\mathsf{const}\ x) \implies t = \mathsf{const}\ x$
$P\,(\mathsf{const}\ x)\,(\mathsf{const}\ x)$
$P\,t\,(u_1\ \$\ u_2) \implies \exists t_1\ t_2.\ t = t_1\ \$\ t_2 \wedge P\,t_1\ u_1 \wedge P\,t_2\ u_2$
$P\,t_1\ u_1 \implies P\,t_2\ u_2 \implies P\,(t_1\ \$\ t_2)\,(u_1\ \$\ u_2)$
term_struct_rel_strong
$P\,(\mathsf{const}\ x)\,t \implies t = \mathsf{const}\ x$
$P\,(u_1\ \$\ u_2)\,t \implies \exists t_1\ t_2.\ t = t_1\ \$\ t_2 \wedge P\,t_1\ u_1 \wedge P\,t_2\ u_2$
sublocale

Figure 4.1: Syntactic locales on terms

4.2.4.3 Syntactic predicates and relations

Predicates and relations on term types that can follow a simple syntactic structure admit lifting of theorems through `match`. Figure 4.1 gives an overview over the locales and their relationships.

Corollary 4.21. *The predicates* `closed` *and* `no_abs` *implement* `simple_syntactic_and`.

Lemma 4.22. *For all predicates P implementing* `simple_syntactic_and`*: If an object t for which $P\,t$ holds matches a pattern with resulting environment σ, then P holds for the range of σ.*

Proof. By induction on the termination relation of `match`. The object gets decomposed only according to `unapp`. If P holds for the entire object, it also holds for all such sub-terms. □

The assumptions of `term_struct_rel` are right-biased: it means that the structure of the second argument determines the structure of the first argument. The locale `term_struct_rel_strong` strengthens this to work symmetrically.

Lemma 4.23. *For all relations P implementing* `term_struct_rel`*: If $P\,t_1\ t_2$ and matching t_2 against a pattern p yields an environment σ_2, then matching t_1 against p yields an environment σ_1 where* `rel` *$P\,\sigma_1\ \sigma_2$.*

The statement becomes stronger when assuming the symmetric locale:

Lemma 4.24. *For all relations P implementing* `term_struct_rel_strong`*: If $P\ t_1\ t_2$, then* `rel` (`rel` P) (`match` $p\,t_1$) (`match` $p\,t_2$).

Unfortunately, no general relational properties about `subst` can be proved, because neither the locales nor the `term` class assume any behaviour on terms with abstractions. A counterexample would be a predicate P ensuring that there are no nested abstractions. A term containing an abstraction could be substituted below an abstraction in another term, creating a term that contains nested abstractions. Proving a restricted result on terms that contain no abstractions however is possible, but not useful enough.

4.2.4.4 Substitution

The lack of general relational results about substitution notwithstanding (§4.2.4.3), it is possible to prove some equational properties of substitution.

Corollary 4.25 (Substitution of non-occurring variables)**.** *If* $\texttt{frees}\ t$ *and* $\texttt{dom}\ \sigma$ *are disjoint, then* $\texttt{subst}\ \sigma\ t = t$*.*

Proof. Iterate Axiom 4.7 until the empty environment is reached. `id_env` [] obviously holds, hence $\texttt{subst}\ \sigma\ t = t$ with Axiom 4.6. □

Corollary 4.26 (Substitution of a closed term)**.** *Substituting any environment into a closed term yields the same term.*

The above results are properties on substitutions that ignore variables that occur freely in a term, leaving the term unchanged. Conversely, it is also possible to restrict substitutions to the set of free variables, having the same result as substituting the full environment. $\texttt{restrict}\ S\ \sigma$ restricts the domain of a map σ to the set S, i.e., mappings outside of S are dropped.

Lemma 4.27 (Substitution of restricted environment)**.** *Let* $\sigma' = \texttt{restrict}\ (\texttt{frees}\ t)\ \sigma$*. Then,* $\texttt{subst}\ \sigma'\ t = \texttt{subst}\ \sigma\ t$*.*

Proof. Clearly, the variables dropped from σ are not in $\texttt{frees}\ t$. In other words, $\sigma' = \sigma - (\texttt{dom}\ \sigma - \texttt{frees}\ t)$. Iterate Axiom 4.7 until σ' is reached. □

Corollary 4.28 (Substitution congruence)**.** *If* σ_1 *and* σ_2 *agree on all free variables in* t*, then* $\texttt{subst}\ \sigma_1\ t = \texttt{subst}\ \sigma_2\ t$*. Formally:*

$$\forall x \in \texttt{frees}\ t.\ \sigma_1\ x = \sigma_2\ x \implies \texttt{subst}\ \sigma_1\ t = \texttt{subst}\ \sigma_2\ t$$

Recall Axiom 4.8:

$$\texttt{closed}\ \sigma_2 \wedge \texttt{disjoint}\ \sigma_1\ \sigma_2 \implies \texttt{subst}\ (\sigma_1 \mathbin{+\!\!+} \sigma_2)\ t = \texttt{subst}\ \sigma_1\ (\texttt{subst}\ \sigma_2\ t)$$

It is now possible to prove this, assuming only the `abs_pred` case that needs to be proved individually for all term instantiations:

Proof. By induction using Axiom 4.11.

Const/App Structural using partial definition axioms (Listing 4.4.a).

Free Let $t = \texttt{free}\ x$. Proof by case distinction on $x \in \texttt{dom}\ \sigma_2$.

- If $x \in \texttt{dom}\ \sigma_2$, then let $u = \sigma\ x$. But because σ_2 is closed, so is u. With Corollary 4.26, obtain $u = \texttt{subst}\ \sigma_1\ u$.
- Otherwise, show that $\texttt{subst}\ (\sigma_1 +\!\!+\ \sigma_2)\ t = \texttt{subst}\ \sigma_1\ t$. This follows directly from another case distinction on $x \in \texttt{dom}\ \sigma_1$. □

In a second step, the assumption that both environments are disjoint can be discharged:

Lemma 4.29 (Substitution independence).

$$\texttt{closed}\ \sigma_2 \implies \texttt{subst}\ (\sigma_1 +\!\!+\ \sigma_2)\ t = \texttt{subst}\ \sigma_1\ (\texttt{subst}\ \sigma_2\ t)$$

Proof.

$$\begin{aligned}
&\texttt{subst}\ (\sigma_1 +\!\!+\ \sigma_2)\ t \\
&= \texttt{subst}\ (\texttt{restrict}\ (\texttt{frees}\ t)\ (\sigma_1 +\!\!+\ \sigma_2))\ t && \text{(Lemma 4.27)} \\
&= \texttt{subst}\ (\texttt{restrict}\ (\texttt{frees}\ t)\ \sigma_1 +\!\!+\ \sigma_2)\ t && \text{(Corollary 4.28)} \\
&= \texttt{subst}\ (\texttt{restrict}\ (\texttt{frees}\ t - \texttt{dom}\ \sigma_2)\ \sigma_1 +\!\!+\ \sigma_2)\ t && \text{(Corollary 4.28)} \\
&= \texttt{subst}\ (\texttt{restrict}\ (\texttt{frees}\ t - \texttt{dom}\ \sigma_2)\ \sigma_1)\ (\texttt{subst}\ \sigma_2\ t) && \text{(Axiom 4.8)}^{\dagger} \\
&= \texttt{subst}\ (\texttt{restrict}\ (\texttt{frees}\ (\texttt{subst}\ \sigma_2\ t))\ \sigma_1)\ (\texttt{subst}\ \sigma_2\ t) && \text{(Axiom 4.9)} \\
&= \texttt{subst}\ \sigma_1\ (\texttt{subst}\ \sigma_2\ t) && \text{(Lemma 4.27)}
\end{aligned}$$

Applying the independence axiom requires proving its assumption (†): I need to show that $\texttt{restrict}\ (\texttt{frees}\ t - \texttt{dom}\ \sigma_2)\ \sigma_1$ and σ_2 are disjoint. This follows because σ_1 is restricted to a set that does not contain the domain of σ_2. □

The reason why it is kept as an assumption in Axiom 4.8 is that it makes the instantiation proofs simpler.

Finally, it is possible to prove abstractly that matching and substitution behave as expected:

- If an object matches a pattern, substituting the resulting environment into the pattern should yield the object.
- Otherwise, there is no substitution that yields the object.

In this formalization, the naive translation of this proposition into Isabelle type checks, but is not as general as expected:

$$\texttt{match}\ t\ u = \texttt{Some}\ \sigma \implies \texttt{subst}\ t\ \sigma = u$$

Because of the premise, the pattern t is fixed to be of type `term` and because of the conclusion, t and u must have the same type. But the desired property should work for all term types, i.e. $u :: \alpha :: \texttt{term}$. The key idea is to use `convert_term` to transfer the pattern into an arbitrary term type. Additionally, both parts of the correctness property must be stated simultaneously; otherwise the induction is not general enough.

Lemma 4.30 (Correctness of matching and substitution). *Let t :: `term` be a linear pattern. The following property holds:*

$$\begin{aligned}&\textbf{case}\ \texttt{match}\ t\ u\ \textbf{of}\\&\quad \texttt{None} \Longrightarrow \forall\sigma.\ \texttt{subst}\ \sigma\ (\texttt{convert\_term}\ t) \neq u\\&\quad \texttt{Some}\ \sigma \Longrightarrow \texttt{subst}\ \sigma\ (\texttt{convert\_term}\ t) = u\end{aligned}$$

Proof. By induction on t, generalizing u. Note that because t is linear, it falls within the domain of `convert_term`.

Const/Free Trivial.

Abs If t is an abstraction, then t cannot be linear. Contradiction.

App Let $t = t_1 \$ t_2$. Hence, $t' = \texttt{subst}\ \sigma\ (\texttt{convert\_term}\ t)$ is an application for arbitrary σ. This case can be proved by a series of case distinctions.

- If u is not an application, then matching fails. But because t' is always an application, $t' \neq u$.
- Otherwise, $u = u_1 \$ u_2$.

  If t_1 and u_1 do not match, then there is no σ_1 that transforms t_1 into u_1, according to the induction hypothesis. With injectivity (Listing 4.2.a), there can be no σ that transforms t into u.

  Similarly for the case when t_2 and u_2 do not match.
- Otherwise, t_1 matches u_1 with σ_1 and t_2 matches u_2 with σ_2. I know from the induction hypotheses that $\texttt{subst}\ \sigma_i\ (\texttt{convert\_term}\ t_i) = u_i$. It remains to show that the environment resulting from matching t and u, namely $\sigma = \sigma_1 +\!\!+\ \sigma_2$, transforms t into u. From Corollary 4.5, I know that $\texttt{dom}\ \sigma_i = \texttt{frees}\ t_i$. Consequently, $\texttt{restrict}\ (\texttt{frees}\ t_i)\ \sigma = \sigma_i$. With Lemma 4.27, the substitution of t_i can be lifted from σ_i to σ. □

Recall Definition 3.3 (overlapping patterns). It was defined on a high level: two patterns and overlap if there is an object that is matched by both. In fact, this is not how it is defined in the formalization, but it is a consequence of the actual definition. The underlying primitives are defined as follows:

```
definition matches :: α ⇒ α ⇒ bool (infix ≲ 50) where
t ≲ u ⟷ (∃σ. subst σ t = u)

definition overlapping :: α ⇒ α ⇒ bool where
overlapping s t ⟷ (∃u. s ≲ u ∧ t ≲ u)
```

Observe that the `match` function does not appear there. The advantage of that phrasing is that it works for arbitrary term types. The desired definition of overlapping patterns now follows as a corollary:

Corollary 4.31 (Overlapping patterns). *If two linear patterns both match the same object, then they overlap.*

Proof. First, let p_1 and p_2 be linear patterns and u be the object; assume that matching with p_1 and p_2 yields the environments σ_1 and σ_2. Observe that u is polymorphic, and so are σ_1 and σ_2. Define $\sigma_i' = \texttt{map convert\_term}\ \sigma_i$, where the target type of the conversion is `term`. Using Lemma 4.14, obtain that `match` p_i (`convert_term` u) = `Some` σ_i'. The conclusion follows from Lemma 4.30. ☐

Corollary 4.32. *If two linear and compatible patterns (Definition 3.4) both match the same object, then they are equal to each other.*

Proof. Corollary 4.31 establishes that the two patterns overlap. But since they are compatible by assumption, Lemma 3.6 implies that they are equal to each other. ☐

4.2.4.5 Matching and rewriting

The matching operation can be lifted to a list of pairs of patterns and arbitrary data. If the data are terms – which is usually the case –, this list is referred to as *clauses*. The first matching pattern is selected.

```
fun find_match :: (term × α) list ⇒ α ⇒ ((string ⇀ α) × term × α) option where
find_match [] _ = None
find_match ((pat, rhs) # cs) t =
  (case match pat t of Some σ ⇒ Some (σ, pat, rhs) | None ⇒ find_match cs t)
```

Results from §4.2.4.3, in particular Lemma 4.24, can be lifted from `match` to `find_match`.

A particularly noteworthy consequence of Corollary 4.32 is that the result of `find_match` is uniquely determined:

Lemma 4.33. *Let cs be clauses such that:*

- *all patterns are mutually compatible,*
- *all patterns are linear, and*
- *there are no two clauses with equal pattern but different terms.*

Furthermore, let $(p, u) \in cs$ *be a clause. If t matches p yielding environment* σ*, then* `find_match` *returns* (σ, p, u).

A single rewrite step comprises matching a term t_1 against a pattern p, producing an environment σ, and subsequent substitution of a term t_2 with that environment. This notion occurs frequently in the formalization.

Recall §4.2.4.2. There, operations to split and combine terms and patterns were introduced. This can be used to define *equations* in the context of term rewriting:

Definition 4.34. *An* equation *is a pair of a pattern* (left-hand side) *and a term* (right-hand side)*, where the pattern is of the form* $f \,\$\$\, ps$. f *is referred to as the* head *of the equation.*

Following term rewriting terminology, I sometimes refer to an equation as *rule*, and a collection thereof as *rules* or *rule set*. Mostly, it is assumed that the left-hand side of a rule is linear.

Note that this definition of equation is largely similar to the kind of equations other tools in Isabelle expect (i.e., the **function** package and the code generator, §2.4).

In a particular compiler phase (§6.3), equations need to be transformed according to the structure of their left-hand sides. For that, it is necessary to split those into a pair of head and patterns. In this section, I will now establish a connection between pattern compatibility and the match operation lifted to lists (§4.2.2).

Recall the informal definition of pattern compatibility (Definition 3.4). Formally, it is defined as follows:

```
fun pattern_compatible :: term ⇒ term ⇒ bool where
pattern_compatible (t₁ $ t₂) (u₁ $ u₂) ⟷
  pattern_compatible t₁ u₁ ∧ (t₁ = u₁  ⟹  pattern_compatible t₂ u₂)
pattern_compatible t u ⟷
  t = u ∨ ¬ overlapping t u
```

Note that pattern compatibility could logically be defined on arbitrary term types. However, it is only ever used on `term`; consequently, I have restricted it to `term`.

This notion can also be lifted to lists, using a predicate combinator:

$$\frac{}{\texttt{rev\_accum\_rel}\ R\ []\ []} \qquad \frac{xs = ys \Longrightarrow R\ x\ y \qquad \texttt{rev\_accum\_rel}\ R\ xs\ ys}{\texttt{rev\_accum\_rel}\ R\ (xs\ @\ [x])\ (ys\ @\ [y])}$$

```
definition patterns_compatible :: term list ⇒ term list ⇒ bool where
patterns_compatible = rev_accum_rel pattern_compatible
```

This definition is admittedly non-obvious. It can be justified by the structure of `list_comb` and pattern compatibility. In fact, I have defined it in such a way that the following lemma is provable:

Lemma 4.35.

$$\frac{\texttt{patterns\_compatible}\ xs\ ys \qquad \texttt{pattern\_compatible}\ f\ g}{\texttt{pattern\_compatible}\ (\texttt{list\_comb}\ f\ xs)\ (\texttt{list\_comb}\ g\ ys)}$$

The proof works by rule induction on `rev_accum_rel` and requires Corollary 4.19.

Finally, the main result can be proved:

Corollary 4.36. *Let ts_1 and ts_2 two linear and compatible lists of terms. If* `matchs` $ts_i\ us$ = `Some` σ_i *for* $i \in \{1, 2\}$, *then* $ts_1 = ts_2$.

Proof. This can be proved using dummy heads. Let *name* be an arbitrary string that will serve as a dummy head. Using Lemma 4.20, establish that `match` $(name \$\$ ts_i)\ (name \$\$ us) =$ `Some` σ_i. Also, from assumptions, deduce that ts_1 and ts_2 have the same length. With Lemma 4.35, obtain that $name\ \$\$\ ts_1$ and $name\ \$\$\ ts_2$ are compatible.

Because these two patterns are compatible, linear, and match the same object, I can use Corollary 4.32 to show that they must be equal to each other. It remains to show that $ts_1 = ts_2$. This follows from injectivity of $\lambda ts.\ name\ \$\$\ ts$ (Corollary 4.18). □

```
datatype nterm =
  Nconst string |
  Nvar string |
  Nabs string nterm |
  Napp nterm nterm
```

(4.5.a) Explicit bound variable names

```
datatype pterm =
  Pconst string |
  Pvar string |
  Pabs ((term × pterm) set) |
  Papp pterm pterm
```

(4.5.b) Explicit pattern matching

```
datatype sterm =
  Sconst string |
  Svar string |
  Sabs ((term × sterm) list) |
  Sapp sterm sterm
```

(4.5.c) Ordered clauses

```
datatype value =
  Vconstr string (value list) |
  Vabs ((term × sterm) list)
       (string ⇀ value) |
  Vrecabs (string ⇀ ((term × sterm) list))
          string
          (string ⇀ value)
```

(4.5.d) Values

Listing 4.5: Intermediate term types

4.3 Term types

Portions of this section are based on the publications "A Verified Compiler from Isabelle/HOL to CakeML", authored by Hupel and Nipkow [65], and the AFP entry "An Algebra for Higher-Order Terms", authored by Hupel [60].

The original `term` type was already given in Listing 2.4.a. The intermediate types are depicted in Listing 4.5. In this section, I will briefly describe them and explain how they differ from each other. The proofs of the term algebra laws (§4.2) are mostly technical, so I omit them here.

4.3.1 de Bruijn terms (`term`)

The definition of `term` is almost an exact copy of Isabelle's internal term type, with the notable omissions of type information (§2.6). The implementation of β-reduction is straightforward via index shifting of bound variables (Listing 4.6). Substitution of free variables does not require any additional assumptions, because capture is impossible. The full instantiation of the `term` class has already been given in Listing 4.3.

4.3.2 Explicit names (`nterm`)

The `nterm` type is similar to `term`, but removes the distinction between *bound* and *free* variables (Listing 4.5.a). Instead, there are only named variables. To avoid capture during substitution, there is an additional constraint that only closed terms may be substituted into terms. This is reflected in the precondition of Axiom 4.9 of the `term` class.

```
definition shift_nat :: nat ⇒ int ⇒ nat where
shift_nat n k = if k ≥ 0 then n + nat k else n - nat |k|

fun incr_bounds :: int ⇒ nat ⇒ term ⇒ term where
incr_bounds inc lev (Bound i) =
  (if i ≥ lev then Bound (shift_nat i inc) else Bound i)
incr_bounds inc lev (Λ u) = Λ incr_bounds inc (lev + 1) u
incr_bounds inc lev (t₁ $ t₂) = incr_bounds inc lev t₁ $ incr_bounds inc lev t₂
incr_bounds _ _ t = t

fun replace_bound :: nat ⇒ term ⇒ term ⇒ term where
replace_bound lev (Bound i) t =
  if i < lev then Bound i
  else if i = lev then incr_bounds (int lev) 0 t
  else Bound (i - 1)
replace_bound lev (t₁ $ t₂) t = replace_bound lev t₁ t $ replace_bound lev t₂ t
replace_bound lev (Λ u) t = Λ replace_bound (lev + 1) u t
replace_bound _ t _ = t

abbreviation β_reduce :: term ⇒ term ⇒ term (_ [_]) where
t [u] ≡ replace_bound 0 t u
```

Listing 4.6: β-reduction on terms using index shifting

Looking at the bigger picture, this is reasonable, because substitutions are only performed with closed terms. However, for proofs, it might still be necessary, so appropriate preconditions need to control when substitution is safe.

The instantiation `nterm :: term` is given below:

$$\begin{aligned}
\mathsf{const} &= \mathsf{Nconst} & \mathsf{consts}\,(\mathsf{Nabs}\;x\;t) &= \mathsf{consts}\;t \\
\mathsf{free} &= \mathsf{Nvar} & \mathsf{frees}\,(\mathsf{Nabs}\;x\;t) &= \mathsf{frees}\;t - \{x\} \\
\mathsf{app} &= \mathsf{Napp} & \mathsf{subst}\;\sigma\,(\mathsf{Nabs}\;x\;t) &= \mathsf{Nabs}\,(\mathsf{subst}\;(\sigma - x)\;t)
\end{aligned}$$
$$\mathsf{abs\_pred}\;P\;t = (\forall x\;t'.\; t = \mathsf{Nabs}\;x\;t' \implies P\;t' \implies P\;t)$$

The main difference to the `term` type is that the bound variable name affects substitution, i.e., it needs to be dropped from the environment.

4.3.3 Explicit pattern matching (`pterm`)

Functions in Isabelle are usually defined using *implicit* pattern matching, that is, the terms p_i occurring on the left-hand side $f\;p_1\;\ldots\;p_n$ of an equation may be constructor patterns. This is also common among functional programming languages like Haskell or OCaml. CakeML only supports *explicit* pattern matching using case expressions. A function definition consisting of multiple defining equations must hence be translated to the form $f = \lambda x.$ **case** x **of** .... The elimination proceeds by iteratively removing the last parameter in the block of equations until none are left (§6.3).

In my formalization, I have decided to conflate the notion of "abstraction" and "case expression", yielding *case abstractions,* represented as the `Pabs` constructor in Listing 4.5.b. This is similar to the `fn` construct in Standard ML, which denotes an anonymous function that immediately matches on its argument [91]. The same construct also exists in Haskell with the `LambdaCase` language extension in Haskell.[1] As in §4.2.4.5, the pairs of patterns and terms are referred to as *clauses.*

I chose this representation for these reasons:

1. It allows for a simpler language grammar because there is only one (shared) constructor for abstraction and case expression.
2. The elimination procedure outlined above does not need to introduce fresh names in the process. Later, when translating to CakeML syntax, fresh names are introduced and proved correct in a separate step (§6.8).

Furthermore, observe that the clauses are unordered and are hence a set. This stems from the unordered nature of the term-rewriting semantics (Listing 2.4.b): if a rule matches, it can be applied, no matter what order the rules were defined in.

As a short-hand notation, $\Lambda\{p_1 \Rightarrow t_1, p_2 \Rightarrow t_2, ...\}$ represents `Pabs` $\{(p_1, t_1), (p_2, t_2), ...\}$.

The instantiation `pterm` :: `term` is a bit more complicated than for `nterm`, because it has to take multiple clauses into account:

$$\begin{aligned}
\texttt{consts}\,(\texttt{Pabs}\ cs) &= \bigcup_{(p,t)\in cs} \texttt{consts}\ t \\
\texttt{frees}\,(\texttt{Pabs}\ cs) &= \bigcup_{(p,t)\in cs} \texttt{frees}\ t - \texttt{frees}\ p \\
\texttt{subst}\ \sigma\,(\texttt{Pabs}\ cs) &= \texttt{Pabs}\ \{(p, \texttt{subst}\ (\sigma - \texttt{frees}\ p)\ t) \mid (p,t) \in cs\} \\
\texttt{abs\_pred}\ P\,t &= (\forall cs.\ t = \texttt{Pabs}\ cs \implies (\forall\ p\,t.\ (p,t) \in cs \implies P\,t) \implies P\,t)
\end{aligned}$$

As for `nterm`s, substitution is only supported for closed terms.

4.3.4 Ordered clauses (`sterm`)

For CakeML, the clauses need to be applied in a deterministic order, i.e., sequentially. The `sterm` type only differs from `pterm` by using `list` instead of `set` (Listing 4.5.c). Hence, case abstractions use list brackets: $\Lambda[p_1 \Rightarrow t_1, p_2 \Rightarrow t_2, ...]$.

The instantiation `sterm` :: `term` is largely similar to above.

The proofs of Axioms 4.9 and 4.10 can be shared between `sterm` and `pterm` with a simple trick. First, I define a function that converts from `sterm` to `pterm`; call it `sterm_to_pterm` (the implementation is given in Listing 6.7). This is always possible, because one can always discard an ordering, i.e., turn a list into a set. Next, I prove that this conversion function

[1]`https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/syntax-extns.html`

is total and that the free variables and constants are preserved by it. Finally, the following lemma allows to transfer results from `sterm` to `pterm`:

Lemma 4.37.

$$\texttt{subst}\ (\texttt{map}\ \texttt{sterm\_to\_pterm}\ \sigma)\ (\texttt{sterm\_to\_pterm}\ t) = \texttt{sterm\_to\_pterm}\ (\texttt{subst}\ \sigma\ t)$$

Taken to its conclusion, this trick could eventually be used to define `pterm` as a quotient type of `sterm` [58].

For all terms that do not contain abstractions, `sterm_to_pterm` coincides with `convert_term`. This applies to all conversion functions between term types that are used in the compiler (§6).

4.3.5 Irreducible terms (`value`)

CakeML distinguishes between *expressions* and *values*. Whereas expressions may contain free variables or β-redexes, values are closed and fully evaluated. Both have a notion of abstraction, but values differ from expressions in that they contain an environment binding free variables.

Consider the expression $(\lambda x.\lambda y.x)\ (\lambda z.z)$, which can be rewritten to $\lambda y.\lambda z.z$ by β-reduction. Note how the bound variable x disappears, since it gets substituted. This is contrary to how programming languages are usually implemented: Evaluation does not happen by substituting the argument term t for the bound variable x, but by recording the binding $x \longmapsto t$ in an environment [74]. A pair of an abstraction and an environment is usually called a *closure* [81, 123]. Similarly to `pterm` and `sterm`, pattern matching is immediate, which is why the body of a closure is a list of clauses.

In CakeML, this means that evaluation of the above expression results in the closure $(\lambda y.x, [\texttt{"x"} \longmapsto (\lambda z.z, [])])$. Note the nested structure of the closure, whose environment itself contains a closure.

To reflect this in the formalization, I introduce a `value` type (Listing 4.5.d) which distinguishes between:

VCONSTR constructor values: a data constructor applied to multiple values

VABS closures: clauses combined with an environment mapping variables to values

VRECABS recursive closures: a group of mutually recursive function bodies with an environment mapping variables to values

The above example evaluates to the closure

$$\texttt{Vabs}\ \big[\langle y\rangle \Longrightarrow \langle x\rangle\big]\ \big[\texttt{"x"} \longmapsto \texttt{Vabs}\ [\langle z\rangle \Longrightarrow \langle z\rangle]\ []\big]$$

Recursive closures The third case for recursive closures only becomes relevant late in the compiler (§6.7). In the initial term-rewriting semantics, there is a clear distinction between constants and variables. Constants and their definitions are recorded in the rule set *rs*. Consequently, recursive calls are straightforward: The appropriate definition for the constant can be looked up there. CakeML knows no such distinction between constants and variables, hence everything has to reside in a single environment σ.

This can be illustrated with the following example. Consider the defining equations of odd and even:

$$\begin{aligned} \mathsf{odd}\ 0 &= \mathsf{False} \\ \mathsf{odd}\ (\mathsf{Suc}\ n) &= \mathsf{even}\ n \\ \mathsf{even}\ 0 &= \mathsf{True} \\ \mathsf{even}\ (\mathsf{Suc}\ n) &= \mathsf{odd}\ n \end{aligned}$$

When evaluating the term odd *k*, the definitions of even and odd themselves must be available in the environment captured in the definition of odd. However, there is no way in Isabelle to inductively construct such a Vabs that refers to itself, because that would be an infinite object. Instead, the expressions used to define odd and even are captured in a recursive closure. There might be an alternative design using coinduction, but I have chosen to follow the same path as CakeML, where it is modelled in a similar way as here.

For the above example, this would result in the following global environment:

$$[\texttt{"odd"} \longmapsto \mathsf{Vrecabs}\ css\ \texttt{"odd"}\ [], \texttt{"even"} \longmapsto \mathsf{Vrecabs}\ css\ \texttt{"even"}\ []]$$

$$\begin{aligned} \text{where } css = [&\texttt{"odd"} \longmapsto [\langle 0\rangle \Longrightarrow \langle\mathsf{False}\rangle, \langle\mathsf{Suc}\ n\rangle \Longrightarrow \langle\mathsf{even}\ n\rangle], \\ &\texttt{"even"} \longmapsto [\langle 0\rangle \Longrightarrow \langle\mathsf{True}\rangle, \langle\mathsf{Suc}\ n\rangle \Longrightarrow \langle\mathsf{odd}\ n\rangle]] \end{aligned}$$

Note that in the first line, the right-hand sides are values, but in *css*, they are expressions. The additional string argument denotes the *active* function. The semantics then takes care to add those to the environment as appropriate.

4.3.5.1 Conversions

The lack of a suitable `term` instantiation notwithstanding, it is possible to establish a relationship with `sterms` by defining conversion functions. Similar to how `linear` characterizes `terms` that lie within the pattern fragment, `is_value` characterizes `sterms` that lie within the value fragment.

$$\frac{}{\mathtt{is\_value}\ (\Lambda\ cs)} \qquad \frac{name \in \mathsf{C} \qquad \forall t \in ts.\ \mathtt{is\_value}\ t}{\mathtt{is\_value}\ (name\ \$\$\ ts)}$$

Informally, a term that satisfies that predicate is referred to as a *term-value*. Observe that this definition is parametrized on C, which is the set of all constructor names in any given theory. This set is automatically generated during deep embedding (§5.2).

The actual conversion from values to terms can be described as executing the substitution from the captured environments:

```
fun value_to_sterm :: value ⟹ sterm where
value_to_sterm (Vconstr name vs) =
  list_comb (Sconst name) (map value_to_sterm vs)
value_to_sterm (Vabs cs σ) =
  Λ [(pat, subst (map value_to_sterm σ - frees pat) t)) | (pat, t) ← cs]
value_to_sterm (Vrecabs css name σ) =
  Λ [(pat, subst (map value_to_sterm σ - frees pat) t)) | (pat, t) ← css name]
```

The opposite direction does not need to perform any substitution, but it has to traverse a term according to its applicative structure (§4.2.4.2):

```
fun sterm_to_value :: sterm ⟹ value where
sterm_to_value t = case strip_comb t of
  (Sconst name, args) ⟹ Vconstr name (map sterm_to_value args)
  (Sabs cs, []) ⟹ Vabs cs [])
```

Note that if the term is an abstraction, the result is a closure with an empty environment.

That function to convert from terms to values is underspecified. Its domain is a superset of all terms-values; specifically, terms that contain non-constructor constants are allowed. All other terms yield an unspecified result. For the purpose of this thesis, it is sufficient to assume that `is_value` is the domain of `sterm_to_value`.

Lemma 4.38. *For all term-values t,* `value_to_sterm` (`sterm_to_value` *t*) = *t.*

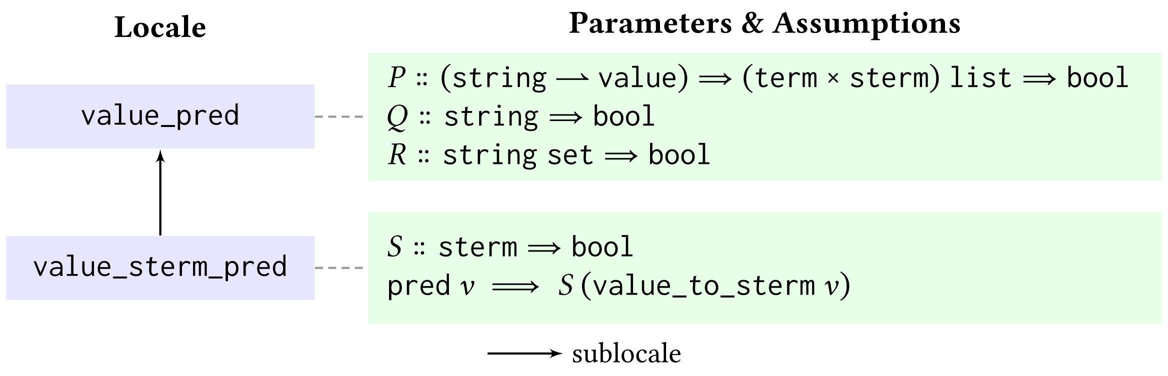
Unfortunately, the converse direction does not hold: when transforming values to terms, the distinction between recursive and non-recursive closures is lost.

4.3.5.2 Syntactic predicates and relations

§4.2.4.3 describes locales providing syntactic predicates on term types. Unfortunately, the `value` type cannot be made an instance of the `term` class, because it has no notion of free variables: values are always thought to be closed.

Nonetheless, just like for terms, there needs to be some predicate that can be used to enforce syntactic properties on values, because the clauses in a closure may be of arbitrary shape. Figure 4.2 lists two locales: `value_pred` introduces a generic predicate `pred` for wellformedness checks (§6.1) on values and `value_sterm_pred` establishes a connection to predicates on `sterm`s. Contrary to the locales on terms, `value_pred` is used to construct a predicate of type `value ⟹ bool`, instead of providing more results about existing predicates. The motivation behind this is that recursion on values – because of their complicated structure – is cumbersome and repetitive, especially termination proofs. This locale avoids such duplication.

The locale parameters and the definition of `pred` itself is sufficiently technical to warrant a high-level explanation. *P* checks a list of clauses against an environment. This can, for example, be used to ensure that all free variables in the clauses are bound in the environment.



where:

$$\begin{aligned}
\texttt{pred}\ (\texttt{Vconstr}\ \mathit{name}\ \mathit{vs}) &= Q\ \mathit{name} \wedge (\forall v \in \mathit{vs}.\ \texttt{pred}\ v)\\
\texttt{pred}\ (\texttt{Vabs}\ \mathit{cs}\ \Gamma) &= P\ \Gamma\ \mathit{cs} \wedge (\forall v \in \texttt{range}\ \Gamma.\ \texttt{pred}\ v)\\
\texttt{pred}\ (\texttt{Vrecabs}\ \mathit{css}\ \mathit{name}\ \Gamma) &= (\forall v \in \texttt{range}\ \Gamma.\ \texttt{pred}\ v) \wedge \mathit{name} \in \texttt{dom}\ \mathit{css} \wedge\\
&\qquad R\ (\texttt{dom}\ \mathit{css}) \wedge (\forall \mathit{name} \in \texttt{dom}\ \mathit{css}.\ P\ \Gamma\ (\Gamma\ \mathit{name}))
\end{aligned}$$

Figure 4.2: Syntactic locales on values (predicates)

Consequently, the instantiation

$$\begin{aligned}
P\ \Gamma\ \mathit{cs} &= (\forall (p, t) \in \mathit{cs}.\ \texttt{frees}\ t \subseteq (\texttt{dom}\ \Gamma \cup \texttt{frees}\ p))\\
Q\ \mathit{name} &= \texttt{True}\\
R\ \mathit{names} &= \texttt{True}
\end{aligned}$$

yields a predicate that is true if and only if a value is closed.

Q and R check the names of the constructor and the closures. This is used to avoid collisions with other kinds of constants. In the compiler, this becomes relevant at multiple stages; for example, to ensure that no constant with a definition is also, say, a data constructor (see also §5 for details).

The pred meta-predicate always checks, regardless of the concrete instantiation, that in a recursive closure, the active function is present. This has merely been done for convenience.

Continuing with the above example, the second locale value_sterm_pred can be instantiated by defining S = closed. It remains to be proved that pred v implies closed (value_to_sterm v), which is mostly technical and uninteresting.

Figure 4.3 introduces a locale for structural relations between values. The $\simeq$ relation represents a basic structural equality that only considers the shape of the value; and in the case of constructors, recursively.

Lemma 4.39. *If $t \simeq u$ and t or u contain no closures, then $t = u$.*

Corollary 4.40. *If $Q\ t\ u$ and t or u contain no closures, then $t = u$.*

Locale	**Parameters & Assumptions**
`value_struct_rel`	$Q :: \texttt{value} \Rightarrow \texttt{value} \Rightarrow \texttt{bool}$ $Q\, t_1\, t_2 \implies t_1 \simeq t_2$ $\texttt{Vconstr}\ name_1\ ts_1 \simeq \texttt{Vconstr}\ name_2\ ts_2 \iff$ $name_1 = name_2 \wedge \texttt{rel}\ Q\ ts_1\ ts_2$

where:

$$\frac{}{\texttt{Vabs}\ cs_1\ \Gamma_1 \simeq \texttt{Vabs}\ cs_2\ \Gamma_2} \qquad \frac{}{\texttt{Vrecabs}\ css_1\ name_1\ \Gamma_1 \simeq \texttt{Vrecabs}\ css_2\ name_2\ \Gamma_2}$$

$$\frac{\texttt{rel}\ (\simeq)\ ts\ us}{\texttt{Vconstr}\ name\ ts \simeq \texttt{Vconstr}\ name\ us}$$

Figure 4.3: Syntactic locale on values (relation)

Corollary 4.41. *$\simeq$ itself is a structural value relation.*

A predicate that ensures the absence of closures can be easily obtained by instantiating `pred` as follows:

$$P\,\Gamma\ cs = \texttt{False}$$
$$Q\ name = \texttt{True}$$
$$R\ names = \texttt{False}$$

All three locales provide further results about matching, but this requires additional material on patterns. I will revisit this in §4.3.6.

4.3.5.3 Extensional equivalence

A special kind of structural relation on terms is extensionality (Listing 4.7). It compares the captured environments based on the *identifiers* that occur in the body of the closure. The set of identifiers comprises the sets of free variables and constants: `ids` t = `frees` $t \cup$ `consts` t.

As outlined earlier in this section, the closures cases of the `value` type are thought to contain environments mapping variables to values. This is only true up until a particular phase in the compiler (§6.7). From that part on, the environments contain mappings for both variables and constants. This explains why the extensional equivalence relation uses the set of identifiers to compare the environments.

Lemma 4.42. *Extensional equivalence is reflexive and a structural value relation.*

4.3.6 Proper patterns (`pat`)

The `value` type, instead of using binary function application as all other term types, uses *n*-ary constructor application. This introduces a conceptual mismatch between (binary) patterns

$$\frac{\texttt{rel}\ (\approx_e)\ vs\ us}{\texttt{Vconstr}\ name\ vs \approx_e \texttt{Vconstr}\ name\ us}$$

$$\frac{\forall x \in \texttt{ids}\ (\Lambda\ cs).\ \sigma_1\ x \approx_e \sigma_2\ x}{\texttt{Vabs}\ cs\ \sigma_1 \approx_e \texttt{Vabs}\ cs\ \sigma_2} \qquad \frac{\forall cs \in \texttt{range}\ css.\ \forall x \in \texttt{ids}\ (\Lambda\ cs).\ \sigma_1\ x \approx_e \sigma_2\ x}{\texttt{Vrecabs}\ css\ name\ \sigma_1 \approx_v \texttt{Vrecabs}\ css\ name\ \sigma_2}$$

Listing 4.7: Extensional equivalence of values

and values. To make some proofs easier, and to introduce an intermediate layer between `term` patterns and CakeML patterns, I have introduced a *dedicated pattern* type of *n*-ary patterns (Listing 4.8). The function `mk_pat :: term ⇒ pat` converts from binary to *n*-ary patterns.

Note that the function is underspecified. Its domain is a superset of all linear terms; specifically, terms that are “almost” linear but in which the same variable occurs twice are allowed. All other terms yield an unspecified result.

The remainder of this section is concerned with defining a matching function and establishing a correspondence to `match`; more specifically, for the `sterm` instantiation of `match`. This works in two steps: first, I define a function that matches a `pat` and an `sterm`, then, a function for `pat` and `value`.

Because patterns have a nested recursion structure, it becomes necessary to introduce some library functions that deal with lists:

$$\begin{aligned} \texttt{those} &:: \alpha\ \texttt{option list} \Rightarrow \alpha\ \texttt{list option} \\ \texttt{map2} &:: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha\ \texttt{list} \Rightarrow \beta\ \texttt{list} \Rightarrow \gamma\ \texttt{list} \end{aligned}$$

With them, it is now possible to define the intermediate matching function:

```
fun match' :: pat ⇒ sterm ⇒ (string ⇀ sterm) option where
match' (Patvar name) t = Some [name ↦ t]
match' (Patconstr name ps) t = case strip_comb t of
  (Sconst name', vs) ⇒
    if name = name' ∧ length ps = length vs then
      map (foldl (++) []) (those (map2 match' ps vs))
    else
      None
  _ ⇒
    None
```

Lemma 4.43. *For all linear patterns p and term-values t:* `match` *p t is equal to* `match' (mk_pat` *p) t.*

The proof requires a custom induction rule on linear patterns:

```
datatype pat =
  Patvar string |
  Patconstr string (pat list)

fun mk_pat :: term ⟹ pat where
mk_pat pat = case strip_comb pat of
  (Const s, args) ⟹ Patconstr s (map mk_pat args)
  (Free s, []) ⟹ Patvar s
```

Listing 4.8: Proper patterns

Lemma 4.44 (Linear induction).

$$\frac{\texttt{linear}\ t \qquad \forall s.\ P\,(\texttt{Free}\ s) \qquad \forall name\ args.\ \texttt{linears}\ args \implies (\forall arg \in args.\ P\ arg) \implies P\,(name\ \$\$\ args)}{P\,t}$$

Proof. The proof proceeds by well-founded induction on the size measure. Clearly, in the recursive case, all constituent patterns are smaller than their combination. Because *t* is linear, the cases are exhaustive. □

Finally, the desired matching function, together with its correctness statement, can be defined:

```
fun vmatch :: pat ⟹ value ⟹ (string ⇀ value) option where
vmatch (Patvar name) v = Some [name ↦ v]
vmatch (Patconstr name ps) (Vconstr name' vs) =
  if name = name' ∧ length ps = length vs then
    map (foldl op ++ []) (those (map2 vmatch ps vs))
  else
    None
vmatch _ _ = None
```

Lemma 4.45 (*n*-ary vs. binary patterns)**.** *For all linear patterns, the result of* `vmatch` *corresponds (with respect to* `value_to_sterm`*) to* `match`*. Formally:*

$$\begin{aligned}\texttt{linear}\ p \implies &\texttt{rel}\,(\texttt{rel}\,(\lambda v\,t.\ t = \texttt{value\_to\_sterm}\ v))\ \sigma_1\ \sigma_2\\ \textit{where}\ \sigma_1 &= \texttt{vmatch}\,(\texttt{mk\_pat}\ p)\ v\\ \sigma_2 &= \texttt{match}\ p\,(\texttt{value\_to\_sterm}\ v)\end{aligned}$$

The proof first establishes a correspondence between `match'` and `vmatch` and then composes the result with Lemma 4.43.

Having introduced proper patterns and matching of values with patterns, Lemma 4.22 can be adapted accordingly in the context of the predicate locale (Figure 4.2):

Lemma 4.46. *If a value v for which* `pred` *v holds matches a pattern with resulting environment σ, then* `pred` *holds for the range of σ.*

Similarly to §4.2.4.3, the relation locale (Figure 4.3) admits relational results on matching. For example, the equivalent to Lemma 4.24 is:

Lemma 4.47. *For all structural value relations Q: If $Q\ t_1\ t_2$, then*

$$\texttt{rel}\ (\texttt{rel}\ Q)\ (\texttt{vmatch}\ p\ t_1)\ (\texttt{vmatch}\ p\ t_2)$$

4.4 Related work

Term rewriting The field of term rewriting is a rich area of research. For the purpose of this thesis, I would like to restrict the analysis of related work to Isabelle formalizations. Most notably in this space is the IsaFoR/CeTA project: "An Isabelle/HOL Formalization of Rewriting for Certified Tool Assertions".[2] The project consists of multiple modules, some of which are available in the Archive of Formal Proofs:

Abstract Rewriting Sternagel and Thiemann [116] give abstract characterizations of term rewriting systems, in particular definitions of properties like completeness and normalization. There is no fixed type of terms that are prescribed for rewriting; all definitions operate on arbitrary relations.

First-Order Terms The same authors [118] provide a general definition of first-order terms, i.e., composed of variables and function applications. Based on this, they introduce matching and unification algorithms.

Z Property Felgenhauer et al. [35] formalized the *Z property* based on work by Dehornoy and van Oostrom [32]. As an example application, they prove that lambda calculus has the Church–Rosser property [29]. Unfortunately, this formalization of higher-order terms does not include the notion of *constants*.

Term algebras Schmidt-Schauß and Siekmann [111] discuss the concept of *unification algebras*. They generalize terms to *objects* and substitutions to *mappings*. A unification problem can be rephrased to finding a mapping such that a set of objects are mapped to the same object. The advantage of this generalization is that other – superficially unrelated – problems like solving algebraic equations or querying logic programs can be seen as unification problems.

In particular, the authors note that among the similarities of such problems are that "objects [have] variables" whose "names do not matter" and "there exists an operation like substituting objects into variables" [111, §1]. The major difference between my formalization in §4.2 and the work by Schmidt-Schauß and Siekmann is that I use concrete types for variables and mappings. Otherwise, some similarities to here can be found.

Eder [33] discusses properties of substitutions with a special focus on a partial ordering between substitutions. However, Eder constructs and uses a concrete type of first-order terms, similarly to Sternagel and Thiemann [118].

Williams [129] defines substitutions as elements in a monoid. In this setting, instantiations can be represented as *monoid actions*. Williams then proceeds to define – for arbitrary sets of

[2] `http://cl-informatik.uibk.ac.at/isafor/`

terms and variables – the notion of *instantiation systems,* heavily drawing on notation from Schmidt-Schauß and Siekmann. Some of the presented axioms [129, §2] are also present in my formalization. Consequently, generic theorems can be derived from those axioms: for example Corollary 4.25,[3] Lemma 4.27,[4] or Axiom 4.8.[5]

Higher-order terms Blanchette et al. have formalized λ-free higher-order terms in Isabelle [6, 17]. Their main application are term orders that are used in automated theorem proving. The formalization is relevant insofar as their term type can be made an instance of the generic `term` class.

`Lambda_Free_Compat`

4.5 Conclusion

In this section, I have presented a formalization of a term algebra that abstracts over operations that are common in term rewriting. This itself is not a novel idea, but it – to the best of my knowledge – the first implementation in a proof assistant. I have demonstrated its applicability to other term formalizations, too, which could then benefit from a wealth of results.

Further work in this topic includes a generalized implementation of unification. While this is not strictly necessary for this formalization, it would avoid duplicated formalization efforts for other applications.

The algebra also allows pain-free definition of different varieties of term types. This simplifies the correctness proofs of the compiler phases, because each phase can use a tailored term type.

[3]Corollary 3.8B: $t\sigma = t$, if $\text{var}(t) \cap \text{dom}(a) = \emptyset$

[4]Corollary 3.8A: $t\sigma = t(\sigma|\text{var}(t))$

[5]Corollary 3.8D: $\sigma + \tau = \sigma\tau$, if $\text{cdm}(\sigma) \cap \text{dom}(\tau) = \emptyset$

5 Deep embedding of terms

Deep down, I happen to be very shallow.

(Pat Paulsen)

The *deep embedding* phase happens after preprocessing (§3) and is special: whereas preprocessing stays entirely within the realm of ML to alter (or derive new) HOL definitions and the compiler (§6) stays entirely within the realm of term languages modelled in Isabelle, deep embedding lifts definitions into a model. It bridges the gap between the ML and the HOL world.

Starting with a HOL definition, this phase constructs a *reified* definition in the deeply embedded term language given in Listing 2.4.a. This term language corresponds closely to the `term` data type of Isabelle's implementation (using de Bruijn indices [23]), but without types and schematic variables. To establish a formal connection between the original and the reified definitions, a "family of relations" is used, "defined by induction on types" [120]. This concept of a *logical relation* is well-understood in literature [54, 120] and can be nicely implemented in HOL using type classes.

This compiler phase can be structured into three parts: the mapping – implemented in ML – between raw Isabelle terms and deeply-embedded `terms` (§5.1), the HOL relations that certify correspondence between both representations (§5.2), and the proof tactics – also implemented in ML – that establish theorems about concrete mappings (§5.3).

5.1 Embedding operation

The embedding operations lifts HOL definitions into the `term` type. I use angle brackets to denote an embedded term: $\langle t \rangle$, where t is an arbitrary HOL expression of any type and the result $\langle t \rangle$ is a HOL value of type `term`. It is a purely syntactic transformation, without preliminary evaluation or reduction, and discards type information. The following examples illustrate the operation and its typographical conventions:

$$\begin{aligned}
\langle x \rangle &= \texttt{Free "x"} \\
\langle \texttt{f} \rangle &= \texttt{Const "f"} \\
\langle x \mathbin{\#} xs \rangle &= \texttt{Const "List.list.Cons"} \mathbin{\$} \langle x \rangle \mathbin{\$} \langle xs \rangle \\
\langle \lambda x y.\ \texttt{f}\ y\ x \rangle &= \Lambda\ (\Lambda\ (\langle \texttt{f} \rangle \mathbin{\$} \mathsf{Bound}\ 0 \mathbin{\$} \mathsf{Bound}\ 1))
\end{aligned}$$

Contrary to the formalization, I will use the $\langle t \rangle$ notation in a more flexible way and allow it to also represent values of the other term types (§4.3), e.g. `pterm`.

The correspondence between an embedded term t to a HOL term a of type τ is expressed with the syntax $R \vdash t \downarrow a$. The relation can be described as "using the rule set R, the term t can be rewritten to $\langle a \rangle$". Most importantly, this means the correspondence relation is aware of the term-rewriting semantics of `term` (Listing 2.4.b), where R is the set of all known defining equations. Furthermore, the notation $R \vdash t \downarrow a$ actually represents a family of relations. Abstractly, it is defined in the `embed` type class.

Before the relations are discussed in detail in §5.2, I will show an example of the embedding operation and explain its implementation.

Example Consider the `map` function on lists:

$$\begin{array}{llll} \texttt{map} & f & [] & = [] \\ \texttt{map} & f & (x \mathbin{\#} xs) & = f\,x \mathbin{\#} \texttt{map}\,f\,xs \end{array}$$

The result of embedding this function is a set of rules `map'`. If written directly by the user, it would look as follows:

```
definition map' :: (term × term) set where
map' =
  {(Const "List.list.map" $ Free "f" $ (Const "List.list.Cons" $ Free "x" $ Free "xs"),
      Const "List.list.Cons" $ (Free "f" $ Free "x") $ ...),
   (Const "List.list.map" $ Free "f" $ Const "List.list.Nil",
      Const "List.list.Nil")}
```

Additionally, the theorem `map'` $\vdash$ `Const "List.list.map"` $\downarrow$ `map` is proved using a custom tactic (§5.3). Constant names like `List.list.map` come from the fully qualified internal names in Isabelle.

Implementation Listing 5.1 shows the Isabelle/ML implementation of the embedding operation on terms. It lifts the Pure term constructors to their equivalents in the `term` type.

Note that the `term` type only knows free variables, but no schematic variables. Depending on the context where this function is used, either schematic or free variables are mapped to free variables in `term`.

This function can then be subsequently used to lift sets of equations. As illustrated by the example, the resulting type is (`term` × `term`) `set`: equations are split into left- and right-hand side (Definition 4.34). The set contains all transitive dependencies according to the code graph (§3.1.3).

Because the set of embedded defining equations is now subject to a term-rewriting semantics, I will refer to it as a rule set (§4.2.4.5).

Observe that both constructor names (e.g. `Cons`) and function names (e.g. `map`) look the same in this representation. But just like in the dictionary construction (§3.1), they require different treatment in some phases of the compiler. For that reason, the deep embedding also produces a set of constructor names, which is a subset of all constants that occur. This is captured in the `constructors` locale (Listing 5.2). The constructor information consists of name, arity, and name of the type that it belongs to.

Additionally, there is also a locale that fixes a rule set and assumes some wellformedness conditions (§6.1). It is a sublocale of `constants`, which ensures that there are no constants

```
fun embed schematic t =
  let
    fun
      aux (Const (n, _)) = @{term Const} $ HOLogic.mk_string n
    | aux (Free (n, _)) =
        if schematic then
          error "free variables are not supported"
        else
          @{term Free} $ HOLogic.mk_string n
    | aux (Bound i) = @{term Bound} $ HOLogic.mk_number @{typ nat} i
    | aux (t $ u) = @{term App} $ aux t $ aux u
    | aux (Abs (_, _, t)) = @{term Abs} $ aux t
    | aux (Var ((n, i), _)) =
        if schematic then
          @{term Free} $ HOLogic.mk_string (n ^ "." ^ Value.print_int i)
        else
          error "schematic variables are not supported"
  in aux t end
```

Listing 5.1: Full ML implementation of the embedding operation on terms

that are both constructors and head of a defining equation. More broadly speaking, the embedding operation does not just define a set of equations, but declares an interpretation of the locale and proves all conditions. Ill-formed rule sets would be flagged at this point.

Deep embedding – being a purely syntactic operation – does not respect *referential transparency* [25, 108]. In particular, constants' names become part of the embedded definition. As an example, consider the two functions $\mathtt{id}_1\ x = x$ and $\mathtt{id}_2\ x = x$. Both can be proved to be equal according to HOL's extensional function equality. However, their deep embeddings differ from each other. For practical purposes, this bears little relevance, because users of the compiler would not routinely alter either the embedding or the generated theorems.

5.2 Embedding relations

The embed type class (Listing 5.3) introduces two distinct relations: $\approx$ *(ground embedding)* and $\downarrow$ *(equational embedding)*. The latter has already been briefly introduced in the previous section. To make the type explicit, both relations can be indexed: $\approx_\tau$ and $\downarrow_\tau$. Both also use the same syntax: An embedded term corresponds to a HOL term a of type τ with respect to a rule set R is written as $R \vdash t \approx a$ or $R \vdash t \downarrow a$. However, both differ in the sets of terms t that correspond to a.

```
type_synonym c_info = nat × string

locale constructors =
  fixes C_info :: string ⇀ c_info
begin

definition C :: string set where
C = dom C_info

end

locale constants = constructors +
  fixes heads :: string set
  assumes heads ∩ C = ∅
begin

definition all_consts :: string set where
all_consts = heads ∪ C

end
```

Listing 5.2: Definitions of the `constructors` and `constants` locales

The definitions for $\downarrow$ and $\approx$ are part of the trusted base, i.e., one needs to be confident in them in order to be confident of the main theorems.

Ground embedding For ground types, an embedding relation can be defined easily. For example, the following two rules define $\approx_{\mathsf{nat}}$:

$$\frac{}{R \vdash \langle \mathsf{Groups.zero\_class.zero} \rangle \approx_{\mathsf{nat}} 0} \qquad \frac{R \vdash \langle t \rangle \approx_{\mathsf{nat}} n}{R \vdash \langle \mathsf{Nat.Suc}\ t \rangle \approx_{\mathsf{nat}} \mathsf{Suc}\ n}$$

Definitions of $\approx$ for arbitrary data types without nested recursion can be derived mechanically in the same fashion as for `nat`, where they constitute one-to-one relations. Every datatype is defined by a set of constructors:

$$(\alpha_1, \ldots, \alpha_n)\ \tau = \mathsf{C}_1\ \kappa_{1,1}\ \ldots\ \kappa_{1,n_1} \mid \cdots \mid \mathsf{C}_k\ \kappa_{1,k}\ \ldots\ \kappa_{1,n_k}$$

Consequently, the predicate

$$\approx_\tau :: (\mathsf{term} \times \mathsf{term})\ \mathsf{set} \Longrightarrow \mathsf{term} \Longrightarrow (\alpha_1 :: \mathsf{embed}, \ldots, \alpha_n :: \mathsf{embed})\ \tau \Longrightarrow \mathsf{bool}$$

is defined with one rule per constructor:

$$\frac{R \vdash t_1 \approx x_1 \qquad \cdots \qquad R \vdash t_{n_i} \approx x_{n_i}}{R \vdash \langle \mathsf{C}_i \rangle \mathbin{\$} t_1 \mathbin{\$} \ldots \mathbin{\$} t_{n_i} \approx_\tau \mathsf{C}_i\ x_1\ \ldots\ x_{n_i}}$$

```
class embed =
  fixes embed :: (term × term) set ⇒ term ⇒ α ⇒ bool (_ ⊢ _ ≈ _)
  assumes R ⊢ t ≈ a  ⟹  wellformed t
begin

definition embed' :: (term × term) set ⇒ term ⇒ α ⇒ bool (_ ⊢ _ ↓ _) where
R ⊢ t ↓ a ⟷ wellformed t ∧ (∃ t'. R ⊢ t ⟶* t' ∧ R ⊢ t' ≈ a)

end
```

Listing 5.3: embed class

Proving the axiom of the type class is trivial, because it only requires wellformedness of `terms`. A `term` is wellformed if there are no dangling de Bruijn indices. In this case, no bound variables occur in the definition of $\approx_\tau$.

An implementation restriction is that currently, nested recursion is not supported. Instead, such datatypes must be expressed with mutual recursion [46], which is supported by the current **datatype** command [18, §2].

Note that for such types, $\approx$ ignores R, because no rewriting happens. The reason why $\approx$ must be parametrized on R will become clear in a moment.

For function types, I follow Myreen and Owen's approach [94]. The statement $R \vdash t \approx f$ can be interpreted as "$t \,\$\, \langle a \rangle$ can be rewritten to $\langle f\, a \rangle$". Because this might involve applying a function definition from R and beta reduction, the $\approx$ relation must be indexed by the rule set. This is where equational embedding comes into play.

Equational embedding The equational embedding relation $R \vdash t \downarrow x$ is defined to mean that there is a t' such that t can be rewritten to t' in R where $R \vdash t' \approx x$. This is apparent in the formalization (Listing 5.3). Formally:

$$R \vdash t \downarrow a \Longleftrightarrow \texttt{wellformed}\, t \wedge (\exists t'.\, R \vdash t \longrightarrow^* t' \wedge R \vdash t' \approx a)$$

Equipped with this, it becomes possible to define $\approx$ for function types:

$$\frac{\texttt{wellformed}\, t \qquad \forall\, x\, u.\, R \vdash u \downarrow_{\tau_1} x \Longrightarrow R \vdash t \,\$\, u \downarrow_{\tau_2} f\, x}{R \vdash t \approx_{\tau_1 \Rightarrow \tau_2} f}$$

This definition is easily motivated: Embedding a constant `f` yields a rule set R, containing rewrite rules for `f` and all other required constants. The embedding routine will produce the theorem $R \vdash \langle \texttt{f} \rangle \approx \texttt{f}$. After applying extensionality, one still needs to apply the rewrite rules to obtain the corresponding function values. Note that $\langle \texttt{f} \rangle$ itself, without being applied to parameters, cannot be rewritten.

5.3 Embedding proofs

The proofs for the correctness of the embedding are fully automatic. To illustrate the mechanics, I will prove the embedding for a list concatenation function:

$$\begin{aligned} \mathsf{append}\ []\ ys &= ys \\ \mathsf{append}\ (x\ \#\ xs)\ ys &= x\ \#\ \mathsf{append}\ xs\ ys \end{aligned}$$

In this simple example, the function is total (according to §3.1.5). Consequently, the final theorem is unconditional and in η-contracted form:

$$R \vdash \langle \mathsf{append} \rangle \approx \mathsf{append}$$

In order to prove this statement, the definition of $\approx$ for functions needs to be unfolded twice:

$$\frac{R \vdash t_{xs} \downarrow xs \qquad R \vdash t_{ys} \downarrow ys}{\exists t'.\ R \vdash \langle \mathsf{append} \rangle \$\ t_{xs}\ \$\ t_{ys} \longrightarrow^* t' \wedge R \vdash t' \approx \mathsf{append}\ xs\ ys}$$

The append function now appears in fully-expanded application form.

The variables t_{xs} :: `term` and t_{ys} :: `term` are *metavariables*. As such, they do not stand for `term`-level variables `Free` xs or `Free` ys, but rather for arbitrary terms. The naming is chosen to illustrate the relationship between t_x and x.

However, the term $\langle \mathsf{append} \rangle \$\ t_{xs}\ \$\ t_{ys}$ cannot be rewritten, because no rule in R matches. In this case, applying induction on xs is sufficient; in general, induction on the termination relation of the function is required.

Consider the case where $xs = []$. The assumption for t_{xs} is instantiated as $R \vdash t_{xs} \downarrow []$. Expanding $\downarrow$ means that there is a t' such that t_{xs} can be rewritten to t' and $R \vdash t' \approx []$. According to the definition of $\approx_{\texttt{list}}$, i.e., the ground embedding of `list`, $t' = \langle [] \rangle =$ `Const "List.list.Nil"`. Now, consider the rewriting of $\langle \mathsf{append} \rangle \$\ t_{xs}\ \$\ t_{ys}$ again:

$$\begin{aligned} \langle \mathsf{append} \rangle \$\ t_{xs}\ \$\ t_{ys} &\longrightarrow^* \langle \mathsf{append} \rangle \$\ \langle [] \rangle\ \$\ t_{ys} \\ &\longrightarrow^* t_{ys} \end{aligned}$$

Since $R \vdash t_{ys} \downarrow ys$ is known, the existential quantifier in the conclusion can be instantiated by $t' = t_{ys}$, solving the case.

For the case $xs = z\ \$\ zs$, the approach is similar. The only difference is that unfolding the definition of $\approx_{\texttt{list}}$ reveals information on both z and zs, meaning the final rewrite chain becomes longer.

5.4 Related work

Fallenstein and Kumar [34] have presented a model of HOL inside HOL including a reflection proof principle. The principle states that for any proposition φ, if the syntactic encoding of

φ is provable, then φ holds. This covers the entirety of of higher-order logic and requires the existence of a large cardinal. The work has been carried out in HOL4. In this thesis, I restrict myself to only the term-rewriting fragment of higher-order logic, meaning that the full generality of the reflection principle is not required.

Kunčar and Popescu [78, 79] introduce a mechanism to turn statements on types into statements on sets, exploiting the common set-theory-based semantics of higher-order logic. They perform a low-level dictionary construction using an extension of the logic. In this work, no extension is required; instead, derived constants are re-defined through the **function** package (§3.1).

6 Compiling HOL terms to CakeML expressions

"The time has come", the Walrus said,
"To talk of many things"

(Lewis Carroll, The Walrus and The Carpenter*)*

Contents

Portions of this chapter are based on the publication “A Verified Compiler from Isabelle/HOL to CakeML”, authored by Hupel and Nipkow [65].

In this section, I will discuss the progression from the de Bruijn based term language with its small-step semantics given in Listing 2.4.a to the final CakeML semantics. The compiler starts out with a term-rewrite system on type `term` and applies multiple phases to eliminate features that are not present in the CakeML source language. The types `term`, `nterm` and `pterm` each have a small-step semantics only. The type `sterm` has a small-step and several intermediate big-step semantics that bridge the gap to CakeML. An overview of the intermediate semantics and compiler phases is given in Figure 6.1. The left-hand column gives an overview of the different phases. The right-hand column gives the types of the rule set and the syntax and semantics for each phase.

6.1 Side conditions

All of the semantics presented in this chapter require some side conditions on the rule set. These conditions are purely syntactic. For example, these are the conditions for the correctness of the first compiler phase:

- Patterns must be linear, and constructors in patterns must be fully applied.
- Definitions must have at least one parameter on the left-hand side (§6.6).
- The right-hand side of an equation refers only to free variables occurring in patterns on the left-hand side and contains no dangling de Bruijn indices.
- There are no two defining equations $lhs = rhs_1$ and $lhs = rhs_2$ such that $rhs_1 \neq rhs_2$.
- For each pair of equations that define the same constant, their arity must be equal and their patterns must be compatible (§6.3).
- There is at least one equation.
- Variable names occurring in patterns must not overlap with constant names (§6.7).
- Any occurring constants must either be defined by an equation or be a constructor.

The conditions for the subsequent phases are sufficiently similar that I do not list them again.

In the formalization, I use locales to fix the rules and assumptions over them. Each phase has its own locale, together with a sublocale proof that after compilation, the conditions are preserved in the new semantics. Some of the correspondence relations are defined within the locale, so that they can access the fixed rules (and potentially additional parameters). For technical reasons, the semantics are usually defined outside of the locales. In the subsequent sections, the modelling as locales and the sublocale proofs are largely ignored, since they offer no new insights into the formalization.

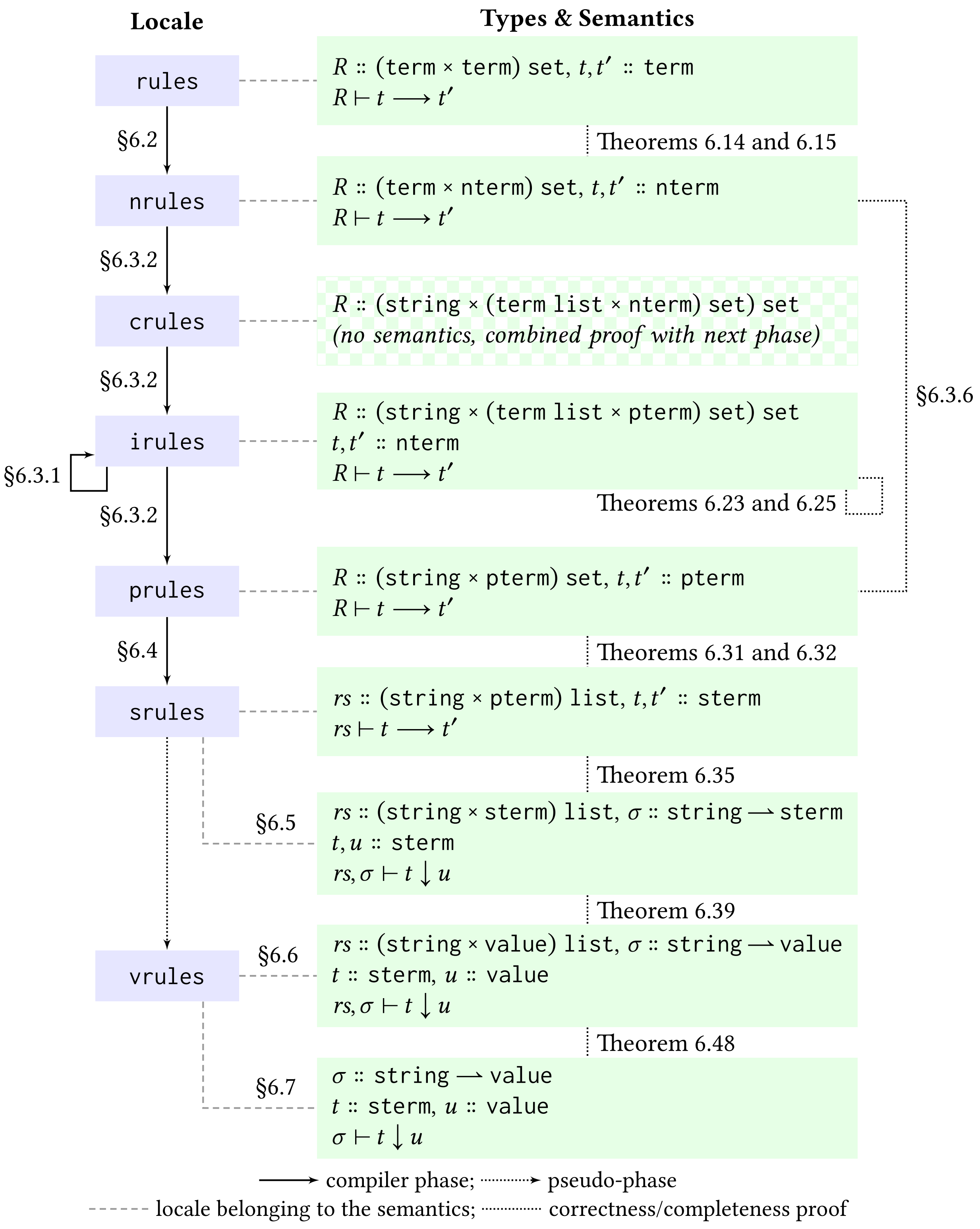

Locale
Types & Semantics
rules
R :: (term × term) set, t, t' :: term
$R \vdash t \longrightarrow t'$
§6.2
Theorems 6.14 and 6.15
nrules
R :: (term × nterm) set, t, t' :: nterm
$R \vdash t \longrightarrow t'$
§6.3.2
crules
R :: (string × (term list × nterm) set) set
(no semantics, combined proof with next phase)
§6.3.2
irules
R :: (string × (term list × pterm) set) set
t, t' :: nterm
$R \vdash t \longrightarrow t'$
§6.3.1
Theorems 6.23 and 6.25
§6.3.6
§6.3.2
prules
R :: (string × pterm) set, t, t' :: pterm
$R \vdash t \longrightarrow t'$
§6.4
Theorems 6.31 and 6.32
srules
rs :: (string × pterm) list, t, t' :: sterm
$rs \vdash t \longrightarrow t'$
Theorem 6.35
§6.5
rs :: (string × sterm) list, σ :: string ⇀ sterm
t, u :: sterm
$rs, \sigma \vdash t \downarrow u$
Theorem 6.39
vrules
§6.6
rs :: (string × value) list, σ :: string ⇀ value
t :: sterm, u :: value
$rs, \sigma \vdash t \downarrow u$
Theorem 6.48
§6.7
σ :: string ⇀ value
t :: sterm, u :: value
$\sigma \vdash t \downarrow u$
compiler phase; pseudo-phase
locale belonging to the semantics; correctness/completeness proof

Figure 6.1: Intermediate semantics and compiler phases

$$\textsc{Step}\ \frac{(\mathit{lhs}, \mathit{rhs}) \in R \qquad \texttt{match}\ \mathit{lhs}\ t = \mathsf{Some}\ \sigma}{R \vdash t \longrightarrow \texttt{subst}\ \sigma\ \mathit{rhs}} \qquad \textsc{Beta}\ \frac{}{R \vdash (\Lambda x.\ t)\ \$\ t' \longrightarrow \texttt{subst}\,[x \mapsto t']\ t}$$

$$\textsc{Fun}\ \frac{R \vdash t \longrightarrow t'}{R \vdash t\ \$\ u \longrightarrow t'\ \$\ u} \qquad \textsc{Arg}\ \frac{R \vdash u \longrightarrow u'}{R \vdash t\ \$\ u \longrightarrow t\ \$\ u'}$$

Listing 6.1: Small-step semantics with explicit bound variable names

6.2 Naming bound variables

Some proofs in this section have been contributed by Yu Zhang.

Portions of this section appear in the AFP entry “An Algebra for Higher-Order Terms”, authored by Hupel [60].

Isabelle uses de Bruijn indices in the term language for the following two reasons: First, for substitution, there is no need to rename bound variables. Second, α-equivalent terms are equal.

In implementations of programming languages, these advantages are not required: Typically, substitutions do not happen inside abstractions, and there is no notion of equality of functions. CakeML is no exception and therefore it uses named variables. In this compilation step, de Bruijn indices are removed.

The *named* semantics is based on the `nterm` type (Listing 4.5). The rules are given in Listing 6.1. Notably, β-reduction reuses the substitution function.

For the correctness proof, I establish a correspondence between `term`s and `nterm`s. Translation from `nterm` to `term` is trivial: Replace bound variables by the number of abstractions between occurrence and where they were bound in, and keep free variables as they are. This function is called `nterm_to_term`.

The other direction is not unique and requires introduction of *fresh* names for bound variables. In the formalization, I have chosen to use a *monad* to produce these names (§6.2.1). This function is called `term_to_nterm`.

Both functions should behave in such a way that conversion back and forth yields the same term. For converting from `term` to `nterm` and back, this is trivial to ensure (§6.2.2). However, the other direction is harder, because when converting from `nterm` to `term`, the original variable names get lost.

6.2.1 The `fresh` monad

Generation of fresh names in general can be thought of as picking a string that is not an element of a (finite) set of already existing names. For Isabelle, the *Nominal* framework [124, 125] provides support for reasoning over fresh names, but unfortunately, its definitions are not executable.

Instead, I chose to model generation of fresh names as a monad α `fresh` with the following high-level interface:

$$\begin{aligned} \texttt{Fresh.create} &:: \texttt{string fresh} \\ \texttt{Fresh.return} &:: \alpha \Rightarrow \alpha\,\texttt{fresh} \\ \texttt{Fresh.bind} &:: \alpha\,\texttt{fresh} \Rightarrow (\alpha \Rightarrow \beta\,\texttt{fresh}) \Rightarrow \beta\,\texttt{fresh} \\ \texttt{Fresh.run} &:: \alpha\,\texttt{fresh} \Rightarrow \texttt{string set} \Rightarrow \alpha \end{aligned}$$

`Fresh.returnFresh.runFresh.bind` With this, it becomes possible to write programs using **do**-notation. Note that such a program only tracks the known names via the argument passed into `run`; it is not possible to declare further names inside the program. This is sufficient for the compiler, because the set of all reserved names is known in advance: I take the union of all known constant names, including constructors.

In the formalization, this is implemented using the state monad, i.e., α `fresh` is a type synonym for a function `string` $\Rightarrow$ (α × `string`). Perhaps counter-intuitively, the state type is just a single `string` instead of a `string set`. It is used to track the highest used name according to the lexicographic order of strings.

Abstractly, this is modelled as a locale that expects two operations `next` :: $\alpha \Rightarrow \alpha$ and `arb` :: α with the axiom `next` $x > x$. The high-level `Fresh.create` operation can then be defined as $\lambda x.$ (`next` x, `next` x).

It remains to be explained how `run` can be implemented, i.e., how the highest used name is computed. The locale expects α to be a `linorder`. Using the `Max` :: $\alpha \Rightarrow \alpha$ `set` combinator from Isabelle's library and the `arb` operation, a derived operation to compute the successor of a set of names can be defined as follows:

```
definition Next :: α set ⇒ α where
Next S = (if S = ∅ then arb else next (Max S))
```

Consequently, `Fresh.run` is merely an abbreviation for first computing the successor of the input set of names, then running the state function with that successor.

Definition 6.1 (Freshness). *A name s' is fresh in a set S if all $s \in S$ are smaller than s'.*

Lemma 6.2. `Next` S *is fresh in* S*, i.e.,* $\forall s \in S.$ `Next` $S > s$.

Corollary 6.3. `Next` $S \notin S$.

This abstract specification of fresh name generation has multiple advantages:

- There may be multiple implementations for a single type; here, I chose suffixing an underscore to the highest name for computing the successor.
- All existing combinators and lemmas on the state monad can be reused.
- Reasoning about freshness of names can be reduced to monotonicity arguments.

The disadvantage is that it is harder to provide an implementation that uses a fixed scheme for fresh names, e.g. "a" with a suffixed number. In particular, it requires a modified linear order for names.

```
fun term_to_nterm :: string list ⇒ term ⇒ nterm fresh where
term_to_nterm _ (Const name) = Fresh.return (Nconst name)
term_to_nterm _ (Free name) = Fresh.return (Nvar name)
term_to_nterm Γ (Bound n) = Fresh.return (Nvar (Γ ! n))
term_to_nterm Γ (Λ t) = do {
  n ← Fresh.create;
  e ← term_to_nterm (n # Γ) t;
  Fresh.return (Λ n. e)
}
term_to_nterm Γ (t₁ $ t₂) = do {
  e₁ ← term_to_nterm Γ t₁;
  e₂ ← term_to_nterm Γ t₂;
  Fresh.return (e₁ $ e₂)
}

fun nterm_to_term :: string list ⇒ nterm ⇒ term where
nterm_to_term _ (Nconst name) = Const name
nterm_to_term Γ (Nvar name) = case find_first name Γ of
    Some n ⇒ Bound n
    None ⇒ Free name
nterm_to_term Γ (t $ u) = nterm_to_term Γ t $ nterm_to_term Γ u
nterm_to_term Γ (Λ x. t) = Λ nterm_to_term (x # Γ) t
```

Listing 6.2: Translations between terms and nterms

6.2.2 Translations between term and nterm

Both translations are recursive functions on the structure of the input terms, with an additional parameter Γ that records the context of bound variable names (Listing 6.2).

For terms that contain no abstractions, term_to_nterm coincides with convert_term for all Γ; the opposite direction coincides only for Γ = [].

Lemma 6.4. *Let t be a wellformed* term *(i.e., without dangling de Bruijn indices). Then, the translation of t from* term *to* nterm *is reversible:*

$$\texttt{nterm\_to\_term}\ []\ (\texttt{run}\ (\texttt{term\_to\_nterm}\ []\ t)\ S) = t,$$

where frees $t \subseteq S$.

The basic proof idea is to use induction over t after suitable generalization for an arbitrary context Γ.

As already indicated earlier, the correctness property of the opposite direction requires α-equivalence. Its definition is given in Listing 6.3. A slight deviation from literature is that I use an explicit context of renamings instead of substituting directly. This is merely for convenience. Two terms can be said to be α-equivalent if there is a context of renamings that transforms one into the other.

$$\frac{}{\Gamma \vdash \mathsf{Nconst}\ x \approx_\alpha \mathsf{Nconst}\ x} \qquad \frac{x \notin \mathsf{dom}\ \Gamma \qquad x \notin \mathsf{range}\ \Gamma}{\Gamma \vdash \mathsf{Nvar}\ x \approx_\alpha \mathsf{Nvar}\ x} \qquad \frac{\Gamma\ x = \mathsf{Some}\ y}{\Gamma \vdash \mathsf{Nvar}\ x \approx_\alpha \mathsf{Nvar}\ y}$$

$$\frac{\Gamma +\!\!+ [x \mapsto y] \vdash t \approx_\alpha u}{\Gamma \vdash \Lambda x.\ t \approx_\alpha \Lambda y.\ u} \qquad \frac{\Gamma \vdash t_1 \approx_\alpha u_1 \qquad \Gamma \vdash t_2 \approx_\alpha u_2}{\Gamma \vdash t_1\ \$\ t_2 \approx_\alpha u_1\ \$\ u_2}$$

Listing 6.3: α-equivalence between terms

Corollary 6.5 (Reflexivity). *For all terms t,* $[] \vdash t \approx_\alpha t$ *holds.*

The correctness property can now be phrased accordingly. Because the term that is translated may not be closed, it is necessary for this direction to take contexts into account:

Lemma 6.6. *Let t be a* `nterm` *and* Γ, Γ' *be contexts with the same length and* `frees` $t \subseteq \Gamma$. *Then, the translation of t from* `nterm` *to* `term` *(with context* Γ'*) and back (with context* Γ*) is α-equivalent to t. The corresponding renaming is the pairing of* Γ *and* Γ'*. Formally:*

$$\mathtt{map\_of}\ (\mathtt{zip}\ \Gamma\ \Gamma') \vdash \mathtt{Fresh.run}\ (\mathtt{term\_to\_nterm}\ \Gamma'\ (\mathtt{nterm\_to\_term}\ \Gamma\ t))\ S \approx_\alpha t$$

Note that this lemma also constructs the renaming. For closed terms, the following simplified corollary can be obtained:

Corollary 6.7. *Let t be a closed* `nterm`. *Then, the translation of t from* `nterm` *to* `term` *and back is α-equivalent to t:*

$$[] \vdash \mathtt{Fresh.run}\ (\mathtt{term\_to\_nterm}\ []\ (\mathtt{nterm\_to\_term}\ []\ t))\ S \approx_\alpha t$$

Furthermore, it is possible to prove that two translation runs from `term` to `nterm` with different contexts yield α-equivalent terms.

Lemma 6.8. *Let* Γ_1 *and* Γ_2 *be two contexts with the same length such that both* Γ_1 *and* Γ_2 *have no duplicate elements. Let t be a closed* `term`. *Let* s_1 *and* s_2 *be names that are fresh in* Γ_1 *and* Γ_2*, respectively. Then:*

$$\mathtt{map\_of}\ (\mathtt{zip}\ \Gamma_1\ \Gamma_2) \vdash \mathtt{fst}\ (\mathtt{term\_to\_nterm}\ \Gamma_1\ t\ s_1) \approx_\alpha \mathtt{fst}\ (\mathtt{term\_to\_nterm}\ \Gamma_2\ t\ s_2)$$

In the above statement, `term_to_nterm` is used as a function with three arguments, because `fresh` is implemented as a state monad, which is in turn a function of one argument. In this case, the third argument is a simple `string`. `Fresh.run` is not used, because it expects a `string set`.
An instantiated version of this lemma with $\Gamma_1 = \Gamma_2 = []$ would make little sense: it yields a special case of Corollary 6.5.

As usual, the proof proceeds by generalization and subsequent induction over the term.

In the remainder of this section, I will explain the relationship between terms and their translations with respect to matching and substitution.

Definition 6.9. *A* `term` *t and an* `nterm` *u are related wrt Γ if t =* `nterm_to_term` *Γ u.*

Corollary 6.10. *Definition 6.9 is a strong structural term relation (§4.2.4.3) for all Γ.*

From this corollary, it follows that matching behaves identically on terms and their translations. However, there are no generic lemmas over substitution (§4.2.4.3).

Lemma 6.11. *Let σ_1 and σ_2 be closed and related environments wrt Γ. If Γ and the domain of σ_1 are disjoint, then* `nterm_to_term` *and* `subst` *commute. Formally:*

$$\texttt{subst}\ \sigma_1\ (\texttt{nterm\_to\_term}\ \Gamma\ t) = \texttt{nterm\_to\_term}\ \Gamma\ (\texttt{subst}\ \sigma_2\ t)$$

The proof proceeds by induction on t and crucially depends on the σ_i being closed.

The next lemma establishes a connection between β-reduction (§2.6) and substitution:

Lemma 6.12. *If t' is closed, then:*

$$\texttt{nterm\_to\_term}\ \Gamma\ (\texttt{subst}\ [x \longmapsto t']\ t) = (\texttt{nterm\_to\_term}\ (x \# \Gamma)\ t)[\texttt{nterm\_to\_term}\ \Gamma\ t']$$

Informally speaking, this means that one can either substitute t' for x in t and then translate the resulting term back, or translate both t and t' back and then perform β-reduction.

6.2.3 Compilation, correctness, & completeness

Having established a low-level translation function from `term` to `nterm` that invents bound variable names based on names from a context, it remains to explain how full rule sets are compiled. The idea is rather straightforward: translate the right-hand side of all rules while leaving the left-hand sides unchanged. The left-hand side is still represented as `term`, for the reason outlined in §4.2.2.

Formally, compilation is defined as follows:

$$\texttt{compile}\ R = \{(p, \texttt{Fresh.run}\ (\texttt{term\_to\_nterm}\ []\ t)\ (\texttt{all\_consts} \cup \texttt{frees}\ p)) \mid (p, t) \in R\}$$

The fresh monad receives two sets of known names. Naturally, names bound by the patterns must be avoided (`frees` p). To avoid collisions of names further down in the compilation pipeline, additionally all the known constant names are supplied (`all_consts`). Recall that that set comprises two subsets (Listing 5.2):

- the names of all constructors from the `constructors` locale
- the names of all equation heads (`heads_of` R)

The side conditions of the locale (§6.1) ensure that `heads_of` R is equal to the set `heads` defined during deep embedding (Listing 5.2).

Corollary 6.13 (Invariance of heads). *The heads of R remain unchanged after compilation.*

Theorem 6.14 (Correctness of compilation)**.** *Assuming a step can be taken with the compiled rule set, this step can be reproduced with the original rule set.*

$$\frac{\texttt{compile}\ R \vdash t \longrightarrow t' \qquad \texttt{closed}\ t}{R \vdash \texttt{nterm\_to\_term}\ t \longrightarrow \texttt{nterm\_to\_term}\ t'}$$

Proof. By rule induction over the semantics (Listing 6.1). The interesting cases are Step and Beta.

Step This follows from Corollary 6.10 and Lemma 6.11.

Beta This follows from Lemma 6.12. □

Theorem 6.15 (Completeness of compilation)**.** *Assuming a step can be taken in the original rule set, this step can be reproduced in the compiled rule set to obtain an α-equivalent term. Formally: if $R \vdash t \longrightarrow t'$ where t is closed and wellformed, then there is a term u' such that:*

$$\texttt{compile}\ R \vdash \texttt{term\_to\_nterm}\ []\ t\ s \longrightarrow u' \wedge [] \vdash u' \approx_\alpha \texttt{term\_to\_nterm}\ []\ t'\ s'$$

6.3 Explicit pattern matching

In this step, rule sets are transformed from implicit to explicit pattern matching (§4.3.3). This is an iterative algorithm that requires two fundamental invariants: all defining equations of one function must

1. have the same number of parameters (guaranteed by the **function** command during preprocessing) and
2. be pattern-compatible (Definition 3.4)

If either invariant is violated, the result of an iteration is unspecified; therefore, they form additional assumptions (§6.1).

Defining equations after elimination of implicit patterns will have the form ⟨f⟩ = Λ C, where C is a set of clauses. The right-hand side contains n nested abstractions, where n is the number of parameters of the defining equations of f.

The implementation strategy requires successive elimination of a single parameter from right to left, in a similar fashion as Slind's pattern matching compiler [113, §3.3.1]. There is an alternative implementation that will be discussed in §6.3.6. Explanation of the semantics in this phase is dependent upon understanding of the elimination procedure, which is why I defer this to §6.3.2.

The running example in this section is the map function from §5.1. It has arity 2. I omit the brackets ⟨⟩ for brevity. Recall its definition:

$$\begin{array}{llll} \texttt{map} & f & [] & = [] \\ \texttt{map} & f & (x \mathbin{\#} xs) & = f\,x \mathbin{\#} \texttt{map}\ f\ xs \end{array}$$

First, the list parameter is eliminated:

$$\begin{aligned}\texttt{map}\ f = \lambda\ [] &\Longrightarrow [] \\ \mid x \# xs &\Longrightarrow f\ x \# \texttt{map}\ f\ xs\end{aligned}$$

Finally, the function parameter is eliminated:

$$\begin{aligned}\texttt{map} = \lambda\ f \Longrightarrow (\lambda\ [] &\Longrightarrow [] \\ \mid x \# xs &\Longrightarrow f\ x \# \texttt{map}\ f\ xs)\end{aligned}$$

This final equation has arity zero and is defined by a twice-nested abstraction.

From this example, it becomes obvious why the first invariant has to hold: the elimination procedure would get stuck for jagged arrays. The second invariant is more subtle. Consider an equivalent definition of map, where the function parameter has a different name in the second equation:

$$\begin{array}{llll}\texttt{map} & f & [] & = [] \\ \texttt{map} & g & (x \# xs) & = g\ x \# \texttt{map}\ g\ xs\end{array}$$

Through elimination, this would turn into:

$$\begin{aligned}\texttt{map} = \lambda\ f &\Longrightarrow (\lambda\ [] \Longrightarrow []) \\ \mid g &\Longrightarrow (\lambda\ x \# xs \Longrightarrow f\ x \# \texttt{map}\ f\ xs)\end{aligned}$$

Even though the original equations were non-overlapping, the resulting abstraction has overlapping patterns. Slind observed a similar problem [113, §3.3.2] in his algorithm. Therefore, he only permits *uniform* equations, as defined by Wadler [106, §5.5]. In the formalization, I am able to give a formal characterization of the requirements as a computable function on pairs of patterns (§4.2.4.5). This compatibility constraint ensures that any two overlapping patterns at the same parameter position are equal and are thus appropriately grouped together in the elimination procedure.

While this rules out some theoretically possible pattern combinations (see also §3.5), in practice, I have not found this to be a problem. If a function's parameters cannot be renamed accordingly, users can transform a set of equations into a single equation using **case** combinators with the **case_of_simps** command by Noschinski and Klein.

Simps_Case_Conv

6.3.1 Elimination procedure

The elimination procedure can be described as an iterative matrix transformation. Functions are processed one at a time. In a similar fashion as Slind's pattern matching compiler [113, §3.3.1], I view the set of defining equations of c as a matrix where each row represents an equation.

$$\begin{array}{lccl}\texttt{c} & p_{1,1} \cdots p_{1,n} & = & rhs_1 \\ \texttt{c} & p_{2,1} \cdots p_{2,n} & = & rhs_2 \\ & & \vdots & \\ \texttt{c} & p_{m,1} \cdots p_{m,n} & = & rhs_m\end{array} \rightsquigarrow \left(\begin{array}{cccc|c} p_{1,1} & p_{1,2} & \cdots & p_{1,n} & rhs_1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} & rhs_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} & rhs_m\end{array}\right)$$

Definition 6.16 (Arity)**.** *The* arity *of a function is the unique number of patterns in each of its defining equations, if it exists, or unspecified otherwise. Two equations are* arity-compatible *if they define different functions or have the same number of patterns.*

Applied to the above example, the arity of `c` is n.

In each step, the equations are grouped by the initial $n-1$ patterns (the *prefix*). More formally, I treat a row in the matrix as an $(n+1)$-tuple $(p_{i,1}, \dots, p_{i,n}, rhs_i)$ and define an equivalence relation $\equiv_p$ such that

$$(p_{i,1}, \dots, p_{i,n}, rhs_i) \equiv_p (p_{j,1}, \dots, p_{j,n}, rhs_j) \iff (p_{i,1}, \dots, p_{i,n-1}) = (p_{j,1}, \dots, p_{j,n-1})$$

The new matrix is constructed from the set of equivalence classes. It consists of n columns and one row per equivalence class. The first $n-1$ columns contain the unique $p_{i,1}, \dots, p_{i,n-1}$, whereas the last column is an abstraction with the set S_i of all $(p_{k,n}, rhs_k)$ in the equivalence class as the set of clauses.

$$\left(\begin{array}{cccc|c} p'_{1,1} & p'_{1,2} & \cdots & p'_{1,n-1} & \Lambda\, S_1 \\ p'_{2,1} & p'_{2,2} & \cdots & p'_{2,n-1} & \Lambda\, S_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p'_{m',1} & p'_{m',2} & \cdots & p'_{m',n-1} & \Lambda\, S_{m'} \end{array}\right)$$

Lemma 6.17 (Invariant)**.** *Given a set of arity- and pattern-compatible equations, the elimination procedure produces another set of arity- and pattern-compatible equations.*

Proof. Consider a set of equations with the same head `c`. Assume arity of `c` is nonzero.

1. Obvious from the matrix construction.
2. Let R_{c} :: `(term list × pterm) set` be all equations for `c`. Also, let R'_{c} be the set after a single elimination. To establish pattern compatibility of a set, pick any two equations (ps_1, rhs_1) and (ps_2, rhs_2) in R'_{c} and prove their pattern compatibility. Consider the origin of these two equations: There must be (at least) one equation $(ps_i \mathbin{@} [p'_i], rhs'_i) \in R_{\mathsf{c}}$ that gave rise to (ps_1, rhs_1) for $i \in \{1, 2\}$. Consequently, using the invariant for R_{c}, $ps_1 \mathbin{@} [p'_1]$ and $ps_2 \mathbin{@} [p'_2]$ must be compatible.

   It remains to be proved that ps_1 and ps_2 are compatible. This follows directly from the definition of `rev_accum_rel` (§4.2.4.5). □

Lemma 6.18 (Termination)**.** *For each function in a rule set R, its arity is zero or decreases by one after a single elimination step.*

Corollary 6.19. *Let* `c` *be the function with the maximum arity n in a rule set R. Eliminating patterns in R for n times yields a rule set R′ where each function has arity zero.*

6.3.2 Compilation

After transformation to `nterm`, equations are represented as a tuple `term × nterm`. But recall from Figure 6.1 that the type of R in this phase is `(string × (term list × pterm) set) set`. The full compilation goes through multiple sub-phases:

$$\textsc{Step}\ \frac{(name, \mathtt{C}) \in R \qquad ([p_1, \dots, p_n], rhs) \in C \qquad \mathtt{match}\ (\mathtt{Pconst}\ name\ \$\ p_1\ \$\ \dots\ \$\ p_n)\ t = \mathsf{Some}\ \sigma}{R \vdash t \longrightarrow \mathtt{subst}\ \sigma\ rhs}$$

$$\textsc{Beta}\ \frac{(pat, rhs) \in \mathtt{C} \qquad \mathtt{match}\ pat\ t = \mathsf{Some}\ \sigma \qquad \mathtt{closed}\ t}{R \vdash (\Lambda\ \mathtt{C})\ \$\ t \longrightarrow \mathtt{subst}\ \sigma\ rhs}$$

$$\textsc{Fun}\ \frac{R \vdash t \longrightarrow t'}{R \vdash t\ \$\ u \longrightarrow t'\ \$\ u} \qquad \textsc{Arg}\ \frac{R \vdash u \longrightarrow u'}{R \vdash t\ \$\ u \longrightarrow t\ \$\ u'}$$

(6.4.a) Combined implicit and explicit pattern matching

$$\textsc{Step'}\ \frac{(name, rhs) \in R}{R \vdash \mathtt{Pconst}\ name \longrightarrow rhs}$$

(6.4.b) Modified Step rule for explicit-only pattern matching

Listing 6.4: Small-step semantics with pattern matching

1. The left-hand side of each equation (of type `term`) is destructured into a tuple (*name*, *pats*) :: `string` × `term list`, where *name* represents the head of the equation and *pats* the list of patterns.
2. The right-hand side (of type `nterm`) can be trivially embedded into `pterm`: An `nterm`-abstraction $\Lambda x.\ t$ is translated to the `pterm`-abstraction $\Lambda\{\langle x\rangle \Longrightarrow t\}$, i.e., a case abstraction with the single clause (`Pvar` x, t).
3. Equations with the same head are grouped together. Different groups are processed separately. A single group has the type (`term list` × `pterm`) `set`. This corresponds to the initial matrix representation as depicted in §6.3.1.
4. For each function (i.e. for each group) that has an arity greater than zero, the elimination procedure is applied. The type does not change here. This can be iterated until all functions have arity zero (Corollary 6.19).

For correctness and completeness purposes, all but the last step have purely syntactic proofs, that is, they do not require arguing about the semantics.

The target semantics has two variants and is given in Listing 6.4. The first variant with the Step rule performs both implicit and explicit pattern matching, to account for functions that may have nonzero arity.

The second variant applies after a post-processing step. When all functions in the rule set are of arity zero, the implicit pattern lists are all empty. Consequently, the rule set can be trivially converted to type (`string` × `pterm`) `set` by stripping the pattern lists. To accommodate for this, the modified Step' rule merely replaces a constant by its definition, without taking arguments into account. Only the Beta rule performs pattern matching.

$$\textsc{Const}\ \frac{}{\texttt{Pconst}\ n \approx_{\mathrm{p}} \texttt{Pconst}\ n} \qquad \textsc{Var}\ \frac{}{\texttt{Pvar}\ n \approx_{\mathrm{p}} \texttt{Pvar}\ n} \qquad \textsc{Comb}\ \frac{t_1 \approx_{\mathrm{p}} t_2 \qquad u_1 \approx_{\mathrm{p}} u_2}{t_1 \mathbin{\$} u_1 \approx_{\mathrm{p}} t_2 \mathbin{\$} u_2}$$

$$\textsc{Ext}\ \frac{\texttt{rel}\ (\lambda(p_1, t_1)\ (p_2, t_2).\ p_1 = p_2 \wedge t_1 \approx_{\mathrm{p}} t_2)\ \texttt{C}_1\ \texttt{C}_2}{\Lambda\ \texttt{C}_1 \approx_{\mathrm{p}} \Lambda\ \texttt{C}_2}$$

$$\textsc{Defer}\ \frac{\texttt{arity}\ R_f > 0 \qquad \begin{array}{c} \forall i.\ \texttt{closed}\ t_i \qquad (f, R_f) \in R \\ \texttt{rel}\ (\lambda(p_1, t_1)\ (p_2, t_2).\ p_1 = p_2 \wedge t_1 \approx_{\mathrm{p}} t_2)\ (\texttt{deferred}\ [t_1, \ldots, t_n]\ R_f)\ \texttt{C} \end{array}}{\texttt{Pconst}\ f \mathbin{\$} t_1 \mathbin{\$} \ldots \mathbin{\$} t_n \approx_{\mathrm{p}} \Lambda\ \texttt{C}}$$

$$\begin{aligned} \texttt{deferred}\ ts\ R_f = \{(p_{n+1}, \texttt{subst}\ \sigma\ rhs) \mid \\ ([p_1, p_2, \ldots, p_{n+1}], rhs) \in R_f \wedge \texttt{matchs}\ [p_1, \ldots, p_n]\ ts = \texttt{Some}\ \sigma\} \end{aligned}$$

Listing 6.5: Left-deferred correspondence

The remainder of this section is concerned with the proofs for the elimination procedure. I will briefly revisit the initial and post-processing steps in §6.3.6.

6.3.3 Correspondence relation

The statement of the semantic correctness property is more difficult than in the previous phase. The obvious property does not hold:

$$\frac{\texttt{compile}\ R \vdash t \longrightarrow u \qquad \texttt{closed}\ t}{R \vdash t \longrightarrow u}$$

Consider the `map` function again. After eliminating once, the defining equation of `map` is of the form ⟨`map` f⟩ = Λ C, which means that the term ⟨`map id`⟩ can be rewritten to Λ C. However, this rewrite step cannot be taken in the original rule set, because the second argument is missing.

Because of the absence of a direct correspondence, I have introduced a relation $\approx_{\mathrm{p}}$ (Section 6.3.3). The ultimate goal is that the following correctness property holds:

$$\frac{\texttt{compile\_single}\ R \vdash u \longrightarrow u' \qquad t \approx_{\mathrm{p}} u \qquad \texttt{closed}\ t}{\exists t'.\ R \vdash t \longrightarrow^{*} t' \wedge t' \approx_{\mathrm{p}} u'}$$

where `compile_single` is a single application of the elimination procedure. For this to work, $\approx_{\mathrm{p}}$ must take the rule set into account.

I will illustrate the meaning of this relation based on the `map` example. Consider its matrix representation:

$$R_{\texttt{map}} = \left(\begin{array}{cc|c} \langle f \rangle & \langle [] \rangle & \langle [] \rangle \\ \langle f \rangle & \langle x \mathbin{\#} xs \rangle & \langle f\ x \mathbin{\#} \texttt{map}\ f\ xs \rangle \end{array} \right)$$

After an elimination step:

$$R'_{\mathsf{map}} = \left(\langle f \rangle \;\middle|\; \left\langle \begin{array}{l} \lambda\, [] \Rightarrow [] \\ \mid x \,\#\, xs \Rightarrow (\lambda y.\ y)\ x \,\#\, \mathsf{map}\ (\lambda y.\ y)\ xs \end{array} \right\rangle \right)$$

In this modified rule set, the term $\langle \mathsf{map}\ (\lambda y.\ y) \rangle$ can be rewritten. In the original rule set, it cannot. I refer to such a rewriting as a *half-step*. Using the Defer rule, the un-reduced and the reduced term are related:

$$\langle \mathsf{map}\ (\lambda y.\ y) \rangle \approx_{\mathrm{p}} \left\langle \begin{array}{l} \lambda\, [] \Rightarrow [] \\ \mid x \,\#\, xs \Rightarrow (\lambda y.\ y)\ x \,\#\, \mathsf{map}\ (\lambda y.\ y)\ xs \end{array} \right\rangle$$

The Defer rule can be explained by examining the `deferred` function. Given a function application for n parameters $\langle \mathsf{c}\ t_1\ \ldots\ t_n \rangle$ of a function with arity $n + 1$, it selects all defining equations that so far match these arguments (minus the n + 1-st one). Each of these equations $\langle \mathsf{c}\ p_1\ \ldots\ p_n\ p_{n+1} = t \rangle$ carries an additional pattern p_{n+1} for the $(n + 1)$st argument which has to be supplied eventually. From these equations, I construct a Λ-abstraction comprising pairs $(\langle p_{n+1} \rangle, \mathsf{subst}\ \sigma \langle t \rangle)$, where σ is the result of matching the initial n patterns.

Based on this understanding, the $\approx_{\mathrm{p}}$ can be described as a *left-deferred* or a *right-extensional* correspondence.

In the case of abstraction-free terms, the relation collapses to equality:

Lemma 6.20. *If $t \approx_{\mathrm{p}} u$ and u is abstraction-free, then $t = u$.*

Proof. By rule induction on $\approx_{\mathrm{p}}$. Neither the Ext nor the Defer rule are applicable; consequently, only Const, Var, and Comb are left. They describe an equality relation. □

This lemma becomes important when considering the iterated elimination, because $\approx_{\mathrm{p}}$ is not transitive.

Lemma 6.21. *$\approx_p$ is reflexive. $\approx_p$ is a structural term relation (§4.2.4.3).*

$\approx_{\mathrm{p}}$ is an example of a structural term relation that is not also a strong structural relation: If t and u are related through the Defer rule, then t may be an application or a constant, whereas u is an abstraction, hence having different shapes. Consequently, Lemma 4.24 would be violated.

Lemma 6.22 (Substitution). *Let t_1 and t_2 be related terms. Also, let σ_1 and σ_2 be closed and related environments. Then, the results of substitution are also related. Formally:* $\mathsf{subst}\ \sigma_1\ t_1 \approx_{\mathrm{p}} \mathsf{subst}\ \sigma_2\ t_2$.

6.3.4 Correctness

Recall the desired correctness property as stated in §6.3.3:

Theorem 6.23.

$$\frac{\mathsf{compile\_single}\ R \vdash u \longrightarrow u' \qquad t \approx_{\mathrm{p}} u \qquad \mathsf{closed}\ t}{\exists t'.\ R \vdash t \longrightarrow^{*} t' \wedge t' \approx_{\mathrm{p}} u'}$$

Observe that the conclusion uses the starred rewrite relation, i.e., its reflexive-transitive closure. The reason is simple: one step in the transformed rule set may correspond to zero or one step in the original rule set.

Proof. By rule induction over the semantics (Listing 6.4). Similarly to the proof of Theorem 6.14, the interesting cases are Step and Beta.

Step In this case, u is an application of a constant to a number of arguments. It holds that `compile_single` $R \vdash \langle \mathsf{c}\ u_1\ \dots\ u_n \rangle \longrightarrow u'$ for some u' and u_i. Case distinction on c's arity in R:

- If c already had arity zero, nothing changed during compilation. In particular, there is a rule in R that matches and can rewrite the function application to u'. The same rewrite step needs to be taken from a t where $t \approx_{\mathrm{p}} u$. This follows from Lemmas 6.21 and 6.22. Consequently, $R \vdash \langle \mathsf{c}\ t_1\ \dots\ t_n \rangle \longrightarrow t'$ where $t' \approx_{\mathrm{p}} u'$ with a single rewrite step.
- If not, the arity of c decreased. This case requires establishing the correctness of the `deferred` set: the transformed rule set is able to perform a half-step, whereas R cannot. Hence, set $t' = t$ with $R \vdash t \longrightarrow^* t'$ because of the reflexive-transitive closure. It remains to be shown that $t \approx_{\mathrm{p}} u'$. This requires Lemma 6.22 and Corollary 4.36.

Beta In this case, u is an application where the function is a case abstraction, i.e., there is an x such that $u = (\Lambda\ C)\ \$\ x$ and `compile_single` $R \vdash (\Lambda\ C)\ x \longrightarrow u'$ for some u'. u' is the result of matching x to a pattern p in C and subsequent substitution. Now, t is a term such that $t \approx_{\mathrm{p}} (\Lambda\ C)\ \$\ x$. Using the Comb rule, there must be t_1 and t_2 with $t = t_1\ \$\ t_2$ and $t_1 \approx_{\mathrm{p}} \Lambda\ C$ and $t_2 \approx_{\mathrm{p}} x$. There are two possible cases in $\approx_{\mathrm{p}}$ that could give rise to $t_1 \approx_{\mathrm{p}} \Lambda\ C$:

Ext This is the simple case, because it reveals that t_1 has an identical structure to $\Lambda\ C$. Hence, there is an equivalent pattern p' in t_1 as in C. The result follows from Lemmas 6.21 and 6.22.

Defer The situation is that t_1 is an application of a constant to a list of arguments. The rewrite step could not be completed because the number of arguments is one less than the arity of that constant. However, since $t = t_1\ \$\ t_2$, the step can now be completed, because t_2 is available as an extra argument. On the other side, $\Lambda\ C$ are precisely the clauses that have been deferred. It now remains to find the matching pattern p in C and establish the existence of a corresponding equation in R. The technical challenge is that the right-hand sides in C have already been partially substituted by the half-step. This can be solved by using Lemmas 4.20 and 4.29, which allow to split the matching and substitution into two parts: first the n-ary prefix and then the additional pattern p. □

Combining this result with Lemma 6.20 yields:

Corollary 6.24 (Correctness for abstraction-free results).

$$\frac{\texttt{compile\_single}\ R \vdash t \longrightarrow u \qquad \texttt{closed}\ t \qquad \texttt{no\_abs}\ u}{R \vdash t \longrightarrow^* u}$$

This corollary can be lifted to `compile`, i.e., the iterated application of `compile_single`.

6.3.5 Completeness

The proof in this section has been contributed by Yu Zhang.

The opposite direction can be stated directly, without the use of an additional relation:

Theorem 6.25.

$$\frac{R \vdash t \longrightarrow t' \qquad \texttt{closed}\ t}{\texttt{compile\_single}\ R \vdash t \longrightarrow^* t'}$$

Whereas the correctness requires the original semantics to perform zero or one step, completeness requires the new semantics to perform one or two steps: if the arity of one constant has been reduced during pattern elimination, both half-steps need to be executed in order. This is also the idea of the proof for the Step case; the other cases are trivial.

6.3.6 Discussion

This compilation phase is both non-trivial and has some minor restrictions on the set of function definitions that can be processed. Instead of eliminating patterns from right to left, patterns could also alternatively be grouped into tuples. The `map` example would be translated into:

$$\begin{aligned}\texttt{map} = \lambda\ &(f, []) \Longrightarrow [] \\ |\ &(f, x \mathbin{\#} xs) \Longrightarrow f\ x \mathbin{\#} \texttt{map}\ f\ xs\end{aligned}$$

The compilation of patterns would then be left for the CakeML compiler, which has no pattern compatibility restriction.

Despite the simpler idea behind the algorithm, there are two disadvantages:

- the compiler phase would require the knowledge of a tuple type in the term language, which is otherwise unaware of concrete datatypes
- the correspondence relation would be harder to specify, because the alternative translation goes directly from arity n to 0.

Finally, I will review the complexity of the proofs in this section. Recall the detailed structure of this compiler phase as outlined in §6.3.2.

The vast majority of the complexity is caused by the iterative elimination procedure. The other parts, namely the massaging of `nterms` to arrive at a syntactically equivalent rule set of `pterms`, and similarly to remove empty implicit pattern lists, only require some basic reasoning on sets. But all steps share the requirement for classical logic. Frequently, I have resorted to using choice operators, for example, when determining the arity of a function definition. Nonetheless, the entire routine is executable (§2.4).

$$\textsc{Step}\ \frac{(name, rhs) \in \texttt{set}\ rs}{rs \vdash \texttt{Sconst}\ name \longrightarrow rhs} \qquad \textsc{Beta}\ \frac{\texttt{find\_match}\ cs\ t = \texttt{Some}\ (\sigma, rhs)}{rs \vdash (\Lambda\ cs)\ \$\ t \longrightarrow \texttt{subst}\ \sigma\ rhs}$$

$$\textsc{Fun}\ \frac{rs \vdash t \longrightarrow t'}{rs \vdash t\ \$\ u \longrightarrow t'\ \$\ u} \qquad \textsc{Arg}\ \frac{rs \vdash u \longrightarrow u'}{rs \vdash t\ \$\ u \longrightarrow t\ \$\ u'}$$

Listing 6.6: Small-step semantics with ordered clauses

6.4 Sequentialization

Some proofs in this section have been contributed by Yu Zhang.

The semantics of `pterm` and `sterm` (Listing 6.6) differ only in rule Beta. Instead of any matching clause, the first matching clause in a case abstraction is picked. For technical reasons, the Step rule is phrased using an additional inductive relation in the formalization. Because the rule set is now a rule list, I use the naming convention *rs* instead of *R*.

For the correctness proof, the order of clauses does not matter: I only need to prove that a step taken in the sequential semantics can be reproduced in the unordered semantics. As long as no rules are dropped, this is trivially true. For that reason, the compiler can choose an arbitrary ordering.

This semantics only sequentializes pattern matching: Rewriting may still be non-deterministic because if there are multiple redexes in a term, either can be picked.

6.4.1 Translations between pterm and sterm

Similarly to §6.2.2, both directions are recursive functions on the structure of the input terms (Listing 6.7). However, no name context is required. For terms that contain no abstractions, both directions also coincide with `convert_term`.

I will first explain the technically more interesting conversion from `pterm` to `sterm`, as this requires imposing an ordering on the unordered set of clauses. For simplicity, I have chosen to use the lexicographic order on the patterns, i.e., `terms`. Then, the problem can be generalized to finding a function $((\alpha :: \texttt{linorder}) \times \beta)\ \texttt{set} \Rightarrow (\alpha \times \beta)\ \texttt{list}$.

```
definition ordered_map :: (α::linorder × β) set ⇒ (α × β) list where
ordered_map S = [
  (k, the_elem {(k', v) | (k', v) ∈ S ∧ k = k'}) |
  k ← sorted_list_of_set {k | (k, v) ∈ S}]
]
```

The functions `the_elem` :: α `set` $\Rightarrow \alpha$ and `sorted_list_of_set` :: α :: `linorder set` $\Rightarrow$ α `list` are defined in the Isabelle library and return the element of a singleton set and the sorted list of elements of a set. Using these functions, `ordered_map` first produces an ordered list of keys, which are then associated with the values from the input set.

```
fun pterm_to_sterm :: pterm ⇒ sterm where
pterm_to_sterm (Pconst name) = Sconst name
pterm_to_sterm (Pvar name) = Svar name
pterm_to_sterm (t $ u) = pterm_to_sterm t $ pterm_to_sterm u
pterm_to_sterm (Pabs cs) = Sabs (ordered_map {(p, pterm_to_sterm t) | (p, t) ∈ cs})

fun sterm_to_pterm :: sterm ⇒ pterm where
sterm_to_pterm (Sconst name) = Pconst name
sterm_to_pterm (Svar name) = Pvar name
sterm_to_pterm (t $ u) = sterm_to_pterm t $ sterm_to_pterm u
sterm_to_pterm (Sabs cs) = Pabs (set (map (map_prod id sterm_to_pterm) cs))
```

Listing 6.7: Translations between `pterm`s and `sterm`s

Naturally, this requires that the input set is a set of pairs where the first elements are unique. This is guaranteed by a side condition (§6.1). Otherwise, the result is unspecified.

Lemma 6.26. *For suitable S,* `set (ordered_map` S`)` $= S$.

The opposite direction is much simpler and has already been explained in §4.3.4.

Corollary 6.27. *For wellformed t,* `sterm_to_pterm (pterm_to_sterm` t`)` $= t$.

The reverse property does not hold directly, because an `sterm` may contain non-ordered clause lists. This property would require some form of α-equivalence that disregards ordering.

Similarly to §6.2.2, I have used the translation functions to establish term correspondence.

Definition 6.28. *A* `pterm` *t is related to an* `sterm` *u if t =* `sterm_to_pterm` *u.*

Definition 6.29. *An* `sterm` *u is related to a* `pterm` *t if* `pterm_to_sterm` *t = u.*

Corollary 6.30. *Definitions 6.28 and 6.29 are strong structural term relations (§4.2.4.3).*

Corresponding versions of Lemma 6.11 can also be proved for these relations.

6.4.2 Compilation, correctness, & completeness

At the same time the terms are translated, the rules also have to be converted from type `(string × pterm) set` to `(string × sterm) list`. Fortunately, the same `ordered_map` function can be used for that, because `string :: lexorder`:

```
definition compile :: (string × pterm) set ⇒ (string × sterm) list where
compile R = ordered_map {(name, pterm_to_sterm t) | (name, t) ∈ R}
```

This looks very similar to the `Pabs` case in `pterm_to_sterm` (Listing 6.7).

$$\textsc{Const}\ \frac{(name, rhs) \in rs}{rs, \sigma \vdash \texttt{Sconst}\ name \downarrow rhs} \qquad \textsc{Var}\ \frac{\sigma\ name = \texttt{Some}\ v}{rs, \sigma \vdash \texttt{Svar}\ name \downarrow v}$$

$$\textsc{Abs}\ \frac{}{rs, \sigma \vdash \Lambda\ cs \downarrow \Lambda\, [(pat, \texttt{subst}\ (\sigma - \texttt{frees}\ pat)\ t \mid (pat, t) \leftarrow cs]}$$

$$\textsc{Comb}\ \frac{\begin{array}{c} rs, \sigma \vdash t \downarrow \Lambda\ cs \\ rs, \sigma \vdash u \downarrow u' \qquad \texttt{find\_match}\ cs\ u' = \texttt{Some}\ (\sigma', rhs) \qquad rs, \sigma +\!\!+\ \sigma' \vdash rhs \downarrow v \end{array}}{rs, \sigma \vdash t \mathbin{\$} u \downarrow v}$$

$$\textsc{Constr}\ \frac{name \in \mathrm{C} \qquad rs, \sigma \vdash t_1 \downarrow u_1 \qquad \cdots \qquad rs, \sigma \vdash t_n \downarrow u_n}{rs, \sigma \vdash \texttt{Sconst}\ name \mathbin{\$} t_1 \mathbin{\$} \ldots \mathbin{\$} t_n \downarrow \texttt{Sconst}\ name \mathbin{\$} u_1 \mathbin{\$} \ldots \mathbin{\$} u_n}$$

Listing 6.8: Big-step semantics for `sterm`

Observe that the `ordered_map` function is used twice in this section, but with different type instantiations. This is the reason why this compiler phase and the proofs are not parametrized on the concrete sequentialization strategy, because locales in Isabelle do not allow polymorphic parameters.

Luckily, both the correctness and the completeness property of this phase are easily stated and proved:

Theorem 6.31 (Correctness).

$$\frac{\texttt{compile}\ R \vdash t \longrightarrow u}{R \vdash \texttt{sterm\_to\_pterm}\ t \longrightarrow \texttt{sterm\_to\_pterm}\ u}$$

Proof. By rule induction on the new semantics (Listing 6.6). As described in the introduction, any step taken in the sequential semantics can trivially be taken in the unordered semantics too. □

Theorem 6.32 (Completeness).

$$\frac{R \vdash t \longrightarrow u}{\texttt{compile}\ R \vdash \texttt{pterm\_to\_sterm}\ t \longrightarrow \texttt{pterm\_to\_sterm}\ u}$$

Proof. By rule induction on the previous semantics (Listing 6.4.b). The challenge here lies within the Beta case, where it needs to be proved that the arbitrary step taken by the previous semantics also corresponds to the first matching clause in the ordered term. But this is guaranteed by Lemma 4.33. □

6.5 Big-step semantics

This big-step semantics for `sterm` is not a compiler phase but moves towards the desired evaluation semantics. In this step, I reuse the `sterm` type for evaluation results, instead of evaluating to the separate type `value` (§4.3.5). This means that I can ignore environment capture in closures for now.

All previous $\longrightarrow$ relations were parametrized by a rule set. Now the big-step predicate is of the form $rs, \sigma \vdash t \downarrow t'$ where $\sigma :: \texttt{string} \rightharpoonup \texttt{sterm}$ is a variable environment. The intended invariant is that evaluated terms are closed when σ is closed.

This semantics also requires the distinction between constructors and defined constants, which I have already touched on in §5.1. If `c` is a constructor, the term $\langle \texttt{c}\ t_1\ \dots\ t_n \rangle$ is evaluated to $\langle \texttt{c}\ t'_1\ \dots\ t'_n \rangle$ where the t'_i are the results of evaluating the t_i. This imposes additional restrictions on the rule set: A constant must not be a constructor and a definition at the same time. The semantics is parametrized on the set `C` of all data constructors via the `constructors` locale (Listing 5.2).

The full set of rules is shown in Listing 6.8. They deserve a short explanation:

Const Constants are retrieved from the rule set *rs*.

Var Variables are retrieved from the environment σ.

Abs In order to achieve the intended invariant, abstractions are evaluated to their fully substituted form. Observe the obvious similarity between this rule and the definition of substitution for `sterm` (§4.3.4).

Comb Function application $t\ \$\ u$ first requires evaluation of t into an abstraction $\Lambda\ cs$ and evaluation of u into an arbitrary term u'. Afterwards, a clause matching u' in cs is searched, which produces a local variable environment σ', possibly overwriting existing variables in σ. Finally, the right-hand side of the clause is evaluated with the combined global and local variable environment.

Constr For a constructor application $\langle \texttt{c}\ t_1\ \dots \rangle$, evaluate all t_i.

Lemma 6.33 (Closedness invariant)**.** *If σ is closed, $\texttt{frees}\ t \subseteq \texttt{dom}\ \sigma$ and $rs, \sigma \vdash t \downarrow t'$, then t' is closed.*

Because of the unchanged term type in this and the previous semantics, the generalized correctness property is easily phrased:

Lemma 6.34. *For any closed environment σ with $\texttt{frees}\ t \subseteq \texttt{dom}\ \sigma$,*

$$rs, \sigma \vdash t \downarrow u \implies rs \vdash \texttt{subst}\ \sigma\, t \longrightarrow^* u$$

This can be proved easily by rule induction on the big-step semantics. The correctness theorem can be obtained by instantiating $\sigma = []$:

Theorem 6.35 (Correctness)**.** $rs, [] \vdash t \downarrow u \wedge \texttt{closed}\ t \implies rs \vdash t \longrightarrow^* u$

The semantics also satisfies some further properties. Even though they are only required in the subsequent proofs, I will describe the statements and their proofs here, because they frequently refer to the rules of the semantics.

Lemma 6.36 (Idempotence)**.** *For all closed term-values (§4.3.5.1), big-step evaluation is idempotent. Formally:* $rs, \sigma \vdash t \downarrow t$.

Proof. By rule induction on term-values. The Abs case requires Corollary 4.26. □

Lemma 6.37 (Pre-substitution)**.** *Let* σ *be a closed term-value environment. Big-step evaluation of a term* t *with* σ *does not change if* t *is first substituted with a environment* $\sigma' \subseteq \sigma$. *Formally:*

$$rs, \sigma \vdash t \downarrow t' \implies rs, \sigma \vdash \texttt{subst}\ \sigma'\ t \downarrow t'$$

Proof. By rule induction on the semantics.

Var If the variable *name* is defined in σ', then substitution of `Svar` *name* yields σ' *name*, which then will be evaluated again. But since σ' is a term-value environment, Lemma 6.36 guarantees that it evaluates to itself.

Const Substituting a constant results in the same constant, therefore the induction hypothesis immediately applies.

Abs It must hold that

$$rs, \sigma \vdash \texttt{subst}\ \sigma'\ (\Lambda\ cs) \downarrow \Lambda\, [(pat, \texttt{subst}\ (\sigma - \texttt{frees}\ pat)\ t \mid (pat, t) \longleftarrow cs]$$

This can be proved in two steps:

1. By applying the Abs rule and unfolding the definition of `subst`, obtain

$$rs, \sigma \vdash \texttt{subst}\ \sigma'\ (\Lambda\ cs) \downarrow \texttt{subst}\ \sigma\, (\texttt{subst}\ \sigma'\ (\Lambda\ cs))$$

2. Generalize the necessary equality

$$\texttt{subst}\ \sigma\, (\texttt{subst}\ \sigma'\ (\Lambda\ cs)) = \texttt{subst}\ \sigma\, (\Lambda\ cs)$$

to arbitrary terms. This follows from Corollary 4.28 and Lemma 4.29 and the representation of the larger environment σ as $\sigma \mathbin{+\!\!+} \sigma'$.

The other cases are uninteresting. □

Lemma 6.38 (Environment coincidence)**.** *Let* σ, σ' *be two closed environments who coincide on the set* S. *Formally:*

$$S \subseteq \mathsf{dom}\ \sigma \wedge S \subseteq \mathsf{dom}\ \sigma' \wedge \forall a \in S.\ \sigma\ a = \sigma'\ a$$

Then, if $\texttt{frees}\ t \subseteq S$, *evaluation in both environments yields the same result:*

$$rs, \sigma \vdash t \downarrow u \iff rs, \sigma' \vdash t \downarrow u$$

The lemma itself is obvious and even desirable for the semantics of a programming language. Its proof is mainly technical; it requires set-theoretic reasoning about domains of mappings.

$$\textsc{Const}\ \frac{(name, rhs) \in rs}{rs, \sigma \vdash \texttt{Sconst}\ name \downarrow rhs} \quad \textsc{Var}\ \frac{\sigma\ name = \texttt{Some}\ v}{rs, \sigma \vdash \texttt{Svar}\ name \downarrow v} \quad \textsc{Abs}\ \frac{}{rs, \sigma \vdash \Lambda\ cs \downarrow \texttt{Vabs}\ cs\ \sigma}$$

$$\textsc{Comb}\ \frac{\begin{array}{c} rs, \sigma \vdash t \downarrow \texttt{Vabs}\ cs\ \sigma' \\ rs, \sigma \vdash u \downarrow v \qquad \texttt{find\_match}\ cs\ v = \texttt{Some}\ (\sigma'', rhs) \qquad rs, \sigma' +\!\!+\ \sigma'' \vdash rhs \downarrow v' \end{array}}{rs, \sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{RecComb}\ \frac{\begin{array}{c} rs, \sigma \vdash t \downarrow \texttt{Vrecabs}\ css\ name\ \sigma' \qquad css\ name = \texttt{Some}\ cs \\ rs, \sigma \vdash u \downarrow v \qquad \texttt{find\_match}\ cs\ v = \texttt{Some}\ (\sigma'', rhs) \qquad rs, \sigma' +\!\!+\ \sigma'' \vdash rhs \downarrow v' \end{array}}{rs, \sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{Constr}\ \frac{name \in \mathsf{C} \qquad rs, \sigma \vdash t_1 \downarrow v_1 \qquad \cdots \qquad rs, \sigma \vdash t_n \downarrow v_n}{rs, \sigma \vdash \texttt{Sconst}\ name\ \$\ t_1\ \$\ \ldots\ \$\ t_n \downarrow \texttt{Vconstr}\ name\ [v_1, \ldots, v_n]}$$

Listing 6.9: Evaluation semantics

6.6 Evaluation semantics

The previous big-step semantic evaluates `sterm`s to `sterm`s. Now, I introduce the concept of values into the semantics, while still keeping the rule set (for constants) and the environment (for variables) separate. The evaluation rules are specified in Listing 6.9. They represent a departure from the original rewriting semantics: a term does not evaluate to another term but to an object of a different type, a `value`. I still use $\downarrow$ as notation, because big-step and evaluation semantics can be disambiguated by their types.

The evaluation model itself is fairly straightforward. As explained in §4.3.5, abstraction terms are evaluated to closures capturing the current variable environment.

I will now examine each rule that has changed substantially from the previous semantics.

ABS Abstraction terms are evaluated to a closure capturing the current environment. The resulting value `Vabs` $cs\ \sigma$ is closed if σ is closed and the free variables occurring in cs are a subset of `dom` σ.

COMB As before, in an application $t\,\$\,u$, t must evaluate to a closure `Vabs` $cs\ \sigma'$. The evaluation result of u is then matched against the clauses cs, producing an environment σ''. The right-hand side of the clause is then evaluated using $\sigma' +\!\!+\ \sigma''$; the original environment σ is effectively discarded.

RECCOMB Similar to COMB. Finding the matching clause is a two-step process: First, the appropriate clause list is selected by name of the currently active function. Then, matching is performed.

In this semantics, recursive closures are not treated differently from non-recursive closures. In a later stage, when rs and σ are merged, this distinction becomes relevant (§6.7).

The semantics uses a variant of `find_match` that matches a `value` against a pattern by first converting it from `term` to `pat` (§4.3.6). This merely simplifies proofs and could be removed completely by fusing the `vmatch` and `mk_pat` functions.

The translations between `sterm` and `value` have already been discussed in §4.3.5.1.

Recall the definition of `value_to_sterm`. The translation rules for `Vabs` and `Vrecabs` are intentionally similar to the Abs rule from the big-step semantics (Listing 6.8). Roughly speaking, the big-step semantics always keeps terms fully substituted, whereas the evaluation semantics defers substitution.

6.6.1 Correctness

The proof of the correctness theorem requires many of the properties proved in §6.5.

Theorem 6.39. *Let σ be a closed environment and t a term which only contains free variables in* dom σ*. Then, an evaluation to a value $rs, \sigma \vdash t \downarrow v$ can be reproduced in the big-step semantics as $rs_0, \sigma_0 \vdash t \downarrow \texttt{value\_to\_sterm}\ v$, where*

$$\begin{aligned} rs_0 &= \texttt{map}\ (\texttt{map}_{\texttt{prod}}\ \texttt{id}\ \texttt{value\_to\_sterm})\ rs \\ \sigma_0 &= \texttt{map}\ \texttt{value\_to\_sterm}\ \sigma \end{aligned}$$

Proof. By rule induction on the evaluation semantics. The interesting cases are Comb and RecComb. Both are roughly identical, save for the additional complication that RecComb has an extra selection step to find the clause set. I will omit that for brevity and focus on the Comb case.

In this case, I need to show that function application behaves the same way in both semantics. Formally:

$$rs_0, \texttt{map}\ \texttt{value\_to\_sterm}\ \sigma \vdash t \$ u \downarrow \texttt{value\_to\_sterm}\ v'$$

The following induction hypotheses are available (side conditions omitted):

$$\begin{aligned} &rs_0, \texttt{map}\ \texttt{value\_to\_sterm}\ \sigma \vdash t \downarrow \texttt{value\_to\_sterm}\ (\texttt{Vabs}\ cs\ \sigma') \\ &rs_0, \texttt{map}\ \texttt{value\_to\_sterm}\ \sigma \vdash u \downarrow \texttt{value\_to\_sterm}\ v \\ &rs_0, \texttt{map}\ \texttt{value\_to\_sterm}\ (\sigma' +\!\!+ \sigma'') \vdash rhs \downarrow \texttt{value\_to\_sterm}\ v' \end{aligned}$$

as well as these premises stemming from the Comb rule:

$$\begin{aligned} &rs, \sigma \vdash t \downarrow \texttt{Vabs}\ cs\ \sigma' \\ &rs, \sigma \vdash u \downarrow v \\ &rs, \sigma' +\!\!+ \sigma'' \vdash rhs \downarrow v' \end{aligned}$$

Furthermore, it is known that `find_match` $cs\ v$ = `Some` (σ'', rhs). Obtain the corresponding pattern pat such that $(pat, rhs) \in cs$.

The first two facts from the hypotheses and the premises line up nicely. The major difference between the two different Comb rules is that in the evaluation semantics, the evaluation of

t to a closure reveals a hidden environment σ' that is subsequently used in the evaluation of rhs. That hidden environment may bear no relation to the active environment σ which is used in the big-step semantics. Hence, I use a trick: Pre-substitute in rhs (Lemma 6.37). Then use Lemma 6.38 to replace σ' for σ:

$$\textsc{Comb}\;\dfrac{\cdots\,\dagger \qquad \text{Lemma 6.38}\;\dfrac{\text{Lemma 6.37}\;\dfrac{\text{IH}\;\dfrac{}{rs_0, \texttt{map value\_to\_sterm}\;(\sigma' \mathbin{+\!\!+} \sigma'') \vdash rhs \downarrow \texttt{value\_to\_sterm}\; v'}}{rs_0, \texttt{map value\_to\_sterm}\;(\sigma' \mathbin{+\!\!+} \sigma'') \vdash rhs_{\sigma'} \downarrow \texttt{value\_to\_sterm}\; v'}}{rs_0, \texttt{map value\_to\_sterm}\;(\sigma \mathbin{+\!\!+} \sigma'') \vdash rhs_{\sigma'} \downarrow \texttt{value\_to\_sterm}\; v'}}{rs_0, \texttt{map value\_to\_sterm}\;\sigma \vdash t\;\$\;u \downarrow \texttt{value\_to\_sterm}\; v'}$$

where $rhs_{\sigma'}$ = `subst` (`map value_to_sterm` σ' − `frees` pat) rhs. I specifically chose this value for $rhs_{\sigma'}$ to be able to combine both lemmas and to coincide with the definition of `value_to_sterm`. The evaluations of t and u in the Comb rule have been omitted (†), because they are trivial.

Recall that the precondition for Lemma 6.37 (instantiated to this case) is that σ' − `frees` $pat \subseteq \sigma' \mathbin{+\!\!+} \sigma''$. However, when adding two environments, the entries on the right override the entries on the left. But from the premises and the `term` axioms, it is known that `dom` σ'' = `frees` pat. It follows that σ' − `frees` pat contains no entries that are also contained in σ'', proving the precondition.

The situation is similar for the application of Lemma 6.38. I can instantiate S with `frees` pat and must show that $\sigma' \mathbin{+\!\!+} \sigma''$ and $\sigma \mathbin{+\!\!+} \sigma''$ agree on this set. This again follows immediately from `dom` σ'' = `frees` pat.

The remainder of the proof is mostly clerical and takes care of lining up further preconditions.

□

6.6.2 Discussion

The correctness theorem states that, for any given evaluation of a term t with given rs, σ containing `values`, that evaluation can be reproduced in the big-step semantics using a derived list of rules rs_0 and an environment σ_0 containing `sterms` that are generated by the `value_to_sterm` function. But recall the diagram in Figure 6.1. In the formalization, it starts with a given rule set of `sterms` that has been compiled from a rule set of `terms`. Hence, the correctness theorem only deals with the opposite direction.

It remains to construct a suitable rs such that applying `value_to_sterm` to it yields the given `sterm` rule set. I exploit the side condition (§6.1) that all bindings define proper functions, i.e., the right-hand sides are abstractions:

Definition 6.40 (Global clause set). *The mapping*

$$\texttt{global\_css} :: \texttt{string} \rightharpoonup ((\texttt{term} \times \texttt{sterm})\;\texttt{list})$$

is obtained by stripping the `Sabs` *constructors from all definitions and converting the resulting list to a mapping. For each non-constructor constant* `c`*, define its closure representation as*

$$\texttt{v}_\texttt{c} = \texttt{Vrecabs global\_css}\; c\; []$$

where c is the string representation of the name `c`.

In other words, each function is now represented by a recursive closure bundling all functions.

Lemma 6.41. *Let* $(c, \Lambda\ cs) \in rs$. *Applying* `value_to_sterm` *to* $\mathtt{v}_\mathtt{c}$ *returns the original definition of* `c`. *Formally:* `value_to_sterm` $\mathtt{v}_\mathtt{c} = \Lambda\ cs$.

Let *rs* denote the original `sterm` rule set and rs_v the environment mapping all constants to their closure representations.

Corollary 6.42. $rs = \mathtt{map}\ (\mathtt{map}_\mathtt{prod}\ \mathtt{id}\ \mathtt{value\_to\_sterm})\ rs_v$

Furthermore, the variable environments σ and σ' can safely be set to the empty mapping, because top-level terms are evaluated without any free variable bindings. Combined, this enables instantiation of Theorem 6.39:

Corollary 6.43 (Correctness). $rs_v, [] \vdash t \downarrow v \implies rs, [] \vdash t \downarrow \mathtt{value\_to\_sterm}\ v$

Even though rs_v is executable (§2.4), this step is not part of the compiler. Instead, it is a refinement of the semantics to support a more modular correctness proof.

Example Recall the `odd` and `even` example from §4.3.5. After compilation to `sterm`, the rule set looks like this:

$$\begin{aligned} rs = \{(\text{"odd"}, \mathtt{Sabs}\ [\langle 0\rangle \Rightarrow \langle\mathtt{False}\rangle, \langle\mathtt{Suc}\ n\rangle \Rightarrow \langle\mathtt{even}\ n\rangle]),\\ (\text{"even"}, \mathtt{Sabs}\ [\langle 0\rangle \Rightarrow \langle\mathtt{True}\rangle, \langle\mathtt{Suc}\ n\rangle \Rightarrow \langle\mathtt{odd}\ n\rangle])\} \end{aligned}$$

This can be easily transformed into the following global clause set:

$$\begin{aligned} \mathtt{global\_css} = [\text{"odd"} \mapsto [\langle 0\rangle \Rightarrow \langle\mathtt{False}\rangle, \langle\mathtt{Suc}\ n\rangle \Rightarrow \langle\mathtt{even}\ n\rangle],\\ \text{"even"} \mapsto [\langle 0\rangle \Rightarrow \langle\mathtt{True}\rangle, \langle\mathtt{Suc}\ n\rangle \Rightarrow \langle\mathtt{odd}\ n\rangle]] \end{aligned}$$

Finally, rs_v is computed by creating a recursive closure for each function:

$$\begin{aligned} rs_\mathrm{v} = [\text{"odd"} \mapsto \mathtt{Vrecabs}\ \mathtt{global\_css}\ \text{"odd"}\ [],\\ \text{"even"} \mapsto \mathtt{Vrecabs}\ \mathtt{global\_css}\ \text{"even"}\ []] \end{aligned}$$

6.7 Evaluation with recursive closures

CakeML distinguishes between non-recursive and recursive closures [94]. This distinction is also present in the `value` type. In this step, I will close that gap by conflating variables with constants. This necessitates a special treatment of recursive closures. Therefore I introduce a new predicate $\sigma \vdash t \downarrow v$ (Listing 6.10) that – in contrast to the previous semantics – only has one environment.

As usual, I will explain the rules that have changed significantly from the previous semantics here:

$$\textsc{Const}\ \frac{name \notin \mathsf{C} \qquad \sigma\ name = \mathsf{Some}\ v}{\sigma \vdash \mathsf{Sconst}\ name \downarrow v}$$

$$\textsc{Var}\ \frac{\sigma\ name = \mathsf{Some}\ v}{\sigma \vdash \mathsf{Svar}\ name \downarrow v} \qquad \textsc{Abs}\ \frac{}{\sigma \vdash \Lambda\ cs \downarrow \mathsf{Vabs}\ cs\ \sigma}$$

$$\textsc{Comb}\ \frac{\begin{array}{c}\sigma \vdash t \downarrow \mathsf{Vabs}\ cs\ \sigma' \\ \sigma \vdash u \downarrow v \qquad \mathsf{find\_match}\ cs\ v = \mathsf{Some}\ (\sigma'', rhs) \qquad \sigma' +\!\!+\ \sigma'' \vdash rhs \downarrow v'\end{array}}{\sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{RecComb}\ \frac{\begin{array}{c}\sigma \vdash t \downarrow \mathsf{Vrecabs}\ css\ name\ \sigma' \qquad css\ name = \mathsf{Some}\ cs \qquad \sigma \vdash u \downarrow v \\ \mathsf{find\_match}\ cs\ v = \mathsf{Some}\ (\sigma'', rhs) \qquad \sigma' +\!\!+\ \mathsf{mk\_rec\_env}\ css\ \sigma' +\!\!+\ \sigma'' \vdash rhs \downarrow v'\end{array}}{\sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{Constr}\ \frac{name \in \mathsf{C} \qquad \sigma \vdash t_1 \downarrow v_1 \qquad \cdots \qquad \sigma \vdash t_n \downarrow v_n}{\sigma \vdash \mathsf{Sconst}\ name\ \$\ t_1\ \$\ \ldots\ \$\ t_n \downarrow \mathsf{Vconstr}\ name\ [v_1, \ldots, v_n]}$$

$$\mathsf{mk\_rec\_env}\ css\ \sigma = \mathsf{map\_of}\ \{\mathsf{Vrecabs}\ css\ name\ \sigma \mid (name, \_) \in css\}$$

Listing 6.10: ML-style evaluation semantics

Const/Var Constant definition and variable values are both retrieved from the same environment σ. I have chosen to keep the distinction between constants and variables in the `sterm` type to avoid the introduction of another intermediate term type. This gap will be closed in the final phase (§6.8). For technical reasons, the Const case here now requires an additional check that it is a non-constructor constant, which was not required in the previous semantics.

Abs Identical to the previous evaluation semantics. Evaluation never creates recursive closures at run-time. Anonymous functions, e.g. in the term ⟨map $(\lambda x.\ x)$⟩, are evaluated to non-recursive closures. Combined with the fact that in my source language, the set of definitions is fixed before running the compiler, it follows that the evaluation semantics never needs to construct a hitherto unseen `Vrecabs` value: these are constructed by the compiler (§6.6.2).

RecComb Almost identical to the evaluation semantics. Additionally, for each function $(name, cs) \in css$, a new recursive closure `Vrecabs` $css\ name\ \sigma'$ is constructed and inserted into the environment. Observe that the cs is ignored during construction, because it is contained in css itself. This ensures that after the first call to a recursive function, the function itself is present in the environment to be called recursively, without having to introduce coinductive environments. Still, some proofs require coinductive reasoning at a later point (Theorem 6.48). The strategy itself is similar to how it is implemented in the CakeML semantics.

By merging the rule set *rs* with the variable environment σ, it becomes necessary to discuss possible clashes. Previously, the syntactic distinction between Svar and Sconst meant that $\langle x\rangle$ (the syntactic variable "x") and $\langle \texttt{x}\rangle$ (the syntactic constant "x") are not ambiguous: all semantics up to the evaluation semantics clearly specify where to look for the substitute. This is not the case in functional languages where definitions and variables are not distinguished syntactically.

Instead, I rely on the fact that the initial rule set only defines constants. All variables are introduced by matching before β-reduction (that is, in the Comb and RecComb rules). The Abs rule does not change the environment. Hence it suffices to assume that variables in patterns must not overlap with constant names, which is guaranteed by a side-condition (§6.1).

The following lemma shows the direct correspondence between rs_v (§6.6.2) and mk_rec_env (Listing 6.10).

Lemma 6.44. map_of rs_v = mk_rec_env global_css []

The proof is technical and requires no further assumptions about the rule set.

Lemma 6.45 (Extensionality)**.** *Evaluation is invariant under extensional equivalence (§4.3.5.3).*

Example Reusing the example from §6.6.2, I will demonstrate the evaluation of the term $\langle \texttt{odd (Suc 0)}\rangle$. Since it is a closed term, there are no free variables that need to have a value. Let $\sigma = rs_\text{v}$ from that example.

$$\textsc{RecComb}\ \dfrac{\cdots \qquad \dagger\ \dfrac{\textsc{RecComb}\ \dfrac{\cdots \qquad \textsc{Var}\ \dfrac{}{\sigma +\!\!+ [n \mapsto \langle 0\rangle] \vdash \langle n\rangle \downarrow \langle 0\rangle}}{\sigma +\!\!+ [n\mapsto\langle 0\rangle] \vdash \langle \texttt{even}\ n\rangle \downarrow \langle\texttt{True}\rangle}}{[\,] +\!\!+ \texttt{mk\_rec\_env}\ css\ [\,] +\!\!+ [n\mapsto \langle 0\rangle] \vdash \langle\texttt{even}\ n\rangle\downarrow\langle\texttt{True}\rangle}}{\sigma \vdash \langle \texttt{odd (Suc 0)}\rangle \downarrow \langle\texttt{True}\rangle}$$

The tree is annotated with the rules from the semantics. The step † is not a rule application, but a simplification, because mk_rec_env *css* [] = σ. This example illustrates that mk_rec_env allows recursive calls without introducing cyclic environments.

6.7.1 Correspondence relation

Both constant definitions and values of variables are recorded in a single environment σ. This also applies to the environment contained in a closure. The correspondence relation thus needs to take a different sets of bindings in closures into account.

Hence, I define a coinductive relation $\approx_\text{v}$ that uses the computed rule set rs_v (§6.6.2) and compares environments (Listing 6.11). I call it *right-conflating*, because in a correspondence $v \approx_\text{v} u$, any bound environment in u is thought to contain both variables and constants (i.e., the environment is conflated), whereas in v, any bound environment contains only variables. In

$$\frac{\texttt{rel}\ (\approx_{\mathrm{v}})\ \mathit{vs}\ \mathit{us}}{\texttt{Vconstr}\ \mathit{name}\ \mathit{vs} \approx_{\mathrm{v}} \texttt{Vconstr}\ \mathit{name}\ \mathit{us}}$$

$$\frac{\forall x \in \texttt{frees}\ (\Lambda\ \mathit{cs}).\ \sigma_1\ x \approx_{\mathrm{v}} \sigma_2\ x \qquad \forall x \in \texttt{consts}\ (\Lambda\ \mathit{cs}).\ \mathit{rs}\ x \approx_{\mathrm{v}} \sigma_2\ x}{\texttt{Vabs}\ \mathit{cs}\ \sigma_1 \approx_{\mathrm{v}} \texttt{Vabs}\ \mathit{cs}\ \sigma_2}$$

$$\frac{\begin{array}{c}\forall \mathit{cs} \in \texttt{range}\ \mathit{css}.\ \forall x \in \texttt{frees}\ (\Lambda\ \mathit{cs}).\ \sigma_1\ x \approx_{\mathrm{v}} \sigma_2\ x \\ \forall \mathit{cs} \in \texttt{range}\ \mathit{css}.\ \forall x \in \texttt{consts}\ (\Lambda\ \mathit{cs}).\ \mathit{rs}_{\mathrm{v}}\ x \approx_{\mathrm{v}} (\sigma_2 \mathbin{+\!\!+} \texttt{mk\_rec\_env}\ \mathit{css}\ \sigma_2)\ x\end{array}}{\texttt{Vrecabs}\ \mathit{css}\ \mathit{name}\ \sigma_1 \approx_{\mathrm{v}} \texttt{Vrecabs}\ \mathit{css}\ \mathit{name}\ \sigma_2}$$

Listing 6.11: Right-conflating correspondence (coinductive)

the correctness property, v and u will originate from the previous and the ML-style semantics, respectively.

It is worth examining the definition of $\approx_{\mathrm{v}}$:

VCONSTR Constructors are compared structurally.

VABS Non-recursive closures must have identical clauses cs. The bound environment σ_2 of the right closure is compared on two subsets:

- on the free variables of cs, it must correspond to the bound environment σ_1 of the left closure, and
- on the constants of cs, it must correspond to the rules rs of the global environment.

VRECABS Recursive closures are treated similarly to non-recursive closures, with one exception: In the constants of the clauses set css, σ_2 is first augmented with $\texttt{mk\_rec\_env}\ css\ \sigma_2$ before relating to rs. This corresponds to the RECCOMB rule of the semantics. It is also the reason why this relation must be coinductive: any derivation tree involving recursive closures will be infinite, because the recursive environment keeps being inserted into σ_2.

This relation has some perhaps surprising properties. Most importantly, it is not reflexive. For example, assuming a constant a is defined, the following proposition does not hold:

$$\texttt{Vabs}\ \mathit{cs}\ [] \approx_{\mathrm{v}} \texttt{Vabs}\ \mathit{cs}\ [] \quad \text{where } \mathit{cs} = [\langle x\rangle \Longrightarrow \langle \mathsf{a}\rangle]$$

Fortunately, $\approx_{\mathrm{v}}$ fits into the usual scheme of relations:

Corollary 6.46. *$\approx_{\mathrm{v}}$ is a structural value relation (§4.3.5.2).*

Similarly to $\approx_{\mathrm{p}}$ (Listing 6.5), this somewhat technical definition implies a more intuitive property:

Lemma 6.47. $u \approx_{\mathrm{v}} v \implies \texttt{value\_to\_sterm}\ u = \texttt{value\_to\_sterm}\ v$

Corollary 4.40 also applies; i.e., for closure-free values, $\approx_{\mathrm{v}}$ collapses to equality.

6.7.2 Correctness

Theorem 6.48. *Let σ be an environment, t be a closed term and v a value such that $\sigma \vdash t \downarrow v$. If for all constants x occurring in t, $rs\ x \approx_\mathrm{v} \sigma\ x$ holds, then there is an u such that $rs, [] \vdash t \downarrow u$ and $u \approx_\mathrm{v} v$.*

As usual, the proof proceeds via induction over the semantics (Listing 6.10). It is important to note that the global clause set construction (§6.6.2) satisfies the preconditions of the theorem:

Lemma 6.49. *If name* ∈ `dom global_css`*, then:*

$$\texttt{Vrecabs global\_css}\ \mathit{name}\ [] \approx_\mathrm{v} \texttt{Vrecabs global\_css}\ \mathit{name}\ []$$

Proof. Because $\approx_\mathrm{v}$ is defined coinductively, the proof of this precondition proceeds by coinduction. In essence, I have to use the `Vrecabs` rule of $\approx_\mathrm{v}$. But because $\sigma_1 = \sigma_2 = []$, the first premise of the rule is trivial and the second one collapses to a correspondence between rs_v and `mk_rec_env global_css`. This is a direct consequence of Lemma 6.44, which shows an equality between those two maps. □

6.8 CakeML

CakeML's semantics has been formalised in Lem [93]. For the correctness proof of the CakeML compiler, its authors have extracted this into HOL theories. In this work, I directly target CakeML abstract syntax trees (thereby bypassing the parser) and use its big-step semantics, which I have extracted into Isabelle [66].

Consequently, there is not a single correctness result in Isabelle, but rather two parts: A frontend from Isabelle to CakeML, and a backend from CakeML to machine code, the latter of which is provided by the ongoing work on CakeML [121].

In order to execute the Isabelle-generated CakeML syntax trees, there are three choices:

1. turning CakeML's big step semantics into an executable function within Isabelle using the predicate compiler [8],
2. executing the functional big-step semantics directly within Isabelle [102], and
3. printing the tree as an s-expression and compiling it with the official CakeML compiler.

`CakeML_`
`Compiler`

6.8.1 CakeML's semantic functions

In §2.7, I have outlined the work that was required to integrate the Lem export of CakeML with Isabelle. Notwithstanding that, the CakeML theories describe an entirely separate universe from the definitions in §4. Notably, it is not possible to instantiate the `term` class for CakeML expressions. In this section, I will discuss the differences between my and CakeML's implementation of term operations.

Almost all of CakeML's semantic functions take (a fragment of) an environment into account. An environment consists of two *namespaces*: the *value* and the *constructor* namespace. The value namespace records existing bindings. It is comparable to the Γ and σ used in the various intermediate semantics. The constructor namespace keeps track of defined datatypes and their constructors, including their arities.

```
fun pat_to_cake :: pat ⇒ Cake.pat where
pat_to_cake (Patvar s) = Cake.Pvar s
pat_to_cake (Patconstr s args) =
  Cake.Pcon (Some (Cake.Short s)) (map pat_to_cake args)
```

Listing 6.12: Translation between patterns

6.8.1.1 CupCakeML: A purely-functional subset of CakeML

The core of CakeML's big-step semantics is organized in three mutually recursive inductive predicates (evaluating an expression, a list of expressions, and a match expression) with a total of over thirty rules. It is that large because it has to deal with exceptions, modules, step counters, and other features that their compiler supports, but I do not need in the formalization. For that reason, I have defined the subset *CupCakeML* which only allows the syntactic forms known from the `sterm` type. It also enforces that all occurring values have an empty module environment and a consistent datatype environment.

I define a smaller big-step semantics comprising a single inductive predicate with twelve cases. Then, I prove equivalence to the original semantics, that is, both correctness and completeness, under the assumption that the expression and initial environments are in the supported fragment. The supported fragment has also been proved to be closed under evaluation.

The namespaces are allowed to vary in closures, since CakeML supports introducing new type definitions inside a program. Tracking this would greatly complicate the proofs. Hence, I assume a fixed constructor namespace and enforce that all values use exactly that one. This information is provided by the `C_info` parameter (Listing 5.2) that is generated during embedding.

6.8.1.2 Patterns and matching

CakeML defines three distinct datatypes for patterns, expressions and values. Because they model the syntax of a real-world programming language, they are much more complex than the types required for term rewriting.

The most notable difference in types is that patterns and constructor values in CakeML are *n*-ary. This corresponds closely to the `value` (§4.3.5) and `pat` (§4.3.6) types that I introduced for the evaluation semantics (§6.6). Consequently, a conversion function between the pattern types is easily defined (Listing 6.12). Additional CakeML complexities, for example pattern wildcards (**case** *t* **of** _ ⇒ *u*) or qualified names (`Module.name`) are excluded from the CupCakeML fragment.

CakeML's matching function has three possible results: `Match`, `No_match`, and `Match_type_error`. The latter result may occur when the pattern and the object disagree about their arity. For example, consider the ill-typed program **case** `T` *x* **of** `T` ⇒ *y*. In my implementation, matching `T` *x* to `T` would result in `None`. However, in CakeML it returns `Match_type_error`.

A similar check happens when constructing constructor values, too. In short, data constructors must always occur with all arguments supplied on right-hand and left-hand sides.

During embedding (§5), all type information is erased. Fully applied constructors in terms can be easily guaranteed by the introduction of constructor functions (§3.3). For patterns however, this must be ensured throughout the compilation pipeline; it is (like other syntactic constraints) another side condition imposed on the rule set (§6.1).

For the purposes of the correctness proof, a CakeML type error is treated equivalently to non-termination. This means that should the evaluation of a program end in a type error, there are no guarantees. However, I avoid this problem by running the CakeML type checker on the generated programs (§7.1).

An additional complication comes from the shallow nature of the typing check. Not all ill-typed constructors yield a type error. For example, matching the object `T U2 (V` x`)` against the pattern `T U1 V` would result in `No_match`. `U2` and `U1` do not match. At that point, the matching function does not proceed further and the arity mismatch in the second argument remains hidden.

Similarly to my implementation of matching, CakeML does not check for linearity of patterns. However, the linearity check happens in the big-step semantics, so it needs to be dealt with there.

6.8.1.3 Closures

Closures are syntactically similar between the two different semantics. In particular, the `mk_rec_env` function I introduced for the ML-style evaluation semantics (§6.7) has been modelled after its counterpart in the CakeML semantics.

It remains to deal with application of closures to values. CakeML models function application as a binary operator, similarly to other operators such as addition on machine words. The CupCakeML fragment enforces that application is the only operator that may occur in generated code, since my formalization never emits operations on machine types. The complication here is that in my semantics, application always requires matching. In CakeML, closures always take exactly one argument which is not matched, but directly bound as a variable. The body then may or may not perform a matching step. I will explain this later when introducing an appropriate correspondence relation (Listing 6.15).

6.8.2 Translation from `sterm` to `exp`

After the series of translations described in the earlier sections, the terms in my formalization are syntactically close to CakeML's terms (`Cake.exp`). The only remaining differences are outlined below:

- CakeML does not combine abstraction and pattern matching. For that reason, I need to translate $\Lambda\ [p_1 \Rightarrow t_1, ...]$ into $\Lambda x.$ **case** x **of** $p_1 \Rightarrow t_1 \mid ...$, where x is a fresh variable name. I reuse the `fresh` monad (§6.2.1) to obtain a bound variable name.
- CakeML has two distinct syntactic categories for identifiers (that can represent variables or constants) and data constructors. My term types however have two distinct syntactic

```
fun
  sterm_to_cake :: string set ⟹ sterm ⟹ Cake.exp and
  clauses_to_cake :: string set ⟹ (term × sterm) list ⟹ (Cake.pat × Cake.exp) list
where
sterm_to_cake _ (Svar s) = Cake.Var (Cake.Short s)
sterm_to_cake _ (Sconst s) = Cake.Var (Cake.Short s)
sterm_to_cake S (t1 $ t2) =
  Cake.App Cake.Opapp [sterm_to_cake S t1, sterm_to_cake S t2]
sterm_to_cake S (Sabs cs) =
  let n = Fresh.run Fresh.create (S ∪ constructors) in
    Cake.Fun n (Cake.Mat (Cake.Var (Cake.Short n)) (clauses_to_cake S cs))
clauses_to_cake S cs =
  [(pat_to_cake (mk_pat pat), sterm_to_cake (frees pat ∪ S) t)) | (pat, t) ⟵ cs]
```

Listing 6.13: Translation from `sterm`s to `exp`s

categories for constants (that can represent function definitions or data constructors) and variables. The necessary prerequisites to deal with this are already present in the ML-style evaluation semantics (§6.7) which conflates constants and variables, but has a dedicated Constr rule for data constructors.

The corresponding translation functions are given in Listing 6.13 (the type prefix `Cake` indicates a CakeML type).

In the `Sabs` case, is not necessary to thread through already created variable names, only existing names. The reason is simple: a generated variable is bound and then immediately used in the body. Shadowing the name somewhere in the body does not cause any problems. This is in contrast to the `term_to_nterm` function (Listing 6.2), which needs to track all bound variable names.

The S parameter then needs to be initialized with all existing constants; similarly to §6.2.3, this can be achieved with `all_consts`.

6.8.3 Correspondence relations

I define two different correspondence relations: one for expressions and one for values. Both share the $\approx_{🍰}$ operator (where 🍰 is the official CakeML icon), because they can be disambiguated by types. There is no separate relation for patterns, because their translation is simple.

First, I will examine the expression correspondence (Listing 6.14).

Var Variables are directly related by identical name.

Const Recall that constructors are treated specially in CakeML. In order to not confuse functions or variables with data constructors themselves, the relation demands that the constant name is not a constructor.

$$\textsc{Var}\ \frac{}{\texttt{Svar}\ n \approx_{\text{🍰}} \texttt{Cake.Var}\ n} \qquad \textsc{Const}\ \frac{n \notin \texttt{C}}{\texttt{Sconst}\ n \approx_{\text{🍰}} \texttt{Cake.Var}\ n}$$

$$\textsc{Constr}\ \frac{n \in \texttt{C} \qquad \texttt{rel}\ (\approx_{\text{🍰}})\ ts\ us}{name\ \$\$\ ts \approx_{\text{🍰}} \texttt{Cake.Con}\ (\texttt{Some}\ (\texttt{Cake.Short}\ name)\ us)}$$

$$\textsc{App}\ \frac{t_1 \approx_{\text{🍰}} u_1 \qquad t_2 \approx_{\text{🍰}} u_2}{t_1\ \$\ t_2 \approx_{\text{🍰}} \texttt{Cake.App}\ \texttt{Cake.Opapp}\ [u_1, u_2]}$$

$$\textsc{Fun}\ \frac{\begin{array}{c} n \notin \texttt{ids}\ (\Lambda\ cs) \qquad n \notin \texttt{all\_consts} \\ \texttt{rel}_{\texttt{list}}\ (\texttt{rel}_{\texttt{prod}}\ (\lambda\ t\ p.\ p = \texttt{term\_to\_pat}\ (\texttt{pat\_to\_cake}\ t))\ (\approx_{\text{🍰}}))\ cs\ ml\_cs \end{array}}{\Lambda\ cs \approx_{\text{🍰}} \texttt{Cake.Fun}\ n\ (\texttt{Cake.Mat}\ (\texttt{Cake.Var}\ n))\ ml\_cs}$$

$$\textsc{Mat}\ \frac{t \approx_{\text{🍰}} u \qquad \texttt{rel}_{\texttt{list}}\ (\texttt{rel}_{\texttt{prod}}\ (\lambda\ t\ p.\ p = \texttt{term\_to\_pat}\ (\texttt{pat\_to\_cake}\ t))\ (\approx_{\text{🍰}}))\ cs\ ml\_cs}{\Lambda\ cs\ \$\ t \approx_{\text{🍰}} \texttt{Cake.Mat}\ u\ ml\_cs}$$

Listing 6.14: Expression correspondence

Constr Constructors are directly related by identical name, and recursively related arguments.

App CakeML does not just support general function application but also unary and binary operators. In fact, function application is represented by the binary operator Opapp. The sterm_to_exp translation function never generates other operators. Consequently, the correspondence is restricted to Opapp.

Fun/Mat Observe the symmetry between these two cases: In my term language, matching and abstraction are combined, which is not the case in CakeML. This means relating a case abstraction to a CakeML function containing a match, and a case abstraction applied to a value to just a CakeML match. The additional requirement is that the bound variable name in a CakeML function must not occur in the clauses nor be a known constant name (Listing 5.2).

The value correspondence is structurally simpler, but unfortunately, syntactically noisier (Listing 6.15).

Constr Constructor values are compared recursively.

Abs Comparing closures has multiple requirements:

- the CakeML closure must take a parameter with the bound variable name *n* that is immediately matched,
- *n* must be a fresh variable,
- the clauses must correspond according to the expression correspondence, and

$$\textsc{Constr}\ \frac{\texttt{rel}\ (\approx_{\text{🍰}})\ \mathit{vs}\ \mathit{us}}{\texttt{Vconstr}\ \mathit{name}\ \mathit{vs} \approx_{\text{🍰}} \texttt{Cake.Conv}\ (\texttt{Some}\ (\mathit{name}, \mathit{type})\ \mathit{us})}$$

$$\textsc{Abs}\ \frac{\begin{array}{c} n \notin \texttt{ids}\ (\Lambda\ \mathit{cs}) \qquad n \notin \texttt{all\_consts} \\ \texttt{rel}_{\texttt{list}}\ (\texttt{rel}_{\texttt{prod}}\ (\lambda\, t\ p.\ p = \texttt{term\_to\_pat}\ (\texttt{pat\_to\_cake}\ t))\ (\approx_{\text{🍰}}))\ \mathit{cs}\ \mathit{ml\_cs} \\ \forall x \in \texttt{ids}\ (\Lambda\ \mathit{cs}).\ \Gamma\ x \approx_{\text{🍰}} \texttt{map\_of}\ \mathit{env}\ x \end{array}}{\texttt{Vabs}\ \mathit{cs}\ \Gamma \approx_{\text{🍰}} \texttt{Cake.Closure}\ \mathit{env}\ n\ (\texttt{Cake.Mat}\ (\texttt{Cake.Var}\ n)\ \mathit{ml\_cs})}$$

Listing 6.15: Value correspondence

- the environments must correspond recursively, but only on the identifiers that occur in the clauses.

RecAbs This case is identical to non-recursive closures, but lifted to lists of clauses. It is omitted in Listing 6.15 because of its syntactic size.

6.8.4 Correctness

Lemma 6.50 (Correctness of `sterm_to_cake`).

$$t \approx_{\text{🍰}} \texttt{sterm\_to\_cake}\ (\texttt{ids}\ t \cup \texttt{all\_consts})\ t$$

Lemma 6.51 (Composition of value and extensional correspondence).

$$v \approx_{\text{🍰}} u \wedge v' \approx_{\mathrm{e}} v \implies v' \approx_{\text{🍰}} u$$

I use the same trick as in §6.6.2 to obtain a suitable environment for CakeML evaluation based on the rule set *rs*. Assuming this environment, the correctness theorem can be stated as follows:

Theorem 6.52 (Correctness). *If a CakeML expression e terminates with a value u in the CakeML semantics and t $\approx_{\text{🍰}}$ e there is a value v such that v $\approx_{\text{🍰}}$ u and t evaluates to v.*

Proof. The proof proceeds by induction on the CakeML evaluation. I will discuss some of the ideas here.

The most interesting case is application. By assumption, $t = t_1 \mathbin{\$} t_2$ (similarly for e). By induction hypothesis, t_1 evaluates to a (possibly recursive) closure. I need to show that t evaluates to a value $v :: \texttt{value}$ corresponding to u.

Assume that e_1 evaluates to a non-recursive closure. The recursive case is similar.

Consequently, according to Listing 6.14, the closure must take a variable with a fresh name and immediately match it against a list of clauses. This additional variable gets added to the environment; hence, Lemmas 6.45 and 6.51 must be used to establish correspondence. ☐

6.9 Composition

After having described the complete compiler pipeline in the previous sections, I will give a high-level overview of how all phases fit together here.

The actual compiler can be characterized with two functions. Firstly, the translation of `term` to `Cake.exp` is a simple composition of each term translation function:

```
definition term_to_cake :: term ⇒ Cake.exp where
term_to_cake = sterm_to_cake ∘ pterm_to_sterm ∘ nterm_to_pterm ∘ term_to_nterm
```

Secondly, the function that translates function definitions by composing the phases as outlined in Figure 6.1, including iterated application of pattern elimination:

```
definition compile :: (term × term) set ⇒ Cake.dec where
compile = Cake.Dletrec ∘ compile_srules_to_cake ∘ compile_prules_to_srules ∘
  compile_irules_to_srules ∘ compile_irules_iter ∘ compile_crules_to_irules ∘
  consts_of ∘ compile_rules_to_nrules
```

Each function `compile_*` corresponds to one compiler phase; the remaining functions are trivial.

This produces a CakeML top-level declaration. I prove that evaluating this declaration in the top-level semantics (`evaluate_prog`) results in an environment `cake_sem_env`. But `cake_sem_env` can also be computed via another instance of the global clause set trick (§6.6.2).

Correctness is justified for each phase between intermediate semantics and correspondence relations, most of which are rather technical. Whereas the compiler may be complex and impenetrable, the trustworthiness of the constructions hinges on the obviousness of those correspondence relations.

Fortunately, under the assumption that terms to be evaluated and the resulting values do not contain abstractions – or closures, respectively – all of the correspondence relations collapse to simple structural equality.

Equipped with these functions and relations, I can state the final correctness theorem:

```
theorem compiled_correct:
  (* If CakeML evaluation of a term succeeds ... *)
  assumes evaluate False cake_sem_env s (term_to_cake t) (s', Rval v)
  (* ... producing a constructor term without closures ... *)
  assumes cake_no_abs v
  (* ... and some syntactic properties of the involved terms hold ... *)
  assumes closed t and ¬ shadows_consts t and welldefined t and wellformed t
  (* ... then this evaluation can be reproduced in the term-rewriting semantics *)
  shows rs ⊢ t ⟶* cake_to_term v
```

This theorem directly relates the evaluation of a term t in the full CakeML (including mutability and exceptions) to the evaluation in the initial higher-order term rewriting semantics. The evaluation of t happens using the environment produced from the initial rule set. Hence, the theorem can be interpreted as the correctness of the pseudo-ML expression **let rec** rs **in** t.

Observe that in the assumption, the conversion goes from my terms to CakeML expressions, whereas in the conclusion, the conversion goes the opposite direction.

6.10 Related work

There is existing work in the Coq [1, 40] and HOL [94] communities for proof producing or verified extraction of functions defined in the logic. Anand et al. [1] present work in progress on a verified compiler from Gallina (Coq's specification language) via untyped intermediate languages to CompCert C light. They plan to connect their extraction routine to the CompCert compiler [83].

Compilation of pattern matching is well understood in literature [2, 106, 113]. In this work, I contribute a transformation of sets of equations with pattern matching on the left-hand side into a single equation with nested pattern matching on the right-hand side. This is implemented and verified inside Isabelle.

Besides CakeML, there are many projects for verified compilers for functional programming languages of various degrees of sophistication and realism [7, 28, 39]. Particularly modular is the work by Neis et al. [96] on a verified compiler for the ML-like imperative source language Pilsner. The main distinguishing feature of this work is that I start from a set of higher-order recursion equations with pattern matching on the left-hand side rather than a lambda calculus with pattern matching on the right-hand side. On the other hand I stand on the shoulders of CakeML which allows me to bypass all complications of machine code generation. Note that much of the formalization is not specific to CakeML and that it would be possible to retarget it to, for example, Pilsner abstract syntax with moderate effort.

7 Conclusion

In this thesis, I have covered the design and implementation of a verified compiler from Isabelle/HOL to CakeML. In this final chapter, I discuss the results and possible avenues for further research.

7.1 Results

The result of this thesis is a substantial formalization of a real-world compiler.

Component	**Size** (ML)	**Size** (Isar)
Dictionary Construction	2,100	800
Higher-Order Term Algebra	n/a	3,800
CakeML *(excluding generated parts)*	300	4,100
CupCakeML	n/a	900
Preprocessing	1,100	300
Compiler	n/a	14,100
	3,500	24,000

The compiler is able to emit concrete syntax (using s-expressions, §6.8) of CakeML that can be consumed by the CakeML compiler, version 2.0. It comes with a correctness proof that covers all phases and a completeness proof that only covers some phases. For that reason, it is not advisable to turn off type inference when running the CakeML compiler.

A remaining issue is performance of the compiler. In particular, the dictionary construction and the deep embedding are expensive operations. The former needs to define new functions which are subject to the overhead of the **function** command. The latter carries out proofs by induction, potentially handling large terms. Luckily, both can be sped up by using parallelism, which is easily achieved with Isabelle/ML.

7.2 Future work

Dictionary construction Currently, there are heuristics for both termination and specifiedness of functions. Neither of them allow customization. For the former, allowing users to carry out manual proofs is a technical, but not theoretical, challenge. The latter admits some basic tuning by providing different congruence rules. It is an interesting research question how far the detection of specifiedness can be automated without these congruence rules. The most difficult challenge is posed by functions that are polymorphic in the return type, i.e. where type variables can be instantiated with functions that may influence the specifiedness of a returned function.

CakeML integration *OpenTheory* is an exchange format between different provers in the HOL family [67]. It is relevant to this thesis in two ways:

1. Large parts of the CakeML formalization and proofs are not available in Lem; instead, they are implemented directly in HOL4. An OpenTheory import tool could make this simpler and eliminate the need for Lem. There is an existing import tool,[1] but major technical obstacles prevent it from being used on large projects like CakeML.
2. Presently, there is no export tool from Isabelle to OpenTheory because of various features that the Isabelle kernel supports, but that are not present in OpenTheory. In the future, the dictionary construction (§3.1) could play a role in implementing such a tool.

OpenTheory could eventually be used to import the full CakeML formalization into Isabelle, avoiding the verification gap between the CakeML compiler proof and this work. Alternatively, it could be feasible to load HOL4 as an Isabelle application, providing its full kernel as a wrapper around the Isabelle/Pure logic.[2] It is not yet clear which is the more promising approach.

Pattern compilation As discussed in §6.3.6, there is an alternative compilation scheme of patterns occurring on the left-hand side of equations. Using that scheme means that the conflation of abstraction and matching in `pterm` and `sterm` could be removed. This would not require changes to the term algebra, but it would require changes to the correctness proofs of all phases after `nterm`.

Targetting machine types While CakeML supports operations on machine types, e.g. words, my compilation toolchain never emits such operations (§6.8.1.1). The reason is simple: the intermediate term types have no notion of machine types, just terms and values composed out of datatype constructors and abstractions. There are two ways to fix this:

1. Keep the intermediate types as they are and implement a heuristic in the final phase that detects machine operations.
2. Introduce a notion of machine values and operations into the intermediate term types.

I estimate the implementation and verification effort for the first option to be much higher than for the second option. Therefore, I will briefly sketch the implementation approach for the second option.

- The intermediate term types have to be made polymorphic over a type variable μ that denotes the type of machine values. This keeps the formalization abstract over the concrete set of types that are supported. In the simplest case, this could be instantiated as μ = 32 `word` for just 32-bit integers.

[1] `https://github.com/xrchz/isabelle-opentheory`
[2] `https://web.archive.org/web/20190204170746/https://sourceforge.net/p/hol/mailman/message/36404016/`

- A sublocale `machine_term` must be added to the `term` class. It needs to be a locale, because it deals with two type variables, where α is the term type and μ the machine type. The locale provides extensions to the substitution axioms and translations to CakeML literals (Listing 1.1). An open question is whether matching should be extended to allow native values in patterns. If not, this must be removed in a suitable preprocessing step, for which there is precedent in Isabelle [47, §7.3].
- Machine operations are still modelled as constants. Because they require special treatment, they must not have defining equations (similarly to constructors). The `constructors` locale needs to be extended with a set of machine operations.
- The rewriting semantics need to be parametrized over an oracle that assigns meaning to machine operations applied to machine values. For example, the term $\langle 3 + 5 \rangle$ where 3 and 5 are words cannot be evaluated using term rewriting. The oracle would provide the arity of the + operation and evaluate $3 + 5$ to 8.
- The embedding phase needs to be made aware of native values and perform the deep embedding accordingly. Some non-datatypes such as words can then be made instances of the `embed` class.
- The final compiler phase that emits CakeML code can then produce machine literals. It's correctness would need be parametrized on the `native_term` locale and the operations oracle.

Bibliography

[1] Abhishek Anand et al. “CertiCoq: A verified compiler for Coq”. In: *CoqPL’17: The Third International Workshop on Coq for Programming Languages.* 2017. URL: http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf (cited on page 130).

[2] Lennart Augustsson. “Compiling pattern matching”. In: *Functional Programming Languages and Computer Architecture.* Springer Berlin Heidelberg, 1985, pp. 368–381. DOI: 10.1007/3-540-15975-4_48 (cited on page 130).

[3] Lennart Augustsson. “Implementing Haskell Overloading”. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* FPCA ’93. Copenhagen, Denmark: ACM, 1993, pp. 65–73. ISBN: 0-89791-595-X. DOI: 10.1145/165180.165191. URL: http://doi.acm.org/10.1145/165180.165191 (cited on page 48).

[4] Jeremy Avigad et al. “Introduction to Milestones in Interactive Theorem Proving”. In: *Journal of Automated Reasoning* 61.1 (June 2018), pp. 1–8. ISSN: 1573-0670. DOI: 10.1007/s10817-018-9465-5. URL: https://doi.org/10.1007/s10817-018-9465-5 (cited on page 12).

[5] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation.* 2018. URL: https://isabelle.in.tum.de/dist/Isabelle2019/doc/locales.pdf (cited on page 60).

[6] Heiko Becker et al. “Formalization of Knuth–Bendix Orders for Lambda-Free Higher-Order Terms”. In: *Archive of Formal Proofs* (Nov. 2016). http://isa-afp.org/entries/Lambda_Free_KBOs.html, Formal proof development. ISSN: 2150-914x (cited on page 86).

[7] Nick Benton and Chung-Kil Hur. “Biorthogonality, step-indexing and compiler correctness”. In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009.* Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 2009, pp. 97–108. DOI: 10.1145/1596550.1596567. URL: http://doi.acm.org/10.1145/1596550.1596567 (cited on page 130).

[8] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. “Turning Inductive into Equational Specifications”. In: *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2009, pp. 131–146. DOI: 10.1007/978-3-642-03359-9_11 (cited on pages 20, 123).

[9] Stefan Berghofer and Tobias Nipkow. “Executing Higher Order Logic”. In: *Types for Proofs and Programs (TYPES 2000).* Ed. by P. Callaghan et al. Vol. 2277. 2002, pp. 24–40 (cited on page 12).

[10] Stefan Berghofer and Markus Wenzel. "Inductive Datatypes in HOL — Lessons Learned in Formal-Logic Engineering". In: *Theorem Proving in Higher Order Logics.* Ed. by Yves Bertot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 19–36. ISBN: 978-3-540-48256-7 (cited on page 58).

[11] Julian Biendarra et al. "Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic". In: *Frontiers of Combining Systems.* Springer International Publishing, 2017, pp. 3–21. DOI: `10.1007/978-3-319-66167-4_1` (cited on page 58).

[12] Richard Bird and Lambert Meertens. "Nested datatypes". English. In: *Mathematics of Program Construction.* Ed. by Johan Jeuring. Vol. 1422. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 52–67. ISBN: 978-3-540-64591-7. DOI: `10.1007/BFb0054285`.

[13] Jasmin Christian Blanchette. "Automatic Proofs and Refutations for Higher-Order Logic". Dissertation. München: Technische Universität München, 2012.

[14] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. "Automatic Proof and Disproof in Isabelle/HOL". In: *Frontiers of Combining Systems.* Springer, 2011, pp. 12–27.

[15] Jasmin Christian Blanchette and Tobias Nipkow. "Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder". English. In: *Interactive Theorem Proving.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 131–146. ISBN: 978-3-642-14051-8. DOI: `10.1007/978-3-642-14052-5_11`.

[16] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. "Operations on Bounded Natural Functors". In: *Archive of Formal Proofs* (Dec. 2017). `http://isa-afp.org/entries/BNF_Operations.html`, Formal proof development. ISSN: 2150-914x (cited on page 39).

[17] Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. "Formalization of Recursive Path Orders for Lambda-Free Higher-Order Terms". In: *Archive of Formal Proofs* (Sept. 2016). `http://isa-afp.org/entries/Lambda_Free_RPOs.html`, Formal proof development. ISSN: 2150-914x (cited on page 86).

[18] Jasmin Christian Blanchette et al. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *Interactive Theorem Proving.* Springer International Publishing, 2014, pp. 93–110. DOI: `10.1007/978-3-319-08970-6_7` (cited on pages 16, 29, 39, 58, 62, 91).

[19] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Certified Programs and Proofs". In: *Certified Programs and Proofs.* Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 362–377 (cited on page 12).

[20] Robert S. Boyer and J. Strother Moore. "Single-Threaded Objects in ACL2". In: *Practical Aspects of Declarative Languages (PADL 2002).* Ed. by Shriram Krishnamurthi and C. R. Ramakrishnan. Vol. 2257. Lecture Notes in Computer Science. Springer, 2002, pp. 9–27. DOI: `10.1007/3-540-45587-6_3`. URL: `https://doi.org/10.1007/3-540-45587-6_3` (cited on page 12).

[21] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.05 (Sept. 2013), pp. 552–593. DOI: `10.1017/s095679681300018x` (cited on page 48).

[22] Joachim Breitner et al. *Certified HLints with Isabelle/HOLCF-Prelude*. Haskell And Rewriting Techniques (HART). June 2013.

[23] N. G. de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: `http://dx.doi.org/10.1016/1385-7258(72)90034-0`. URL: `http://www.sciencedirect.com/science/article/pii/1385725872900340` (cited on pages 21, 87).

[24] Lukas Bulwahn. "The New Quickcheck for Isabelle". English. In: *Certified Programs and Proofs*. Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 92–108. ISBN: 978-3-642-35307-9. DOI: `10.1007/978-3-642-35308-6_10` (cited on page 19).

[25] Herman Cappelen and Ernest Lepore. "Quotation". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2017. Metaphysics Research Lab, Stanford University, 2017 (cited on page 89).

[26] Kung Chen, Paul Hudak, and Martin Odersky. "Parametric Type Classes". In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. San Francisco, California, USA: ACM, 1992, pp. 170–181. ISBN: 0-89791-481-3. DOI: `10.1145/141471.141536`. URL: `http://doi.acm.org/10.1145/141471.141536` (cited on page 48).

[27] J. H. Cheng and C. B. Jones. "On the usability of logics which handle partial functions". In: *3rd Refinement Workshop*. Ed. by C. Morgan and J. C. P. Woodcock. Springer-Verlag, 1991, pp. 51–69 (cited on page 49).

[28] Adam Chlipala. "A verified compiler for an impure functional language". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 93–106. DOI: `10.1145/1706299.1706312`. URL: `http://doi.acm.org/10.1145/1706299.1706312` (cited on page 130).

[29] Alonzo Church and J. B. Rosser. "Some properties of conversion". In: *Transactions of the American Mathematical Society* 39.3 (Mar. 1936), pp. 472–472. DOI: `10.1090/s0002-9947-1936-1501858-0` (cited on page 85).

[30] Luc J. M. Claesen and Michael J. C. Gordon, eds. *Higher Order Logic Theorem Proving and its Applications, Proceedings of the IFIP TC10/WG10.2 Workshop HOL'92, Leuven, Belgium, 21-24 September 1992*. Vol. A-20. IFIP Transactions. North-Holland/Elsevier, 1993. ISBN: 0-444-89880-8.

[31] Judy Crow et al. *Evaluating, Testing, and Animating PVS Specifications*. Tech. rep. Menlo Park, CA: Computer Science Laboratory, SRI International, Mar. 2001 (cited on page 12).

[32] P Dehornoy and V van Oostrom. “Z, proving confluence by monotonic single-step upperbound functions”. In: *Logical Models of Reasoning and Computation (LMRC-08)* (2008). URL: https://mitpress.mit.edu/books/proof-language-and-interaction (cited on page 85).

[33] Elmar Eder. “Properties of substitutions and unifications”. In: *Journal of Symbolic Computation* 1.1 (Mar. 1985), pp. 31–46. DOI: 10.1016/s0747-7171(85)80027-4 (cited on page 85).

[34] Benja Fallenstein and Ramana Kumar. “Proof-Producing Reflection for HOL”. In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang. Springer International Publishing, 2015, pp. 170–186. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_11 (cited on page 92).

[35] Bertram Felgenhauer et al. “The Z Property”. In: *Archive of Formal Proofs* (June 2016). http://isa-afp.org/entries/Rewriting_Z.html, Formal proof development. ISSN: 2150-914x (cited on page 85).

[36] Amy Felty. “A logic programming approach to implementing higher-Order term rewriting”. In: *Extensions of Logic Programming*. Ed. by L. -H. Eriksson, L. Hallnäs, and P. Schroeder-Heister. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 135–161. ISBN: 978-3-540-47114-1. DOI: 10.1007/BFb0013606 (cited on page 41).

[37] Amy Felty. “Higher-Order Conditional Rewriting in the L-lambda Logic Programming Language”. 1992. URL: http://www.site.uottawa.ca/~afelty/dist/welp92a.pdf (cited on page 41).

[38] Simon Finn, Michael P. Fourman, and John Longley. “Partial Functions in a Total Setting”. In: *Journal of Automated Reasoning* 18.1 (Feb. 1997), pp. 85–104. ISSN: 1573-0670. DOI: 10.1023/a:1005702928286. URL: https://doi.org/10.1023/A:1005702928286 (cited on page 48).

[39] Arthur D. Flatau. “A Verified Implementation of an Applicative Language with Dynamic Storage Allocation”. PhD thesis. University of Texas at Austin, 1992 (cited on page 130).

[40] Yannick Forster and Fabian Kunze. “Verified Extraction from Coq to a Lambda-Calculus”. In: *The 8th Coq Workshop*. 2016. URL: http://www.ps.uni-saarland.de/~forster/coq-workshop-16/abstract-coq-ws-16.pdf (cited on page 130).

[41] Jan Gilcher, Andreas Lochbihler, and Dmitriy Traytel. “Conditional Parametricity in Isabelle/HOL”. Extended abstract. 2017. URL: https://people.inf.ethz.ch/trayteld/papers/itp17poster-cond_param/cond.pdf (cited on pages 38, 39).

[42] Andrew D. Gordon. “A Tutorial on Co-induction and Functional Programming”. In: *Functional Programming, Glasgow 1994*. Ed. by Kevin Hammond, David N. Turner, and Patrick M. Sansom. London: Springer London, 1995, pp. 78–95. ISBN: 978-1-4471-3573-9 (cited on page 21).

[43] Andrew D. Gordon and Cédric Fournet. “Principles and Applications of Refinement Types”. In: *NATO Science for Peace and Security Series - D: Information and Communication Security* 25.Logics and Languages for Reliability and Security (2010), pp. 73–104. ISSN: 1874-6268. DOI: `10.3233/978-1-60750-100-8-73` (cited on page 40).

[44] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* New York, NY, USA: Cambridge University Press, 1993. ISBN: 0-521-44189-7 (cited on page 16).

[45] David A. Greve et al. “Efficient execution in an automated reasoning environment”. In: *J. Funct. Program.* 18.1 (2008), pp. 15–46. DOI: `10.1017/S0956796807006338`. URL: `https://doi.org/10.1017/S0956796807006338` (cited on page 12).

[46] Elsa L. Gunter. “Why we can’t have SML-style datatype Declarations in HOL”. In: *Higher Order Logic Theorem Proving and its Applications, Proceedings of the IFIP TC10/WG10.2 Workshop HOL’92, Leuven, Belgium, 21-24 September 1992.* Ed. by Luc J. M. Claesen and Michael J. C. Gordon. Vol. A-20. IFIP Transactions. North-Holland/Elsevier, 1992, pp. 561–568. ISBN: 0-444-89880-8 (cited on pages 58, 91).

[47] Florian Haftmann. *Code generation from Isabelle/HOL theories.* URL: `https://isabelle.in.tum.de/dist/Isabelle2019/doc/codegen.pdf` (cited on page 133).

[48] Florian Haftmann. “From higher-order logic to Haskell: there and back again”. In: *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM. 2010, pp. 155–158 (cited on page 18).

[49] Florian Haftmann and Tobias Nipkow. “Code Generation via Higher-Order Rewrite Systems”. In: *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings.* Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 103–117. ISBN: 978-3-642-12251-4. DOI: `10.1007/978-3-642-12251-4_9`. URL: `http://dx.doi.org/10.1007/978-3-642-12251-4_9` (cited on pages 18, 26, 27, 30, 48).

[50] Florian Haftmann and Makarius Wenzel. “Constructive Type Classes in Isabelle”. In: *Types for Proofs and Programs: International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers.* Ed. by Thorsten Altenkirch and Conor McBride. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 160–174. ISBN: 978-3-540-74464-1. DOI: `10.1007/978-3-540-74464-1_11`. URL: `http://dx.doi.org/10.1007/978-3-540-74464-1_11` (cited on page 26).

[51] Florian Haftmann et al. “Data Refinement in Isabelle/HOL”. In: *Interactive Theorem Proving (ITP 2013).* Ed. by S. Blazy, C. Paulin-Mohring, and D. Pichardie. Vol. 7998. Lecture Notes in Computer Science. 2013, pp. 100–115 (cited on page 53).

[52] Cordelia V. Hall et al. “Type classes in Haskell”. In: *ACM Transactions on Programming Languages and Systems* 18.2 (Mar. 1996), pp. 109–138. DOI: `10.1145/227699.227700` (cited on page 28).

[53] Philipp Haller et al. *Futures and Promises.* 2012. URL: `http://docs.scala-lang.org/overviews/core/futures.html`.

[54] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. “Logical Relations and Parametricity – A Reynolds Programme for Category Theory and Programming Languages”. In: *Electronic Notes in Theoretical Computer Science* 303 (2014), pp. 149–180. ISSN: 1571-0661. DOI: `http://dx.doi.org/10.1016/j.entcs.2014.02.008`. URL: `http://www.sciencedirect.com/science/article/pii/S1571066114000346` (cited on page 87).

[55] J. Roger Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (Dec. 1969), p. 29. DOI: `10.2307/1995158` (cited on page 18).

[56] Ralf Hinze and Ross Paterson. “Finger trees: a simple general-purpose data structure”. In: *Journal of Functional Programming* 16.02 (2006), pp. 197–217.

[57] Paul Hudak. “Conception, Evolution, and Application of Functional Programming Languages”. In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300. DOI: `10.1145/72551.72554`. URL: `http://doi.acm.org/10.1145/72551.72554` (cited on page 19).

[58] Brian Huffman and Ondřej Kunčar. “Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL”. In: *Certified Programs and Proofs.* Ed. by Georges Gonthier and Michael Norrish. Cham: Springer International Publishing, 2013, pp. 131–146. ISBN: 978-3-319-03545-1 (cited on page 78).

[59] John Hughes. “A novel representation of lists and its application to the function “reverse””. In: *Information Processing Letters* 22.3 (Mar. 1986), pp. 141–144. DOI: `10.1016/0020-0190(86)90059-1` (cited on page 52).

[60] Lars Hupel. “An Algebra for Higher-Order Terms”. In: *Archive of Formal Proofs* (Jan. 2019). `http://isa-afp.org/entries/Higher_Order_Terms.html`, Formal proof development. ISSN: 2150-914x (cited on pages 13, 61, 75, 98).

[61] Lars Hupel. “Certifying Dictionary Construction in Isabelle/HOL”. Preprint. 2018. URL: `https://lars.hupel.info/pub/dict.pdf` (cited on pages 13, 26).

[62] Lars Hupel. “Constructor Functions”. In: *Archive of Formal Proofs* (Apr. 2017). `http://isa-afp.org/entries/Constructor_Funs.shtml`, Formal proof development. ISSN: 2150-914x (cited on pages 13, 50).

[63] Lars Hupel. “Dictionary Construction”. In: *Archive of Formal Proofs* (May 2017). `http://isa-afp.org/entries/Dict_Construction.html`, Formal proof development. ISSN: 2150-914x (cited on pages 13, 26).

[64] Lars Hupel. “Lazifying case constants”. In: *Archive of Formal Proofs* (Apr. 2017). `http://isa-afp.org/entries/Lazy_Case.shtml`, Formal proof development. ISSN: 2150-914x (cited on pages 13, 49).

[65] Lars Hupel and Tobias Nipkow. “A Verified Compiler from Isabelle/HOL to CakeML”. In: *Programming Languages and Systems.* Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 999–1026. ISBN: 978-3-319-89884-1 (cited on pages 13, 75, 96).

[66] Lars Hupel and Yu Zhang. “CakeML”. In: *Archive of Formal Proofs* (Mar. 2018). `http://isa-afp.org/entries/CakeML.html`, Formal proof development. ISSN: 2150-914x (cited on pages 11, 13, 123).

[67] Joe Hurd. “The OpenTheory Standard Theory Library”. In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 177–191. ISBN: 978-3-642-20398-5 (cited on page 132).

[68] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994. ISBN: 9780511663086. DOI: `10.1017/cbo9780511663086` (cited on page 33).

[69] Alexander Krauss. “Automating Recursive Definitions and Termination Proofs in Higher-Order Logic”. Dissertation. München: Technische Universität München, 2009 (cited on pages 16, 37, 41, 55).

[70] Alexander Krauss. “Partial and Nested Recursive Function Definitions in Higher-order Logic”. In: *Journal of Automated Reasoning* 44.4 (2010), pp. 303–336. ISSN: 1573-0670. DOI: `10.1007/s10817-009-9157-2`. URL: `http://dx.doi.org/10.1007/s10817-009-9157-2` (cited on pages 16, 41).

[71] Alexander Krauss and Andreas Schropp. “A Mechanized Translation from Higher-Order Logic to Set Theory”. In: *Interactive Theorem Proving*. Springer Berlin Heidelberg, 2010, pp. 323–338. DOI: `10.1007/978-3-642-14052-5_23`.

[72] Ramana Kumar. *Self-compilation and self-verification*. Tech. rep. UCAM-CL-TR-879. University of Cambridge, Computer Laboratory, Feb. 2016. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-879.pdf` (cited on page 9).

[73] Ramana Kumar and Magnus O. Myreen. “Clocked Definitions in HOL”. In: *CoRR* abs/1803.03417 (2018). arXiv: `1803.03417`. URL: `http://arxiv.org/abs/1803.03417` (cited on page 22).

[74] Ramana Kumar et al. “CakeML: A Verified Implementation of ML”. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’14. San Diego, California, USA: ACM, 2014, pp. 179–191. ISBN: 978-1-4503-2544-8. DOI: `10.1145/2535838.2535841`. URL: `http://doi.acm.org/10.1145/2535838.2535841` (cited on pages 10, 12, 78).

[75] Ramana Kumar et al. “Self-Formalisation of Higher-Order Logic”. In: *Journal of Automated Reasoning* 56.3 (Mar. 2016), pp. 221–259. ISSN: 1573-0670. DOI: `10.1007/s10817-015-9357-x`. URL: `https://doi.org/10.1007/s10817-015-9357-x` (cited on page 9).

[76] Viktor Kuncak. “Developing Verified Software Using Leon”. In: *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. 2015, pp. 12–15.

[77] Ondřej Kunčar. “Correctness of Isabelle’s Cyclicity Checker: Implementability of Overloading in Proof Assistants”. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. CPP ’15. Mumbai, India: ACM, 2015, pp. 85–94. ISBN: 978-1-4503-3296-5. DOI: `10.1145/2676724.2693175`.

[78] Ondřej Kunčar and Andrei Popescu. “A Consistent Foundation for Isabelle/HOL”. In: *Journal of Automated Reasoning* (Jan. 2018). ISSN: 1573-0670. DOI: `10.1007/s10817-018-9454-8`. URL: `https://doi.org/10.1007/s10817-018-9454-8` (cited on pages 18, 93).

[79] Ondřej Kunčar and Andrei Popescu. “From Types to Sets by Local Type Definition in Higher-Order Logic”. In: *Journal of Automated Reasoning* (June 2018). ISSN: 1573-0670. DOI: `10.1007/s10817-018-9464-6`. URL: `https://doi.org/10.1007/s10817-018-9464-6` (cited on page 93).

[80] Ralf Lämmel and Simon Peyton Jones. “Scrap Your Boilerplate with Class: Extensible Generic Functions”. In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’05. Tallinn, Estonia: ACM, 2005, pp. 204–215. ISBN: 1-59593-064-7. DOI: `10.1145/1086365.1086391`. URL: `http://doi.acm.org/10.1145/1086365.1086391` (cited on page 48).

[81] P. J. Landin. “The Mechanical Evaluation of Expressions”. In: *The Computer Journal* 6.4 (Jan. 1964), pp. 308–320. DOI: `10.1093/comjnl/6.4.308` (cited on page 78).

[82] Didier Le Botlan and Didier Rémy. “MLF: Raising ML to the Power of System F”. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’03. Uppsala, Sweden: ACM, 2003, pp. 27–38. ISBN: 1-58113-756-7. DOI: `10.1145/944705.944709`. URL: `http://doi.acm.org.eaccess.ub.tum.de/10.1145/944705.944709` (cited on page 18).

[83] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: `10.1145/1538788.1538814`. URL: `http://doi.acm.org/10.1145/1538788.1538814` (cited on page 130).

[84] Pierre Letouzey. “A New Extraction for Coq”. In: *Types for Proofs and Programs*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Springer, 2003, pp. 200–219 (cited on page 12).

[85] Andreas Lochbihler. “Probabilistic Functions and Cryptographic Oracles in Higher Order Logic”. In: *Programming Languages and Systems (ESOP 2016)*. Ed. by Peter Thiemann. Vol. 9632. LNCS. Springer, 2016, pp. 503–531. DOI: `10.1007/978-3-662-49498-1_20` (cited on pages 38, 39).

[86] Zohar Manna and Amir Pnueli. “Formalization of Properties of Functional Programs”. In: *Journal of the ACM* 17.3 (July 1970), pp. 555–569. DOI: `10.1145/321592.321606` (cited on page 55).

[87] Simon Marlow, ed. *Haskell 2010 Language Report*. 2010. URL: `https://www.haskell.org/onlinereport/haskell2010/` (cited on page 60).

[88] David CJ Matthews and Makarius Wenzel. “Efficient parallel programming in Poly/ML and Isabelle/ML”. In: *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*. ACM. 2010, pp. 53–62.

[89] Jia Meng and Lawrence C. Paulson. "Translating Higher-Order Clauses to First-Order Clauses". In: *Journal of Automated Reasoning* 40.1 (Sept. 2007), pp. 35–60. DOI: 10.1007/s10817-007-9085-y.

[90] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (Dec. 1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cited on page 18).

[91] Robin Milner et al. *The definition of Standard ML (revised)*. MIT Press, 1997 (cited on page 77).

[92] John Garrett Morris. "Type Classes and Instance Chains: A Relational Approach". PhD thesis. Portland State University, 2013. URL: http://homepages.inf.ed.ac.uk/jmorri14/pubs/morris-dissertation.pdf (cited on page 48).

[93] Dominic P. Mulligan et al. "Lem: Reusable Engineering of Real-world Semantics". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 175–188. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628143. URL: http://doi.acm.org/10.1145/2628136.2628143 (cited on pages 10, 123).

[94] Magnus O. Myreen and Scott Owens. "Proof-producing translation of higher-order logic into pure and stateful ML". In: *Journal of Functional Programming* 24.2-3 (May 2014), pp. 284–315. DOI: 10.1017/S0956796813000282. URL: https://www.cambridge.org/core/article/proof-producing-translation-of-higher-order-logic-into-pure-and-stateful-ml/4836EEA7A25F733339A9D23CD6C9F208 (cited on pages 12, 41, 91, 119, 130).

[95] Wolfgang Naraschewski and Markus Wenzel. "Object-oriented verification based on record subtyping in Higher-Order Logic". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 349–366. DOI: 10.1007/bfb0055146 (cited on page 29).

[96] Georg Neis et al. "Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 166–178. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784764. URL: http://doi.acm.org/10.1145/2784731.2784764 (cited on page 130).

[97] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. Springer, Dec. 15, 2014. 298 pp. ISBN: 3319105418. URL: http://www.concrete-semantics.org/ (cited on pages 9, 12, 20).

[98] Tobias Nipkow and Christian Prehofer. "Type Reconstruction for Type Classes". In: *Journal of Functional Programming* 5.2 (1995), pp. 201–224. DOI: 10.1017/S0956796800001325 (cited on pages 33, 35).

[99] Tobias Nipkow and Gregor Snelting. "Type classes and overloading resolution via order-sorted unification". In: *Functional Programming Languages and Computer Architecture*. Springer Berlin Heidelberg, 1991, pp. 1–14. DOI: 10.1007/3540543961_1 (cited on pages 33, 35).

[100] Benedikt Nordhoff, Stefan Körner, and Pet Lammich. “Finger Trees”. In: *Archive of Formal Proofs* (Oct. 2010). https://isa-afp.org/entries/Finger-Trees.shtml, Formal proof development. ISSN: 2150-914x.

[101] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. “Type Classes As Objects and Implicits”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications.* OOPSLA ’10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 341–360. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869489. URL: http://doi.acm.org/10.1145/1869459.1869489 (cited on page 48).

[102] Scott Owens et al. “Functional Big-Step Semantics”. In: *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings.* Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 589–615. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_23. URL: http://dx.doi.org/10.1007/978-3-662-49498-1_23 (cited on pages 11, 20, 123).

[103] Lawrence C Paulson. “A generic tableau prover and its integration with Isabelle”. In: *Journal of Universal Computer Science* 5.3 (1999), pp. 73–87 (cited on page 16).

[104] Lawrence C Paulson. *Isabelle: The Next 700 Theorem Provers.* Tech. rep. Computer Laboratory, University of Cambridge, 1990. URL: https://www.cl.cam.ac.uk/~lp15/papers/Isabelle/chap700.pdf (cited on page 15).

[105] John Peterson and Mark Jones. “Implementing Type Classes”. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation.* PLDI ’93. Albuquerque, New Mexico, USA: ACM, 1993, pp. 227–236. ISBN: 0-89791-598-4. DOI: 10.1145/155090.155112. URL: http://doi.acm.org/10.1145/155090.155112 (cited on page 48).

[106] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X (cited on pages 53, 104, 130).

[107] Gordon Plotkin, Colin Stirling, and Mads Tofte, eds. *Proof, Language, and Interaction: Essays in Honour of Robin Milner (Foundations of Computing).* The MIT Press, 2000. ISBN: 9780262161886 (cited on page 9).

[108] Willard Van Orman Quine. *Word and Object.* MIT University Press Group Ltd, Jan. 11, 1960. ISBN: 0262670011. URL: https://www.ebook.de/de/product/3238067/willard_van_orman_quine_word_and_object.html (cited on page 89).

[109] John C. Reynolds. “Types, Abstraction and Parametric Polymorphism”. In: *IFIP Congress.* 1983, pp. 513–523 (cited on pages 38, 39).

[110] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2011. DOI: 10.1017/CBO9780511777110 (cited on page 21).

[111] Manfred Schmidt-Schauß and Jörg Siekmann. *Unification Algebras: An Axiomatic Approach to Unification, Equation Solving and Constraint Solving*. Tech. rep. SEKI-report SR-88-09. FB Informatik, Universität Kaiserslautern, 1988. URL: `http://www.ki.informatik.uni-frankfurt.de/papers/schauss/unif-algebr.pdf` (cited on page 85).

[112] Natarajan Shankar. “Static Analysis for Safe Destructive Updates in a Functional Language”. In: *Logic Based Program Synthesis and Transformation (LOPSTR 2001)*. Ed. by Alberto Pettorossi. Vol. 2372. Lecture Notes in Computer Science. Springer, 2001, pp. 1–24 (cited on page 12).

[113] Konrad Slind. “Reasoning about Terminating Functional Programs”. PhD thesis. Technische Universität München, 1999. URL: `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss1999111516455` (cited on pages 41, 42, 53, 103, 104, 130).

[114] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 278–293. DOI: `10.1007/978-3-540-71067-7_23`.

[115] Antal Spector-Zabusky et al. “Total Haskell is Reasonable Coq”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: ACM, 2018, pp. 14–27. ISBN: 978-1-4503-5586-5. DOI: `10.1145/3167092`. URL: `http://doi.acm.org/10.1145/3167092` (cited on page 20).

[116] Christian Sternagel and René Thiemann. “Abstract Rewriting”. In: *Archive of Formal Proofs* (June 2010). `http://isa-afp.org/entries/Abstract-Rewriting.html`, Formal proof development. ISSN: 2150-914x (cited on page 85).

[117] Christian Sternagel and René Thiemann. “Deriving class instances for datatypes”. In: *Archive of Formal Proofs* (Mar. 2015). `http://isa-afp.org/entries/Deriving.shtml`, Formal proof development. ISSN: 2150-914x (cited on page 58).

[118] Christian Sternagel and René Thiemann. “First-Order Terms”. In: *Archive of Formal Proofs* (Feb. 2018). `http://isa-afp.org/entries/First_Order_Terms.html`, Formal proof development. ISSN: 2150-914x (cited on page 85).

[119] Christian Sternagel and René Thiemann. “Haskell’s Show Class in Isabelle/HOL”. In: *Archive of Formal Proofs* (July 2014). `http://isa-afp.org/entries/Show.html`, Formal proof development. ISSN: 2150-914x (cited on page 52).

[120] W. W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212. ISSN: 00224812. DOI: `10.2307/2271658`. URL: `http://www.jstor.org/stable/2271658` (cited on page 87).

[121] Yong Kiam Tan et al. “A new verified compiler backend for CakeML”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*. Association for Computing Machinery (ACM), 2016. DOI: `10.1145/2951913.2951924` (cited on pages 10, 12, 123).

[122] *The HOL System Description*. 2014. URL: `https://hol-theorem-prover.org/` (cited on page 10).

[123] D. A. Turner. “Some History of Functional Programming Languages”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 1–20. DOI: `10.1007/978-3-642-40447-4_1` (cited on page 78).

[124] Christian Urban. “Nominal Techniques in Isabelle/HOL”. In: *Journal of Automated Reasoning* 40.4 (2008), pp. 327–356. ISSN: 1573-0670. DOI: `10.1007/s10817-008-9097-2`. URL: `http://dx.doi.org/10.1007/s10817-008-9097-2` (cited on page 98).

[125] Christian Urban, Stefan Berghofer, and Cezary Kaliszyk. “Nominal 2”. In: *Archive of Formal Proofs* (Feb. 2013). `http://isa-afp.org/entries/Nominal2.shtml`, Formal proof development. ISSN: 2150-914x (cited on page 98).

[126] Philip Wadler and Steven Blott. “How to Make Ad-hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: ACM, 1989, pp. 60–76. ISBN: 0-89791-294-2. DOI: `10.1145/75277.75283`. URL: `http://doi.acm.org/10.1145/75277.75283` (cited on pages 26, 48).

[127] Makarius Wenzel. “Isabelle/Isar — a versatile environment for human-readable formal proof documents”. PhD thesis. Technische Universität München, 2002. URL: `https://mediatum.ub.tum.de/doc/601724/601724.pdf` (cited on page 16).

[128] Markus Wenzel. “Type classes and overloading in higher-order logic”. In: *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs ’97 Murray Hill, NJ, USA, August 19–22, 1997 Proceedings*. Ed. by Elsa L. Gunter and Amy Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 307–322. ISBN: 978-3-540-69526-4. DOI: `10.1007/BFb0028402`. URL: `http://dx.doi.org/10.1007/BFb0028402` (cited on page 26).

[129] James G. Williams. *Instantiation Theory*. Springer-Verlag, 1991. DOI: `10.1007/bfb0031932`. URL: `https://doi.org/10.1007%2Fbfb0031932` (cited on pages 85, 86).

[130] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. “Verified Memoization and Dynamic Programming”. In: *Interactive Theorem Proving*. Springer International Publishing, 2018, pp. 579–596. DOI: `10.1007/978-3-319-94821-8_34` (cited on page 40).

[131] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Feb. 11, 1993. 384 pp. ISBN: 9780262231695. URL: `https://mitpress.mit.edu/books/formal-semantics-programming-languages` (cited on page 19).

Index