# Theorem Provers as Libraries – An Approach to Formally Verifying Functional Programs

By

Evan Austin

Submitted to the Electrical Engineering and Computer Science Department and the
Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

| | |
|---|---|
| | Perry Alexander, Chairperson |
| | Arvin Agah |
| Committee members | Andy Gill |
| | Prasad Kulkarni |
| | Erik Van Vleck |

Date defended: ______________________

The Dissertation Committee for Evan Austin certifies

that this is the approved version of the following dissertation :

Theorem Provers as Libraries – An Approach to Formally Verifying Functional Programs

Perry Alexander, Chairperson

Date approved:

## Abstract

Property-directed verification of functional programs tends to take one of two paths. First, is the traditional testing approach, where properties are expressed in the original programming language and checked with a collection of test data. Alternatively, for those desiring a more rigorous approach, properties can be written and checked with a formal tool; typically, an external proof system. This dissertation details a hybrid approach that captures the best of both worlds: the formality of a proof system paired with the native integration of an embedded, domain specific language (EDSL) for testing. At the heart of this hybridization is the titular concept – a theorem prover as a library. The verification capabilities of this prover, HaskHOL, are introduced to a Haskell development environment as a GHC compiler plugin. Operating at the compiler level provides for a comparatively simpler integration and allows verification to co-exist with the numerous other passes that stand between source code and program.

# Acknowledgements

TODO

# Contents

# List of Figures

# List of Code Examples

# 1 Introduction

Property-directed verification of functional programs tends to follow one of two paths: testing or formal reasoning. Each approach has its own advantages and disadvantages. Software test cases, while easy to implement, are unable to provide a general guarantee of their completeness. Conversely, formal reasoning, with its basis in sound logical and mathematical principles, provides that missing guarantee; but at a cost of more complex system integration and increased pre-requisite knowledge. This dissertation details a hybridization of these approaches with the goal of providing a path to verification that captures the "best of both worlds" with as little of the "worst" as possible.

My primary technical contribution is the development of an interactive, higher-order logic (HOL) [29] theorem prover that can integrate natively with a functional programming language. This integration allows both implementation and verification to reside within the same environment, permitting the construction of the logical bridge between the two at any level of development. This freedom enables what I feel is the most important, and novel, portion of my contribution: a linkage between implementation and verification that exists at the core, or intermediate language, level. Given that functional programming languages are rooted in the same lambda calculus [14, 15] that serves as the foundation of HOL, this linkage exposes a number of benefits not seen in related verification workflows.

Specifically, it is my claim that my work provides the following benefits not currently observed by the existing, related work:

- Integrating natively is simpler, in that no external tools are required to marshall information from program to prover, or vice versa.
- Using the compiler to desugar higher-level language features before translation reduces the complexity of the required representational logic.
- Implementing formal reasoning capabilities in a DSL style allows for their reuse in a variety of applications beyond verification.

Again, these benefits arise as a result of melding formal reasoning with the predominant method of verification, software testing.

Testing's pervasive use is not surprising when you consider its chief appeal: software implementation and software testing can be handled by the same individual. The last statement is made under the belief that writing a program and writing its associated test cases requires the same skill set and knowledge. Take, for example, a basic implementation of lists and their `append` and `reverse` operations written in a functional language, as shown in Listing 1.1.

Listing 1.1: A Basic List Implementation

```
data List a = Nil | Cons a (List a)

append :: [a] -> [a] -> [a]
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

reverse :: [a] -> [a]
reverse Nil = Nil
reverse (Cons x xs) = append (reverse xs) (Cons x Nil)
```

One possible set of tests for this implementation is shown in Listing 1.2. Note that these unit tests require only basic knowledge of list construction and the expected behavior of `reverse`; knowledge that the implementor should already possess. The implementation of `reverse`, including its internal use of `append`, can be treated as a black box and ignored. Conducting tests in this way, especially when aided by a testing framework or library, works well. The consequence of approaching verification in such a manner, is that showing a set of tests exhaustively cover a property ranges from difficult to impossible. Thus, the argument of correctness frequently cannot claim to be an irrefutable guarantee. In this specific case, to make such a claim, the inductive nature of lists needs to be leveraged. Shifting from informal testing to a formal structural induction proof provides us with the coverage guarantee we desire.

Listing 1.2: Testing a List Implementation

```
do test (reverse Nil == Nil)
        "Test (reverse []) failed."
   test (reverse (Cons 1 Nil) == (Cons 1 Nil))
        "Test (reverse [1]) failed."
   test (reverse (Cons 1 (Cons 2 Nil)) == (Cons 2 (Cons 1 Nil)))
        "Test (reverse [1, 2]) failed."
```

Given the applicative nature of functional languages, the formal reasoning tools most commonly turned to for verification purposes utilize a logic that supports equational reasoning. These tools generally take the form of a theorem prover, with verification proceeding as the backwards, or sometimes forwards, proof of propositional goals. Continuing with the example above, one could formally demonstrate the correctness of `reverse` by proving the function's implementation to be involutary:

```
forall xs. reverse (reverse xs) == xs
```

As was hinted at in the preceding paragraph, the proof of the above property involves an inductive, case-wise analysis of the structure of the quantified list:

```
reverse (reverse Nil) == Nil
reverse (reverse (Cons x xs')) == Cons x xs'
```

The resultant subgoal proofs are a series of equational rewrites based on the definitions of `append` and `reverse` and the distributive property of `append`; these proofs are shown in Figures 1.1 and 1.2 accordingly. The simplicity of these proofs reflects the appeal of equational reasoning and referential transparency: when provided with a context containing the requisite definitions and lemmas, most verification obligations can be discharged via induction and simplification (rewriting).

Figure 1.1: A Proof of `reverse`'s Involutarity (Part 1)

Base Case:

$$
\begin{aligned}
& \mathit{reverse}\ (\mathit{reverse}\ \mathit{Nil}) & & \\
&= \mathit{reverse}\ \mathit{Nil} & &\text{By Definition of } \mathit{reverse} \quad (1)\\
&= \mathit{Nil} & &\text{By Definition of } \mathit{reverse} \quad (2)
\end{aligned}
$$

Inductive Case:

$$\mathit{reverse}\ (\mathit{append}\ \mathit{xs}\ \mathit{ys}) = \mathit{append}\ (\mathit{reverse}\ \mathit{ys})\ (\mathit{reverse}\ \mathit{xs}) \qquad \text{(Distributive Lemma)}$$

$$\begin{aligned}
&\mathit{reverse}\ (\mathit{reverse}\ (\mathit{Cons}\ x\ \mathit{xs}')) && \\
&= \mathit{reverse}\ (\mathit{append}\ (\mathit{reverse}\ \mathit{xs}')\ (\mathit{Cons}\ x\ \mathit{Nil})) && \text{By Definition of } \mathit{reverse} \quad (1)\\
&= \mathit{append}\ (\mathit{reverse}\ (\mathit{Cons}\ x\ \mathit{Nil}))\ (\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}')) && \text{By Distributive Lemma} \quad (2)\\
&= \mathit{append}\ (\mathit{append}\ (\mathit{reverse}\ \mathit{Nil})(\mathit{Cons}\ x\ \mathit{Nil}))\ (\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}')) && \text{By Definition of } \mathit{reverse} \quad (3)\\
&= \mathit{append}\ (\mathit{append}\ \mathit{Nil}\ (\mathit{Cons}\ x\ \mathit{Nil}))\ (\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}')) && \text{By Definition of } \mathit{reverse} \quad (4)\\
&= \mathit{append}\ (\mathit{Cons}\ x\ \mathit{Nil})\ (\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}')) && \text{By Definition of } \mathit{append} \quad (5)\\
&= \mathit{Cons}\ x\ (\mathit{append}\ \mathit{Nil}\ (\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}'))) && \text{By Definition of } \mathit{append} \quad (6)\\
&= \mathit{Cons}\ x\ (\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}')) && \text{By Definition of } \mathit{append} \quad (7)\\
&= \mathit{Cons}\ x\ \mathit{xs}' && \text{By Inductive Hypothesis} \quad (8)
\end{aligned}$$

Figure 1.2: A Proof of `reverse`'s Involutarity (Part 2)

While the proof process can be quite simple, using proof tools themselves can be quite the opposite. Where as software testing can be performed with minimal knowledge of the source language and problem domain, interacting with a theorem prover for the purposes of program verification requires expert knowledge of the afore mentioned topics, as well as the proof system itself. An additional challenge is introduced when following the formal reasoning approach – the implementation and verification of a program now reside in separate systems with separate languages. Even if these languages are logically close, any conversion between the two implies a necessarily more complex system integration than is required by the testing approach.

This is not to say that such integrations are impossible, or even uncommon. There have been a number of successful verification efforts involving functional programs, ranging from the small [1, 45] to the quite large [53, 58]. The chosen routes from implementation to verification in these projects also vary quite widely. Some elect to iteratively abstract from implementation to specification, providing a bridge to the verification in incremental steps [54]. Others elect a more direct approach, utilizing external tools to automatically generate either a specification from an implementation [35], or vice versa [60]. In either case, the development pipeline is now augmented with a large proof system that tends to be overkill for all but the most complex of problems.

It is my belief that the simplicity of the testing approach is rooted in the fact that the verification effort never leaves the source language's environment. In order to bring that same simplicity to the formal reasoning approach, the tool to be utilized must be implemented in a way such that its proof artifacts can be handled as first class objects in the target source language. For the work of this dissertation, I choose to satisfy this requirement by implementing a HOL theorem prover as an embedded domain specific language (EDSL), with proof terms and theorems acting as the principal data types for the DSL.

This works exposes two interesting, auxiliary benefits of note. First, the LCF implementation style [31] popularized by HOL theorem provers pairs quite naturally with the EDSL approach [44], leading to, what is in my opinion, a comparatively clear and direct implementation of a general theorem prover, as will be discussed in Chapter 4. Second, given that many of the popular testing frameworks for functional languages are also implemented as EDSLs [17] the resultant proof system can reuse a large number of existing test cases as verification targets. This was one of the primary motivations of the work, as will be explained in more detail in Chapter 2.

The techniques presented in this dissertation should be applicable to any functional language with the required features, namely support for user-written compiler plugins. The concrete implementation of said techniques utilizes the Haskell programming language [62]; specifically, the version of Haskell supported by the Glasgow Haskell Compiler (GHC) [25]. The resultant HOL EDSL, cleverly (or not so) named *HaskHOL* is available as a series of Haskell packages.

Stable versions of some of these packages are hosted on the Hackage package repository (`https://hackage.haskell.org`). Unstable, but cutting edge, versions are available from my personal Github account (`https://github.com/ecaustin`). Additional information regarding the HaskHOL system, as well as related publications, is available at `http://haskhol.org`.

# 2 Motivation

The motivation for the work in this dissertation is rooted in the origin story of the HaskHOL proof system:

My time as a graduate student began in support of Rosetta, a specification language for system-level design [3] whose development was spearheaded by the System Level Design Group at the University of Kansas. By the time I joined the research group, a fairly robust tool suite had already been developed to accompany the Rosetta language standard – with tools ranging from type checking frontends to hardware synthesis backends. What was missing, though, was a way to formally reason about the specifications we were writing.

Attempts had been made to correct this deficiency before my arrival; most notably involving work with the VSPEC [8] and Prufrock [92] systems. As will be discussed in the next subsection, the issue with attempting verification using external systems such as these is that, in general, it is extremely difficult to restructure their output to be meaningful within the context of the original specification. We desired a system that could integrate seamlessly into the Rosetta tool suite to avoid this problem.

The rest of the Rosetta tool suite was implemented using Haskell. At the time, there was a dearth of Haskell-based formal reasoning tools so, regrettably, we found no off-the-shelf solution for what we wanted. Guided by prior experience and familiarity with proof systems, including PVS [20] and the previously mentioned Prufrock, we started down the path of developing a purpose-built theorem prover for Rosetta specifications. We decided on a foundational logic for the prover when a literature search lead us to higher-order logic (HOL). In addition to having a rich history and an active community behind it, HOL is a great formalism for hardware verification [30, 64] which was the primary domain for Rosetta specification at the time.

After surveying the popular members of the HOL theorem prover family tree [81, 69, 41] we decided to follow the titular “lightweight” approach of John Harrison’s HOL Light system. In fact, our earliest attempt to implement a proof tool for Rosetta was a naive translation of HOL Light’s OCaml implementation to Haskell [6]; thus, HaskHOL was born. Having gone through a number of revisions and reimplementations since then, HaskHOL has morphed into a more general tool; a library designed to provide proof capabilities to any Haskell-based program that imports it [5]. This transformation opened the door to the work of this dissertation: a novel approach to formally verifying functional languages. The following sections detail the motivation behind the design of this new workflow, as well as what I see as the two most obvious cases to motivate its use.

## 2.1 A Closed System Approach to Verification

In physics, a *closed system* is a system which does not allow transfers of certain types across its boundaries. For example, a thermodynamically closed system can exchange heat or work with its surroundings, but not matter. The science of calorimetry depends on systems restricted in this way, as the process of calculating a reaction's net energy is greatly simplified when changes in mass can be removed from the equation. Analogously, it is my belief that a verification effort can be simplified by removing the marshaling of information from its "equation."

The problem with most verification workflows is that they require a specification to be "thrown over the wall" that stands between the development environment and the verification environment. Provided that both environments are rooted in compatible formalisms, this "tossing" of a specification back and forth reduces to a translation between language and logic. Focusing specifically on theorem proving, Florian Haftmann has shown that, following the spirit of the Curry-Howard isomorphism [83], these translations between proof and program can be automated [35]. Unfortunately, the information contained in, and/or represented by, the resultant artifacts of either environment can not as easily be shared.

The opening to this chapter stated the problem quite clearly: "in general, it is extremely difficult to restructure [proof objects] to be meaningful within the [development environment]." Outside of a prover, a theorem represents nothing more than an affirmation of the validity of a property. Its structure cannot be dissected or analyzed, its requisite assumptions cannot be checked, and its proof cannot be rerun or otherwise validated. The consequence is that most workflows match the diagram shown in Figure 2.1; development and verification are distinct and separate systems with compatible input and incompatible outputs.

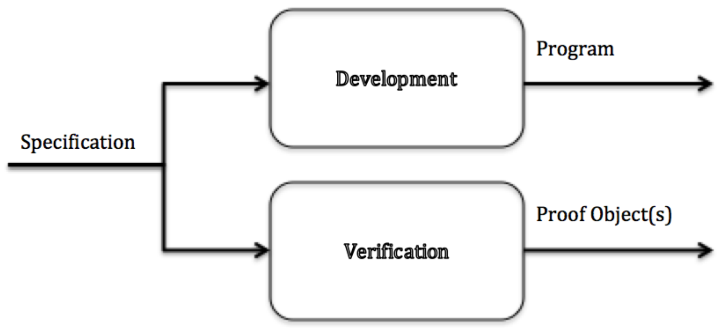
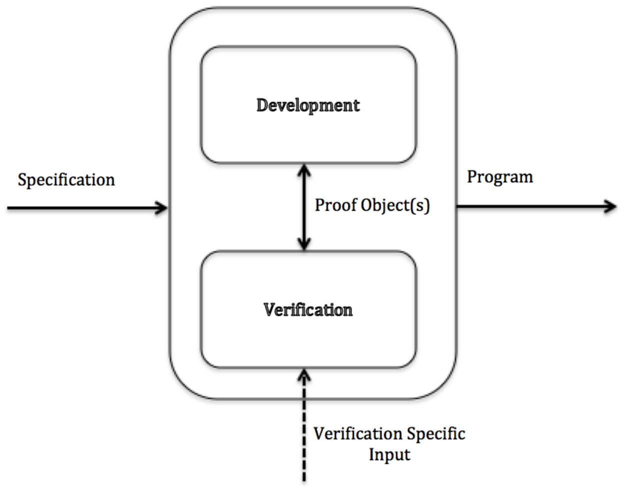


Figure 2.1: An Open System Approach to Verification

The work in this dissertation is motivated by a different workflow, shown in Figure 2.2, where development and verification are subsystems within a larger system closed to proof objects. There are two keys to this design that should be observed. First, the development and verification environments share a host language, such that both can interact natively with the abstract data representation of proof objects. Second, the boundary to this host system is semi-permeable, such that a user can interact with the verification system, but not the proof objects themselves. These design requirements are critical to the success and soundness of the overall system.

Figure 2.2: A Closed System Approach to Verification



## 2.2 A Proposal for Type Class Laws

One of Haskell's more unique contributions to programming language research is its approach to ad hoc polymorphism: *type classes* [37]. Operations can be made overloadable by enclosing their declaration within a class that binds the type variable to be made polymorphic. For example, the equality operator, `(==)`, is contained within the `Eq` type class:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Overloads are introduced via type class instances which instantiate the class parameter and provide definitions for the monomorphic view of the class operations. Continuing with the equality example, one possible instance of `Eq` for `Bool`eans is shown below:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

When an overloaded operation is used in another definition or expression, the requisite type class is introduced as a constraint to the overall type, e.g.:

```
elem :: Eq a => a -> [a] -> Bool
```

At compile time, type inference is used to resolve which type, if any, has an instance that satisfies this constraint and can unify with the polymorphic type variable [50]. The pertinent definitions from that type's instance are inlined into the original body of code, providing a dictionary passing implementation of polymorphism. Given that constraint resolution is entirely type directed, it is critical that definitions in class instances are well-typed. This typing is currently the only sanity or safety check that is performed on class instances, such that the compiler will gladly accept "incorrect" definitions for overloads:

```
instance Eq Bool where
  _  == _ = False
```

The author of a type class usually defines '*correctness* by pairing the class declaration with a set of properties to guide the implementation of class instances. In the case of value equality, there are a number of properties that instances of `Eq` should satisfy – the simplest being the reflexive property: `forall x. x == x = True`. The incorrect definition shown above is obviously invalidated by this property, so why does the compiler continue to accept it? In short, *there is no formal way to specify or check these properties within the Haskell language.*

Typically, properties are documented in source code comments tied to type class declarations. Instance implementors reason away their obligation to correctness with arguments ranging from the hand waving, to formal proofs of correctness. The more rigorous end of this spectrum is most commonly occupied by test suites that cover the definitions of an instance. This technique works well for basic type classes, such as `Eq`, but can fail in more complicated cases. The main problem is that testing frameworks tend to struggle with code that is polymorphic [10]; a fact that will be expounded upon in the next section.

The discussion up to this point has dealt with a type class whose parameter is of kind `*`, such that type instantiation results in monomorphic code. In the case where the bound variable is higher kinded[1], e.g. `* -> *`, type instantiation results in parametrically polymorphic code. The primitive type classes of Haskell based on category theory constructs, `Functor`, `Monad`, etc., are common, real-world examples.

The correctness of `Monad` instances is of particular note, given that Haskell's desugaring of `do` notation depends on rewrite rules which are literal translations of the monad laws. Haskell's declaration of the `Monad` class, along with its correctness properties, is shown in Listing 2.1.

[1]Higher kinded type classes are sometimes referred to as constructor classes, referring to the type constructors that instantiate their parameter

Listing 2.1: The `Monad` Type Class

```
{-
Instances of 'Monad' should satisfy the following laws:

> return a >>= k  ==  k a
> m >>= return  ==  m
> m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
-}
class Monad m where
  (>>=)  :: forall a b. m a -> (a -> m b) -> m b
  return :: a -> m a
```

Instantiating the `Monad` class with even the simplest possible type constructor, `Identity`, creates issues for testing. This instantiation, along with a derivation of the corresponding instance of the left identity monad law, is shown in Figure 2.3. By inlining the definitions of both `>>=` and `return`, we are left with the testing property formed by equating the final two steps in the figure. This property contains two universally quantified variables, both of which are polymorphic. The full property, including its type quantifications, is shown below:

```
FORALL A B. forall (k :: A -> Identity B) (a :: A).
    k (runIdentity (Identity a)) == k a
```

Figure 2.3: The `Identity` `Monad`

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

Left Identity Law:

$$\begin{aligned}
& return\ a >\!>= k && \\
&= k\ (runIdentity\ (return\ a)) && \text{By Definition of } (>\!>=) \quad (1)\\
&= k\ (runIdentity\ (Identity\ a)) && \text{By Definition of } return \quad (2)\\
&= k\ a && \text{By Definition of } runIdentity \quad (3)
\end{aligned}$$

In the previously cited paper, Bernardy et. al present an approach to testing polymorphic properties, including an extension that supports properties with multiple type parameters. However, it is not readily apparent if that technique is applicable in cases where a parameter needs to represent an arbitrary, functional value, as is the case for the parameter `k` in this example property. Claessen presents a refinement of his work with Bernardy et. al specifically to address this problem; however, the newer approach still has limitations of its own [16]. Following either technique, there is a risk of producing an inefficient, and potentially incomplete, test suite bogged down by unnecessary redundancy. Or worse yet, the resultant test suite will be logically weaker than you need it to be.

Conversely, the "paper proof" from Figure 2.3 can be easily, and formally, proved using a higher-order logic that supports type quantification. One possible, subgoal-directed proof is shown in Figure 2.4.

Figure 2.4: Proof of Left Identity Law for `Identity` Monad

Original Goal:

$$FORALL\ A\ B.\ forall\ (k :: A \to Identity\ B)\ (a :: A).\ k\ (runIdentity\ (Identity\ a)) = k\ a \tag{1}$$

$$forall\ (k :: A' \to Identity\ B')\ (a :: A').\ k\ (runIdentity\ (Identity\ a)) = k\ a \quad \text{By Repeated Type Generalization} \tag{2}$$

$$k'\ (runIdentity\ (Identity\ a')) = k'\ a' \quad \text{By Repeated Term Generalization} \tag{3}$$

Equality of Functions Subgoal:

$$k' = k' \tag{4}$$

$$True \quad \text{By Reflexivity} \tag{5}$$

Equality of Arguments Subgoal:

$$runIdentity\ (Identity\ a') = a' \tag{6}$$

$$a' = a' \quad \text{By Definition of } runIdentity \tag{7}$$

$$True \quad \text{By Reflexivity} \tag{8}$$

This motivating example has two key parts that guided the work presented in this dissertation. First, it was clear that the Haskell community as a whole would benefit from a standardized way to formally state type class properties. Second, it was my belief that a mechanization of the steps shown in Listing 2.1, Figure 2.3, and Figure 2.4 would lead to a higher level of assurance than was currently provided by a testing approach. This mechanization could be realized as an implementation and application of the verification workflow shown in the preceding section.

## 2.3 Augmenting Informal Tests with Formal Proofs

To paraphrase a theme of the previous section, there are times when a testing approach to verification may be inadequate. The previously cited work of Bernardy, Jansson, and Claessen focused on a specific instance of this problem – finding solutions for testing otherwise untestable polymorphic properties. In a Haskell mailing list post announcing the publication of this work[2], Bernardy summarizes the crux of their solution. In short, you *can* test polymorphic properties, but to do so you must coerce them to be monomorphic. We follow Bernardy's posted example to both explain this approach and document the remaining open issues.

The property in Listing 2.2 is flawed, as the arguments to `(++)` are not properly commuted on the right-hand side of the equation. Fixing the polymorphic type variable by defaulting to the simplest type, `()`, results in a monomorphic view of this property that obscures this flaw rather than reveals it. The cause of the obscurity lies in the definition of the unit type, `()`; so named because it is inhabited by only a single term value. Given that every element in a list of type `[()]` must be equal, comparisons between such lists can be reduced to a comparison of their lengths. Provided that the implementations of `reverse` and `(++)` are length preserving, as they should be, fixing the type of `prop_reverse_append` to `[()] -> [()] -> Bool` makes it a tautology, even though its polymorphic counterpart is refutable.

Listing 2.2: An Example Polymorphic Testing Property

```
prop_reverse_append :: Eq a => [a] -> [a] -> Bool
prop_reverse_append xs ys =
    reverse (xs ++ ys) == reverse xs ++ reverse ys
```

[2]https://www.haskell.org/pipermail/haskell/2009-October/021657.html

Prior to the publication of Bernardy et. al's work, the popular technique was to fix polymorphic type variables to a type with a very large term space, usually `Integer`, specifically to avoid this problem. Even with a countably infinite term space to draw from, there are still potential issues that can arise when selecting test data. The most concerning of these issues is that a seemingly complete set of tests may actually be significantly weaker than necessary to verify their target property.

Recall that the flaw in `prop_reverse_append` is a missing commutation of arguments when appending lists. When the argument lists are equal, though, the commuted and uncommuted versions of an append expression are reflexively equal. Thus, when the input lists `xs` and `ys` are equal, `prop_reverse_append` will always test true. Following from the previous discussion, this should be an obvious result for lists that hold homogeneous values, e.g. `['a'], [1, 1], [(), (), ()]`, etc., regardless of which monomorphic type is selected. However, when testing with equal, heterogeneous lists, it may not be immediately obvious that weakened test data has been selected.

Take, for example, the following set of test data where each item is a pairing of inputs for the lists `xs` and `ys` accordingly: `([], []), ([1], [1]), and ([1, 2], [1, 2])`. Upon first inspection it appears that all of the necessary test cases are covered – empty lists, unary lists, and `n`-ary lists. However, as discussed in the preceding paragraph, testing with this data will not reveal the flaw in `prop_reverse_append` given that each pair is comprised of equal lists. The inclusion of an additional test pair that doesn't satisfy this property, specifically `([], [1])`, is still not enough to strengthen the test suite to the point of finding a refutation, as it is a member of another logically weak class of tests for this property; `prop_reverse_append` is vacuously true for list pairs whose combined length is 1 or less.

Randomly generating test data can strength its collective argument by making it probabilistically unlikely that every case is weak. However, random generation can simultaneously aggravate secondary issues with testing – inefficiency and incompleteness. As noted above, `prop_reverse_append` is true for any pairing of a null and unary list. Thus, if the property is tested with the pair `([], [1])` then repeating the test with either the pairing's permutation or a different unary list would represent a wasted effort. Given that only a fixed number of tests will be performed, in addition to being inefficient, redundant tests jeopardize a test suite's odds of revealing a refutation, should any exist

Analyzing all of the possible ways a property can construct values from its input data can reveal techniques for mitigating the potential weakness and redundancy of said data. In the case of `prop_reverse_append`, there are two input lists, `xs` and `ys`, that are constrained by the same polymorphic variable. If the combined elements of the two lists form a set of entirely distinct values then weakness is no longer a concern, as `xs` and ys can never be equal. Generating lists of exclusively unique values can also help reduce redundancy, as their complexity subsumes that of similarly lengthed lists, e.g. testing with `[1, 2, 3]` sufficiently replaces tests for `[1, 1, 1]`, `[1, 1, 2]`, etc. Thus, the property only needs to be tested once for each combination of list lengths.

This approach, especially Claessen's later refinements of it, goes a long way towards making polymorphic properties testable. However, there are still open problems that need to be solved. For one, this approach to data generation cannot be made general for higher-order polymorphic arguments, only first-order ones. Secondly, identifying methods of construction for a property can be extremely difficult, especially in the presence of type classes or other methods of indirection. Third, it is unclear if this approach is applicable to non-equational properties, nor is it clear how it would need to be modified in order to make it so. Finally, as has been harped upon up to this point, this approach to data generation still does not address the issue of guaranteeing the completeness of a test suite.

Even with these outstanding problems, it is my belief that testing still has a place somewhere in the overall verification of a system. Formal reasoning tools can provide strong, mathematical arguments for the correctness of a property when a verification effort succeeds, however, when a verification effort fails it typically reveals very little about the problem at hand. Conversely, when a test is found that refutes a property, it represents a concrete counter-example to work from; something that can be very hard to derive when working with formal systems, proof tools especially. Additionally, given the "black box" nature of testing, an initial verification attempt can be made with minimal to no knowledge of the implementation of the system involved.

Given these benefits, a hybrid approach to verification where testing is used to preface formal reasoning would seem to provide the best of both worlds: a quick path to possible refutation and a formal verification to support an otherwise informal argument of correctness. The workflow presented in Section 2.1 can be adapted to mechanize this proposed hybridization. The only difference compared to the motivating example from Section 2.2 is that properties are now captured via code fragments in the implementation language itself rather than in a special language construct, such that they must be translated alongside the code they target.

A true hybridization would generate both a monomorphic testing property and a proof obligation from the same polymorphic source. More recent versions of the QuickCheck library [17] provide an experimental mechanism for automatically monomorphising a property, satisfying the testing half of the proposed hybrid. Template Haskell [80] is used to inspect the property's polymorphic type signature, generate a monomorphic view of it, and ascribe this new type to the original property:

```
prop_reverse_append' :: Integer -> Integer -> Bool
prop_reverse_append' = $(monomorphic 'prop_reverse_append)
```

When test suites are implemented in this manner we can essentially "recycle" them for formal verification.

# 3 Background

The requisite knowledge to understand the technical content of this dissertation falls into two main categories:

1. Functional Programming
2. Higher-Order Logic

Section 3.1 covers the functional programming background material. Starting with a concise introduction to the Lambda Calculus, select axes of Barendregt's Lambda Cube are explored to develop a basic understanding of the type theory related to the foundational logics of both Haskell and HaskHOL.

Section 3.2 covers the HOL background material. The basics of higher-order logic, as mechanized by the original HOL theorem prover, are presented building from the logical kernel up.

## 3.1 Functional Programming

In the early 1930s, Alonzo Church was pursuing a way to formally model a theory of computable functions [13]. Alan Turing, a student of Church, followed this work with his own model of computability [88], eventually showing the two to share the same level of expressivity [89]. Subsequently, these two formalisms grew to become the prominent models of computation in computer science: the $\lambda$-calculus and the Turing Machine, respectively. Modern functional languages can be viewed as natural elaborations of the original $\lambda$-calculus, frequently utilizing some variant of it as their core language. This section works through a derivation of these elaborations, starting with Church's original formulation.

### Church's $\lambda$-calculus

The chief appeal of the $\lambda$-calculus is its simplicity. Its syntax is defined by only three possible constructs: variables, abstractions, and combinations. The grammar for this language is shown back in Figure 3.1. Briefly summarized:

- Variables model an identifier for a term.
- Abstractions model a function definition by binding a variable name, such that it can be referenced in the body of the abstraction.
- Combinations model a function application by pairing a function/operator term with an argument/operand term.

Figure 3.1: BNF Grammar for the $\lambda$-calculus

$$\begin{array}{lcl}
\text{variable} & ::= & \text{a} \mid \text{b} \mid \ldots \\
\\
\text{term} & ::= & \text{variable} \\
 & \mid & \lambda\ \text{variable}\ .\ \text{term} \\
 & \mid & \text{term term}
\end{array}$$

The only method of computation in the pure $\lambda$-calculus is reduction, as powered by variable substitution. In the body of the term $\lambda x.\ x\ y$ there are two variable occurrences, $x$ and $y$. The variable $x$ is considered *bound*, as it is introduced within the scope of a lambda binding, $\lambda x$. Conversely, the variable $y$ is considered *free* as it has no introduction to the term, i.e. it is scoped globally. In the instance where a variable has multiple occurrences in a term, each occurrence's scope is calculated from the inside out. For example, in the term $\lambda x.\ y\ (\lambda x.\ \lambda y.\ x\ y)$, the outermost binding of $x$ is unused, the outermost occurrence of $y$ is free, and the innermost occurrences of $x$ and $y$ are both bound by the corresponding, remaining lambdas.

Variable substitution is notated by the short-hand $[x \mapsto e]\, t$, indicating that occurrences of the variable $x$ in the term $t$ are replaced by the expression $e$. In the interest of brevity, the actual definition of substitution is not shown. It should be noted, though, that it satisfies two important properties:

1. Only *free* occurrences of $x$ in $t$ are substituted.
2. A substitution will not capture variables, i.e. previously *free* occurrences never become *bound*.

In the event where a substitution would violate either one of these properties, a renaming of bound variables can be performed to avoid the problem.

There are three classes of reduction for the untyped $\lambda$-calculus described above. The first, $\alpha$-reduction, is used to rename the bound variable of a lambda expression:

$$\lambda x.\ t \underset{\alpha}{\Longrightarrow} \lambda y.\ [x \mapsto y]\, t$$

Term equality in the pure $\lambda$-calculus is intensional, such that two terms are equal if they have equivalent representations modulo renaming of bound variables. For example, the terms $\lambda x.\ x$ and $\lambda y.\ y$ are considered to be equal in value, but the terms $x$ and $y$ are not. The $\alpha$-reduction rule facilitates the conversion between alpha-equivalent terms and is critical to ensuring the correctness of the definition of substitution described above.

The second, and most important, class is $\beta$-reduction:

$$(\lambda x.\ t_1)\ t_2 \underset{\beta}{\Longrightarrow} [x \mapsto t_2]\, t_1$$

The above rule represents the primary method of computation in the $\lambda$-calculus, as it is the only mechanism that can reduce a combination of terms. The application of a lambda term to an argument, known as a $\beta$-redex[1], is simplified by substituting the operand for every occurrence of the bound variable in the body of the operator. For example, the expression $(\lambda x.\ x)\ y$ can be $\beta$-reduced to $y$, capturing the notion of the application of the identity function to a variable value.

The final class, $\eta$-reduction, can be used to simplify extraneously abstract lambda terms:

$$(\lambda x.\ t)\ x \underset{\eta}{\Longrightarrow} t$$

Starting with the term $\lambda x.\ (\lambda y.\ y)\ x$, we can see that $\eta$-reduction is essentially a sequential application of $\beta$-reduction, $\lambda x.\ (\lambda y.\ y)\ x \underset{\beta}{\Longrightarrow} \lambda x.\ x$ and $\alpha$-reduction, $\lambda x.\ x \underset{\alpha}{\Longrightarrow} \lambda y.\ y$. These two reductions can be safely combined into a single step in the instance where the outermost binding is used immediately in the innermost application. The proof of the extensionality property of functions[2] in the $\lambda$-calculus also relies on $\eta$-reduction.

Even though it is framed by only the three language constructs and three classes of reduction described above, the $\lambda$-calculus is an amazingly expressive language[3] . It has the ability to capture complicated aspects of a programming language quite clearly, though in most cases not succinctly. For example, the notion of looping is captured in the $\lambda$-calculus by the fixed-point, or $Y$, combinator: $\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$. Non-termination, e.g. infinite looping, can be implemented by applying the $Y$-combinator to the identity function, as shown in Figure 3.2.

[1]redex = reducible expression

[2]$(\forall x.\ f(x) = g(x)) \Rightarrow f = g$

[3]The $\lambda$-calculus is Turing complete, as explained by Turing's work cited above.

$$
\begin{array}{l}
(\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ (\lambda x.\ x) \underset{\beta}{\Longrightarrow} \\
(\lambda x.\ (\lambda x.\ x)\ (x\ x))\ (\lambda x.\ (\lambda x.\ x)\ (x\ x)) \underset{\beta}{\Longrightarrow} \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \underset{\beta}{\Longrightarrow} \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \underset{\beta}{\Longrightarrow} \\
\ldots
\end{array}
$$

Figure 3.2: Derivation of the $\Omega$ Combinator

In this example, repeated application of the $\beta$-reduction rule will always return the expression $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$; cleverly named, the $\Omega$ combinator as it is the "last" term in Church's $\lambda$-calculus. The derivation of the $\Omega$ combinator is a strong motivating example for the discussion of the evaluation semantics of the $\lambda$-calculus. Note that there are two $\beta$-redexes in the first step of the derivation shown in Figure 3.2, the overall expression and the body of the first lambda term, thus there are two possible reduction strategies. The derivation as presented follows a *normal* order, reducing the leftmost, outermost redex first. The alternative strategy, an *applicative* order, reduces the leftmost, innermost redex first. Note that following an applicative evaluation order in this case would lead to a divergence at the inner redex, preventing the derivation of the $\Omega$ combinator as encoded.

Unless otherwise noted, the examples in this section will follow the *call-by-value* semantics shown in Figure 3.3. The evaluation relation $e \longrightarrow e'$ indicates that an expression is reduced, or *steps*, to a simpler form; in this case through $\beta$-reduction. This relation is formally defined with an operational semantics where an expression's overall reduction is structured by the sub-term reductions shown above the dividing line. In these semantics an applicative evaluation order is enforced by reducing the function term of an application to a value before continuing with the reduction of the application's argument term. Note that the evaluation order is further restricted by forcing the argument of an application order to also be reduced to a value before $\beta$-reduction can occur, hence the name call-by-value.

$$
\begin{array}{lcl}
\text{term} & ::= & \text{variable} \\
 & | & \text{term term} \\
 & | & \lambda \text{ variable . term} \\
 & & \\
\text{value} & ::= & \lambda \text{ variable . term}
\end{array}
$$

$$Eval_{App1}\frac{term_1 \longrightarrow term_1'}{term_1\ term_2 \longrightarrow term_1'\ term_2}$$

$$Eval_{App2}\frac{term \longrightarrow term'}{value\ term \longrightarrow value\ term}$$

$$Eval_{Abs}\frac{}{(\lambda x.\ term)\ value \longrightarrow [x \mapsto value]\ term}$$

Figure 3.3: Evaluation Semantics of the λ-calculus

Reduction proceeds in this manner until the expression reaches an irreducible, or *normal*, form. The appeal of a call-by-value semantics is that the normal forms of a language are all elements of its value space. For the pure λ-calculus, this value space is defined as all arbitrary lambda terms. Languages that always terminate their evaluation at a term in normal form are said to be *strongly normalizing.* The existence of the Ω combinator acts as proof that the pure λ-calculus is not strongly normalizing; though all Turing complete languages are not strongly normalizing by definition.

Notably absent from the language shown in the previous subsection is any construct for the inclusion of constants. This makes it quite difficult to use the λ-calculus to model otherwise simple formalisms; for example, Peano arithmetic [72]. Peano's formulation is simplistic in that it requires just nine axioms to establish the canonical definition of natural numbers. These axioms can be used to inductively define basic arithmetic functions, such as addition of natural numbers. Even with such a simple formulation, though, it is a non-trivial exercise to capture the semantics of Peano addition in the λ-calculus.

Kleene's Church-Turing thesis demonstrates that any data type, and any computation over it, can be represented in the λ-calculus [52]. The Church numerals are one such encoding of natural numbers. Their basic premise is that a natural number, $n$, can be represented by $n$ applications of a function, $f$, to a base value, $x$; both of which must be bound variables. Thus, counting in Church numerals gives the term stream: $\lambda f.\ \lambda x.\ x$, $\lambda f.\ \lambda x.\ f\ x$, $\lambda f.\ \lambda x.\ f\ (f\ x)$, etc.

The addition of two numerals can be represented as the composition of their function applications:

$$add(m, n) = m + n \Longrightarrow \lambda f.\ \lambda x.\ f^m\ (f^n\ x)$$

Given that a Church numeral is encoded as a series of function applications, the expression $f^n$ can be represented in the λ-calculus by passing $f$ as the function argument to a numeral $n$, i.e. $f^n\ x \Longrightarrow \lambda n.\ \lambda f.\ \lambda x.\ n\ f\ x$. The resultant encoding for the addition function, $m + n$, is shown below:

$$\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

Using this definition, we can derive other functions. The derivation for the successor function, *succ*, as well as a simple test, are shown in Figures 3.4 and 3.5 accordingly. Note that this derivation matches the intuitive definition of *succ*, $f(f^n x)$, and the test for $succ(0)$ produces the correct Church numeral for the value 1.

Figure 3.4: Derivation of the *succ* Function

$succ(n) = add(1, n)$ :
$(\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x))\ (\lambda f.\ \lambda x.\ f\ x) \underset{\beta}{\Longrightarrow}$

$\lambda n.\ \lambda f.\ \lambda x.\ (\lambda f.\ \lambda x.\ f\ x)\ f\ (n\ f\ x) \underset{\beta}{\Longrightarrow}$

$\lambda n.\ \lambda f.\ \lambda x.\ (\lambda x.\ f\ x)\ f\ (n\ f\ x) \underset{\beta}{\Longrightarrow}$

$\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$

$succ(0) = 1:$

$(\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x))\ (\lambda f.\ \lambda x.\ x) \underset{\beta}{\Longrightarrow}$

$\lambda f.\ \lambda x.\ f\ ((\lambda f.\ \lambda x.\ x)\ f\ x) \underset{\beta}{\Longrightarrow}$

$\lambda f.\ \lambda x.\ f\ (\lambda x.\ x)\ x \underset{\beta}{\Longrightarrow}$

$\lambda f.\ \lambda x.\ f\ x$

Figure 3.5: Testing the *succ* Function

A similar encoding can be implemented for boolean constants. Given their finite value space, Church booleans are implemented as a set of higher-order functions that select and return one of their parameters, each representing a single possibility from the enumerated value space:

- $true$ is encoded as the function that chooses the first parameter - $\lambda t.\ \lambda f.\ t$
- $false$ is encoded as the function that chooses the second parameter - $\lambda t.\ \lambda f.\ f$

Primitive boolean operators can be similarly encoded as functions that implement the appropriate truth tables. For reference, the truth tables for $NOT(P) = R$ and $AND(P, Q) = R$ are shown in Figure 3.6.

Figure 3.6: Boolean Truth Tables

Truth Table for $NOT$

| P | R |
|---|---|
| $true$ | $false$ |
| $false$ | $true$ |

Truth Table for $AND$

| P | Q | R |
|---|---|---|
| $true$ | $true$ | $true$ |
| $true$ | $false$ | $false$ |
| $false$ | $true$ | $false$ |
| $false$ | $false$ | $false$ |

Following the truth table, the $NOT$ operator is encoded as a function that returns the inverse of its Church boolean parameter:

$$NOT(P) \Longrightarrow \lambda p.\ p\ F\ T$$

Operators with more than one parameter can be encoded by scanning the truth table from left to right, branching at each parameter, providing saturated applications of the remaining parameters for the true and false arguments. For example, the encoding of the $AND$ operator consists of saturated applications of $Q$ for each branch of $P$:

$$AND(P, Q) \Longrightarrow \lambda p.\ \lambda q.\ p\ (q\ T\ F)\ (q\ F\ F)$$

A benefit of encoding constant values with selector functions in this manner is that a number of rewrite rules follow as corollaries. In cases where a selector is applied to the same value twice, the expression can be reduced to just that value:

$$sel\ x\ x \underset{ID_1}{\Rightarrow} x$$

In cases where an argument to a selector is equal to its respective parameter value, the expression can be rewritten to replace said argument with the selector itself:

$$sel\ T\ x \underset{ID_{2_a}}{\Rightarrow} sel\ sel\ x$$
$$sel\ x\ F \underset{ID_{2_b}}{\Rightarrow} sel\ x\ sel$$

Following from a composition of the previous laws, in the case where a selector function is applied to values in an order that matches their parameterized representation, the expression can be safely reduced to that selector:

$$sel\ T\ F \underset{ID_3}{\Rightarrow} sel$$

$$\begin{aligned}
&\lambda p.\ \lambda q.\ p\ (q\ T\ F)\ (q\ F\ F) \underset{ID_1}{\Rightarrow}\\
&\lambda p.\ \lambda q.\ p\ (q\ T\ F)\ F \underset{ID_3}{\Rightarrow}\\
&\lambda p.\ \lambda q.\ p\ q\ F \underset{ID_{2_b}}{\Rightarrow}\\
&\lambda p.\ \lambda q.\ p\ q\ p
\end{aligned}$$

Figure 3.7: Derivation of Alternative $\lambda$-calculus Encodings for $AND$

Applications of these rules allow us to reduce the previous encoding for $AND$ to a simpler form, as shown in Figure 3.7.

**The Simply Typed $\lambda$-Calculus**

The consequence of implementing constants with Church encodings is that constructing a language with multiple classes of constants presents a certain level of danger. Take, for example, an application of the previously derived *succ* function to a Church boolean, as shown in Figure 3.8.

In this example, we can proceed with $\beta$-reduction because the application forms a valid $\beta$-redex, however, the resultant value is meaningless; it takes the structural shape of neither a Church numeral nor boolean. This should be of no surprise, though, given that we are attempting to apply *succ* to $true$, a value not covered by Peano's formulation. The solution to this problem is to separate the value spaces of numerals and booleans, such that ill-formed expressions can be identified before reduction is attempted or computed.

Figure 3.8: An Unsafe Mixing of Church Numerals and Booleans

$$\begin{aligned}
&(\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x))\ (\lambda p.\ \lambda q.\ p) \underset{\beta}{\Longrightarrow}\\
&\lambda f.\ \lambda x.\ f\ ((\lambda p.\ \lambda q.\ p)\ f\ x)) \underset{\beta}{\Longrightarrow}\\
&\lambda f.\ \lambda x.\ f\ ((\lambda q.\ f)\ x)) \underset{\beta}{\Longrightarrow}\\
&\lambda f.\ \lambda x.\ f\ f
\end{aligned}$$

| | | |
|---|---|---|
| variable | ::= | a \| b \| ... |
| value | ::= | $\lambda$ variable : type . term |
| | \| | $c$ *where* $c$ is a constant value |
| term | ::= | variable |
| | \| | term term |
| | \| | $\lambda$ variable : type . term |
| | \| | $c$ *where* $c$ is a constant term |
| type | ::= | type $\rightarrow$ type |
| | \| | $T$ *where* $T$ is a base type |

Figure 3.9: BNF Grammar for the Simply Typed $\lambda$-calculus

Church's typed formulation of the $\lambda$-calculus implements this solution by adding an additional level of abstraction to the language to express the *types* of *terms* [14]. The grammar for this language is shown in Figure 3.9. In the simply typed $\lambda$-calculus, the type level has only a single constructor, $\rightarrow$, used to build function types. For example, the type $Nat \rightarrow Bool$ is the type of a function whose domain is natural numbers and range is booleans. Note that there are no variable types in this language, thus there is no notion of polymorphism or type substitution. Additionally, the only place a type can be, and must be, introduced in a term is as the type of the bound variable of a lambda term.

The evaluation of the simply typed $\lambda$-calculus is guarded by only reducing *well-typed* expressions. We say an expression, $e$, is well-typed when, under a typing context, $\Gamma$, we can compute a type, $T$, for it. We denote this typing relation with the syntax $\Gamma \vdash e : T$. The typing relation itself is defined by the set of rules shown in Figure 3.10. These inference rules follow the standard form $\frac{A_1 ... A_n}{C}$, stating that the conclusion $C$ holds under the premise that the assumptions $A_1 ... A_n$ are all true.

$$Type_{Con}\frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T} \qquad Type_{Var}\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$Type_{Lam}\frac{\Gamma, (x : T_1) \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.\ e) : (T_1 \to T_2)} \qquad Type_{App}\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T_2}$$

Figure 3.10: Typing Rules for the Simply Typed λ-calculus

The first and simplest rule, $Type_{Con}$, states that a constant is typed by the set of values it belongs to, e.g. $\vdash$ *true* : *Bool*. The second rule, $Type_{Var}$, appears equally simple, though there is a slight wrinkle for cases with an empty typing context. A variable, $x$, is of type $T$ when the judgement $x : T$ is a member of the typing context. Note that the typing context is only extended when a variable is bound by a lambda term, as shown in rule $Type_{Lam}$. Thus, free variables are always ill-typed under an empty typing context. Expressions that are well-typed under the empty context, i.e. they contain no free variables, are referred to as *closed*.

The final two rules introduce and eliminate function types. As noted above, typing a lambda term extends the context with the type ascription contained in its binding. The $Type_{Lam}$ rule constructs a function type for a lambda term where the domain is drawn from this ascription and the range is the type of its body under the extended context. The final rule, $Type_{App}$, acts as the type-checking equivalent of *modus ponens*. The type of an application, $e_1\ e_2$, is a reduction to $e_1$'s range type when its domain type matches the type of $e_2$.

The simply typed λ-calculus is considered a "safe" language because it is strongly normalizing for well-typed terms. Given that type-checking this language is a static computation, it provides a "compile-time" guarantee that the evaluation of an expression will terminate with a value that is correct according to the semantics of the language. Hence why it was previously noted that typing typically precedes evaluation for most λ-calculus based systems.

$$
\begin{array}{lcl}
\text{nat} & ::= & 0 \mid \mathit{succ}\ \text{nat} \\
\text{value} & ::= & \mathit{true} \mid \mathit{false} \mid \text{nat} \mid \ldots \\
\text{term} & ::= & \mathit{true} \mid \mathit{false} \mid 0 \mid \mathit{succ}\ \text{term} \mid \ldots \\
\text{type} & ::= & \mathit{Bool} \mid \mathit{Nat} \mid \ldots
\end{array}
$$

$$
Type_{True}\frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool}} \qquad Type_{False}\frac{}{\Gamma \vdash \mathit{false} : \mathit{Bool}}
$$

$$
Type_{Zero}\frac{}{\Gamma \vdash 0 : Nat} \qquad Type_{Succ}\frac{\Gamma \vdash n : Nat}{\Gamma \vdash \mathit{succ}\ n : Nat}
$$

$$
Eval_{Succ}\frac{e \longrightarrow e'}{\mathit{succ}\ e \rightarrow \mathit{succ}\ e'}
$$

Figure 3.11: Extended Semantics for Naturals and Booleans

Stated more formally, a language is considered safe when it satisfies the following properties:

- *Progress* - A well-typed term is either a value or it can be reduced by one of the evaluation rules.
- *Preservation* - If a well-typed term is reduced by one of the evaluation rules then the resultant term or value remains well typed.

We can step through a semi-formal proof of these properties for the extension of the simply typed $\lambda$-calculus that accommodates booleans and naturals shown in Figure 3.11.

The proof of progress proceeds by rule induction over the possible derivations for the typing judgements of well-typed terms. The cases for $Type_{True}$, $Type_{False}$, and $Type_{Zero}$ are vacuously true given that the terms in their judgments are all values. Per the inductive hypothesis, the assumption judgement of the $Type_{Succ}$ case, $\Gamma \vdash n : Nat$, is a witness for a well-typed term that itself is either a value or progresses. In the sub-case where $n$ is a value, it must be a $Nat$ value, thus the grammar of the language dictates that $\mathit{succ}\ n$ is a value as well. In the sub-case where $n \longrightarrow n'$, then $\mathit{succ}\ n \longrightarrow \mathit{succ}\ n'$ by the $Eval_{Succ}$ evaluation rule.

The proof of preservation proceeds similarly, again by rule induction over the possible derivations for the typing judgements of well-typed terms. The cases for $Type_{True}$, $Type_{False}$, and $Type_{Zero}$ are again vacuously true, this time by contradiction given that values can not be reduced. For the $Type_{Succ}$ case there is only one possible reduction we can perform, $succ\ term \longrightarrow succ\ term'$ by the $Eval_{Succ}$ evaluation rule. By the induction hypothesis, the sub-term reduction $term \longrightarrow term'$ must be type preserving, thus $term' : Nat$ and $succ\,term' : Nat$ per $Type_{Succ}$, preserving the type of $succ\,term$.

With the language formally defined and its safety informally proven, we can show that the previously discussed computation $succ(true)$ is ill typed and will result in a stuck computation:

$$\dfrac{\dfrac{\dfrac{\Gamma, (n : Nat) \vdash n : Nat}{\Gamma, (n : Nat) \vdash (succ\,n) : Nat}}{\Gamma \vdash (\lambda n : Nat.\ succ\,n) : Nat \rightarrow Nat} \qquad true : Bool}{\Gamma \vdash ((\lambda n : Nat.\ succ\,n)\ true) : < error >}$$

**Polymorphic $\lambda$-Calculi (System F and Friends)**

The simply typed $\lambda$-calculus makes for a nice introduction to type systems, but being limited to a fixed set of monomorphic base types makes it an impractical language for actual use. The lack of polymorphism in the language almost completely eliminates any notion of the reusability that computer scientists value so much. Additionally, the monomorphism of lambda bindings precludes the previously discussed Church encoding technique, meaning that new types can only be introduced by extending a language's semantics and implementation both. Using the $\lambda_{NB}$ language defined in the previous subsection as the basis of discussion, note that there are several possible operations that should intuitively work over both of the primitive types; the identity operation, $\lambda x.\ x$, is just one such example.

The lack of polymorphism in this language means that you must write a new identity function for each type, e.g. $\lambda x : Nat.\ x$ and $\lambda x : Bool.\ x$. The style of implicit typing introduced by Haskell Curry et. al in their combinatory logic systems can be adapted for the $\lambda$-calculus to make these terms intrinsically equal [21, 22]. However, the consequence of removing type ascriptions and inferring the type of lambda terms from their arguments is that it makes typing an undecidable procedure in general.

Continuing with $\lambda_{NB}$, note that there are no primitive operations over booleans defined by the language. Additionally, the language has no mechanisms for scrutinizing data types or branching based on their construction, so the only method we have to introduce new operations is through semantic extension. We could attempt to redefine booleans through a Church encoding as we did in a previous subsection, but the type system of the simply typed $\lambda$-calculus will shut us down quickly.

Assuming our working language has an arbitrary type value, $A$, among its base types, we can derive the following typing judgement for the Church encoding of $true$:

$$(\Gamma \vdash \lambda t : A.\ \lambda f : A.\ t) : (A \rightarrow A \rightarrow A)$$

Given that the types in the following example will grow quite large, we abbreviate the type $A \rightarrow A \rightarrow A$ as $Bool$ and $Bool \rightarrow Bool \rightarrow Bool$ as $Sel$. With these abbreviations in mind, we show the failed derivation of a typing judgement for the Church encoding of $NOT(true)$ below:

$$\dfrac{\dfrac{\dfrac{\Gamma, (p : Sel) \vdash p : Sel \qquad \Gamma' \vdash false : Bool}{\Gamma, (p : Sel) \vdash (p\ false) : (Bool \rightarrow Bool)} \qquad \Gamma' \vdash true : Bool}{\Gamma \vdash (\lambda p : Sel.\ p\ false\ true) : (Sel \rightarrow Bool)} \qquad \Gamma \vdash true : Bool}{\Gamma \vdash ((\lambda p : Sel.\ p\ false\ true)\ true) : < error >}$$

$$\begin{array}{lll} \text{type\_variable} & ::= & \text{A} \mid \text{B} \mid ... \\ \text{value} & ::= & \Lambda \text{ type\_variable . term} \mid ... \\ \text{term} & ::= & \Lambda \text{ type\_variable . term} \mid \text{term [ type ]} \mid ... \\ \text{type} & ::= & \text{type\_variable} \mid \forall \text{ type\_variable . type} \mid ... \end{array}$$

$$Type_{APP_{TY}} \frac{\Gamma \vdash term : \forall X.\ type_2}{\Gamma \vdash term\ [type_1] : [X \mapsto type_1]\, type_2}$$

$$Type_{ABS_{TY}} \frac{\Gamma, X \vdash term : type}{\Gamma \vdash (\Lambda X.\ term) : \forall X.\ type}$$

$$Eval_{APP_{TY}} \frac{term \longrightarrow term'}{term\ [type] \longrightarrow term'\ [type]}$$

$$Eval_{ABS_{TY}} \frac{}{(\Lambda X.\ term)\ [type] \longrightarrow [X \mapsto type]\, term}$$

Figure 3.12: Extended Semantics for System F

Functionally, $NOT$ operates as a binary selector, much like $true$ and $false$ do. It makes sense, therefore, that all of their types share the shape $* \to * \to *$. We can see in the example above, though, that in spite of the similar shape to the type, there is no way to derive a correct judgement for an application of $NOT$. Preemptively modifying the type of $true$ to match the expected type of the argument to $NOT$, i.e. changing it from $Bool$ to $Sel$, just leads to another tripling of $NOT$'s domain type to $Sel \to Sel \to Sel$. What we need is a way to ascribe a more general type to $true$, such that it can be unified with any witness to the type shape $* \to * \to *$.

Jean-Yves Girard [27] and John Reynolds [75] independently discovered the solution to this problem, a polymorphic $\lambda$-calculus. The resultant system, extends the simply typed $\lambda$-calculus with the notion of universally quantified types. The extended semantics of this *second-order* $\lambda$-calculus, commonly referred to as System F, are shown in Figure 3.12.

The key construct of System F, $\Lambda X.\ t$, allows for the binding of a type variable at the term level, such that it is in scope for later ascriptions. Demonstrating this new abstraction, the polymorphic identity function is expressed as $\Lambda A.\ \lambda x : A.\ x$. Following from the $Type_{ABS_{TY}}$ typing rule, the type of this function is $\forall A.\ A \rightarrow A$, where the $\forall$ binder is the universal type quantifier. System F extends the $\beta$-reduction relation with the $Eval_{APP_{TY}}$ and $Eval_{ABS_{TY}}$ rules to define how to reduce type applications. Continuing with the above example, a monomorphic instance of the identity function can be produced by applying it to the desired type, e.g.:

$$(\Lambda A.\ \lambda x : A.\ x)\ [Nat] \underset{\beta}{\Longrightarrow} \lambda x : Nat.\ x$$

To reiterate the problem that motivated System F's introduction in this chapter, the simply-typed $\lambda$-calculus is incapable of representing Church booleans because its type system is too restrictive. Recall that we needed a way to express an all-inclusive type of shape $* \rightarrow * \rightarrow *$, but could not do so given the lack of type polymorphism. System F, however, can express this type as $\forall A.\ A \rightarrow A \rightarrow A$. Figure 3.13 shows the resulting formulation of Church booleans and a successful type derivation for the previously failing $NOT(true)$ example.

The key to the success of this derivation is the reduction of the type application. Because the $Bool$ type is being used to instantiate itself, we observe an expansion at the type level; the tripling effect that was noted earlier:

$$(p : Bool)\ [Bool] \underset{\beta}{\Longrightarrow} p : Bool \rightarrow Bool \rightarrow Bool$$

Figure 3.13: A Polymorphic Implementation of Church Booleans

- $Bool$ - $\forall A.\ A \rightarrow A \rightarrow A$
- $true$ - $\Lambda A.\ \lambda t : A.\ \lambda f : A.\ t$
- $false$ - $\Lambda A.\ \lambda t : A.\ \lambda f : A.\ f$
- $NOT$ - $\lambda p : Bool.\ p\ [Bool]\ false\ true$

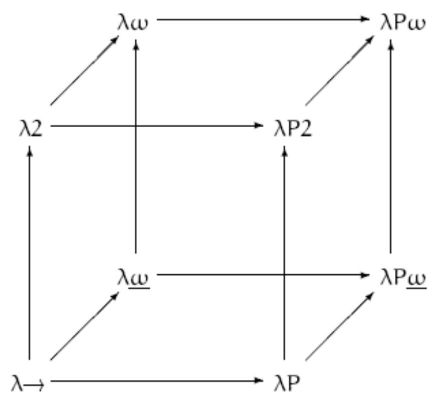
$$\dfrac{\dfrac{\dfrac{\Gamma, (p : Bool) \vdash p : Bool \qquad \Gamma' \vdash false : Bool}{\Gamma, (p : Bool) \vdash (p\ false) : (Bool \rightarrow Bool)} \qquad \Gamma' \vdash true : Bool}{\Gamma \vdash (\lambda p : Bool.\ p\ [Bool]\ false\ true) : (Bool \rightarrow Bool)} \qquad \Gamma \vdash true : Bool}{\Gamma \vdash ((\lambda p : Bool.\ p\ [Bool]\ false\ true)\ true) : Bool}$$

This is permissible in System F because the domain of the $\forall$ operator covers all possible types, such that the type variables it binds can be instantiated with other universally quantified types. While this allows for an incredibly expressive type system, it makes type checking in System F impredicative and undecidable in general [11].

Impredicativity can be avoided by enforcing a hierarchy of types and using it to restrict the instantiation of universally quantified type variables. For example, one possible hierarchy separates types into two sets: *small* types that contain no quantifiers, and types with unrestricted quantification. By permitting only types that satisfy the smallness property to instantiate quantifiers, the type system can be made predicative [73, 90]. Similar restrictions are found in the Hindley-Milner inspired type systems employed by most modern functional languages [42, 66]

System F's polymorphic language features can be more generally explained as the extension of the simply-typed $\lambda$-calculus, from here on referred to as $\lambda_{\rightarrow}$, with constructs to support *terms that depend on types*. The $\lambda_{\rightarrow}$ language can be similarly extended with constructs for *types that depend on types* and *types that depend on terms*. Henk Barendgt modeled these extensions, and their possible combinations, as corners of his Lambda Cube [9], as shown in Figure 3.14[4].

Figure 3.14: Barendgt's Lambda Cube



[4]Image courtesy of `http://commons.wikimedia.org/wiki/File:Lambda_cube.png`

The origin point of this cube is $\lambda_{\to}$, a language that, in Barendgt's terms, provides an abstraction for *terms that depend on terms.* As was stated in the preceding paragraph, System F, labeled as $\lambda_2$ on the cube to indicate its second-order nature, is the extension of $\lambda_{\to}$ along the axis of polymorphism; this is Barendgt's first axis. Notable features of this axis include universal type quantification, term-level type abstraction, and term-level type application.

Barendgt's second axis is the axis of type functions. This axis can be explained as a direct promotion of the untyped $\lambda$-calculus to the type level of a language in order to provide abstractions for *types that depend on types.* The most commonly used type functions are type operators, or type constructors – parametric type constants that can construct new types from provided arguments. The $\lambda_{\to}$ language already includes one such operator, $\to$, which constructs a function type when given arguments for its domain and range type. Other type operators commonly seen in languages provide for the construction of list types, pair types, and sum types.

Extending the $\lambda_{\to}$ language along this axis produces Barendgt's $\lambda_{\underline{\omega}}$, a language that is not suited for practical use for reasons similar to those of $\lambda_{\to}$. However, when paired with Barendgt's first axis, polymorphism, it produces System $F_{\omega}$, the basis for many functional languages [74]. System $F_{\omega}$, labeled as $\lambda_{\omega}$ by Barendgt, also defines the notion of *kinds* – the types of types.

In their simplest form, kinds are used to indicate the arity of a type operator. I have already borrowed the common notation for kinds when discussing the "shape" of a type, e.g. $* \to * \to *$. Nullary type operators and fully saturated type applications have the kind $*$ indicating that they can be used anywhere a type is expected. A type operator that requires an argument has a functional kind, denoted by recycling the $\to$ operator from the type level. For example, the list type operator, $[\,]$, has the kind $* \to *$.

More complicated type systems may introduce classes of kinds, much like extensions of $\lambda_{\rightarrow}$ introduce classes of base types. The most popular example of such a language is System $\mathrm{F}_{\underset{C}{\uparrow}}$ [95], an extension of System $\mathrm{F}_C$ [86] that is itself an extension of System $\mathrm{F}_\omega$. System $\mathrm{F}_{\underset{C}{\uparrow}}$ is notable because it serves as the foundation of Haskell's intermediate language. In this language, all data type definitions are automatically promoted to the kind level, such that you can write kind ascriptions identical to the type ascriptions available in System F. These kind ascriptions guard advanced, *kind-safe* computation at the type level, allowing System $\mathrm{F}_{\underset{C}{\uparrow}}$ to simulate some capabilities of dependent typing.

Incidentally, dependent typing is Barendgt's third and final axis. When extending $\lambda_{\rightarrow}$ along this axis the $\lambda\Pi$-calculus is produced, the meta-language of the LF logical framework [39]. This calculus allows bindings of the form $\Pi$*term. type* at the type level, providing an abstraction for *types that depend on terms.* The most common examples of dependent typing tend to deal with encoding correctness properties at the type level, e.g.:

```
append :: Vector a m -> Vector a n -> Vector a (n + m)
```

When all three axes are followed you arrive at the point opposite $\lambda_{\rightarrow}$ on the Lambda Cube, $\lambda\Pi_\omega$. This language corresponds to Thierry Coquand's Calculus of Constructions type theory [19] which is the foundational basis of the Coq theorem prover [87]. Coq will be discussed as related work, however, its complexity and differing foundational logic greatly distances it from my HaskHOL system. Understanding the basic notions of System $\mathrm{F}_\omega$ should be sufficient for following the presentation of HaskHOL and its application, i.e. the work of this dissertation.

## 3.2 Higher-Order Logic

Higher-order logic (HOL) is an extension of the first-order predicate calculus that allows quantification to cover all objects of a given type, not just simple values [52]. Per the Curry-Howard Isomorphism, it is the logical correspondent to the the simply-typed lambda calculus discussed in Chapter 3 [83]. HOL has served as the logical foundation for a number of proof systems; most obviously the HOL family of theorem provers that HaskHOL is a member of. This section presents a high-level overview of HOL, as originally formalized by Mike Gordon's HOL proof system [29, 32] and its direct descendants.

Following from higher-order logic's extension of the predicate calculus, the HOL proof system was born as an extension of Robin Milner's Logic for Computable Functions (LCF) [34]. HOL extends LCF with the capability to reason about higher-order predicates, guarding their construction and application with a simple, polymorphic type theory. In addition to inheriting its base predicate logic, HOL follows LCF's implementation technique of focusing on the development of a small, trusted, logical kernel from which more advanced reasoning features are bootstrapped from.

This technique, known colloquially as the "LCF style," is employed by a number of proof systems, both inside and out of the HOL family. The appeal of the LCF style is that a system's implementation can be reduced to a composition of a small set of primitive operations that collectively form its trusted code base. If the soundness and completeness of the kernel defined by these primitives can be guaranteed, that guarantee can easily be extended to cover the entirety of the system. This section's presentation follows the bootstrapping nature of the LCF style, working from the primitive, abstract types of a HOL system up to its advanced proof capabilities.

## Types and Terms

As was mentioned in the introduction to this section, the term language of a HOL system corresponds to the simply-typed $\lambda$-calculus that was shown in Figure 3.9. To aid comparison, the grammar for a HOL language is shown in Figure 3.15. Note that this language differs in two major ways:

- Type ascriptions are present at every variable and constant occurrence, not just at variable bindings.
- Function types have been replaced with a more general type application construct.

Both of these changes are motivated by the primary application of this language – representation, rather than computation.

Recall from the previous section that the purpose of types in typed extensions of the $\lambda$-calculus is to guarantee the safe evaluation of expressions. For example, the evaluation of the expression $x$ will fail as there is no way to reduce it to a value of the language, where values are defined as all lambda expressions and constants. We can attest to this failure statically because free variables are ill-typed terms. In other words, free variables are meaningless in the $\lambda$-calculus.

Figure 3.15: BNF Grammar for HOL's Term Language

```
typed_var ::= identifier : type

term      ::= typed_var
            |  term term
            |  λ typed_var . term
            |  c : type where c is a constant term

type      ::= identifier
            |  c [type] where c is a constant type
```

In HOL, however, free variables are used to represent the *unknowns* of a logical term. Most commonly, these unknowns act as placeholders for future instantiations. For example, the proposition $P \vee \neg P$ is a classically true term for every substitution of the free variable $P$. It is critical to the soundness of any logic that the validity of its terms is preserved through these substitutions, otherwise contradictions can easily be derived. Corresponding to the $\lambda$-calculi, types can be used to enforce this preservation property.

In his previously cited formulation of HOL, Gordon demonstrates the danger of an untyped higher-order logic with the predicate $Px = \neg(xx)$. Performing the substitution $[x \mapsto P](Px = \neg(xx))$ results in the term $PP = \neg(PP)$, an obvious contradiction. In a typed logic, the original predicate $Px = \neg(xx)$ could not have been constructed, as there is no satisfying assignment for the type of $x$. This is evident from just the right-hand side of the equation, where $x$ simultaneously has the general types of $A \rightarrow \mathit{Bool}$ and $A$. There exists no finite type that satisfies their unification, thus the entire proposition is ill typed.

For all of the examples seen up to this point, the types of free variables could be inferred from their use. This is not true in general, though, hence why all variables and constants in HOL must be paired with a type ascription. With these ascriptions in place, HOL's typing rules, shown in Figure 3.16[5], are comparatively simpler than those of any of the previously seen $\lambda$-calculi, as a typing context is no longer required.

Figure 3.16: Typing Rules for HOL

$$\mathit{Type}_{As}\frac{x \text{ is a constant or variable}}{\vdash (x : T) : T}$$

$$\mathit{Type}_{Lam}\frac{\vdash e : T_2}{\vdash (\lambda x : T_1.\ e) : (T_1 \rightarrow T_2)}$$

$$\mathit{Type}_{App}\frac{\vdash e_1 : T_1 \rightarrow T_2 \qquad \vdash e_2 : T_1}{\vdash (e_1\ e_2) : T_2}$$

[5]Note that I have combined the typing rules for constants and variables into a single rule, $\mathit{Type}_{As}$, for a more concise display.

The absence of a typing context motivates an interesting point regarding term equality in HOL. For all of the versions of the $\lambda$-calculus that were discussed in Chapter 3, the equality of variables depended on only two factors: their names and their scope. Under HOL's typing rules, the term $\lambda x : Nat.\ (x : Bool)$ has the type $Nat \to Bool$, such that $x$ simultaneously has the types $Nat$ and $Bool$, which again can not be unified. This would seem to indicate that $\lambda x : Nat.\ (x : Bool)$ is an ill-typed term, however, it is, in fact, well typed because the two occurrences of $x$ are different variables. This is due to the equality of variables in HOL depending not just on their names and scope, but on the equality of their types as well.

Given this, to clarify an earlier statement, the predicate $Px = \neg(xx)$ can be well typed in HOL provided that, at minimum, the two occurrences of $x$ on the right-hand side of the equation are different variables. This can be achieved through manual construction, but is typically impossible for a parser to do. Recall from the discussion of implicitly typed lambda terms in the previous section that type inference for such terms is undecidable in general. Most parsers, including those of most HOL systems, solve this problem by making the simplifying assumption that all occurrences of a name in a given scope refer to the same variable. Any of these occurrences that lack a type ascription are assigned a fresh type variable to make typing explicit and decidable. For example, the input `\ x. x` would be parsed as the term $\lambda x : A.\ (x : A)$, but the input `\ x:Nat. (x:Bool)` would be rejected.

As an aside, most pretty-printers for proof systems do not display the types of variables in order to maintain a clean and concise presentation of terms. For most HOL systems, this means that they can print the predicate $Px = \neg(xx)$, but not parse it. Freek Wiedijk has dubbed this problem Pollack-inconsistency [94]. I will attempt to avoid related issues in the remainder of this section by introducing additional type ascriptions to pretty-printed terms where necessary.

Compared to the simply-typed $\lambda$-calculus, the other major difference of this HOL language is its decision to represent constant types with type functions. This approach is similar to that of the $\lambda_{\underline{\omega}}$ language, however, no kind system is introduced, variable type operators are not permitted, and partial applications of types are not allowed. As an example, the various HOL implementations all have different sets of base constants for their logics, but at minimum each must have a constant for term equality. This necessitates the existence of constant types for functions and booleans, i.e. $= : a \rightarrow a \rightarrow bool$. Assuming that strings are an adequate representation of identifiers, this constant can be expressed abstractly as `"=" : ("->", ["a", ("->", [ "a", ("bool", [])])])`, where type applications are simple pairings of an operator and its list of argument types.

Additional constants, both terms and types, are introduced in HOL through definitional extension which will be covered in a following subsection. The important point to be made, though, is that their primitive representations follow from that of equality's, i.e. they will use the same general forms of $(identifier : type)$ and $identifier\ [type]$. Using this general, abstract syntax greatly simplifies the implementation of the logical kernel, as regardless of a constant's definitional complexity, it can be constructed in only one way. For most primitive operations this means that all constants are handled with a single conditional or case scrutinization branch, as opposed to numerous branches that all do the same thing.

### Theorems and Rules

The principle data type of a HOL system is the theorem type, represented as a sequent of boolean terms: $a_1, ..., a_n \vdash c$. In this notation, the term $c$ represents the conclusion of a theorem which is held true under the conjunctive assumption that terms $a_1, ..., a_n$ are also true; hence their name – the assumption list. For example, the theorem $x \wedge y \vdash x$ concludes that x is true under the assumption that its conjunction with another variable, $y$, is true. A theorem with an empty assumption list, such as $\vdash P \vee \neg P$, is a tautology.

Following the restrictions of the LCF style, theorems can not be manually constructed. They can only be introduced by the logical kernel of a system in one of three ways:

- As an axiom.
- Through a primitive rule of inference.
- Or as a consequence of definitional extension.

Axiomatic introduction of a theorem is the acceptance of a term as true without any logical support to make the claim. There are a number of reasons you may wish to introduce an axiom; the most logically sound of these reasons is to assert a theorem you know to be true but *cannot* prove in a system's logic. For example, some, but not all, HOL systems include a primitive inference rule for $\eta$-reduction. In the absence of that rule, a system must axiomatize the universal eta property of lambda terms in order to implement $\eta$-reduction as a derived rule:

$$newAxiom \quad (!\, t : A \rightarrow B.\ (\lambda x.\ t\ x) = t)$$

Similarly, axioms can also be used to assert a theorem that *can* be proved in a system's logic for cases where you wish to delay doing the work for one reason or another. You must be careful when doing so, however, as there are no checks of correctness for axiomatic terms beyond ensuring that they are of a boolean type. Even if a single axiom is not a contradiction on its own, it may raise an inconsistency in the presence of other theorems or axioms. Thus, unrestricted axiomatic introduction is an easy way to jeopardize the soundness of the entire system, intentionally or unintentionally.

The safe alternative to axioms is the introduction of theorems through inference. Collectively, a system's primitive rules of inference frame its foundational logic. It is important to note that there is no one definitive set of rules for a HOL prover; systems are free to include any rule as primitive, rather than derived, and many do for reasons of efficiency. More commonly, though, the sets of primitive rules differ because there are any number of places to "draw a line in the sand" between the logical kernel and the rest of the system.

$$\text{DISCH}\frac{A \vdash q}{A - p \vdash p \Rightarrow q}$$

$$\text{DEDUCT\_ANTISYM\_RULE}\frac{A \vdash p \qquad B \vdash q}{(A - q) \cup (B - p) \vdash p = q}$$

Figure 3.17: Primitive Rules for Assumption Elimination

As an example, the $DISCH$ rule acts as the primitive assumption elimination rule for some systems. However, if a logical kernel does not include the propositional connectives, or at the very least implication, this rule cannot be used. Instead, the only way to remove an assumption is through deductive antisymmetry. These two approaches to assumption elimination are shown in Figure 3.17.

These rules are presented in sequent calculus form, a notation credited to Gerhard Gentzen [24]. The theorems above the horizontal line of a rule are referred to as the *hypotheses*, and the theorem below is referred to as the *result.* Much like the sequent representation of theorems themselves, these rules state that a result can be deduced from the truth of its hypotheses. Again, there is a logical correspondence between this notation and the inference rules for the $\lambda$-calculus that we have already seen.

Though HOL systems vary in their selection of the rules included in their logical kernels, minimally they require methods for:

- Reflexive Equality of Terms
- Introduction and Elimination of Assumptions
- Term Substitution
- Type Instantiation
- Beta Reduction
- Congruence of Abstractions and Combinations
- $Modus\ Ponens$

Note that biconditionality logically subsumes implication, such that the *modus ponens* rule can be written to reduce equivalences of boolean terms if a kernel lacks the inclusion of implication.

$$\text{REFL}\frac{}{\vdash t = t} \qquad \text{ASSUME}\frac{}{t \vdash t}$$

$$\text{DEDUCT\_ANTISYM\_RULE}\frac{A \vdash p \quad B \vdash q}{(A - q) \cup (B - p) \vdash p = q}$$

$$\text{INST}\frac{A \vdash t}{A\,[t_1, ..., t_n/x_1, ..., x_n] \vdash t\,[t_1, ..., t_n/x_1, ..., x_n]}$$

$$\text{INST\_TYPE}\frac{A \vdash t}{A\,[ty_1, ..., ty_n/tv_1, ..., tv_n] \vdash t\,[ty_1, ..., ty_n/tv_1, ..., tv_n]}$$

$$\text{BETA}\frac{}{\vdash (\lambda x.\ t)\ x = t\,[x]}$$

$$\text{ABS}\frac{A \vdash t1 = t2 \quad x\ not\ free\ in\ A}{A \vdash (\lambda x.\ t1) = (\lambda x.\ t2)} \qquad \text{MK\_COMB}\frac{A1 \vdash f = g \quad A2 \vdash x = y}{A1 \cup A2 \vdash f\ x = g\ y}$$

$$\text{EQ\_MP}\frac{A1 \vdash t1 = t2 \quad A2 \vdash t1}{A1 \cup A2 \vdash t2} \qquad \text{TRANS}\frac{A1 \vdash t1 = t2 \quad A2 \vdash t2 = t3}{A1 \cup A2 \vdash t1 = t3}$$

Figure 3.18: Primitive Inference Rules of HOL Light

John Harrison's HOL Light system was developed with a minimalistic and lightweight implementation in mind [41]. As such, the set of primitive inference rules for that system are close to the previously enumerated list. I have included the specifications of these rules in Figure 3.18, both to show what the complete logic of a HOL system looks like, and because HaskHOL reuses these primitive rules as some of its own. Note that HOL Light does not elect to include the definition of propositional connectives in its kernel, such that its rules for assumption elimination and *modus ponens* are modified following the discussions from preceding paragraphs. Additionally, HOL Light includes a primitive rule for transitivity, $TRANS$, even though it can be derived from the others.

Listing 3.1: A Primitive Implementation of Transitivity

```
primTRANS :: HOLThm -> HOLThm -> Either String HOLThm
primTRANS (ThmIn as1 c1) (ThmIn as2 c2) =
  case (destEq c1, destEq c2) of
    (Just (l, m1), Just (m2, r))
      | m1 `aConv` m2 ->
          let as' = as1 `termUnion` as2 in
            Right . ThmIn as' $ safeMkEq l r
      | otherwise -> Left "primTRANS: middle terms don't
         agree"
    _ -> Left "primTRANS: not both equations"
```

The difference between a primitive rule and a derived rule is that primitive rules have access to the internal constructors of the theorems they operate over and derived rules do not. For example, a possible Haskell implementation of $TRANS$ as a primitive is shown in Listing 3.1. In Haskell, pattern matching requires access to the internal constructors of a data type in order to work. HOL theorems can be safely destructed without pattern matching, though, by binding the results of a call to a destruction method: `let (as1, c1)= destThm th1 in ...` The more important use of the internal constructor, `ThmIn`, is when the `primTRANS` rule constructs a new theorem value for its successful return case: `Right . ThmIn as' $ safeMkEq l r` Following from the LCF style, only primitive rules are able to construct theorems in this way. An alternative, derived implementation of transitivity is shown in Listing 3.2.

In this implementation, the internal constructor of the theorem type is never used. The theorem is destructed following the previously described alternative technique and theorem construction is a result of the composition of the primitive inference rules `primMK_COMB`, `primEQ_MP`, and `primREFL`.

Listing 3.2: A Derived Implementation of Transitivity

```
ruleTRANS :: HOLThm -> HOLThm -> Either String HOLThm
ruleTRANS th1 th2 =
    let (_, c) = destThm th1 in
      do (eq, m) <- note "ruleTRANS: not an equation"
                     (destComb =<< rator c)
         th3 <- primMK_COMB (primREFL eq) th1
         th4 <- primMK_COMB th3 th2
         primEQ_MP th4 (primREFL m)
```

Compared to its primitive implementation, the derived version of $TRANS$ has a number of inefficiencies. The overall work to manipulate the assumption lists of the provided theorems remains the same, however, their conclusions are repeatedly destructed and compared with each call to `primMK_COMB` and `primEQ_MP`. Additionally, the construction of each intermediate theorem increases the total allocation of the rule. For small proofs these inefficiencies are relatively benign, however, they have a significant impact on performance when a rule is called thousands to millions of times. This is the general justification behind including additional rules as primitives of a system.

**Definitional Extension and Theories**

The final method for introducing theorems is as a consequence of definitional extension. Following from the LCF style's overall bootstrapping approach, new constants of the language are defined using existing types and terms accordingly. This results in a hierarchy of constant definitions that is rooted in the primitive constructs of the language. These definitions are returned as the conclusions of theorems that can be viewed as axiomatic specifications of constants.

The introduction of a new constant term is essentially the assignment of an identifier to act as an alias for a given expression. This assignment is captured by a theorem of the form $\vdash c = t$ where $c$ is the new constant with the desired name and $t$ is its definitional term. In order to successfully define a constant in this way, there are a number of restrictions that must be satisfied:

- The name for $c$ must not be associated with an existing constant.
- The term $t$ must be closed, i.e. it can have no free variables.
- The new constant $c$ can not appear anywhere in its own definition.

$$
\begin{array}{lll}
\vdash P = (x : \mathit{bool}) & & (1) \\
\vdash P = T & [x \mapsto T]\,(1) & (2) \\
\vdash P = F & [x \mapsto F]\,(1) & (3) \\
\vdash T = P & \text{By Symmetry of (2)} & (4) \\
\vdash T = F & \text{By Transitivity of (4) and (3)} & (5)
\end{array}
$$

Figure 3.19: Proof of Contradiction Through Open Definitions

Requiring definitions to be closed prevents substitution from introducing inconsistencies. For example, if the variable definition $\vdash P = (x : \mathit{bool})$ was permitted it could be instantiated to produce both of the theorems $\vdash P = T$ and $\vdash P = F$. Figure 3.19 shows that trivial manipulations of these theorems lead to a proof of $T = F$, demonstrating the unsoundness of open definitions.

The restriction on the use of the constant $c$ in its own definition is to preclude *unbounded* recursive definitions. For example, the definition of the fixed point operator, $fix = \lambda f.\ f\ (fix\ f)$, is infinitely recursive. Recall from the discussion of the $\lambda$-calculi in Chapter 3 that infinite recursion, or any other form of non-termination, prevents a language from being strongly normalizing. The logical correspondence of strong normalization is consistency, thus, a logic that permits infinitely recursive definitions is necessarily inconsistent.

Most HOL systems have derived proof tools external to the kernel that do permit *bounded* recursive definitions, however. Without this capability, we could not represent recursive data types and their associated methods, such as the `List` example from Listing 1.1. Using `append` as an example, its definition can be represented as the conjunction of two clauses:

$$(!ys.\ append\ Nil\ ys = ys) \wedge (!x\ xs\ ys.\ append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys))$$

A general recursive definition, like `append`, must satisfy two properties to be admissible in a HOL logic:

- Its clauses must not be mutually inconsistent, e.g. only one clause applies for any given value.
- The recursion must be well-founded, e.g. it operates over a well-ordered set with a minimal element, such that recursion is provably terminating.

The definition of `append` clearly satisfies the first property because there are distinct clauses for each possible list constructor. It satisfies the second property because its recursive clause destructs lists element by element, with each recursive call bein guaranteed to use a list smaller than the original argument. This process repeats until recursion terminates with the $Nil$ constructor clause. The recursive definition of another list function, `repeat`, would be rejected, though, as its single, recursive clause has a stream-like, constructive behavior: $!x.\ repeat\ x = Cons\ x\ (repeat\ x)$. There is no way to prove this recursion to be well-founded because it has no terminating condition.

HOL also permits the definition of primitive constants that have no associated definition. The equality operator is an example of one such constant that is included in every HOL kernel. These primitive constants are essentially variables that are bound at an inaccessible, global level. This prevents them from being replaced by substitution or rewritten with standard simplification methods. Additionally, primitive constants paired with axioms can be used to circumvent the restrictions of the definitional methods described above. This is useful for constructing definitions whose well-foundedness cannot be automatically proved; however, as with any other use of axioms, the user should be careful. For example, we can demonstrate the construction of the previously mentioned unsound theorem $\vdash P = (x : bool)$ with the following pseudo-code:

```
let p = newConstant "P" tyBool in
  newAxiom (mkEq p (mkVar "x" tyBool))
```

The introduction of new constant types is slightly more complicated due to the fact that HOL does not support computation at the type level to the same degree it does at the term level. Specifically, there is no way to assert the equality of types which precludes following the equational style of definition used for constant terms. There does exist a mechanism for defining type synonyms, however, they are intended to be used exclusively by the parser; they function purely as syntactic sugar, similar to their counterparts in Haskell and other programming languages.

The following example is derived from the explanation of types in Gordon and Melham's introduction to the original HOL system [33]. Suppose we want to define a calendar date as a triple of naturals representing the day, month, and year. Assuming that we have definitions for the *nat* type and the product type operator, #, we can easily define *date* as a type abbreviation:

$$newTypeAbbrev \quad \text{“}date\text{”} \quad (nat\#nat\#nat)$$

This allows us to ascribe *date* as a type, however, it does nothing to restrict its value space, e.g. $(32, 13, 2015) : date$ is a valid term. Furthermore, any value of type $nat\#nat\#nat$, or one of its other synonyms, will be accepted as a *date* whether that is its intended meaning or not. If we model time in an identical manner, using a triple of naturals to represent hours, minutes, and seconds, then the parser will accept the application $(f : date \to bool)\ (a : time)$. It would be preferable, though, if the term was rejected with a type-checking error.

These problems are solved by defining a new type in bijection with an existing type, using term-level constants to move between value spaces. Continuing with the *date* example, these constants would be:

$$absDate :: (nat\#nat\#nat) \to date$$

and

$$repDate :: date \to (nat\#nat\#nat)$$

The prefixes of these constants indicate their role in the isomorphism; they are the *abstraction function* and *representation function* that move between the new and old type. Simultaneous to the definition of these new constants, *date* is introduced as a new type operator, not just a synonym. Thus, unlike the preceding example, attempting to match *date* and *time* will produce the desired type error.

In more programmer-friendly terms, the abstraction and representation functions can be viewed as the most general constructor and destructor for a type, respectively. Therefore, we can restrict the value space of a new type by associating its abstraction function with a predicate that must be satisfied for a successful construction. Following the above example, assume that we want to constrain the minimum and maximum values for days and months to those found on the Gregorian calendar. We can define this predicate as the *validDate* function shown below:

$$validDate = \lambda(day, month, year).\ (0 < day \wedge day < 32) \wedge (0 < month \wedge month < 13)$$

In addition to acting as a precondition to construction, *validDate* doubles as an assertion to the possible representative values after deconstruction. This creates a biconditional relationship between abstract terms and their representative values:

$$absDate\ (day, month, year) \Leftrightarrow validDate\ (day, month, year)$$

This relationship is important because it is a critical component to proving that the abstraction and representation functions have an inverse relationship. If we start with a value of the representative type, we know that the composition of construction and destruction will return the original value if and only if the original value satisfied the corresponding predicate:

$$\vdash validDate\ x \Leftrightarrow repDate\ (absDate\ x) = x$$

We can make a corresponding statement about the reverse of this composition:

$$validDate\ x \Leftrightarrow absDate\ (repDate\ x) = x$$

However, this proposition is stronger than it needs to be. If we start with a value of the abstract type then we know that the value was correctly constructed to begin with, thus we can exclude the predicate from the final theorem:

$$\vdash absDate\ (repDate\ x) = x$$

Beyond serving to frame the inverse relationship between abstraction and representation, a predicate is used to prove inhabitation of a new type. In HOL, there are only two base types from which other types can be defined: boolean and individual. The boolean type is primitive to all HOL kernels, with its base values, $T$ and $F$, being introduced either as primitives or through definitional extension. Once classical logic is admitted in the prover, the inductive and enumerative properties of booleans are derived, such that its value space is provably closed to just these two values.

Individuals are a notion inherited from some formulations of set theory; they represent an infinite, non-empty set of distinct atoms. Again, in most HOL systems the individual type is introduced as a primitive and its infinite nature is specified through axiomatic specification. Given that both base types are inhabited and that it is impossible to define a bijection from a non-empty set to an empty set, all new types in HOL must also be inhabited. Proving inhabitation is as easy as providing a theorem that demonstrates that a type's predicate is satisfied by a witness:

$$newBasicTypeDefinition \quad date \quad (absDate, repDate) \quad \vdash validDate\ (1, 1, 1)$$

The definition of abstract data types (ADTs) in HOL follows from the implementation of ADTs in most functional languages [2]. Data types are defined as tagged unions of constructors where the tags can be used to build paths to direct injective construction and projective destruction. The definition of the `Either` data type is a literal translation of this idea for binary unions, where possible values are tagged as `Left` or `Right` constructions:

```
data Either a b = Left a | Right b
```

More complicated unions of constructors can be derived from nested applications of `Either`. Two such examples are shown in Figure 3.20. As the names imply, the `Three` type has three possible constructors and the `One` type has only one. Given that these new types are defined as synonyms, their constructors are simply compositions of `Either`'s constructors.

The implementation of `Three`'s constructors is straight forward. The implementation of `One`'s, however, is slightly more challenging given that we need to find some way to indicate that its `Right` construction is impossible. Borrowing an idea from type theory, we can fix `One`'s right values to the *bottom* type which is uninhabited. As an aside, note that in Haskell a bottom value is implicitly included in every data type. This is due to bottom being associated with non-termination and other "errors" of the language, `undefined`. The `exBottom` example from Figure 3.20 demonstrates this fact.

Figure 3.20: Derived Union Types

```
type Three a b c =               type One a = Either a Bottom
    Either a (Either b c)        data Bottom

conA :: a -> Three a b c         conOne :: a -> One a
conA = Left                      conOne = Left

conB :: b -> Three a b c         conBottom :: Bottom -> One a
conB = Left . Left               conBottom = Right

conC :: c -> Three a b c         exBottom :: One a
conC = Right . Left              exBottom = bottom
                                   where bottom = bottom
```

```
data A = None | Some B
data B = Var Nat
```

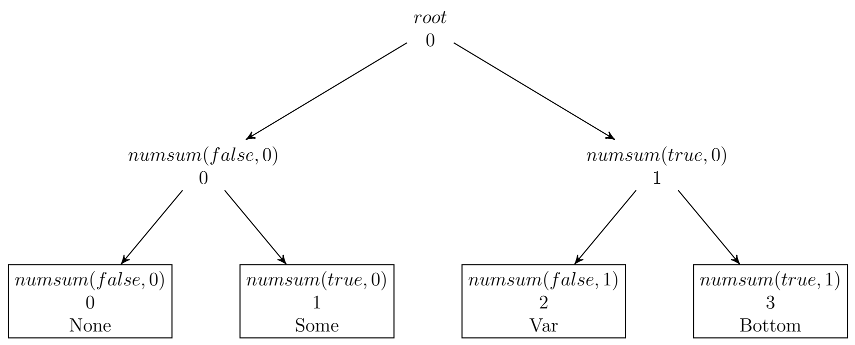


Figure 3.21: HOL Representation of Inductive Types

HOL refines this basic approach with a hierarchy of injective functions based on set theory. These functions allow for the definition of mutually recursive data types in the most general way possible [40]. Key to this technique is the inclusion of `Bottom`, allowing for a balanced binary tree representation of union types, rather than the right or left leaning trees that result from associative nestings of `Either`, such as in `Three`'s implementation in the previous example. A rudimentary example is shown in Figure 3.21.

In this example, the constructors for each type are all combined into a single set and distributed as leaves of a binary tree. This allows each constructor to be indexed by a natural number that can be computed from its position relative to the root:

$$numsum(b, x) = if\ b\ then\ 2 * x\ else\ 2 * x + 1$$

In the above equation, $b$ is a boolean flag indicating a left or right traversal and $x$ is the index of its parent node, with the root's value being set to 0. HOL provides a multivariate constant, $CONSTR$, that accepts these one of these indices and a list of arguments terminated by a $BOTTOM$ value.

These arguments are constructed in one of three ways, following from HOL's set of injective functions:

- Constructors with no arguments are provided an arbitrary value for their argument list. This value is built using Hilbert's choice operator: $\epsilon x.T$.
- Base values share a polymorphic, injective function, such that they can be passed nearly directly.
- Values of a type belonging to the current family are constructed with the appropriate recursive call to $CONSTR$; the constant $CONSTR\_REC$ is used for recursive constructions.

The resultant definitions for the constructors contained in Figure 3.21 are shown below:

```
None = CONSTR 0 (@v. T) BOTTOM
Some b = CONSTR 1 (CONSTR_REC b) BOTTOM
Var n = CONSTR 2 n BOTTOM
```

With the constants for each constructor in place, the type operator for an ADT can itself be defined using a disjunction of its constructors as the predicate. This predicate can obviously be witnessed by supplying any saturated application of one of its constructors:

$$\begin{aligned}&newBasicTypeDefinition \quad A \quad (mkA, destA)\\&\qquad \vdash (\lambda a.\ (\forall a.\ a = None \lor (\exists b.\ a = Some\ b))\ a)\ None\end{aligned}$$

Note that the above definitions and predicate are greatly simplified compared to their actual HOL equivalents. There are various indirections and lifting functions that are critical for a number of reasons, but they are not particularly important to this high-level explanation.

HOL's overall approach to type definition is comparable to predicate sub-typing, as found in the PVS proof system [76]. The primary difference in HOL is that computation of side-conditions introduced by type predicates are performed at the term-level. Conversely, PVS statically tests predicates as part of type-checking, leaving assumptions it can not prove automatically as "type correctness conditions" (TCCs) to be handled by the user. An example of one such TCC is type inhabitation; something that is required in HOL, but optional in PVS. Joe Hurd has shown how to implement actual predicate types in a HOL system, however, to my knowledge no system has elected to do so as of yet [48].

Mirroring definitions at the term level, HOL permits the introduction of primitive types without definition. This does not mean that these types are uninhabited, rather, the predicate that defines their inhabitation is simply unknown. As was previously mentioned, booleans and individuals are two such examples, as they are both introduced through primitive extension; until their respective properties are derived, their sets of values remain unknown. As was the case with primitive terms, be careful pairing primitive types with axioms as unsound definitions can be introduced.

Collectively, the set of definitions and axioms introduced by a user are referred to as the current working theory. These theories may also contain other auxiliary context values, such as parser extensions, configuration flags, rewrite lemmas, etc., that are beneficial to the user but have no impact on the underlying logic of the system. The contents of a theory are left out of the display of theorems as their intended meanings are typically understood without the additional information. For example, the theorem $\vdash T$ explicitly requires a definition for the truth constant. This is not always the case, though, as, unless you are overly familiar with HOL systems, it might not be obvious that the theorem $\vdash P \vee \neg P$ implicitly requires the introduction of several axioms[6].

Any theory can be reconstructed by tracing its extensions back to the logical kernel of the system, e.g. the definition of term pairs requires the definition of conjunction, which requires the definition of truth, which requires the boolean type from the logical kernel. Rather than repeatedly rebuilding theories, theory checkpoints are set when a large collection of related context values can be isolated. For example, the boolean theory checkpoint contains the definitions for truth, falsity, and the propositional connectives, as well as their associated parser/printer extensions.

[6]The admission of classical logic necessitates several axioms in most HOL system.

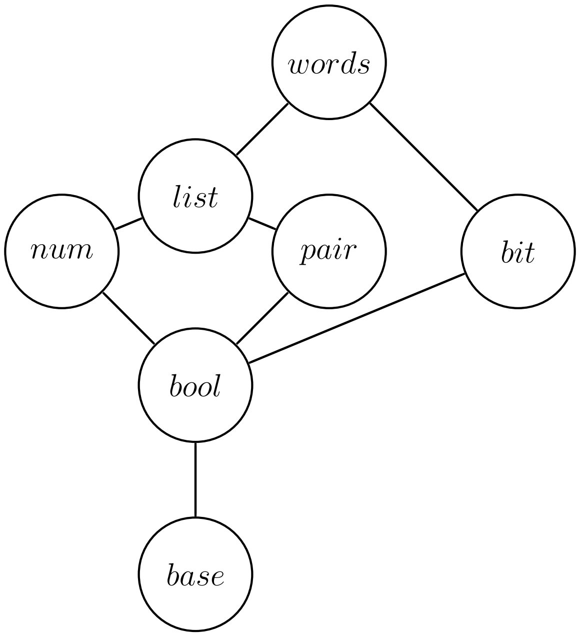


Figure 3.22: A Subset of HOL Theories

The organizational structure of these checkpoints forms a semi-lattice, with each edge indicating a relationship between two theories. To demonstrate this visually, a subset of the theories provided by HOL is shown in Figure 3.22. The *base* theory is the "bottom" of this lattice from which other theories are extended. The line between *base* and *bool* indicates not only a dependency, but also inheritence; *bool* contains the entirety of the *base* theory. Most systems provide a large collection of theories as a prelude, leaving users free to manually include any other theories they need for their proofs.

**Forward and Backward Proof**

In its simplest form, a proof in HOL is the construction of a theorem through repeated applications of primitive inference rules. This process may be aided by introducing lemmas as axioms or through the definitional methods described in the preceding subsection. Proceeding in this manner is referred to as *forward proof* because each new theorem, intermediate and final, is derived from knowledge that came before it.

Comparatively, *backward proof* of a proposition is conducted by showing the sum of its constituent parts to be true. Backward proof is also frequently referred to as subgoal-directed proof because the destruction of a *goal* into smaller *subgoals* is what drives the process. These subgoals are themselves destructed until they can be reduced to a provably true value, at which point they are *accepted* and their proof obligation is discharged. When no subgoals remain, the proof of the original goal is considered complete.

Previous chapters of this dissertation contained examples of proof in both of these forms. Figures 1.1 and 1.2 from Chapter 1 contained a specialized version of forward proof – *equational reasoning*. Figure 2.4 from Chapter 2 contained a backward proof, as might have been obvious from its use of the text "goal" and "subgoal." Though not well illustrated by these examples, each approach has its advantages and disadvantages.

Forward proof is best applied when there is a direct and obvious path for theorem construction. For this reason, it is most commonly used when transforming a general statement to a more specific form. This was the case in the previously cited examples where polymorphic properties were proved for specific instantiations. Where forward proof tends to fall short, though, is in the construction of very large or very complex theorems.

Inference rules paired with methods of automation can assist in these cases, but this is the logical equivalent of swinging a very large hammer, i.e. it is not always the best tool for the job. When dealing with difficult proofs, working backwards tends to be preferable as each subgoal necessarily represents a simpler problem to solve. The trade off, though, is that backwards, logical leaps are not always obvious, especially when re-examining a completed proof. Reading a subgoal-directed proof requires additional effort to "glue the pieces back together," compared to following the step-by-step presentation of a forward proof.

The main tool of forward proof, derived inference rules, was already introduced earlier in this chapter. To reiterate, mirroring both the general flow of information in forward proof and the bootstrapping nature of the LCF style, derived rules are composed of other rules already in existence. The previous discussion of theorem construction compared two possible implementations of the $TRANS$ rule, one of which was derived from applications of primitive rules. Derived rules can, of course, also utilize other derived rules, as is the case in the implementation of the $\alpha$-equivalence rule:

```
ruleALPHA :: HOLTerm -> HOLTerm -> Either String HOLThm
ruleALPHA tm1 tm2 =
    ruleTRANS (primREFL tm1) (primREFL tm2)
```

One class of derived rules worth covering in more detail is HOL's conversion language. A conversion, as the name implies, is a specialized rule that accepts a term, converts it to a new form, and returns a theorem asserting these forms' equivalence. The simplest, non-failing example is reflexivity, where a term is asserted to be equal to itself:

$$primREFL\; x \rightsquigarrow \vdash x = x$$

The primitive $BETA$ rule is another example of a conversion, however, it is typically replaced by a derived implementation that allows for reduction with values of the same type as, but not necessarily equal to, the bound variable:

$$convBETA\; (\lambda x.\; t)\; y \rightsquigarrow \vdash (\lambda x.\; t) = [x \mapsto y]\, t \quad when\; type(x) = type(y)$$

Conversions are unique among derived rules because they have an inherent notion of success and failure; if they succeed, the result will always be a theorem of the form $\vdash x = y$. Knowing the structure of the resultant theorems allows for the definition of a conversion combinator language. For example, given two conversions that return the theorems $\vdash x = y$ and $\vdash y = z$, their sequential application can be performed by using the $TRANS$ rule to combine these intermediate outputs.

Listing 3.3: A Conversion Combinator Language

```
type Conversion = HOLTerm -> Either String HOLTerm

convALL :: Conversion
convALL tm = return (primREFL  tm)

convFAIL :: String -> Conversion
convFAIL str _ = Left str

convTHEN :: Conversion -> Conversion -> Conversion
convTHEN conv1 conv2 tm =
    do th1 <- conv1 tm
       tm' <- note "" rand (concl th1)
       th2 <- conv2 tm'
       ruleTRANS th1 th2

convORELSE :: Conversion -> Conversion -> Conversion
convORELSE conv1 conv2 tm =
    either (\ _ -> conv2 tm) return (conv1 tm)
```

This sequencing combinator is shown in Listing 3.3 as part of a basic combinator language. Just like the other implementations of rules shown up to this point, this language relies on the `Either` type for basic exception handling. This allows for relatively simple definitions of combinators for success (`convALL`), failure (`convFAIL`), sequencing (`convTHEN`), and alternating (`convORELSE`). Much like the derived rules they are designed to support, these four basic combinators can be combined to form more complicated language constructs, as shown in Listing 3.4.

Listing 3.4: Derived Conversion Combinators

```
convFIRST :: [Conversion] -> Conversion
convFIRST [] = convFAIL "convFIRST: empty list"
convFIRST xs = foldr1 _ORELSE xs

convTRY :: Conversion -> Conversion
convTRY conv = conv `convORELSE` convALL

convREPEAT :: Conversion -> Conversion
convREPEAT conv =
    (conv `convTHEN` convREPEAT conv) `convORELSE` convALL
```

The derived combinators `convTRY` and `convREPEAT` are two examples of conversionals, functions that modify the behavior of a conversion. In this case, the evaluation result is guarded, ensuring that one or many applications of a conversion, accordingly, are wrapped by an outer conversion that will never fail. Conversionals can also be used to modify the target of a conversion, applying it to only a portion of a term, rather than its entirety. These subterm conversionals are most useful when you either can not, or do not want to, bind an intermediate term in order to convert it. For example, the `convABS` conversional will retarget a conversion to the body of an abstraction, returning a theorem of the form $\vdash (\lambda x.\ t) = (\lambda x.\ t')$.

Collectively, the set of subterm conversionals can also be used to define depth-based traversal combinators. These traversals navigate the structure of a term from top-down or bottom-up, applying a given conversion to a specified set of subterms: only the first set of subterms it succeeds on, every subterm, or every subterm, repeatedly, until there are no longer any changes. The traversing conversion `(convREDEPTH convBETA)` is one such example; it attempts to $\beta$-reduce every subterm, repeatedly, in a bottom-up manner. This has the net result of producing a theorem that asserts the equivalence of a term and its $\beta$-redex-free form.

The main tool of backward proof, tactics, are the approximate, logical opposite of conversions. Rather than being used for the purpose of construction, tactics are destructive methods for splitting goals into sets of subgoals. Much like a conversion builds an equivalence between old and new terms, a tactic builds a *justification* that can be used to reconstruct a goal from its constituent subgoals. For example, the $tacCONJ$ tactic destructs a conjunctive goal to produce two subgoals, one for each clause. The justification for this tactic reintroduces the conjunction with the $ruleCONJ$ rule, combining the proofs for each subgoal.

In each step of a tactic-based proof, the current set of subgoals and their corresponding justification is tracked by the *goalstate*. This goalstate also records any metavariables introduced by tactics and carries an instantiation for terms, types, and higher-order data. This instantiation is used by the justification to allow for the acceptance of more general proofs of subgoals. A tactic-based proof is considered complete when the goalstate is void of subgoals and the justification returns a theorem concluding the original goal. In the event that this theorem has a non-empty assumption list, it is an indication that an invalid tactic was used at some point in the proof.

Tactics can be interacted with using a combinator language much like conversions; in fact, a number of the same primitive combinators are shared, e.g. success, failure, sequencing, alternating, etc. The implementations of these combinators mirror their conversion equivalents with the exception of sequencing. There is no set return value for a tactic, in that any number of subgoals may be produced. Sequencing of tactics, therefore, must replicate the second tactic and apply it each and every subgoal produced by the first tactic. Alternatively, a list of tactics can be accepted as the second argument to sequencing, mapping each one to a corresponding subgoal. Both of these combinators, shown in Listing 3.5, rely on a sequencing function, `tacsequence`, to ensure that the many intermediate instantiations and justifications are properly composed within the new goalstate.

Listing 3.5: Sequencing Combinators for Tactics

```
_THEN :: Tactic -> Tactic -> Tactic
_THEN tac1 tac2 g =
    do gstate@(GS _ gls _) <- tac1 g
       let tacs = replicate (length gls) tac2
       tacsequence gstate tacs

_THENL :: Tactic -> [Tactic] -> Tactic
_THENL tac1 tacs g =
    do gstate@(GS _ gls _) <- tac1 g
       let tacs' = if null gls then [] else tacs
       tacsequence gstate tacs'
```

As was mentioned in the introduction to this section, a subgoal can be discharged by accepting a theorem that proves it. This action is performed by a theorem tactical, a specialized form of tactic that accepts a theorem as an argument: `tacACCEPT :: HOLThm -> Tactic`. The `tacACCEPT` tactical can also be used to construct tactics from existing rules:

```
tacREFL :: Tactic
tacREFL g@(Goal _ (Comb _ x)) = tacACCEPT (primREFL x) g
tacREFL _ = fail "tacREFL: goal not a reflexivity."
```

The derivation of tactics from rules cannot be generalized, however, derivation from conversions can be: `tacCONV :: Conversion -> Tactic`. The tactic produced by `tacCONV` can destruct a goal in one of three ways. When the provided conversion is applied to the conclusion of the goal, $w$, and the result is a proof of an $\alpha$-equivalent term, $w'$, the goal is accepted. If the result is instead a theorem of the form $\vdash w' = T$, the goal can still be accepted by first eliminating the truth equality in the theorem using the $EQT\_ELIM$ rule. Finally, if the result is a theorem of the form $\vdash w' = x$, then a new subgoal for $x$ is created, using this equivalence theorem as its justification.

The interaction between conversions and tactics is best witnessed by the rewrite engine of a system. This engine is powered by rules that, obviously, *rewrite* terms, transforming them based on a provided body of knowledge. The most common application of these rewrite rules is for implementing simplification and other evaluation strategies. For example, when provided with sufficient knowledge of the propositional connectives, rewrites can be used to automatically prove the proposition $x \wedge y \Rightarrow x$ by showing that it simplifies to $T$. The advantage of proving propositions through rewriting, as opposed to using derived decision procedures, is that it tends to be more a direct and, therefore, more efficient process.

The primitive basis for rewriting is captured by the `convREWR` conversion. This conversion attempts to convert a term by matching it with the left-hand side of an equational rewrite lemma. If a match is possible, `convREWR` asserts the equality of the original term and the instantiated right-hand side of the lemma. For example:

$$convREWR\ (\vdash (P \vee \neg P) \Leftrightarrow T)\ (x \vee \neg x) \rightsquigarrow \vdash (x \vee \neg x) \Leftrightarrow T$$

To improve its applicability, this basic rewriting process is extended in two ways. First, it is typically paired with the previously discussed traversal combinators, such that rewrites can be performed in a term somewhere other than just the top level. Second, rather than accepting only a single theorem, rewrites are attempted using a collection of lemmas. These two extensions can be easily combined in a naive manner by mapping `convREWR` over a list of lemmas and combining the resultant conversions with the desired combinators:

```
convREWRS ths = convTOP_DEPTH (convFIRST (map convREWR ths))
```

It is important to note, however, that even partial matching of terms is a fairly computationally expensive operation. Thus, for rewrites involving large terms and/or large collections of theorems, a naive approach quickly becomes infeasible due to the multiplicative complexity of the matching involved. This problem can be solved by organizing rewrites in a data structure that provides for more efficient, matching lookups for terms – a *conversion net*.

The basic idea behind a conversion net is that all of the term patterns in a theorem can be identified and organized ahead of matching to make the process faster. Similarly, the list of conversions that can succeed for each of these patterns can also be constructed ahead of time. Thus, as each subterm is encountered in a traversal, its pattern is identified and searched for in the net. A matching instantiation is computed if, and only if, a list of potentially applicable rewrites is returned by this search. Nets have the added benefit of efficient extension, such that a net constructed from the rewrite rules of a system "installed" at compile-time can easily be extended by theorems provided by the user at run-time.

The appeal of a net-based approach to rewriting is that it can be made highly configurable by defining a general rewrite conversion:

```
convGENERAL_REWRITE :: (Conversion -> Conversion)-> Net -> [HOLThm] -> Conversion
```

The user can select the traversal they want, the conversion net they wish to start with, and any additional lemmas they want to provide. For example, “pure” rewriting can be achieved by supplying an empty conversion net, such that rewrites are based only on the provided lemmas:

```
convPURE_REWRITE = convGENREAL_REWRITE convTOP_DEPTH netEmpty
```

These conversions can also be lifted into tactics using `tacCONV` to allow rewriting to be utilized in forward and backward proof both:

```
tacPURE_REWRITE thl = tacCONV (convPURE_REWRITE thl)
```

This is not to imply that users are restricted to working purely forwards or backwards in a proof. Both approaches can be, and frequently are, combined by constructing lemmas in the middle of a tactic-based proof. An example of a forward proof, backward proof, and combined proof are all shown in Listing 3.6.

Listing 3.6: Examples of Forward Backward and Combined Proof

```
-- Forward Proof
ruleTAUT [str| (a <=> b) ==> a ==> b |]

-- Backward Proof
prove [str| !t1 t2. t1 /\ t2 <=> t2 /\ t1 |] tacITAUT

-- Combined Proof
prove [str| (?b. P b) <=> P T \/ P F |] $
  tacMATCH_MP (ruleTAUT [str| (~p <=> ~q) ==> (p <=> q)
      |]) `_THEN`
  tacREWRITE [thmDE_MORGAN, thmNOT_EXISTS, thmFORALL_BOOL]
```

# 4 HaskHOL

The verification workflow described in Chapters 1 and 2 demands a formal reasoning tool that possess two critical characteristics:

1. The foundational logic of the tool must be capable of representing artifacts of the target language.
2. The tool itself must be capable of integrating with the target language natively.

This chapter describes a proof system that was specifically developed with these characteristics in mind. As is inferable from its name, HaskHOL is a Haskell (*Hask*) implementation of Higher-Order Logic (*HOL*) theorem proving [5]. More specifically, HaskHOL is an embedded domain specific language (EDSL) that provides HOL proof techniques through an application programming interface (API) heavily inspired by the related HOL Light proof system [41]. This API can be used to implement new proof tools or, alternatively, integrate proof capabilities into existing tools; as is the case for this dissertation work.

The implementation of HaskHOL follows the standard approach to building EDSLs: the primitive data types of the language are defined, effectful computations over these types are structured using a monad, and a set of monadic combinators is implemented and exposed to the user as the interface to the language [44]. This approach is analogous to the traditional implementation technique behind HOL systems, the LCF style [28, 31]. In both cases, the entirety of a codebase is bootstrapped from a small kernel that frames the principal features of a language.

The sections of this chapter are intended to explain HaskHOL's implementation in detail. They are organized as follows:

- Section 4.1 provides a more detailed introduction to the LCF style, noting the challenges involved with implementing it with a monadic approach.
- Section 4.2 discusses in depth the details of HaskHOL's foundational logic, a stateless higher-order logic with quantified types.

## 4.1 Challenges Implementing an LCF-Style Prover with Haskell

There is an ideological split in the functional programming community regarding the role and importance of purity in a language's design. Implementors of interactive theorem provers, however, have almost unanimously elected to rely on impure languages to build their systems. Browse through the documentation and source repositories of the most popular proof tools and you will find some variation of the usual suspects, Lisp and ML. There is another shared trend to be noticed – a boot-strapping implementation style that dates back to early LCF systems [34]. This ubiquitous pairing of impure languages with the LCF style has spawned an impressive number of successful theorem prover systems; among them: Isabelle [69], Coq [87], and every member of the HOL family.

When I began designing a monadic proof system suitable to Haskell's standard methodologies, I noted an ancillary trend amongst the previously mentioned provers that were now serving as my inspiration. The pervasive use of impure side-effects in the implementation of LCF-style provers had made it an almost unintentional, secondary characteristic of the approach. As such, not only did I struggle to translate many of the defining features of these provers to Haskell, but I failed to find anyone else who had succeeded at similar attempts.

### Characterizing an LCF-Style Prover

The central tenet of the LCF approach is an abstraction of theorems to a type whose construction is precisely constrained. When this constraint is obeyed, it forces a bootstrapping approach to deriving new methods of construction not found in the core logic. Stated more concretely, when following this implementation technique, the advanced proof capabilities of a system must be reducible to a composition of that system's primitive inference rules. Thus, the soundness of an entire system can be assumed, provided that its logical kernel is itself shown to be sound; this is the essence of the LCF style.

Critically paired with the LCF style is a host language that can faithfully implement the necessary restrictions on the construction of theorems. At minimum, this language must be strongly typed and have a mechanism for controlling the visibility of data type constructors. Typically, this mechanism is provided in the form of the language's module system, its design of abstract data types, or a combination of the two. Implementation of a proof system, therefore, can proceed directly by mapping its core logic to functions in the target language that construct and destruct theorems appropriately.

The LCF style has a more general analog in functional programming: domain specific languages. As was mentioned in the introduction to this chapter, the LCF style is traditionally facilitated through impure features, however, the process for implementing DSLs in pure languages is both equally common and well understood. The only additional required step in a pure approach is that effects that were previously introduced implicitly by the host language must now be explicitly structured with a computational monad.

As a brief aside, it is worth noting that the LCF style does not itself mandate the use of impure side-effects, nor does it dictate how the implementation of a proof system must be structured. Although they have a shared heritage, the systems listed in the introduction differ quite greatly both in design and degree of impurity:

- Isabelle is implemented with Standard ML (SML) [67], a language that is typically referred to as having impure aspects as opposed to being labeled entirely impure. This system makes great distinction between its host language and the various meta-languages it provides, e.g. Isabelle/HOL, by defining them in terms of an intermediate logical framework, Isabelle/Pure. The use of the impure features of SML in Isabelle is mostly constrained to the implementation of this framework, with only sporadic usages occurring beyond this boundary.
- HOL4 [81] is the most-direct descendent of the early LCF and HOL systems, so it should be of no surprise that it too relies on SML as its implementation language. In addition to admitting only a single meta-language, HOL4 differs from Isabelle in that it is less concerned with restricting the use of impure SML features in its implementation.

- Coq is a proof system of complexity similar to that of Isabelle/HOL and HOL4. The main difference between these systems, beyond their differing foundational logics, is that OCaml [59] is used for Coq's implementation language. Compared to SML, OCaml is significantly more impure. Coq takes significant advantage of this impurity in the implementation of its underlying infrastructure, utilizing everything from mutable data structures to unsafe type coercions.

- HOL Light is yet another proof system that elects to use OCaml as its implementation language. As is implied by its name, this system is notable for its comparatively lightweight implementation. The meta-language exposed to the user is simply OCaml extended with quasi-quotations for concrete syntax. As such, the use of impure language features can be found everywhere in this system, including the implementation of logical theories.

- Stateless HOL [93] is a modification of HOL Light that attempts to bound its use of side-effects. Unlike Isabelle or Coq this system is designed to remove impurity from the logical kernel, rather than constrain it there. The goal of this approach is to make the use of mutable state a matter of convenience, rather than necessity; beyond the kernel, the overall purity of the system is unchanged.

Given that the motivation from Chapter 2 requires a lightweight system, the presented monadic approach follows closely from the design of HOL Light and its extensions/modifications. This is an important fact to keep in mind when reading the following subsections, as a number of critical implementation choices were made specifically with this "lightweight" goal in mind. To reiterate earlier statements, HaskHOL is essentially a shallow embedding of HOL theorem proving in Haskell, rather than a full-fledged proof system like Isabelle, HOL4, or Coq.

### A Monadic Approach to the LCF Style

When simulating the side-effects of an LCF-style theorem prover via a monad, it is important to recognize that a portion of the trusted code base is being shifted from the host language to the prover itself. For example, instead of being able to rely on a compiler's implementation of references in good faith, it is now upon the monad to correctly structure and simulate mutable state. Depending on where the definition of this monad is introduced in the proof system, a variety of potential issues could arise.

Implementing the computational monad inside the logical kernel of a system puts it in the critical path of constructing theorems, i.e. the primitive inference rules become monadic computations. The HOL background examples from Chapter 3 utilized the `Either` monad, however, for the purposes of this section, a more general implementation is assumed, e.g.:

```
primTRANS :: SomeMonad m => HOLThm -> HOLThm -> m HOLThm
```

The guarantee of soundness provided by the LCF approach is now at risk, as this monad represents a potential mechanism for bypassing the constraints on theorem construction. If the monad implementation instead resides at a higher level of the system, as it might in a monadic variant of the previously mentioned Stateless HOL, there are still a number of ways to make the system inconsistent, intentionally or otherwise.

Protecting the soundness of a monadic kernel can be achieved by extending the LCF style's core tenet to additionally constrain the construction of monadic computations. Hiding the internal constructors of the monad and its argument types, just as is done for the abstract theorem type, sufficiently implements this requirement. This process is trivially straightforward, so I will focus the remaining discussion on the harder problem: preserving the consistency of a monadic system.

One potential technique for implementing a monad is with the use of monad transformers. These transformers implement one class of effects each, such that they can be combined in a stack-like manner to form a single monad providing a closed set of effects [61]. Take, for example, a basic `State` and `IO` monad transformer stack that could be used to model the effects in a HOL system, as shown in Listing 4.1.

Listing 4.1: The `HOL` Monad Through Transformers

```
-- Type of a Theory Context
type Context = ...

type HOL = StateT Context IO

runHOL :: HOL a -> Context -> IO (a, Context)
runHOL = runStateT
```

This process promotes the reuse of standard library definitions, that can be beneficial, however, it is important to understand the resultant consequences. In order to facilitate arbitrary transformer stacks, a monad's effects are defined by methods contained within type classes associated with each transformer. In this example, the `StateT` transformer is associated with the `MonadState` class that provides, most notably, the `get` and `put` methods. Note that these methods serve as duals such that, when using the standard transformer libraries, it is impossible to expose one for use without also exposing the other.

This presents a potential issue, as `put` can be used to inject an inconsistent theory context into a computation. This can happen in a variety of ways, although the simplest is a restoration of an old context after a proof is performed:

```
bad1 :: HOL HOLThm
bad1 =
    do ctxt <- get        -- store old theory context
       newAxiom ax        -- introduce an axiom
       th <- someProof    -- requires ax to succeed
       put ctxt           -- restore the old context
       return th
```

This problem is not unique to the methods provided by the `MonadState` class. The definition of `HOL` from Listing 4.1 also utilizes the `IO` monad that itself has an associated class, `MonadIO`. This class provides the `liftIO` method that can be used to lift an arbitrary I/O computation into any transformer stack that is rooted with the `IO` monad. Again, this can be used to introduce inconsistency to the system by discarding theory context updates:

```
bad2 :: HOL HOLThm
bad2 =
    do ctxt <- get
       liftIO $ evalState bad2' ctxt
       -- 'evalState' discards the modified context
  where bad2' = newAxiom ax >> someProof
```

As a general rule of thumb, it can be assumed that any transformer that provides a method for lifting computations or destructively updating any of its argument types exposes a pathway to inconsistency. Regrettably, providing an indirect monad definition via a `newtype` wrapper is not sufficient to protect against these issues. The combination of the language extensions `StandaloneDeriving` and `GeneralizedNewtypeDeriving` allows a user to circumvent this wrapper to generate a type class instance, even where one was not previously provided[1]:

```
newtype HOL a = HOL (StateT Context IO a)
    deriving Monad

deriving instance (MonadState Context) HOL
```

An indirect definition that instead uses a `data` wrapper provides the desired protection, but not without negatively affecting the performance of the monad. The preferable approach, shown in Listing 4.2, is to manually construct a flattened version of the desired transformer stack. The requisite methods can then be defined individually, such that their visibility can be controlled just like any other kernel method.

Listing 4.2: The `HOL` Monad Through Manual Construction

```
module HOL (get) where

newtype HOL a =
    HOL { runHOL :: Context -> IO (a, Context) }

get :: HOL Context
get = HOL $ \ s -> return (s, s)

-- Not exposed external to the HOL module
put :: Context -> HOL ()
put s = HOL $ \ _ -> return ((), s)
```

[1]http://www.haskell.org/ghc/docs/7.10.1/html/users_guide/deriving.html

**Type-Directed Extensible State**

Note that the type of theory contexts was left undefined in the examples from the previous subsection. This was intentional, as it is not immediately obvious how to model these contexts for a monadic approach. Even among LCF-style theorem provers in the same family, there are differences in implementation. Using members of the HOL family as an example:

- HOL Light – Models the theory context pragmatically as an implicit collection of top-level, mutable references.
- HOL4 – Models the theory context as a global symbol table maintained by the pre-kernel system of its underlying infrastructure.
- Isabelle/HOL – Models the theory context as an Isabelle/Pure generic proof context[2], again being created and otherwise maintained by Isabelle's underlying system infrastructure.

Shared among these approaches is the notion that the theory context is both heterogenous and extensible outside of the logical kernel. Unfortunately, while Haskell provides a standard analog for most of the commonly occurring side-effects leveraged by LCF-style proof systems, it lacks an agreed upon technique for modeling extensible state. HOL Light's approach *can* be simulated, however, it would require the use of `unsafePerformIO` that is at best a code smell and at worst a serious threat to the type safety of the system. Instead, the following discussion presents an approach similar to the one utilized by HOL4 and Isabelle/HOL.

The use of a central symbol table to carry configuration data is seen in other large Haskell systems. These tables are implemented using a standard Haskell technique for heterogeneity – existential, abstract data types:

```
class Typeable a => ExtClass a where
    initValue :: a

data ExtState = forall a. ExtClass a => ExtState a
```

[2]These proof contexts are a symbol table for a theory, much like the one stored by HOL4, extended with a number of other values and functions to facilitate stateless, parallel proof.

The `ExtState` constructor shown on the previous page can be used to box values of different types, provided they have an instance of the `ExtClass` class, making their collection well typed. The inclusion of the `ExtClass` class constraint serves two purposes. First, it acts as a subclass to other necessary type class constraints. Second, it provides a mechanism to define initial values for individual pieces of the theory context. Defining initial values for context extensions allows static configuration information to be passed in a dictionary manner, rather than via a computation's state that can help with performance.

The theory context is modeled as a map whose indices are serializations of the types of the context's extensions. These serializations are produced via the `Typeable` class, as introduced through the `ExtClass` class. Context retrieval is implemented as a guarded, type-safe casting from the existential type to the type of the target context extension. Note that this approach mandates that each piece of the theory context has a unique type to prevent future extensions from overlapping the index of an old extension. This uniqueness can easily be enforced by utilizing `newtype` wrappers for context types that are not exported outside of the module they are defined.

Listing 4.3: Extensible State Through Existential Types

```
newtype BinderOps = BinderOps [String]
    deriving Typeable

instance ExtClass BinderOps where
    initValue = BinderOps ["\\"]

parseAsBinder :: String -> HOL ()
parseAsBinder op =
    modifyExt (\ (BinderOps ops) ->
                   BinderOps $ op `insert` ops)

binders :: HOL [String]
binders =
    do (BinderOps ops) <- getExt
       return ops
```

An example of these pieces in play is shown in Listing 4.3. Briefly explained, a new, unique type for binder operator tokens and the necessary class instance are defined. This allows a system implementor to write methods for adding and retrieving binder operators to be used during term parsing. In this example, `modifyExt` and `getExt` are primitive monadic computations that follow from the general forms of context map manipulation discussed in the previous paragraph.

### Practical Implications of Stateful Monads

Compound monadic computations expand to sequences of binds that enforce an ordering of effects. For example, the `binders` computation from Listing 4.3 desugars to the following:

```
binders = getExt >>= \ x -> case x of (BinderOps ops) -> return ops
```

As such, a proof that makes explicit use of all the context modifications it depends on is guaranteed to succeed, as a satisfying context is constructed during its evaluation. Listing 4.4 contains once such example where a proof of the truth theorem, `thmTRUTH`, depends only on the definition of the truth constant, introduced to the working theory context by `defT`. This proof will always succeed, assuming a bug-free implementation of the system up to and including its definition.

Listing 4.4: Explicit and Implicit Stateful Effects

```
defT :: HOL HOLThm
defT = newBasicDefinition "T"
  [str| T = ((\ p:bool . p) = (\ p:bool . p)) |]

defI :: HOL HOLThm
defI = newDefinition "I" [str| I = \x:A. x |]

thmTRUTH :: HOL HOLThm
thmTRUTH =
    do tm <- toHTm [str| \p:bool. p |]
       tdef <- defT
       either fail return $
         do th1 <- ruleSYM tdef
            primEQ_MP th1 (primREFL tm)

thmI :: HOL HOLThm
thmI = prove "!x:A. I x = x" $ tacREWRITE [defI]
```

At first glance the other proof computation shown in Listing 4.4, `thmI`, appears to exhibit the same guarantee. However, it implicitly relies on a number of context modifications to parse a term (requires the definition of the universal quantifier `(!)`) and then prove it through rewriting (requires extending the set of basic rewrites). This proof could be changed to make these dependencies explicit, however that is an intractable solution in general. Furthermore, doing so would defeat one of the main purposes of using a mechanized prover, which is to lessen the amount of work involved in a proof.

A proof like `thmI` can remain implicit in its dependencies provided that we guarantee the construction of a satisfying context before the proof is attempted. The simplest way to do this is to lift the set of context modifications for a theory to a separate, top-level computation that is evaluated before any proof effort:

```
loadCtxt :: HOL ()
loadCtxt = defFORALL >> extendBasicRewrites [...]

runHOL (loadCtxt >> thmI) ctxtBase
```

This technique is a monadic approximation of the "implementation of theories as scripts" approach employed by REPL-style proof systems, such as HOL Light. Those familiar with such systems can point out a serious drawback of their use – a significant amount of time is spent preparing the proof environment every time the system is launched. This delay is amplified by the monadic approach described above, as contexts are rebuilt not just once per session, but for each and every evaluation.

More complex systems, such as the recurringly mentioned Isabelle, HOL4, and Coq, avoid this problem by compiling or otherwise storing their theories in a way that allows them to be used on an as-needed basis without requiring repeated work. The implementations of lightweight systems typically provide an embedding that is too shallow to mimic this approach. In HaskHOL, for example, theories are simply a collection of Haskell functions, such that we lack the ability to inspect or otherwise interact with their construction.

We can, however, serialize just the values associated with theories, i.e. the previously discussed theory contexts, through a simple adaptation of the approach to extensible state presented earlier. Most Haskell libraries for persistent data utilize a method of type-directed serialization compatible with the base techniques of this approach. To concretize discussion, the following examples utilize the acid-state library from the Happstack project [79], although the general ideas should be applicable to other libraries.

Data is serialized by acid-state as a collection of binary files whose access is carefully guarded to provide guarantees of the ACID properties. Its API, inspired by database management systems, provides variations on four basic functions: opening a connection to a data store, querying its value, updating its value, and closing the connection. A reimplementation of the `binders` computation using acid-state is shown in Listing 4.5.

The primary change in this reimplementation is that the state of the `HOL` monad is now a file path where a theory context is stored. The `openLocalStateHOL` method opens a connection to the desired theory context value guided by this file path and an initial value, reusing `initValue` from the `ExtClass` class. Once a connection is formed, this value can be queried, as is done in the example, or updated: `updateHOL acid (InsertBinder op)`.

Listing 4.5: Persistent State Through Existential Types

```haskell
newtype HOL a = HOL { runHOL :: FilePath -> IO a }

openLocalStateHOL ast =
    HOL $ \ fp -> openLocalStateFrom fp ast

binders :: HOL [String]
binders =
    do acid <- openLocalStateHOL (initValue :: BinderOps)
       binds <- queryHOL acid GetBinders
       closeAcidStateHOL acid
       return binds
```

The `GetBinders` and `InsertBinder` constructors used here are defunctionalizations of user-written access methods that are automatically generated by the acid-state library. For example, `GetBinders` maps to a method that looks similar to the previous implementation of `binders`, further demonstrating the compatibility of acid-state and the previous approach:

```
getBinders :: Query BinderOps [Text]
getBinders =
    do (BinderOps ops) <- ask
       return ops
```

Contexts are written to disk during the evaluation of their "loading" computations, e.g. `loadCtxt`. By pairing these computations with the context they extend and their associated file path, this creation can be done dynamically:

```
data Context = Context
    { ctxtFP :: FilePath
    , ctxtBase :: Maybe Context
    , ctxtLoad :: HOL ()
    }

ctxtNew = Context "new" (Just ctxtBase) loadCtxt

runHOL' thmI ctxtNew
```

In the above code, `runHOL'` is a wrapper to the previously seen `runHOL` function. This new evaluation method checks for the existence of a theory context before using it. If it exists, its file path is passed to `runHOL` and evaluation proceeds as before, otherwise it is created by first evaluating its `ctxtLoad` computation. This ensures that the effort to create a context is never repeated, giving us a lightweight approximation of the compilation process performed by more complex systems.

### Optimizing Monadic Proof

As was demonstrated in the preceding subsections, it is certainly possible to construct a monad definition that sufficiently simulates the side-effects required in an LCF-style proof system. This implementation is not necessarily performant, though. Perhaps the most commonly cited criticism of using monads is their impact on computational efficiency within a single thread of execution when compared to equivalent, impure programs. The potential problem is demonstrated by the small, example program shown below:

```
main = print . (flip evalState) 35 $
   do x <- f
      y <- f
      return (x, y)
  where f :: State Int Int
        f = gets fib
```

The important thing to note in this example is that there is an expensive sub-computation, `f`, that is repeated. This computation depends on the monadic state value that we can visually identify as remaining constant between evaluations. Unfortunately, the compiler cannot make the same observation, thus `f` is evaluated twice. The program *can* be manually optimized by sinking the point of evaluation within the `where` clause, as shown:

```
main = print $
   let x = f
       y = f in
     (x, y)
  where f :: Int
        f = evalState (gets fib) 35
```

Because `f` is now a pure binding, rather than a monadic one, the compiler is able to perform the appropriate inlining and subexpression elimination, effectively cutting runtime time in half

This example can be shifted to the domain of theorem proving by imagining `f` as a lemma used within a larger proof. Recall from the discussion of monad implementation techniques that forcing the evaluation of a proof computation and then using the resultant theorem can raise inconsistency issues. The general problem is that context modifications critical to a proof could potentially be discarded. This can be protected against by enforcing two properties:

1. Only non-context-modifying computations can be forcefully evaluated.
2. A theorem can only be used within a context consistent with the one in which it was originally proved.

Both properties can be witnessed statically at the type-level by tagging monadic proof computations with phantom type variables:

```
newtype HOL cls thry a = HOL {runHOL :: Context thry -> IO a}


evalHOL :: HOL cls thry a -> Context thry -> IO a
evalHOL m = liftM fst $ runHOL m
```

The first type variable in the definition of `HOL`, `cls`, records a tag for the classification of a computation. This variable is inhabited by one of two possible empty `data` declarations: `Theory`, for theory context-modifying computations; or `Proof`, for effect-free, proof computations. The classification type of a computation is inferred from its component, primitive computations. For example, the previously shown `updateHOL` method would be classified as a `Theory` computation given that it is used to update a context extension value. The classification of effect-free computations are left fully polymorphic, otherwise type inference would disallow their mixing with `Theory` computations. The `Proof` tag, therefore, is only explicitly used when a witness to the first property is needed.

The second type variable, `thry`, records a tag for the working theory required by a computation. These tags are unique, empty data declarations that should be generated for each theory context checkpoint that is associated with a library. For example, the proof computation for the truth theorem, `|- T`, would carry a tag of `BoolThry` indicating that it requires the definition of `T` from the Boolean logic library. Note that the `thry` type variable also haunts the `Context` type definition to guarantee that theory context values stay tightly coupled to their respective tags.

With these tags in place, it is trivial to define an alias to the evaluation function that provides a guarantee of the first property. For the sake of simplified discussion, assume that there exists a method, `runIO`, that can be used internal to this evaluation function to safely escape the `IO` monad[3]:

```
safeEval :: HOL Proof thry a -> Context thry -> a
safeEval m = runIO . evalHOL m
```

However, this definition provides no guarantee of the second property. Once `safeEval` returns a pure value, it can be lifted into any computation via the `return` function, regardless if it is consistent with the theory context or not.

Rather than return an unprotected pure value, the resultant value should also be tagged with the theory context used to compute it. This process can be made general for all possible values by using an open type family that defines methods for both protecting and using protected data safely. On the next page, Listing 4.6 shows one possible implementation that, when paired with `safeEval`, provides guarantees for both of the desired properties.

[3]This assumption holds as long as primitive computations with non-benign `IO` effects are correctly tagged as `Theory` computations

Listing 4.6: Protecting Context-Sensitive Data

```
class Protected a where
    data PData a thry
    protect :: Context thry -> a -> PData a thry
    serve :: PData a thry -> HOL cls thry a

instance Protected HOLThm where
    data PData HOLThm thry = PThm HOLThm
    protect _ = PThm
    serve (PThm thm) = return thm


safeProof :: HOL Proof thry HOLThm -> Context thry
          -> PData HOLThm thry
safeProof mthm ctxt = protect ctxt $ safeEval mthm ctxt
```

### Polymorphic Protection

The `safeProof` method included in Listing 4.6 can be used to safely optimize monadic proof in a number of convenient ways, ranging from run-time memoization to staged, compile-time computation. When paired with the encoding of theory context tags as described in the previous section it too strongly enforces the second, enumerated, safety property. With their current formulation, for each call to `protect` the type checker can infer only a single, possible type for the theory tag, such that protected values can be reused *only* in the context in which they were computed.

In the absence of primitive "undefinition" methods, LCF-style theory contexts can be assumed to be monotonic. Thus, a theory context is always consistent, not only with itself, but with any new context formed through its extension. A protected value, therefore, should be reusable in the context in which it was computed *and* any of its subsequent extensions. To safely permit this, a theory tag must be polymorphic, with its possible instantiations constrained to the appropriate set of contexts.

As was the case with the abstract representations of theory contexts themselves, the HOL family of provers differs in how they model context hierarchies. Again, HOL Light takes a pragmatic approach and leverages its host language's script-like execution scheme to build a strict, linear ordering of theory contexts. HOL4 and Isabelle/HOL, on the other hand, both have much more complex representations that resemble the semi-lattice structure discussed in the HOL background section. In either case, constraining the theory tag follows directly from translating a context hierarchy to an equivalent type representation.

The technique utilized by the previous subsections reified theory contexts to the type-level by recording only the most recently seen checkpoint. In order to construct a sufficiently polymorphic view of contexts, we must instead record information about all the checkpoints seen in the construction of a theory, including their relationships. With this information available, constraints on the `thry` type can be expressed as tests for the existence of requisite checkpoints in the type representation of a theory context.

In the interest of simplifying the following discussion, a linear ordering of theories, like HOL Light's, is assumed. Under this assumption, the type of a theory context can be encoded directly by promoting the ordered list of its theory checkpoints to the type level. We perform this promotion by mimicking the syntax of list construction, using the type `ExtThry c t` to denote an extension of theory `t` with the checkpoint `c` and the type `BaseThry` to denote the unextended theory of our logical kernel. The existence of a checkpoint can be tested for, therefore, by recursively searching this linear type, much as one would a list.

Listing 4.7: Constructing Polymorphic Theory Constraints

```
data BoolThry
type instance BoolThry == BoolThry = 'True
type BoolType = ExtThry BoolThry BaseThry

type family BoolContext a :: Bool where
    BoolContext BaseThry = 'False
    BoolContext (ExtThry a b) =
        (a == BoolThry) || (BoolContext b)

type family BoolCtxt a :: Constraint where
    BoolCtxt a = (BaseCtxt a, BoolContext a ~ 'True)

type instance PolyTheory BoolType b = BoolCtxt b
```

Using Boolean logic as the example theory, Listing 4.7 shows the pertinent types and instances for the construction of its theory type constraint, `BoolCtxt`. The crux of the presented solution lies in the closed type family definition of `BoolContext`. The two instances for this family collectively define the recursive search pattern for theory types, with the first instance serving as the terminating, base case.

This search pattern relies on a promotion of boolean values to the type level in order to define type-level functions for both equality, `(==)`, and disjunction, `(||)`. Applications of these functions are reduced by a constraint solver which uses type family instances as rewrite rules. Each checkpoint tag has its reflexive equality asserted by a type instance, such that when a constraint is applied to a satisfying theory type it will reduce to the promoted value `'True`. Constraints that are not satisfied will statically fail with one of the following error messages:

- `Couldn't match type ''False' with ''True'` - The base theory context was provided and it failed to satisfy the constraint.
- `Couldn't match type 'A == B' with ''True'` - A theory context was provided with the last seen checkpoint `A`, however, the constraint requires the checkpoint `B` or later, e.g. `Couldn't match type 'BoolThry == SimpThry' ....`

In order to constrain a theory tag to require it to contain the Boolean logic checkpoint, an equality constraint of the form `BoolContext thry ~ 'True` is introduced to the type context. The `BoolCtxt` type both provides a shorthand for this constraint and asserts the constraint of the theory the Boolean logic library extends, `BaseCtxt`. For cases where multiple checkpoints are required, e.g. `(BoolCtxt thry, TheoremsCtxt thry, SimpCtxt thry)`, enforcing the linear ordering inside type constraints allows us to write only the most inclusive constraint, `(SimpCtxt thry)`, as it necessarily implies the others.

The `PolyTheory` type is used to relax a monomorphic theory tag to its corresponding polymorphic constraint. This allows us to redefine `safeProof`, as shown in Listing 4.8. Protected values, such as `pthmTRUTH`, are now assigned the correct, polymorphic type when they are constructed. This theorem can now be safely served in the context it was constructed, `ctxtBool`, and one of its extensions, `ctxtSimp`, but not in a context lacking the Boolean logic checkpoint, `ctxtBase`.

Listing 4.8: A Refinement of `safeProof`

```
safeProof :: PolyTheory thry thry'
          => HOL Proof thry HOLThm -> Context thry
          -> PData HOLThm thry'
safeProof mthm ctxt = protect ctxt $ safeEval mthm ctxt

pthmTRUTH :: BoolCtxt thry => PData thry HOLThm
pthmTRUTH = safeProof (...) ctxtBool

-- These will succeed
testGood1 = evalHOL (serve pthmTRUTH) ctxtBool
testGood2 = evalHOL (serve pthmTRUTH) ctxtSimp

-- This will fail
testBad = evalHOL (serve pthmTRUTH) ctxtBase
```

## 4.2 Stateless Higher-Order Logic with Quantified Types

From the theorem proving perspective, the bootstrapping approach to HaskHOL's implementation is critical to being able to ensure the soundness of the system. To continue harping upon the LCF style's key contribution, by limiting the trusted computing base of a proof system to a small, logical kernel, the lines of code that need to be verified is greatly reduced. From the language design perspective, however, this design pattern is less critical than it is useful.

Specifically, the benefit of implementing a language in the manner described in the previous section is that it forces you to consider the design of its features from the inside out. As was demonstrated, this can be extremely helpful when developing a side-effectful EDSL using a pure host language, as the primitive methods and constructors will dictate the shape of the computational monad. Take, for example, the subset of the HOL Light logical kernel shown in Listing 4.9 that provides for definitional extension of type constants. Using this code as a guide, we see that in order to implement a pure version of `new_type`, we must account for the two previously identified classes of side-effects: exceptions and global state.

Listing 4.9: A Subset of the HOL Light Logical Kernel

```
let the_type_constants = ref ["bool",0; "fun",2]

let get_type_arity s = assoc s (!the_type_constants)

let new_type(name,arity) =
  if can get_type_arity name then
    failwith ("new_type: type "^name^" has already been
        declared")
  else the_type_constants :=
      (name,arity)::(!the_type_constants)
```

Listing 4.10: A Naive Approach to Effects

```
type HOL a = StateT [(String, Int)] IO a

getTypeArity :: String -> HOL Int
getTypeArity name =
    do tys <- get
       maybe (fail $ "getTypeArity: type " ++ name ++
                     " has not been defined.")
         return $ lookup name tys

newType :: (String, Int) -> HOL ()
newType (name, arity) =
    do whenM (can getTypeArity name)
         (fail $ "newType: type " ++ name ++
                 " has already been declared.")
       modify (\ tys -> ((name, arity):tys))
```

Listing 4.10 includes a naive translation of this HOL Light subset that corresponds approximately to the implementation of the `HOL` monad from Listing 4.1 in Section 4.1. Comparing the HOL Light implementation to the code above, specifically `get_type_arity` to `getTypeArity`, highlights a major drawback of a monadic implementation; the introduction of structured side-effects can quite easily complicate code. For that reason, the logical kernel of HaskHOL is designed to minimize the use of side-effects as much as possible.

## A Stateless Kernel

With this goal in mind, the primitive data types of HaskHOL are modeled after the Stateless HOL approach developed by Freek Wiedijk [93]. To summarize Wiedijk's solution, auxiliary information about HOL data types is embedded directly in their construction, rather than being stored via stateful effects. Stateless introduction of a new HOL constant involves creating either a new constant tag or type operator, which can then be provided as an argument when building a HOL term or type accordingly.

Listing 4.11: The Stateless HOL Approach to Effects

```
let new_prim_type_op(name,arity) =
  Typrim(name,arity)

let new_type'(name,tyop) =
  if can get_type_arity name then
    failwith ("new_type: type "^name^" has already been
        declared")
  else the_type_constants :=
     (name,tyop)::(!the_type_constants)

let new_type(name,arity) =
  new_type'(name,new_prim_type_op(name,arity))
```

Stateless HOL's implementation of `new_type` is shown above in Listing 4.11. The crux of this implementation is the segregation, and eventual combination, of stateful (`new_type'`) and stateless (`new_prim_type_op`) methods. The stateless methods of the system act as smart constructor wrappers for the primitive data types; in this case, for `Typrim`. When these smart constructors are fully applied they create closures for tag and operator objects that can be reused in common sub-terms, giving the data types that form Stateless HOL theories a graph-like shape.

In order to maintain comparatively similar performance to the stateful HOL Light system it's based on, Stateless HOL relies on OCaml's object comparison model. By testing the pointers to embedded information for physical equality, they can be compared without having to traverse and, therefore, evaluate the closures they point to. This allows HOL terms and types to be efficiently and, most importantly, statelessly checked for definitional equality with only a minimal memory overhead cost. In essence, the observable sharing in Stateless HOL data types can be gleaned almost for free.

Regrettably, this approach is not applicable in HaskHOL for a number of reasons. The most troublesome obstruction is the lack of a pure, physical equality mechanism in Haskell. The `StableName` [4] and `StablePointer` [5] data types could be used to provide similar functionality, however, neither is an ideal solution. Creation of a `StableName` or `StablePointer` requires use of the `IO` monad. Thus, we are either forced to introduce unwanted effects to the lowest levels of the logical kernel or abuse the *dreaded* `unsafePerformIO` method; both are "solutions" I would prefer to avoid.

Additionally, the implementation of the `StableName` and `StablePointer` data types makes internal use of compiler-specific primitives, `StableName#` and `StablePtr#` repsectively, precluding the derivation of instances for several useful type classes. Interactions with these constructors are restricted to primitive operations defined in GHC's base libraries. These operations include creation, destruction, comparison, and a handful of purpose built methods, e.g. hashing and casting. It is impossible, therefore, to define an instance of Template Haskell's `Lift` class, preventing HaskHOL from including a number useful features, such as quasi-quotation of types and terms. In fact, any instance not already defined for `StableName` and `StablePointer` values cannot be derived without modification to the GHC system itself.

Andy Gill developed a solution for extracting the observable sharing of a DSL where `StableName` values appear in the intermediate construction of an abstract syntax graph (ASG), but not the final result [26]. Given that the `StableName` generated for an object can vary pre and post evaluation, Gill's `reifyGraph` function forces the complete traversal and strict evaluation of a target data structure before its sharing is observed. This brings to surface the secondary concern of implementing stateless data types in Haskell; fully evaluated stateless values utilize exponentially more constructors than their stateful equivalents. Thus, any operations that require a complete traversal of these constructors will necessarily be computationally more expensive.

[4]`http://hackage.haskell.org/package/base-4.7.0.1/docs/System-Mem-StableName.html`
[5]`http://hackage.haskell.org/package/base-4.7.0.1/docs/Foreign-StablePtr.html`

An example of one such operation is the persistence of theory context values via serialization to disk, as described towards the end of Section 4.1. Storing stateless values in their normal, AST representation will produce serializations exponentially larger than their stateful equivalents. Extracting any observable sharing and instead storing their ASG representation does not improve things much, as this process adds a non-trivial amount of run-time overhead. ASG values would need to be converted back to their AST form or, alternatively, primitive term construction and destruction methods would need to be modified to perform the necessary graph merging and updating. The latter option would be comparable to Stateless HOL's implementation of axiom contexts, a process that was shown to significantly slow down the proof system. This slow down would be amplified in this case, give that term manipulations happen much more frequently than theorem manipulations do.

Given the above issues, HaskHOL's implementation uses a "psuedo-stateless" compromise to Stateless HOL's approach. Auxiliary information with uses beyond just comparison, e.g. constant names, arities, etc., are embedded directly as before. However, information used purely for comparison purposes, e.g. definitional terms and theorems, are stored as an embedded hash instead of their actual value. Example implementations of constant tags and type operators are shown in Listing 4.12.

Listing 4.12: Psuedo-Stateless Data Types

```
data ConstTag
    = PrimitiveIn
    | DefinedIn       Int           -- hash
    | MkAbstractIn    Text Int Int  -- name, arity, hash
    | DestAbstractIn  Text Int Int  -- name, arity, hash

data TypeOp
    = TyOpVarIn       Text          -- name
    | TyPrimitiveIn   Text Int      -- name, arity
    | TyDefinedIn     Text Int Int  -- name, arity, hash
```

In practice, this provides the same efficient comparison as the stateless approach without the exponential blowup in constructors. In theory, though, there is the obvious potential issue where a hash collision could lead to an incorrect equality comparison. Hashes are only compared, though, when the names of constant tags match. Thus, the collision problem can only arise in a context with redefined constants. This is only possible if users are manually driving the prover at the stateless level or utilizing non-monotonic theory extension methods, such as undefinition, at the stateful level. Both of these techniques can be beneficial when developing proofs, however, final verifications should avoid using them to guarantee that the soundness of the system is maintained.

### A Polymorphic Type System

Knowing that GHC Core would be a major verification target for HaskHOL, an additional extension was made to its primitive data types to introduce higher-order polymorphism to its type system. This extension allows for direct representation of Core's types, modulo a few limitations:

- Types must be simply kinded, i.e. only `*` or `->` kinds are allowed.
- Following from the above restriction, polymorphism is restricted to the second order.
- Instantiation of bound type variables is limited to non-universal types.

These limitations preclude reasoning about code that leverages some of GHC's more advanced extensions, e.g. `DataKinds`[6] or `RankNTypes`[7]. However, language features that other provers struggle to represent, for example type classes, can be represented and reasoned about in HaskHOL directly.

[6] `https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/promotion.html`

[7] `https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/other-type-extensions.html`

Again, this extension was inspired by a related HOL system; in this case, Norbert Voelker's HOL2P [90]. The "2P" in this system's name refers to the second-order, polymorphic lambda calculus, making it a logical correspondent to the System F language discussed in Section 3.1. The enumerated limitations of HaskHOL's type system, inherited directly from HOL2P, are due to Coquand. In his *A New Paradox in Type Theory*, it was shown that the combination of System F and HOL is inconsistent [18].

This inconsistency is largely because of System F's impredicative type system. HaskHOL and HOL2P both address this problem by following the previously discussed solution of asserting a hierarchy of types and using it to restrict the possible instantiations of universally quantified type variables. Both systems implement a "smallness" constraint to indicate if a type is universal or not, such that only small types may be bound.

In HOL2P, the distinctions between types are syntactic in nature. Identifier prefixes are used to distinguish between the numerous varieties of type variables since small type variables, large type variables, and type operator variables all utilize the same data type constructor. HaskHOL instead elects to make these distinctions structural by including a boolean field in the type variable constructor that indicates smallness; type operator variables are already structurally distinct due to the stateless approach. HaskHOL's primitive data types for its type system are shown in Listing 4.13, including the `TypeOP` type already shown in Listing 4.12.

Listing 4.13: HaskHOL's Type System

```
data HOLType
    = TyVarIn        Bool     Text
    | TyAppIn        TypeOp   [HOLType]
    | UTypeIn        HOLType  HOLType

data TypeOp
    = TyOpVarIn      Text
    | TyPrimitiveIn  Text     Int
    | TyDefinedIn    Text     Int  Int
```

The separation of type operators into a new data type caused signification issues when blending the concepts of Stateless HOL and HOL2P [7]. Recall the primitive $INST\_TYPE$ rule shown in Figure 3.18 back in Section 3.2:

$$\text{INST\_TYPE}\frac{A \vdash t}{A\,[ty_1, ..., ty_n/tv_1, ..., tv_n] \vdash t\,[ty_1, ..., ty_n/tv_1, ..., tv_n]}$$

Implicit in the semantics for this rule is the definition of a type instantiation function. In the simplest case, type instantiation is a substitution performed over type variables. For Stateless HOL, this substitution is trivially defined by the following equations:

$$\begin{aligned} x\,[ty/x] &= ty && \\ y\,[ty/x] &= y && (y \neq x) \\ (c\ a_1 \ldots a_n)\,[ty/x] &= (c\ a'_1 \ldots a'_n) && \end{aligned}$$

Note that these equations use a tick notation to indicate recursive application of substitution. The intent here is to show that in the `TyAppIn` case only the first field is left unchanged. This is not the case in HOL2P, though, as the first argument to a type application may be a type operator variable and, therefore, may be subject to substitution. This system's additional rules for type variable substitution, including the name capture avoiding rules for universal types, is shown below[8]:

$$\begin{aligned} (\_x\ a_1 \ldots a_n)\,[c/\_x] &= (c\ a'_1 \ldots a'_n) && (arity\ c = n) \\ (\_x\ a_1 \ldots a_n)\,[\Pi b_1. \ldots .\Pi b_m.\ ty/\_x] &= ty\,[a'_1/b_1 \ldots a'_n/b_m] && (m = n,\ ty\ is\ small) \\ (\Pi x.a)\,[ty/x] &= (\Pi x.\ a) && \\ (\Pi y.a)\,[ty/x] &= (\Pi y.\ a') && (y \neq x,\ y\ is\ not\ free\ in\ ty) \end{aligned}$$

[8]The notation $\_x$ indicates that $x$ is a type operator variable

Note that these rules disagree on the contents of their substitution environments. The first two rules have [*type*/*type operator*] substitution pairs, while the second two rules have [*type*/*type*] pairs. In essence, HOL2P dictates that type operators have two different, but conceptually equivalent, representations. The disagreement between rules is eliminated by asserting an isomorphism between these representations. More specifically, type operators in HOL2P are stored as string values, such that its `type_subst` function can use `mk_tyvar` and `dest_tyvar` as coercion methods.

The type variable representation of operators is nonsensical, though, as there are no terms that can inhabit such a types, i.e. there are no typing rules that will return a judgement of the form $\Gamma \vdash \textit{term} : \textit{type operator}$. The only real purpose of this representation is to serve as a mechanism for making the implementation of the type substitution function well-typed in the host language. This solution is troublesome, though, as these coercions leak into the other rules for type instantiation. For example, in the case where a type constant is to be substituted for a type operator variable, the same trick must be used to represent both the constant and the operator as type variables.

HaskHOL's differing approach to type instantiation was primarily motivated by a desire to remove any reliance on disingenuous type representations. Regardless of my qualms with HOL2P's design, though, the integration of a stateless approach eliminated the isomorphic representations work-around described above. In a stateless system, there is no way to reconstruct a type given only its identifier, i.e. there is no `mk_type` function available at the kernel level. Therefore, once a constant or operator is converted to a type variable representation, there is no way to return to its original form. This prevents type operators from being derived during substitution in the same manner as HOL2P, thus they must be constructed beforehand.

[*type*/*type*] Substitution (In all cases *ty* must preserve the smallness of $x$):

$$\begin{aligned}
x\,[ty/x] &= ty \\
y\,[ty/x] &= y && (y \neq x) \\
(c\ a_1 \ldots a_n)\,[ty/x] &= (c\ a_1' \ldots a_n') \\
(\Pi x.\ a)\,[ty/x] &= (\Pi x.\ a) \\
(\Pi y.\ a)\,[ty/x] &= (\Pi y.\ a') && (y \neq x,\ y\ is\ not\ free\ in\ ty)
\end{aligned}$$

[*type operator*/*type operator*] Substitution:

$$\begin{aligned}
x\,[c/\_x] &= x \\
(\_x\ a_1 \ldots a_n)\,[c/\_x] &= (c\ a_1' \ldots a_n') && (arity\ c = n) \\
(\Pi x.\ a)\,[c/\_x] &= (\Pi x.\ a')
\end{aligned}$$

[*type operator*/*type*] Substitution:

$$\begin{aligned}
x\,[\Pi b_1. \ldots .\Pi b_m.\ ty/\_x] &= x \\
(\_x\ a_1 \ldots a_n)\,[\Pi b_1. \ldots .\Pi b_m.\ ty/\_x] &= ty\,[a_1'/b_1 \ldots a_n'/b_m] && (m = n,\ ty\ is\ small) \\
(\Pi y.\ a)\,[\Pi b_1. \ldots .\Pi b_m.\ ty/\_x] &= (\Pi y.\ a') && (y\ is\ not\ free\ in\ ty)
\end{aligned}$$

Figure 4.1: Substitution Rules for Type Instantiation in HaskHOL

Given that type instantiation can no longer accept a singular form for substitution pairs, the previously shown rules must be split into three separate functions – one for smallness preserving type substitutions ([*type*/*type*] pairs), one for substitution of constants types for type operator variables ([*type*/*type operator*] pairs), and one for reduction of universal types by type operator variables ([*type*/*type operator*] pairs). The rules for these functions are shown in Figure 4.1.

# 5 HaskHOL as a Library

Among the most commonly cited advantages of working with purely functional languages is that reasoning about program behavior is significantly easier in the presence of referential transparency. Unfortunately, despite all of the benefits that the functional paradigm provides, it does nothing to assuage a major problem in software verification: marshaling knowledge between implementation and verification environments is, more often than not, just as complex and error prone as attempting proofs by hand. Consequently, conceptually "easy" verifications are frequently left unattempted or incomplete as a matter of inconvenience.

The contents of this chapter detail my approach to eliminating this inconvenience through the titular technique of using theorem provers as libraries. Following from the workflow detailed in Chapter 2, the capabilities of a formal reasoning tool are integrated natively with its host language. This integration allows developers to verify their code in an environment that is already both well established and familiar to them. In addition to mitigating the previously mentioned communication issues, this reduces the verification learning curve to the equivalent of understanding a new DSL; something functional programmers, especially Haskell users, are quite adept at.

The previous chapter covered the implementation of the formal reasoning tool in question, the HaskHOL library for HOL theorem proving. The sections of this chapter are intended to demonstrate HaskHOL's applicability for verifying Haskell programs and explain its linkage with the Glasgow Haskell Compiler (GHC). They are are organized as follows:

- Section 5.1 works through an example verification, noting the correspondence between HaskHOL's foundational logic and GHC's core language.
- Section 5.2 details GHC's compiler plugin framework and the HERMIT library, the chosen method for integration.
- Section 5.3 formalizes HaskHOL's connection to GHC Core and notes current limitations of the work.

## 5.1 Verifying Constructor Classes

To reiterate earlier background material, property directed verification is carried out through repeated substitutions of equivalent forms. Much like the evaluation strategy of the host language itself, properties are rewritten with function and data type definitions until they are reduced into true or false statements; this is the essence of equational reasoning. In Haskell, these correctness properties are often self-introduced, whether the programmer realizes it or not. Listing 2.1 from Chapter 2 contained one such example – a signature for the `Monad` type class paired with its categorical laws.

The catch with equational reasoning is that the effort required is directly proportional to the complexity of both the proof obligation and its requisite definitions. The `Identity` monad was intentionally targeted as the motivating example in Chapter 2 because it has the simplest possible `Monad` instance. Conversely, the definition of HaskHOL's computational monad relies on smart constructors, complex data structures, and other monads. Attempting a paper proof with its definition, while certainly possible, would be both time consuming and error prone. This is of no surprise to anyone familiar with the motivations behind mechanizing proof theory.

This problem only further reinforces the point made in the introduction to this chapter – "easy" proofs about functional programs are typically done either informally or not at all. The formal verifications of functional programs that do exist are spread among a diverse set of proof assistants, logics, and representation techniques; each with their own pros and cons. Focusing on examples related to monads, proofs of the monad laws frequently arise as ad hoc lemmas in larger efforts [38, 85]. The monads in these verifications are "built to order", such that it is difficult to restructure their proofs to be useful outside of their original context. In other words, these proofs are not of immediate benefit to anyone attempting to verify existing and unrelated implementations.

Brian Huffman has presented work that more generally formalizes Haskell type classes [45], including `Monad`, using Isabelle/HOLCF [68] and its Tycon library [46]. Unfortunately, by the author's own admission, portions of the presented technique are impractical for widespread use. The domain-theoretic model of Haskell's type system is logically distant from what the average functional programmer is familiar with and an advanced working knowledge of the underlying proof system and logic is required for the more complicated cases.

Huffman et. al's early work [47] took place at a time before type classes were common, first-class constructs of proof languages. Since then, at least two popular proof systems, Coq and Isabelle/HOL, have been extended with implementations of type classes similar to Haskell's [84, 36]. These systems can be used to more clearly model and verify `Monad` instances; and in the case of working with Isabelle/HOL, there even exists a tool to automate the necessary translations from existing Haskell code [35]. However, Coq and Isabelle are both large, complex systems that offer more reasoning power than is necessary in most cases. Furthermore, they can be overwhelming to work with for new users requiring significant background experience to use effectively.

This section presents an alternative approach to verification of the monad laws in a prover built in Haskell, for Haskell – HaskHOL. Rather than working at the source level, as the systems from the preceding discussion do, HaskHOL instead performs its verifications by translating Haskell programs at the intermediate language level. Given that both Haskell and HaskHOL are based on derivatives of the simply typed $\lambda$-calculus, this makes for a much more direct translation and proof.

### Monads in HaskHOL

Following from GHC's intermediate representation of type classes as dictionary-passing constructors, the `Monad` class is represented in HaskHOL as a constant term whose type is dictated by its parameter and methods:

```
forall (m :: * -> *).
  (forall a b. m a -> (a -> m b) -> m b) ->
  (forall a. a -> m a) -> Monad m
```

Recall from Chapter 4 that HaskHOL's polymorphism is restricted in that universally quantified type variables may only be instantiated by *small* types, i.e. types that are not themselves quantified. Additionally, type operators are distinct from regular types in the language such that they also cannot be used to instantiate quantified type variables. This leads to the construction of monads shown in Listing 5.1[1] where the type operator variable representing the class's parameter, `_M`, is left globally free.

Listing 5.1: The Construction of Monads in HaskHOL

```
MONAD
  (bind : % 'A 'B. 'A _M -> ('A -> 'B _M) -> 'B _M)
  (return : % 'A. 'A -> 'A _M)  =
((!! 'A 'B.
   ! (a: 'A) (f:'A -> 'B _M).
     bind [: 'A] [: 'B] (return [: 'A] a) f = f a) /\
 (!! 'A.
   ! (m: 'A _M).
     bind m return = m) /\
 (!! 'A 'B 'C.
   ! (m: 'A _M) (f: 'A -> 'B _M) (g: 'B -> 'C _M).
     bind (bind m f) g = bind m (\ x. bind (f x) g)))
```

[1]A HaskHOL syntax cheat sheet:
`'A` - Small type variables
`_M` - Type operator variables
`(\) / (\\)` - Term-level term/type abstraction
`(!) / (!!)` - Term-level, universal term/type quantification
`[: 'A]`- Term-level type application
`$` - Type-level universal type quantification
`'A _M` - ML-Style, type-level type application

Listing 5.2: The `Identity` Monad and its HaskHOL Proof

```
prove [str| MONAD (\\ 'A 'B. \ m k. k (RunIdentity m))
              Identity |] $
  tacREWRITE [defMONAD] `_THEN` tacCONJ `_THENL`
  [ _REPEAT tacGEN_TY `_THEN`
    tacREWRITE [defIdentity, defRunIdentity]
  , tacCONJ `_THENL`
    [ tacGEN_TY `_THEN`
      tacMATCH_MP inductionIdentity `_THEN`
      tacREWRITE [defIdentity, defRunIdentity]
    , _REPEAT tacGEN_TY `_THEN`
      tacACCEPT thmTRUTH ] ]
```

The `MONAD` constant is defined as the conjunction of the monad laws, translated to HOL propositions. This translation includes making type abstractions and applications in the laws explicit, similar to how the GHC rewrite system requires pattern variables to be explicitly bound in rules. The specific proof obligation for the `Identity` instance can be achieved by supplying appropriate definitions for `bind` and `return`. Following from the definitions of these methods shown in Figure 2.3, HaskHOL constants for the `Identity` constructor and `runIdentity` destructor, shown in Listing 5.3, are introduced to the working theory. Using these constants, a definition for the `Identity` monad instance is constructed, as shown in Listing 5.2[2]

Listing 5.3: A Definition of `Identity` in HaskHOL

```
defineType [str| Identity = IdentityIn A |]

newDefinition [str| Identity = \\ 'a. \ x:'a. IdentityIn x |]

newRecursiveDefinition recursionIdentity
    [str| runIdentity (IdentityIn x) = x |]
```

[2]The `str` quasi-quoter prepares a `String` for HaskHOL's parser, notably removing the need to escape special characters.

```
proveConsClass consDef indThm thms =
    tacREWRITE [consDef] `_THEN` _REPEAT
    (_TRY (tacCONJ `_THEN` _REPEAT tacGEN_TY)
     `_THEN` _TRY (tacMATCH_MP indThm)
     `_THEN` tacREWRITE thms)

proveIDMonad =
    proveConsClass defMONAD inductionIdentity
      [defIdentity, defRunIdentity]
```

Figure 5.1: A More General Tactic for Constructor Classes

Included in this listing is a proof tactic for this term. The term is rewritten using the definition of the `MONAD` constant to bring the instantiated monad laws into view. Each law is separated from the rest via a conjunctive split (`tacCONJ`), its bound types are generalized (`tacGEN_TY`), and the resultant subgoal is proved by rewriting (`tacREWRITE`). Given that the definition of `runIdentity` depends on pattern matching against the `Identity` constructor, proving the left-identity law subgoal requires an extra step. Manual, rule induction is performed to handle pattern matching by invoking the primitive recursion theorem for the `Identity` type via the `tacMATCH_MP` tactic.

This proof tactic may seem daunting to those unfamiliar with HOL systems. However, it is simply a Haskell value that can be manipulated in the same ways as any other Haskell value. Specifically, the constant values that represent HOL theorems can be abstracted out and tactic combinators can be used to make the structure more general. The `proveConsClass` function shown in Figure 5.1 achieves this through judicious use of `_TRY` and `_REPEAT`. Utilizing the more general theorem tactical, a user only needs to be able to identify three pieces of information to complete their proof:

1. The definitional theorem for the constructor class.
2. The recursion theorem to use for induction.
3. The list of additional theorems to use for rewriting.

This proof procedure has two hiccups that prevent it from being fully generalized. First, astute readers may have noticed that Listing 5.3 contained multiple constructors for the `Identity` data type. Depending on where a constructor appears in GHC's core language, its most general type may or may not be universally quantified. HaskHOL's method for type definition, `defineType`, leaves constructors' type variables globally free. This necessitates the definition of an additional, smart constructor to introduce quantifiers when necessary.

The second problem is that not every constructor class proof will have the same induction pattern. For proofs not requiring induction, an `undefined` value can be supplied for the recursion theorem to intentionally fail, and therefore skip, that proof step. Proofs requiring more complicated induction schemes, however, will require the user to modify the structure of this tactic or develop a tactic of their own. Both of these issues will be addressed in more detail in Chapter 6.

## 5.2 HaskHOL as a Plugin

The primary goal of this work is to mitigate or eliminate as many barriers to entry for formal reasoning as possible. Rather than working at the source level, as many of the related approaches do, the intermediate language of GHC is targeted. Working at the core level of the compiler is beneficial for a number of reasons. Chief amongst them is that using the compiler to desugar and simplify definitions to a core syntax allows for a more direct translation to a corresponding higher-order logic. The resultant syntax this translation needs to account for is comparatively simpler and almost exactly mirrors the term language provided by HaskHOL. The following, additional benefits are also observed:

- The core syntax is extremely stable compared to the source syntax.
- Terms can safely be assumed to be well-typed and correctly constructed before translation.
- Type information is stored locally, simplifying translation.
- Integration with GHC, Cabal, and other system tools is significantly easier.

The final point listed above, simplicity of integration, is in large part thanks to GHC's compiler plugin architecture. A GHC plugin modifies the compilation phases of the compiler pipeline with a function of type `[CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]`. In this signature, the abstract type `CoreToDo` represents a compilation pass, such that a plugin author can add, remove, modify, and reorder passes as they please by manipulating the `[CoreToDo]` argument. Thus, verification can be made a part of the compilation process by inserting it as a new pass into the compilation pipeline at the desired point – before simplification, after simplification, or at any alternative, intermediate step.

Listing 5.4: A Subset of the `ModGuts` Data Type

```
data ModGuts = ModGuts
{ mg_module     :: Module       -- The module itself
, mg_tcs        :: [TyCon]      -- Type constructors
, mg_insts      :: [ClsInst]    -- Class instances
, mg_fam_insts  :: [FamInst]    -- Family instances
, mg_patsyns    :: [PatSyn]     -- Pattern synonyms
, mg_rules      :: [CoreRule]   -- Rewrite rules
, mg_binds      :: CoreProgram  -- Bindings
}
```

The critical piece of any user-defined plugin is the function it uses to interact with a module. This function is of the type `ModGuts -> CoreM ModGuts`. Aptly named, the `ModGuts` data type stores all of the information contained in the "guts" of the module currently being compiled. There are a number of fields in the `ModGuts` type that a verification plugin may be interested in, a subset of which are shown in Listing 5.4. However, the translation from Haskell to HaskHOL is primarily concerned with the `CoreProgram` value stored in the `mg_binds` field. As can be inferred from the name of the field, this data type holds all the top-level bindings in a module, stored as a list of variable and expression pairs. These expressions are values of the core type `CoreExpr` that will be examined in more detail in the next section.

Ultimately, the required translation function will be of type `CoreExpr -> HOLTerm`, where `HOLTerm` is HaskHOL's abstract data type for HOL terms, as previously introduced in Chapter 3. This function could be defined using the `GhcPlugins` module provided as part of the GHC API. However, browsing this module's documentation[3] makes it clear that there would be a significant amount of integration work to do before translation could even be attempted. Instead, the Haskell Equational Reasoning Model-to-Implementation Tunnel (HERMIT) library is used to do most of the heavy lifting.

[3]`https://downloads.haskell.org/~ghc/7.10.1/docs/html/libraries/ghc-7.10.1/GhcPlugins.html`

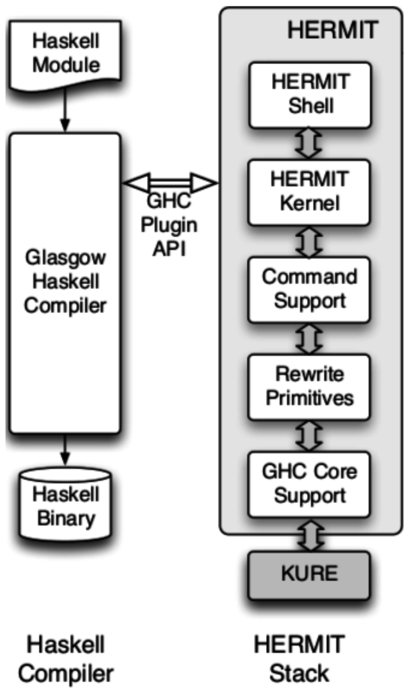


Figure 5.2: The HERMIT Framework

**The HERMIT with the KURE**

Originally built as a tool for interactively developing new optimization passes, HERMIT has since evolved to serve as a generalized framework for constructing new compilation passes of all forms. Figure 5.2, courtesy of Farmer et al., shows the key components of HERMIT [23]. At the root of HERMIT is the Kansas University Rewrite Engine (KURE), an EDSL for strategic rewriting [78]. HERMIT specializes the primitive combinators of KURE to provide generic traversals for GHC's core data types, as indicated by the "GHC Core Support" box in the aforementioned figure.

These traversals are structured by KURE's `Transform` data type that abstractly models a transformation from type `a` to monadic values of type `m b`, with a given context of type `c`:

```
data Transform c m a b = Transform { applyT :: c -> a -> m b }
```

For HERMIT, `Transform` is specialized by the following type synonym:

```
type TransformH a b = Transform HermitC HermitM a b
```

The `HermitM` monad is essentially GHC's core monad, `CoreM`, augmented with error handling as provided by KURE. There are additional effects structured by `HermitM`, but they are not utilized by this work. The context for HERMIT transformations, `HermitC`, tracks all of the bindings in a module from the top-level down. The context also tracks the location inside of a module that a HERMIT computation is manipulating or accessing. These locations are stored with the digital equivalent of a trail of breadcrumbs that leads back to the root of the module. These paths of crumbs are an abstraction, again provided by KURE, that can also be used to dictate how to traverse an expression for transformation or rewriting.

HERMIT provides a dictionary of navigations that can compute a path to a target location in a module. The `bindingOfT` function, whose signature is shown below, returns the path to the first binding it encounters whose variable satisfies a given predicate.

```
type LocalPathH = LocalPath Crumb

bindingOfT :: (Var -> Bool) -> TransformH CoreTC LocalPathH
```

Note that `bindingOfT` is itself a transformation that converts the constructors it encounters to their corresponding `Crumbs`. Given that a path could traverse any type in `ModGuts` from `CoreProgram` all the way down to `CoreExpr`, HERMIT defines sum types that bundle entire data type hierarchies together. These sum types, including the `CoreTC` type used by `bindingOfT`, are shown in Listing 5.5.

Listing 5.5: HERMIT's Sum Data Type Hierarchy

```
data Core = GutsCore  ModGuts
          | ProgCore  CoreProg
          | BindCore  CoreBind
          | DefCore   CoreDef
          | ExprCore  CoreExpr
          | AltCore   CoreAlt

data TyCo = TypeCore Type
          | CoercionCore Coercion

data CoreTC = Core Core
            | TyCo TyCo
```

As was mentioned previously, HERMIT was originally developed as an interactive tool. For this reason, its plugin architecture is based upon yet another monad, `HPM`, whose primary purpose is to structure user interaction:

```
hermitPlugin :: ([CommandLineOption] -> HPM ())-> Plugin
```

HERMIT provides a number of combinators to lift transformations into this monad; two of which, `query` and `at`, are worth examining in more detail:

```
query :: (Injection ModGuts g, Walker HermitC g)
      => TransformH g a -> HPM a


at :: TransformH CoreTC LocalPathH -> HPM a -> HPM a
```

The `query` combinator converts a transformation to an `HPM` computation provided it satisfies two requirements. First, the data type to be transformed must be defined in injection with the `ModGuts` type. Second, this data type must be traversable by KURE combinators when guided by a `HermitC` context. All of the previously shown sum types satisfy these requirements.

The `at` combinator applies a given `HPM` computation at a specified location. This location is provided to `at` as a transformation that computes the correct `LocalPathH` value. Given that the `CoreTC` type is the most inclusive of the sum types, it is an obvious target for this path transformation, hence its involvement with the previously shown `bindingOfT` transformation When paired with the `cmpString2Var` function, `bindingOfT` makes for a clean and concise way to target definitions of a given name. For example, the following computation would apply a transformation to the binding that carries the intermediate representation of `Identity`'s `Monad` instance:

```
at (bindingOfT . cmpString2Var $ "$fMonadIdentity")(query trans)
```

The types of the at and query combinators constrain the possible types of the example transformation function, trans. Refining the statement from earlier in this section, this translation is not of type CoreExpr -> HOLTerm, but rather TransformH CoreTC HOLSum, where HOLSum is the sum of HaskHOL's primitive data types that corresponds to CoreTC. Given that the initial evaluations of this workflow are expression-level, not whole program, verifications, no data types above CoreExpr in HERMIT's hierarchy will be utilized. Thus, provided that a transformation of type TransformH CoreExp HOLTerm can be defined, KURE's Injection class and HERMIT's promotion combinators can be used to bridge the gap from CoreExpr to CoreTC. The general technique for constructing such a transformation in HERMIT is demonstraed by focusing on a "dummy" translation of variables.

HERMIT's combinator for transforming CoreExpr variables is shown at the top of Listing 5.6. A Var is essentially a wrapper for typed identifiers, capable of representing terms and types both. The varT combinator lifts a transformation of identifiers into a transformation of expressions that consist of a single variable occurrence. It unboxes a Var value, applies the sub-transformation to its internal value, and extends the working context with a Crumb to indicate that the transformation has descended into a variable. The dummy transformation takes the name of an Id and uses it to create a new, variable HOL term of an arbitrary type.

Listing 5.6: Transforming Vars in HERMIT

```
varT :: TransformH Id b -> TransformH CoreExpr b
varT t = transform $ \ c -> \case
    Var v -> applyT t (c @@ Var_Id) v
    _      -> fail "not a variable."

dummy :: TransformH Id HOLSum
dummy = contextfreeT $ \ id -> Tm $
    Tm $ mkVarType (name v) tyA
  where name = pack . unqualified . varName

trans :: TransformH CoreExpr HOLSum
trans =
       varT dummy <+
    <+ appT trans trans (\ (Tm x) (Tm y) -> Tm (mkComb x y))
    <+ ...
```

When `varT` is applied to `dummy`, a transformation of the desired type is produced. Note that the `varT` combinator will fail if presented with a `CoreExpr` value not constructed with `Var`. The cases for other constructors are handled by writing transformations for them and then threading everything together with combinators that handle the exceptions. The example in Listing 5.6 uses the `(<+)` combinator, the KURE equivalent of `<|>` from the `Alternative` type class. The formal semantics for the complete translation function will be shown in the next section and the implementation of this translation will be discussed in Chapter 6.

## 5.3 A Formalized Translation

The intermediate language of GHC is an implementation of System $F_C^{\uparrow}$ [95], an extension of System $F_C$ [86] that supports data type promotion. System $F_C$ is itself is an extension of the the well known System F [74] that supports type-level equalities through coercion. Following from previous discussions in Chapters 3 and 4, System F, and all of its extensions, are necessarily more expressive than HaskHOL's logic due to its consistent nature. Thus, HaskHOL can only represent a subset of GHC's core language.

The following list contains the assumptions made by the translation semantics presented in this section. Collectively they define the current limitations of HaskHOL's ability to reason about Haskell programs, some of which were previously enumerated in Chapter 4:

1. Types are simply kinded, i.e. $\star$ or $k_1 \rightarrow k_2$, and/or are otherwise safe to ignore.
2. Reducing a type application results in a substitution or instantiation that obeys the restrictions of HaskHOL's polymorphism.
3. Casts and other coercions are discharged or otherwise removed before translation, e.g. `newtype` definitions are replaced with equivalent `data` versions.
4. Binding groups can be reduced to a list of possibly self recursive, but not mutually recursive, expressions.
5. All of the bindings in a group reside within a single module.
6. Constants of the language, primitives and user-defined alike, all map to analogous constants in an existing HaskHOL theory.

Assumptions 1 and 2 are due to the previously covered limitations of HaskHOL's type system. Though there are HOL systems that can reason about kinds [43], HaskHOL currently lacks that capability. As for the restrictions of HaskHOL's polymorphism, they are in place to maintain consistency of its logic; something that is critical for a proof system but not necessarily important for a type system.

Assumption 3 is primarily due to the immaturity of the translation semantics. In practice, the coercion of `newtype` values was the only type casting encountered by the evaluations to be discussed in Chapter 6. At the time of this writing, it seems an acceptable compromise to manually convert these definitions to `data` values and leave the translation of coercions as possible future work. However, it should be noted that as more and more computation is shifted to the type level in Haskell, coercions may become more prevalent in the intermediate code that HaskHOL targets.

Assumption 4 is again due to a limitation of HaskHOL. Following from the logic of HOL Light that it is based on, HaskHOL permits mutually recursive data types, but not mutually recursive function definitions. Again, there are HOL systems that do have this capability [56], but HaskHOL is not currently one of them.

Assumptions 5 and 6 are due to limitations and design choices of HERMIT and the GHC plugin architecture. The only `ModGuts` value available to a plugin is the value for the module currently being compiled. While there is likely a way to persist the necessary information while compiling a sequence of modules, this technique would be infeasible for any body of code that depends on a library whose source is not immediately available for recompilation; programs depending on GHC's base library would be the most obvious example. Additionally, HERMIT does not currently provide combinators to transform the `TyCon` values that store type definitions, so translating constants would be supported for terms only without a sizable amount of additional work. Again, at the time of this writing, it seems an acceptable compromise to depend on HaskHOL's constant definitions for now, leaving the exploration of alternative solutions as possible future work.

```
data Type                          data CoreExpr
  = TyVarTy  Id                      = Var  Id
  | AppTy    Type  Type              | App  CoreExpr CoreExpr
  | TyConApp TyCon [Type]            | Lam  Id       CoreExpr
  | FunTy    Type  Type              | Type Type
  | ForAllTy Id    Type
```

Figure 5.3: GHC's Core – Simplified

**From GHC Core to HaskHOL**

A simplified view of GHC's core data types is shown in Figure 5.3. The `Cast`, `Lit`, and `LitTy` constructors have been removed following from the previously enumerated list of assumptions. The `Tick` constructor has also been removed, as its primary purpose is annotating information for profiling and debugging purposes, such that it is not relevant to this translation. Finally, the `Case` and `Let` constructors have been removed as both map quite directly to meta-constructs in HaskHOL's term language. This makes their translations both awkward to formalize and comparatively uninteresting.

A corresponding, simplified view of the primitive data types of HaskHOL is shown in Figure 5.4. Constructors and constructor fields that are not critical to our translation have been converted to simpler types or removed entirely. This has mainly resulted in the removal of types that facilitated HaskHOL's semi-stateless features, e.g. `ConstTag`, leaving behind a term language very similar to that of HOL2P. Additionally, we have defined the `HOLSum` sum type that mirrors HERMIT's `CoreTC` type, minus coercions.

Figure 5.4: HaskHOL's Primitive Types – Simplified

```
data HOLSum
    = Tm HOLTerm | Ty HOLType | TyOp TypeOp

data TypeOp                        data HOLTerm
    = TyOpVar   String                 = Var    String  HOLType
    | TyPrim    String Int             | Const  String  HOLType
data HOLType                           | Comb   HOLTerm HOLTerm
    = TyVar String                     | Abs    HOLTerm HOLTerm
    | TyApp TypeOp  [HOLType]          | TyComb HOLTerm HOLType
    | UType HOLType HOLType            | TyAbs  HOLType HOLTerm
```

`Id` to `HOLSum`

$$\frac{varType\ id \to ty \qquad x = \left\{ \begin{array}{ll} \text{if } (isId\ id): & Tm\ (Var\ (varName\ id)\ ty) \\ \text{otherwise}: & Ty\ (TyVar\ (varName\ id)) \end{array} \right\}}{id \to x}$$

`TyCon` to `TypeOp`

$$\frac{x = \left\{ \begin{array}{ll} \text{if } (isFunTyCon\ con): & TyPrim\ \text{“}fun\text{”}\ 2 \\ \text{otherwise}: & TyPrim\ (tyConName\ con)\ (tyConArity\ con) \end{array} \right\}}{con \to x}$$

Figure 5.5: Translating `Id`s and `TyCon`s

Figure 5.5 defines the predicate-based translations for the `Id` and `TyCon` types. Term variables translate directly between language and logic. As was mentioned in the introductory assumptions, when translating type variables their kinds are discarded such that only their names are kept. When translating type constructors, everything is mapped to primitive type operators in HaskHOL, making sure to handle the special case of the function type constructor. Primitive type operators are used rather than variable operators so that the translation can maintain and track the arity of types.

Figures 5.6 and 5.7 define the translations for the constructors of the `Type` data type. Translations not related to type application proceed directly.

Figure 5.6: Translating `Type`s, Part I

`TyVarTy` to `HOLSum`

$$\frac{id \to id'}{TyVarTy\ id \to id'}$$

`ForAllTy` to `HOLSum`

$$\frac{id \to Ty\ id' \qquad ty \to Ty\ ty'}{ForallTy\ id\ ty \to Ty\ (UType\ id'\ ty')}$$

`FunTy` to `HOLSum`

$$\frac{ty1 \to Ty\ ty1' \qquad ty2 \to Ty\ ty2'}{FunTy\ ty1\ ty2 \to Ty\ (TyApp\ (TyPrim\ \text{“}fun\text{”}\ 2)\ [ty1', ty2'])}$$

`AppTy` to `HOLSum`

$$\frac{id \rightarrow Ty\ (TyVar\ x) \qquad ty \rightarrow Ty\ ty'}{AppTy\ (TyVarTy\ id)\ ty \rightarrow Ty\ (TyApp\ (TyOpVar\ x)\ [ty'])}$$

$$\frac{ty1 \rightarrow Ty\ (TyApp\ op\ tys) \qquad ty2 \rightarrow Ty\ ty2'}{AppTy\ ty1\ ty2 \rightarrow Ty\ (TyApp\ op\ (tys +\!\!+\ [ty2']))}$$

`TyConApp` to `HOLSum`

$$\frac{op \rightarrow TyOp\ op' \qquad x = \left\{ \begin{array}{ll} \text{if } (arity\ op' = 0): & Ty\ (TyApp\ op'\ []) \\ \text{otherwise}: & TyOp\ op' \end{array} \right\}}{TyConApp\ op\ [] \rightarrow x}$$

$$\frac{op \rightarrow TyOp\ op' \qquad for\ each\ i \in n,\ ty_i \rightarrow Ty\ ty'_i}{TyConApp\ op\ [ty_1, ..., ty_n] \rightarrow Ty\ (TyApp\ op'\ [ty'_1, ..., ty'_n])}$$

Figure 5.7: Translating `Type`s, Part II

Non-constructor type application translations depend on whether the operator of the application is a type variable or another application. In the case of a type variable, it is converted to a type operator variable and a HOL type application is built. A variable type operator is used rather than a primitive type operator for two reasons. First, the correct arity to give to the operator is not known due to the erasure of kinds. Second, it makes type substitutions performed when translating `CoreExpr` values slightly easier.

Constructor type application translations depend on the length of the argument type list. When passing a constructor as an argument to a type application, GHC will pair it with an empty type argument list in a `TyConApp` to satisfy the type of the `CoreExpr` `Type` constructor. HaskHOL, however, does not permit partial applications of type operators, so the translation must check to see if a type operator is actually nullary or not before handling such applications. If the operator is nullary then the appropriate HOL type application is constructed, otherwise just the type operator itself is returned. Note that this requires adding an additional translation case for `App` constructs to handle applying type operators.

`Var` to `HOLSum`

$$\frac{id \rightarrow id'}{Var\ id \rightarrow id'}$$

`Type` to `HOLSum`

$$\frac{ty \rightarrow ty'}{Type\ ty \rightarrow ty'}$$

`Lam` to `HOLSum`

$$\frac{id \rightarrow id' \qquad tm \rightarrow Tm\ tm' \qquad x = \left\{ \begin{array}{ll} \text{if } (id' = Ty\ ty): & TyAbs\ ty\ tm' \\ \text{if } (id' = Tm\ id'') \wedge \neg(isDict\ id'')): & \\ \quad Abs\ id''\ tm' & \\ \text{otherwise}: & tm' \end{array} \right\}}{Lam\ id\ tm \rightarrow Tm\ x}$$

Figure 5.8: Translating `CoreExprs`, Part I

Figures 5.8 and 5.9 define the translations for the constructors of the `CoreExpr` data type. The rules for translating `Var` and `Type` are trivial. The rules for translating `App` and `Lam` values, however, are fairly complex. Given that the translation maps all constants to their equivalent values in HaskHOL, type classes and other dictionary values do not themselves need to be translated. Each of these constructors, therefore, has a case that essentially erases dictionary arguments or parameters accordingly, adjusting types as needed.

Figure 5.9: Translating `CoreExprs`, Part II

`App` to `HOLSum`

$$\frac{id \rightarrow Tm\ (Var\ x\ (UType\ ty_1\ ty_2)) \qquad ty \rightarrow ty'}{App\ (Var\ id)\ (Type\ ty) \rightarrow Tm\ (Var\ x\ [ty_1 \mapsto ty']\ ty_2))}$$

$$\frac{f \rightarrow Tm\ f' \qquad ty \rightarrow Ty\ ty'}{App\ f\ (Type\ ty) \rightarrow Tm\ (TyComb\ f'\ ty')}$$

$$\frac{id_1 \rightarrow Tm\ id_1' \qquad id_2 \rightarrow Tm\ id_2' \qquad x = \left\{ \begin{array}{ll} \text{if } \neg(isDict\ id_2'): & Comb\ id_1'\ id_2' \\ \text{otherwise}: & Var\ name\ ty_2 \\ \quad where\ id_2'@(name : ty_1 \rightarrow ty_2) & \end{array} \right\}}{App\ (Var\ id_1)\ (Var\ id_2) \rightarrow Tm\ x}$$

$$\frac{f \rightarrow Tm\ f' \qquad a \rightarrow Tm\ a'}{App\ f\ a \rightarrow Tm\ (Comb\ f'\ a')}$$

There are also additional rules for the `App` constructor to force the evaluation of type applications during translation where possible. This is done primarily because GHC's core language permits passing type constructors as arguments to term-level, type applications:

```
(x : forall _m. forall a b. _m a b)(->)
```

In the above term, the bound variable `_m` can be instantiated by any type constructor of kind $* \rightarrow *$, such that the example can be reduced to the following:

```
x : forall a b. a -> b
```

The HaskHOL equivalent is malformed for a number of reasons, though:

```
(x : % _m. % 'a 'b. (a, b)_m)[: (->)]
```

In addition to not being able to bind type operator variables, HaskHOL does not permit partial type applications. Thus, attempting to supply the type `(->)` as an argument for type application is impossible. The reduced term, though, can be represented without issue, hence why the application is forced at translation time:

```
x : % 'a 'b. 'a -> 'b
```

While not necessary, the application of non-operator types are also forced. In most cases this produces terms that more closely match HaskHOL constants, as the majority were defined with the intention of matching their HOL Light definitions; a system that does not support term-level type applications.

# 6 Evaluation

The goal of this chapter is to piece together the concepts from the preceding chapters to evaluate the feasibility and applicability of the verification workflow described in Chapter 2. This evaluation is performed using a HERMIT plugin that implements the translation semantics discussed in Chapter 5. This plugin was applied to examples from each of the motivating problem classes, verification of type class laws and formal "re-verification" of existing test cases, with both quantitative and qualitative results being gathered.

The sections of this chapter are organized as follows:

- Section 6.1 covers the low-level implementation details of the verification plugin itself. Additionally, a general verification procedure that utilizes this plugin is prescribed.
- Section 6.2 uses this procedure to verify the example cases, displaying the results. Following from the presentation of Section 6.1, multiple, intermediate representations of each verification target is shown. Additionally, each verification is timed and any outlying behavior is noted.
- Finally, Section 6.3 discusses the current state of the work, as framed by the results in Section 6.2. The previously enumerated limitations of the system will be reiterated and my thoughts on their possible solutions will be noted as potential future work.

## 6.1 A HERMIT Plugin for Verification

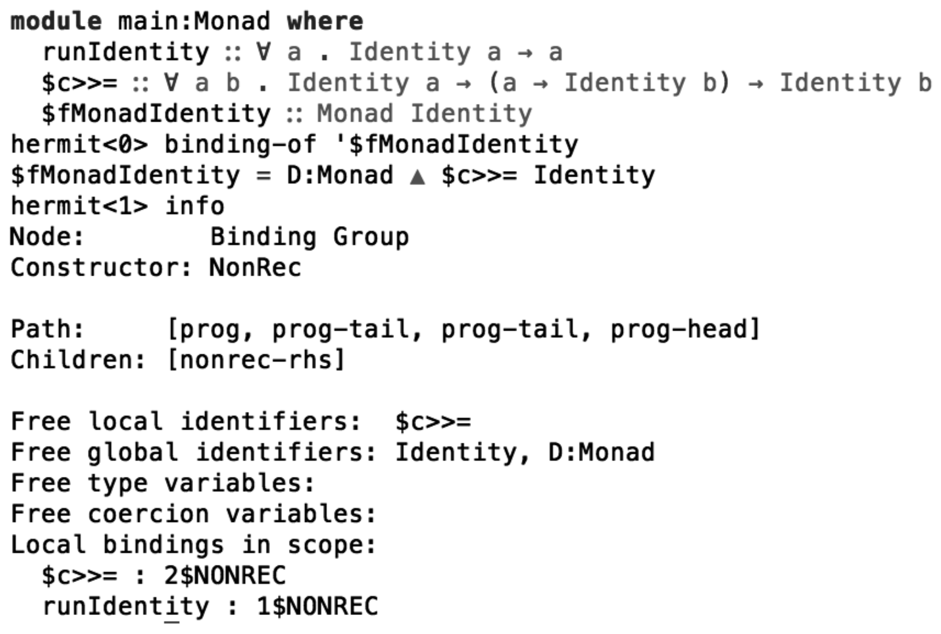
This section will detail the implementation of a HERMIT-based plugin for verification. To aid discussion, this implementation will be derived by following a verification of the monad laws for the `Identity` data type; the example that his driven much of the content of this dissertation. Recall that, due to current limitations of the integration with GHC, all relevant definitions must be contained within a single module. The first step to a verification, therefore, is to construct a satisfying module. For this specific case, a module must contain definitions of the `Identity` type, the `Monad` class, and their intersection. A minimal construction of this module is shown in Listing 6.1.

The remainder of the verification is carried out by the following procedure:

1. GHC's intermediate representation of the target type class instance is translated.
2. The translated term is deconstructed into the definitions of the corresponding class methods.
3. Any bindings within these definitions that are locally available are expanded with their own translations.
4. The final versions of these definitions are recombined, replacing the dictionary constructor with the `Monad` constant as the head term of the application.
5. The resultant term is proved correct.

Listing 6.1: The `Monad` Module

```
{-# LANGUAGE ExplicitForAll #-}
module Monad where

import Prelude hiding (Monad, return, (>>=))

data Identity a = Identity a

runIdentity :: Identity a -> a
runIdentity (Identity a) = a

class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a

instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```



```
module main:Monad where
  runIdentity :: ∀ a . Identity a → a
  $c>>= :: ∀ a b . Identity a → (a → Identity b) → Identity b
  $fMonadIdentity :: Monad Identity
hermit<0> binding-of '$fMonadIdentity
$fMonadIdentity = D:Monad ▲ $c>>= Identity
hermit<1> info
Node:        Binding Group
Constructor: NonRec

Path:     [prog, prog-tail, prog-tail, prog-head]
Children: [nonrec-rhs]

Free local identifiers:  $c>>=
Free global identifiers: Identity, D:Monad
Free type variables:
Free coercion variables:
Local bindings in scope:
  $c>>= : 2$NONREC
  runIdentity : 1$NONREC
```

Figure 6.1: `hermit`'s View of the `Monad.hs` Module

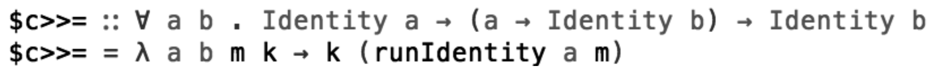
The `hermit` executable provided by the HERMIT library is an invaluable tool that can help at each step of this process. This program compiles a module, launches HERMIT's interactive shell, and populates a context with information pulled from the module's `ModGuts`. An example execution of `hermit` on the `Monad.hs` module is shown in Figure 6.1. Using this tool, `$fMonadIdentity` is identified as the name of the binding that corresponds to the target type class instance. The names of other type class bindings will follow this general pattern of $\$ < var >< ClassName >< InstanceType >$, e.g. `$fMonadMaybe`, `$fMonadEither`, etc.

A HERMIT user can expand this binding and examine its definition with the `binding-of` command. Note that this command is the interactive equivalent of the previously discussed `bindingOfT` transformation. The definition of `$fMonadIdentity` consists of the application of four terms: the dictionary constructor, the instance type[1], and the two definitions for the `Identity` monad's methods. As their names and their locations within this dictionary would imply, the `$c>>=` and `Identity` terms are the definitions for `(>>=)` and `return` accordingly.

[1]HERMIT's default pretty-printer replaces type applications with triangles for brevity's sake.

Querying for information about the binding group with the `info` command reveals that `Identity` is a globally free identifier. This, paired with the fact that its name has a leading capital letter, is a good indication that it is a constructor for a data type. Unfortunately, due to the previously mentioned limitations of HERMIT relating to type constructors, no other information about this identifier can be retrieved, and no other translation work can be done.

The `$c>>=` term, however, refers to another local binding in scope, so there is additional work to be done. After backtracking to the top-level of the module, the binding for `$c>>=` can be expanded to see its definition. Unless the current working theory already has a constant that maps to `Identity`'s instance for `(>>=)`, this expression must be translated in order to produce a verifiable proof obligation:



```
$c>>= :: ∀ a b . Identity a → (a → Identity b) → Identity b
$c>>= = λ a b m k → k (runIdentity a m)
```

Note that the translation of the above expression will match with the `bind` definition supplied as part of the proof obligation in Listing 5.1 from Chapter 5; a good indication that the verification is on the right path. In fact, translation produces an identical obligation, at least visually. Careful readers probably noted that the translation semantics presented in the previous chapter never return constant terms. It is up to the users of the translated terms to substitute in constants for variables where appropriate. This substitution is relatively easy to mechanize thanks to the combinators and methods provided by HERMIT and HaskHOL. One possible implementation for variable substitution is shown in Listing 6.2.

Listing 6.2: Substituting Constants for Variables

```
repVar :: [(String, HOLTerm)] -> String -> HOLType
       -> HOL Proof thry HOLTerm
repVar tmmap i ty = (tryFind (\ (key, Const name ty') ->
    if key == i && isJust (typeMatch ty' ty ([], [], []))
    then mkMConst name ty
    else fail "") tmmap) <|> (return $! mkVar i ty)
```

The `repVar` function works by traversing an association list of `String` and constant `HOLTerm` pairs, looking for all terms indexed by a given variable name. These constants are provided in the same form they are stored in the theory context, with their most general type. Thus, lookups also compute a type match in addition to checking equality with the indexing names. This allows variable substitution to accept multiple possible replacements for a given name, supporting the previously mentioned need to have versions of constants with, and without, quantified types.

Given that all of the constant names in this example mirror their HaskHOL mappings, the term map provided to `repVar` can simply be the list of constants tracked in the theory context, as returned by the `constants` function. However, to account for cases when the names might not line up, this plugin also accepts an auxiliary, user-provided term map. Assuming this mapping is stored as an external configuration file, the following functions can be used to parse it and join it with the relevant constants from the current working theory context:

```
-- Prepare the constant list
prepConsts :: String -> HPM (Map Text b) -> HPM [(Text, b)]
prepConsts file mcnsts =
    do cmap <- liftIO $ parse file
       cnsts <- mcnsts
       let cnsts' = catMaybes $ map (\ (x, y) ->
                        do y' <- mapLookup y cnsts
                           return (x, y')) cmap
       return (cnsts' ++ mapToList cnsts)

-- Lift HOL computations into the HPM monad
liftHOL :: HOL Proof HaskellType a -> HPM a
liftHOL m = liftIO $ runHOLProof True m ctxtHaskell

-- Parse the term constants from a file
prepConsts "terms.h2h" $ liftHOL constants
```

If a matching constant skeleton is found by the search in `repVar`, the `mkMConst` method is used to construct an instantiation of the constant that matches the provided type. If no match is found, a variable of the provided name and type is returned instead. This behavior fits nicely as a reconstruction function in a KURE transformation of `HOLTerm` values:

```
-- The type of the transformation
type TransHOL thry a b = Transform (Context thry) IO a b

-- Transformation to replace variables using 'repVar'
replaceTerm :: [(String, HOLTerm)] -> TransHOL thry HOLTerm HOLTerm
replaceTerm tmmap = repTerm
  where repTerm :: TransHOL thry HOLTerm HOLTerm
        repTerm =
                contextfreeT (\ tm@Const{} -> return tm)
            <+ hvarT (contextfreeT return) (contextfreeT return) repVar
            <+ ...
```

Structuring `replaceTerm` with the `HOL` monad and providing it with a HaskHOL theory context allows for the safe and sound use of `HOL` computations, e.g. `mkMConst`. The definition of other transformations, such as the replacement of type operator variables with type constants, follows similarly. This entirety of the Haskell-to-HOL transformation process requires three major passes:

1. The initial translation from GHC's intermediate representation to a HaskHOL term.
2. The replacement of type variables with matching type constants.
3. The replacement of term variables with matching term constants.

Listing 6.3: Translation and Replacement Passes

```
trans :: TransformH CoreTC HOLTerm

pass :: HOLTerm -> HPM HOLTerm
pass ctxt tm = liftIO $
    do tm' <- applyT (replaceType tyMap) ctxtHaskell tm
       applyT (replaceTerm tmMap) ctxtHaskell tm'

consClassPass :: HOLTerm -> HPM HOLTerm
consClassPass tm =
    do liftIO $ putStrLn "Translating Arguments..."
       let (_, args) = stripComb tm
       args' <- mapM trans' args
       liftIO $ putStrLn "Building Class Instance..."
       monad <- liftHOL $ mkConstFull "MONAD" ([],[],[])
       maybe (fail "Reconstruction of class instance failed.")
         return $ foldlM mkIComb monad args'

  where trans' :: HOLTerm -> HPM HOLTerm
        trans' x@(Var name _) = pass =<< query
            (do lp <- lookupBind $ unpack name
                case lp of
                  Nothing -> return x
                  Just res -> localPathT res trans)
        trans' x = pass x

        lookupBind :: String
                   -> TransformH CoreTC (Maybe LocalPathH)
        lookupBind x = catchesT
            [ liftM Just . bindingOfT $ cmpString2Var x
            , return Nothing
            ]
```

Listing 6.3 shows how these passes are composed to transform a constructor class instance. In this listing, the first signature, `trans`, is the transformation from `CoreTC` to `HOLTerm` defined by the translation semantics. The second definition, `pass`, sequences the replacement of type constants and term constants. The ordering of these replacements is critical given that the replacement of term constants depends on type matches being correctly computed. The third definition, `consClassPass`, combines the two previous definitions in a way specialized for constructor classes. This function destructs a type class instance, translates its constituent methods, and reconstructs it using the appropriate constant term.

The translation of the class methods is controlled by the `trans'` function which builds in more finely grained exception handling than HERMIT's base set of combinators provides. If a binding can be found for a method's name then an additional translation is performed, otherwise only the replacement pass is performed. This allows the `trans'` function to handle locally bound (`$c>>=`) and globally free (`Identity`) methods both, providing some uniformity to the definition of the plugin. Note that additional translations are only performed one level deep. In this example, `runIdentity` will not replaced with its definition as it is assumed that it has a matching constant in the HaskHOL theory that provides the definition of the `Identity` type.

The entirety of the evaluation plugin's implementation, including the definition for the `trans` function, is included in Addendum A as a reference. The major steps it follows are concisely explained below, followed by an example execution of the plugin.

First, the user-provided type and term maps are parsed and prepared:

```
tyMap <- prepConsts "types.h2h" $ liftHOL types
tmMap <- prepConsts "terms.h2h" $ liftHOL constants
```

Next, GHC's intermediate representation of the verification target is translated:

```
tm <- at (lookupBind "$fMonadIdentity") $ query trans
```

The resultant term is transformed by the secondary translation and replacement pass:

```
tm' <- consClassPass tm pass
```

Finally, the verification is completed using a tactic specified by the user:

```
tacMonadIdentity :: HaskellCtxt thry => Tactic cls thry
tacMonadIdentity = proveConsClass defMonad
    inductionIdentity
     [defIdentity, defRunIdentity]


liftHOL $ printHOL =<< prove tm' tac
```

HaskHOL provides a method for safe, run-time interpretation of string values as `HOL` computations. This allows a user to specify the required proof tactic in the same way they would declare the verification target and other configuration options. The plugin used for the evaluations in the next section parses these options from a configuration file contained in the same directory as the target module. An example configuration file tailored to this specific verification is shown in Listing 6.4.

Listing 6.4: An Example Verification Configuration File

```
binding: $fMonadIdentity
bindingType: ConsClass
class: Monad
proofType: Tactic
proofModule: HaskHOL.Lib.Haskell
proofName: tacMonadIdentity
```

As the plugin performs a verification, a user's terminal updates them as each step in the process is completed:

```
> ghc --make Monad.hs -fforce-recomp -O2
    -fplugin=HaskHOL.Haskell.Plugin
...
> Parsing constant mappings...
> Translating from Core to HOL...
> Translating Arguments...
> Building Class Instance...
> Proving...
> |- MONAD (\\'a 'b.(\ m k . k (runIdentity m)))
        Identity
```

If the verification is successful, a theorem is displayed, demonstrating the correctness of the generated proof obligation. If the verification fails, GHC throws an exception, just as it would for any other compilation error.

## 6.2 Verification Results

This section contains the results of example verifications drawn from the two motivating problem classes introduced in Chapter 2. For each example, the following information is shown:

- The Haskell module containing the relevant source-level definitions.
- The associated GHC core-level definitions, as produced by the `hermit` tool.
- The configuration file required by the verification plugin.
- The HaskHOL proof obligation generated by the verification plugin.
- An average total verification time and average isolated proof time.

All timing metrics are recorded using an Apple Macbook Pro laptop with the following configuration:

| | |
|---|---|
| Generation | Retina, Mid 2012 |
| Processor | 2.6 GHz Intel Core i7 |
| Memory | 8 GB 1600 MHz DDR3 |
| Operating System | OS X 10.10.2 |

The average total verification time is the *real* time reported by the following command, averaged over ten executions:

```
time ghc --make Monad.hs -fforce-recomp -O2 -fplugin=HaskHOL.Haskell.Plugin
```

The average isolated proof time is the time reported by `ghci` using the `:seti+s` command. The proof effort is isolated by timing the evaluation of the `prove` method applied to a pre-constructed term and tactic matching those used by the plugin, again averaged over ten executions.

It should be noted that using two different mechanisms for timing makes the results not directly comparable. The average isolated proof time is intended to be used only as an approximation of the amount of total verification time inherited from the proof effort.

- `Identity Functor`
  - Module:

```
module Functor where

import Prelude hiding (Functor, fmap)

data Identity a = Identity a

runIdentity :: Identity a -> a
runIdentity (Identity a) = a

class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor Identity where
    fmap f id = Identity (f (runIdentity id))
```

  - Core:

```
Node:        Module
Constructor: ModGuts


Free global identifiers: Identity, D:Functor
Local bindings in scope:
  $cfmap : 0$TOPLEVEL
  $fFunctorIdentity : 0$TOPLEVEL
  runIdentity : 0$TOPLEVEL

$cfmap = λ △ △ f id → Identity △ (f (runIdentity △ id))

$fFunctorIdentity = D:Functor △ $cfmap
```

  - Configuration:

```
binding: $fFunctorIdentity
bindingType: ConsClass
class: Functor
proofType: Tactic
proofModule: HaskHOL.Lib.Haskell
proofName: tacFunctorIdentity
# Prelude Term Mappings
("Identity", "IdentityIn")
```

  - Proof Obligation:

```
Functor (\\'a 'b. (\ f id . IdentityIn (f (runIdentity id))))
```

  - Performance:

| | |
|---|---|
| Average Total Verification Time | 18.23s |
| Average Proof Time | 8.90s |

- `Identity Monad`
  - Module:

```
{-# LANGUAGE ExplicitForAll #-}
module Monad where

import Prelude hiding (Monad, return, (>>=))

data Identity a = Identity a

runIdentity :: Identity a -> a
runIdentity (Identity a) = a

class Monad m where
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b
    return      :: a -> m a

instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

  - Core:

```
Node:        Module
Constructor: ModGuts

Free global identifiers: Identity, D:Monad
Local bindings in scope:
  $c>>= : 0$TOPLEVEL
  $fMonadIdentity : 0$TOPLEVEL
  runIdentity : 0$TOPLEVEL

$c>>= =  λ △ △ m k → k (runIdentity △ m)

$fMonadIdentity = D:Monad △ $c>>= Identity
```

  - Configuration:

```
binding: $fMonadIdentity
bindingType: ConsClass
class: Monad
proofType: Tactic
proofModule: HaskHOL.Lib.Haskell
proofName: tacMonadIdentity
```

  - Proof Obligation:

```
Monad (\\'a 'b. (\ m k . k (runIdentity m)))Identity
```

  - Performance:

| | |
|---|---|
| Average Total Verification Time | 21.57s |
| Average Proof Time | 13.68s |

- `Maybe Monad`
  - Module:

```
{-# LANGUAGE ExplicitForAll #-}
module Monad where

import Prelude hiding (Maybe(..), Monad, return, (>>=))

data Maybe a = Nothing | Just a

class Monad m where
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b
    return      :: a -> m a

instance Monad Maybe where
    return a       = Just a
    Nothing  >>= k = Nothing
    (Just x) >>= k = k x
```

  - Core:

```
Node:        Module
Constructor: ModGuts

Free global identifiers: Nothing, Just, D:Monad
Local bindings in scope:
  $c>>= : 0$TOPLEVEL
  $fMonadMaybe : 0$TOPLEVEL

$c>>= = λ △ △ ds k →
  case ds of wild △
    Nothing → Nothing △
    Just x → k x

$fMonadMaybe = D:Monad △ $c>>= Just
```

  - Configuration:

```
binding: $fMonadMaybe
bindingType: ConsClass
class: Monad
proofType: Tactic
proofModule: HaskHOL.Lib.Haskell
proofName: tacMonadMaybe
# Prelude Term Mappings
("Just", "JustIn")
```

  - Proof Obligation:

```
Monad (\\'a 'b. (\ ds k .
  match ds with Nothing -> Nothing | JustIn x -> k x)) Just
```

  - Performance:

| | |
|---|---|
| Average Total Verification Time | 2m41.20s |
| Average Proof Time | 1m45.62s |

## A Case Study in Formalizing Test Suites

- `List Test`
  - Module:

```
{-# LANGUAGE TemplateHaskell #-}
module List where

import Test.QuickCheck

myrev :: [a] -> [a]
myrev [] = []
myrev (x:xs) = myrev xs `myapp` [x]

myapp :: [a] -> [a] -> [a]
myapp [] ys = ys
myapp (x:xs) ys = x : myapp xs ys

class MyEq a where
  myeq :: a -> a -> Bool

instance MyEq a => MyEq [a] where
  myeq [] [] = True
  myeq (x:xs) (y:ys)
      | x `myeq` y = myeq xs ys
      | otherwise = False
  myeq _ _ = False

instance MyEq Bool where
  myeq True True = True
  myeq False False = True
  myeq _ _ = False

prop :: MyEq a => [a] -> [a] -> Bool
prop xs ys = (myrev (xs `myapp` ys)) `myeq` (myrev ys `myapp` myrev xs)

return []
main = $quickCheckAll
```

  - Core:

```
Node:        Module
Constructor: ModGuts

Free global identifiers: void#, :, False, [], True, Eq#, (,), (,,),
    tacMATCH_ACCEPT, myeq, quickCheckResult, thmREVERSE_APPEND, D:MyEq,
    $fHOLThmRepHOLclsthry, runQuickCheckAll, $fTestableProperty
Local bindings in scope:
  $cmyeq : O$TOPLEVEL
  $cmyeq : O$TOPLEVEL
  $fMyEq[] : O$TOPLEVEL
  $fMyEqBool : O$TOPLEVEL
  prop : O$TOPLEVEL
  proof : O$TOPLEVEL
  myrev : O$TOPLEVEL
  myapp : O$TOPLEVEL
  main : O$TOPLEVEL

prop = λ △ $dMyEq →
  let $dMyEq = $fMyEq[] △ $dMyEq
  in λ xs ys →
       myeq △ $dMyEq (myrev △ (myapp △ xs ys))
            (myapp △ (myrev △ ys) (myrev △ xs))
```

- Configuration:

```
binding: prop
bindingType: Test
proofType: HOLThm
proofModule: HaskHOL.Lib.Lists
proofName: thmREVERSE_APPEND2
# Prelude Type Mappings
("Bool", "bool")
("[]", "list")
# Prelude Term Mappings
("myrev", "REVERSE")
("myapp", "APPEND")
("myeq", "=")
```

- Proof Obligation:

```
!!'a. !(xs: 'a list) ys.
  REVERSE (APPEND xs ys) = APPEND (REVERSE ys) (REVERSE xs)
```

- Performance:

| | |
|---|---|
| Average Total Verification Time | 20.47s |
| Average Proof Time | 2.55s |

## 6.3 Observations and Future Work

The example verifications contained in Section 6.2 are ordered based on their respective complexities. The first, and simplest, example is a verification of the functor laws for the `Identity` type. Given that this data type was previously covered, but functors were not, HaskHOL's definition of the `Functor` constant is shown below to aid discussion:

```
Functor (fmap : % 'A 'B. ('A -> 'B) -> 'A _F -> 'B _F)
  = ((!! 'A. fmap (I:'A -> 'A) = I) /\
     (!! 'A 'B 'C. ! (f:'B -> 'C) (g:'A -> 'B).
         fmap (f o g) = fmap f o fmap g))
```

Just like the `Monad` class, the definition of the `Functor` class is borrowed from category theory. A data type is a functor if it has a structure preserving traversal, referred to as its functor map or `fmap`, that applies a given function to each element in a set of values grouped by the type. The `List` type is the most commonly cited example of a functor, as its `map` operation is `fmap` specialized for lists.

As is shown in the definition above, the `fmap` function has two laws it should obey. The first states that `fmap` is identity preserving, such that mapping the identity function, represented by the constant `I` in HaskHOL, over a functor is equivalent to applying the identity function directly. The second law states that `fmap` is distributive over function composition, represented by the infix `o` operator in HaskHOL.

As was the case with its monad instance, the `Identity` type is the simplest possible functor; it has only a single constructor that can hold only a single value. These instances are similar enough, in fact, that both verifications are performed using the same general proof tactic for constructor classes, `proveConsClass`, that was defined back in Chapter 5. When compared with the previously seen verification of the monad laws for `Identity` there is only one major difference: the internal constructor for the type is used in the definition of the class's method.

This necessitates providing an additional term mapping when performing the verification. Given the definition of the substitution function `repVar` shown in Listing 6.2 in Section 6.1, the net impact of this change is that a superfluous type matching may be performed based on the ordering of constants in the provided term map. This may present a problem for the construction of very large proof obligations, however, in this case the added computation seems to be relatively inconsequential. When comparing the verification times of the functor laws and monad laws for `Identity`, both the average total verification time and the approximated average proof time are within seconds of each other.

Serving as a contrasting example, the verification of the monad laws for the `Maybe` type takes approximately eight times as long as for the `Identity` type. The majority of this time is spent conducting the proof, however, the construction of the proof obligation also takes an alarming amount of time; almost a full minute compared to roughly ten seconds for the other constructor class examples. This obligation is notably different from the others due to its inclusion of derived term constructs to represent a pattern-matching case statement.

When these derived terms are expanded by their definitions, such that the proof obligation is expressed using only primitive constructors, the resultant term is significantly larger than those of the other examples. Thus, some amount of additional construction and proof time is to be expected, but not to the degree that was observed. Clearly, this example demonstrates a major inefficiency in the HaskHOL system that needs further investigation. Additional constructor class verifications were not attempted past this, as pattern matching is such a critical component of functional programming that I could not think of a non-trivial example that would not be affected by this issue.

The final example included in Section 6.2 is a formal "re-verification" of a test case corresponding to the property shown in Listing 2.2 from Chapter 2. Following from the technique described in that chapter, this test was prepared as a polymorphic proposition that could be monomorphised and tested using the QuickCheck library [17]. In order to avoid the previously mentioned issues with pattern matching, new versions of the reverse and append operations were defined, `myrev` and `myapp` respectively, to exactly match those provided by HaskHOL's list theory. This allows the plugin to focus on translating just the test case, `prop`.

Much like constructor class instances require a specialized pre-processing pass in order to construct a valid proof obligation, so do test cases. After erasing dictionary arguments and forcing the evaluation of type applications, as the translation semantics from Chapter 5 dictate, the `prop` test is reduced to a functional term:

```
\ xs ys. (myrev (myapp xs ys))= (myapp (myrev ys)(myrev xs))
```

The QuickCheck library tests this function by providing a randomized set of input data for `xs` and `ys`, effectively simulating an implicit universal quantification of these arguments. In order to verify this term with HaskHOL, these quantifications must be made explicit:

```
let (bvs, bod) = stripTyAbs tm
    (bvs', bod') = stripAbs bod in
  do x <- pass bod'
     bvs'' <- mapM pass bvs'
     liftHOL $ flip (foldrM mkTyAll) bvs =<<
       listMkForall bvs'' x
```

The above code performs this quantification by stripping the term of any bound variables, term and type both, and applying the translation and replacement transformation, `pass`, to the remaining body. Afterwards, the previously stripped variables are reintroduced, this time bound by universal quantifiers.

The overall term construction process for this example takes approximately twice as long as those of the constructor class verifications of the `Identity` type. However, it is not immediately clear if this additional time is due to the need to quantify bound variables or because of the increased number of term and type mappings required by the example. In either case, the additional cost should scale linearly with the number of bound variables or the number of replacements, such that it won't be a dominating factor in larger verification efforts.

Given that QuickCheck tests do not follow any general form beyond being testable functions, they can not all share a general proof tactic like constructor classes can. Proof search tactics, like `tacMESON`, could be used, but their successful application is not guaranteed. Rather, I elected to extend the capabilities of the verification plugin to allow the user to provide a pre-constructed theorem as an alternative to a proof tactic, as is done in this case. This theorem is paired with the `tacACCEPT` tactic, such that the only proof computation that needs to be performed is to check that the theorem concludes a term $\alpha$-equivalent to the obligation. This is why the average proof time for this example is so comparatively short.

### Future Work

The observations in this section make a key point painfully obvious – the success of this verification workflow is critically dependent on the formal reasoning tool it utilizes. In that regard, all of the potential paths for future work are based on a desire to improve the reasoning capabilities of the HaskHOL proof system in order to enable verification of larger and more complex bodies of Haskell code.

The most apparent, lingering issue with the HaskHOL system was demonstrated by the verification of the monad laws for the `Maybe` data type – manipulating terms of non-trivial complexity is currently a prohibitively time-consuming process. The domain of these terms does not seem to be of importance, only that they are constructed with derived constants that expand to significantly large compositions of primitive constructors.

Proofs of terms with match statements, let bindings, complex arithmetic expressions, etc. all take orders of magnitude longer to complete in HaskHOL than in algorithmically comparable systems. The good news, though, is that up to this point practically no attempt has been made to optimize HaskHOL outside of its logical kernel, so its inefficiencies are of no particular surprise. Provided with enough time, I am confident that I can significantly improve HaskHOL's performance, pushing its limitations beyond the boundary currently witnessed by the `Maybe` monad verification.

Speaking of limitations, Chapter 5 contains an enumerated list of the currently known deficiencies of the semantics that define the translation from Haskell to HaskHOL. These open problems are split roughly down the middle, with half being due to limitations of HaskHOL's foundational logic and the other half being due to the immaturity of the work. My research that has been best received up to this point was the original formulation of HaskHOL's logic, so I am looking forward to pushing its evolution and development in new ways.

Specifically, I believe that with relatively minor changes to the representation of term-level type abstractions and type combinations, the binding and application of type operators would be admissible in the logic. This would be a big step in simplifying the presented translation semantics and would allow terms to be represented more closely to their original Haskell constructions. An extension of HaskHOL's polymorphism to the kind level, like the related HOL-Omega [43] system has, would move things even further in this direction. Extending the translation semantics to handle coercions may or may not also require a modification of HaskHOL's logic. Unfortunately, I have lacked the available time to perform even a cursory examination of this problem.

The remaining limitations of the translation are consequences of the immaturity of this work. The previously mentioned efficiency problems unfortunately lie in the critical path of implementing support for general recursive functions in HaskHOL. As such, its capability to reason about recursive functions is limited to those with relatively simple inductive definitions. Similarly, there are a significant number of types and functions that HaskHOL cannot currently reason about simply because corresponding theories have not been developed for the system. It is my intention to address this problem by either extending the current Haskell theory for HaskHOL or exploring an integration with the OpenTheory tool [49]. This would allow HaskHOL to take advantage of theories that already exist for other HOL systems.

The remaining work I would like to pursue is mainly a collection of “quality of life” improvements for the verification plugin. Currently, the plugin is quite obviously designed and tuned for the specific examples included in this dissertation. Deeper translations of Haskell terms are not performed because they are not required, configuration options are limited because there exists only two problem classes and two styles of proof, and augmenting the plugin to support a new problem class, or really modifying it in any way, requires recompilation of the entire package it is included in. All of these problems should be mitigated, or entirely eliminated, by wrapping the plugin in a user interface with a more robust set of configuration options.

# 7 Related Work

This work has always been challenging to present because it sits somewhere in between the well established worlds of functional programming, interactive theorem proving, and software engineering with applied formal methods. Conference submissions focused too heavily on any one of these topics have frequently been rejected with reviews requesting more content from the others. Attempting to frame this work relative to other research endeavors has been an equally frustrating process, if only because there are so many things it is tangentially related to.

In this chapter I do my best to constrain the related work into a concise presentation split into two categories, each tied to one of the dominant chapters of the dissertation.

- Section 7.1 presents the work related to Chapter 4, focusing on the innumerable other theorem provers that I have encountered during my development of HaskHOL.
- Section 7.2 presents the work related to Chapter 5, focusing on comparable approaches to verifying functional programs that use formal proof in at least some capacity.

## 7.1 Related Theorem Provers

Before I compare HaskHOL to the many leaves on the HOL theorem prover family tree, I think it is only appropriate to extend my sincere gratitude to the developers of each and every one of those systems. Almost the entirety of my knowledge of higher-order logic and theorem proving in general was gained by studying the implementations of these systems and their supporting publications. Without the work that came before it, HaskHOL would quite literally not exist today.

At the root of this tree that HaskHOL is now a proud part of is Milner, Paulson, Gordon, et. al's work on the early LCF systems from which HOL was born [65, 34, 71]. The trunk of that tree has since split into three major branches, each well represented today by a proof system in popular use.

The main branch follows the natural evolution of Gordon's original HOL system [32]. The later systems in this lineage, HOL90 and HOL4 [81], were precipitated by a move to Standard ML as their implementation language. HOL4 is widely considered to be today's de facto standard of a HOL theorem prover, serving as the basis of experimentation for a number of extensions of the HOL logic. The previously cited HOL-Omega system is the result of one such experiementation, the extension of HOL4 with a kind system [43].

Parallel to the development of HOL4, John Harrison followed an alternative, lightweight approach to implementing a HOL theorem prover. The resultant proof system, HOL Light, is HaskHOL's primary source of inspiration [41]. Much like HOL4, HOL Light has also served as a basis for experimentation. HaskHOL's other major sources of inspiration, Freek Wiedijk's Stateless HOL [93] and Norbert Völker's HOL2P [90], are two such examples.

The third and final branch is occupied by the Isabelle proof system [69]. If HOL4 is the direct descendent of Edinburgh LCF then Isabelle, following in the spirit of Cambridge LCF, is its close sibling. Isabelle separates itself from other HOL proof systems by serving as a more general logical framework from which more complicated logics can be built upon. In addition to HOL, Isabelle's theory libraries notably support reasoning with first-order logic, set theory, and domain theory. A more thorough comparison of HaskHOL with all of these proof systems, focused on their low-level implementations, was included in Chapter 4.

Aside from HOL, there are a number of other logics that theorem provers use to frame their primitive methods of inference. One such example, the calculus of constructions (CoC) [19], was previously mentioned in Chapter 3 as the ultimate vertex of Barendregt's lambda cube [9]. The inductive extension of CoC serves as the foundational logic of the Coq theorem prover [87].

Coq's notion of universes is used to assert a hierarchy of types, allowing for a significantly more complex, but still consistent, type system than HOL provers can admit. Specifically, Coq's inclusion of dependent types is frequently touted as one of its key features. Where as HOL systems typically conduct proof through some form of equational reasoning, Coq instead encodes propositions with its type system and proves them by providing a term that inhabits that type. Per the Curry-Howard Isomorphism, Coq's method of proof is the logical correspondent of type-checking [83].

There are a number of other systems that dwell in the dependent type space. Cayenne [4] and Epigram [63] follow more traditional implementations of functional languages with dependent types added on top. However, relative new comers to the field, Agda [70] and Idris [12], are blending the line between language environment and proof environment. A number of verification efforts that utilize Agda will be mentioned in the next section.

On the fringe of being related to HaskHOL are proof systems like PVS [20], NuPRL [57], and ACL2 [51]. PVS shares an equational reasoning approach to theorem proving, however, it differs from the HOL family of provers in that it is not particularly concerned with guaranteeing the soundness of its logical kernel, i.e. it is not implemented following the traditional LCF style. NuPRL *is* an LCF-style system, however, its basis in intuitionistic type theory places it quite logically distant from HOL. ACL2 is an almost equally far distance in the opposite direction; based on a first-order logic, it is closer to hybrid systems like Agda and Idris than it is HaskHOL.

Similarly, a number Haskell-specific proof systems are cropping up that are related to HaskHOL only because of the language they target. The ZENO prover [82] attempts to automatically construct proofs of attribute properties by finding chains of equivalences between Haskell terms. If a proof is found, it is reconstituted as an Isabelle specification for further checking. Like HaskHOL, it operates at the intermediate, compiler level.

Microsoft's HALO system [91] has similar goals, however, it is not itself a prover. HALO simply structures the translation from Haskell to first-order logic using a denotational semantics. One in a FOL representation, the verification work is passed off to another system.

## 7.2 Related Verification Efforts

Possibly the closest related work to the verification workflow presented in this dissertation is a recent extension of HERMIT itself. HERMIT has always supported equational reasoning, insomuch that it could be used to mechanize the rewriting of Haskell terms. This reasoning was extremely restricted, though, as it could only be initiated from existing, not arbitrary, terms. Additionally, this reasoning was necessarily destructive because HERMIT had no notion of theorems or saved proofs. A rewrite could be saved as a script, however, the only way to "prove" it correct was to apply it and actually transform a term into its goal state.

When the GHC rewrite rule system was changed to permit inactive rules, it provided HERMIT with a source of core expressions that could be modified without affecting anything else in the compilation environment. HERMIT's rewrite engine was extended to allow direct reasoning over these rules, and a notion of lemmas was formalized to act as proof objects that could be saved and reused as attestations of equivalence [77]. Essentially, HERMIT lemmas were designed and implemented to provide a mechanism for safe and verified term rewriting.

One of the biggest differences between HaskHOL's logic and HERMIT's logic is that HERMIT's implementation of equational reasoning eschews the typical concerns of soundness in favor of practicality. Additionally, the expressivity of HERMIT's lemmas is significantly limited compared to the term languages of HaskHOL and other more general proof systems. For example, HERMIT lemmas can only express equivalences, not implications or any other statements based on non-equational, propositional connectives. That being said, the early work cited above would seem to indicate that HERMIT's extended equational reasoning system works well for its intended purpose.

The Haskabelle tool cited in the introduction is likely the next closest piece of related work. Like HaskHOL and HERMIT, the goal of Haskabelle is to facilitate equational reasoning of Haskell programs. The principal difference, though, is that Haskabelle operates at the source level, rather than at the intermediate level. The primary advantage of working at the source level is that the resultant specification and proof terms more closely resemble the original implementation. This advantage comes at a steep cost, though, as the Haskell programming language, or more specifically the GHC implementation of it, is constantly changing and adding new syntax that must be accounted for. Comparatively, the core language of GHC changes at a much slower rate.

The secondary consequence of working with Haskabelle is that it requires your verifications to be performed with the Isabelle system. This is not intended as an insult or backhanded comment about that system, it is simply a statement of fact. Isabelle is an incredibly impressive proof system, however, its reasoning capability is significantly overkill for most verifications that I care about at this point in time. It is my opinion simple verifications should be completed with simple tools, and that the larger, more complex systems should be reserved for the larger, more complex problems.

Outside of the Haskell universe, there have been a number of other attempts to integrate formal reasoning tools with programming languages. One such example that we are familiar with is Köksal, et. al's work on integrating the Z3 SMT solver with the Scala programming language [55]. This integration differed from our work, in that their goal was to use Z3 to provide Scala with additional reasoning power, rather than use it to verify Scala programs. Additionally, they elected to integrate Z3 as a library using Scala's equivalent of Haskell's foreign function interface, rather than at the compiler level.

Both of these factors make their work much closer to any of the SAT and SMT binding libraries available on Hackage than ours. However, given that a number of these libraries are incomplete, abandoned, or both, it would seem to indicate that there are significant challenges to integrating reasoning tools with this approach. At the same time though, we will be the first to admit that there were significant challenges in reimplementing a formal logic in our language of choice rather than integrating with an existing tool. In either case, we point to the work of Köksal as an example of how beneficial a symbiotic relationship between formal reasoning tool and programming language can be.

# 8 Conclusion

The primary technical contribution of my work covered in this dissertation is the ongoing development of the HaskHOL proof system, as introduced in Chapter 4. In addition to helping to satisfy a growing demand for formal reasoning tools developed in, and for, Haskell, HaskHOL's lightweight implementation enables the novel verification workflow described in Chapter 2. This workflow was imagined with the specific goals of mitigating issues related to integrating development and verification environments, and leveling out the learning curve for users new to proof-based formal verification.

It is my belief that the implementation of this workflow described in Chapter 5 and evaluated in Chapter 6 has met both of these goals. While there is still a significant amount of work to be done in order to realize its full potential, the results observed so far are an exciting first step towards a method of semi-automated verification that is approachable by functional programmers of a less-than-expert level. This approachability is due largely in part to HaskHOL's implementation as an embedded domain specific language.

Embedding HaskHOL within its host language allows users to interact with a theorem prover just as they would any other library. It also provides for a comparatively simpler integration of proof capabilities with existing Haskell-based applications, e.g. the Glasgow Haskell Compiler (GHC). This native integration with GHC allows formal verification to coexist with, and support, informal testing without having to rely on complex, external reasoning tools.

A secondary benefit of this integration is that GHC can be leveraged to perform desugaring and other simplifications before a verification is attempted, essentially letting HaskHOL target the intermediate language of the compiler. HaskHOL's foundational logic was developed with the specific purpose of corresponding to this language in order to allow for a near direction translation from program to proof. This logic, a polymorphic derivative of the simply-typed $\lambda$-calculus, should appear familiar, and easily understandable, for functional programmers.

The previously noted automation was introduced to this workflow by mechanizing the translation from Haskell to HaskHOL. GHC's compiler plugin framework provides direct access to target terms, such that they can be interacted with programatically. The HERMIT library was used to facilitate this access and structure both the translation and verification of core expressions as a phase of compilation. The resultant implementation, shown in Addendum A, was applied in the evaluation of this work by tackling several examples from the motivating problem classes – verification of type class laws and formal "re-verification" of existing test cases.

This evaluation had largely positive results, however, it made it clear that it scaling this approach to verification to handle "real world" problems would be a serious undertaking. Inefficiencies of the HaskHOL proof system need to be addressed, a more complete theory for Haskell reasoning needs to be written, and a more robust user interface needs to be developed to allow for the interactive proof of targets that don't have easily generalizable structures. All of these items represent possible paths for continuing work, should my future endeavors permit it.

Should you, the reader, desire to experiment with this verification workflow yourself, all of the requisite HaskHOL packages are available from my personal Github, `https://github.com/ecaustin/`. The `haskhol-haskell` package on this site contains the verification plugin itself, as well as a README containing installation and execution instructions. Please note that the development of this verification workflow is active research, such that the implementation of the plugin at the time you download it may differ from its presentation in this dissertation; hopefully because of improvements to it. More information regarding the HaskHOL system in general is also available at `http://haskhol.org`.

## References

[1] Abel, A., Benke, M., Bove, A., Hughes, J., & Norell, U. (2005). Verifying Haskell Programs Using Constructive Type Theory. In *In Haskell'05*: ACM Press.

[2] Aitken, W. E. & Reppy, J. H. (1992). *Abstract Value Constructors: Symbolic Constants for Standard ML*. Technical Report TR 92-1290, Department of Computer Science, Cornell University. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[3] Alexander, P. (2006). *System Level Design with Rosetta*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[4] Augustsson, L. (1999). Cayenne — A Language with Dependent Types. In S. Swierstra, J. Oliveira, & P. Henriques (Eds.), *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science* (pp. 240–267). Springer Berlin Heidelberg.

[5] Austin, E. (2011). Haskhol: A haskell hosted domain specific language for higher-order logic theorem proving. Master's thesis, University of Kansas.

[6] Austin, E. & Alexander, P. (2010). HaskHOL: A Haskell Hosted Domain Specific Language Representatino of HOL Light. In *Trends in Functional Programming Pre-Proceedings*.

[7] Austin, E. & Alexander, P. (2013). Stateless Higher-Order Logic with Quantified Types. In S. Blazy, C. Paulin-Mohring, & D. Pichardie (Eds.), *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science* (pp. 469–476). Springer Berlin Heidelberg.

[8] Baraona, P., Penix, J., & Alexander, P. (1995). VSPEC: A Declarative Requirements Specification Language for VHDL. In J.-M. Berge, O. Levia, & J. Rouillard (Eds.), *High-Level System Modeling: Specification Languages*, volume 3 of *Current Issues in Electronic Modeling* chapter 3, (pp. 51–75). Boston, MA: Kluwer Academic Publishers.

[9] Barendregt, H. et al. (1991). Introduction to generalized type systems. *J. Funct. Program.*, 1(2), 125–154.

[10] Bernardy, J.-P., Jansson, P., & Claessen, K. (2010). Testing Polymorphic Properties. In A. Gordon (Ed.), *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science* (pp. 125–144). Springer Berlin Heidelberg.

[11] Boehm, H.-J. (1985). Partial polymorphic type inference is undecidable. In *Foundations of Computer Science, 1985., 26th Annual Symposium on* (pp. 339–345).

[12] Brady, E. C. (2011). Idris —: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11 (pp. 43–54). New York, NY, USA: ACM.

[13] Church, A. (1932). A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 33(2). http://www.jstor.org/stable/1968702Electronic Edition.

[14] Church, A. (1940). A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5, 56–68.

[15] Church, A. (1985). *The Calculi of Lambda Conversion*. Princeton, NJ, USA: Princeton University Press.

[16] Claessen, K. (2012). Shrinking and Showing Functions: (Functional Pearl). In *Proceedings of the 2012 Haskell Symposium*, Haskell '12 (pp. 73–80). New York, NY, USA: ACM.

[17] Claessen, K. & Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00 (pp. 268–279). New York, NY, USA: ACM.

[18] Coquand, T. (1994). A New Paradox in Type Theory. In *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science* (pp. 7–14).: Elsevier.

[19] Coquand, T. & Huet, G. (1988). The calculus of constructions. *Information and Computation*, 76(2–3), 95 – 120.

[20] Crow, J., Rushby, J., Shankar, N., & Srivas, M. (1995). *A Tutorial Introduction to PVS*. SRI International, Menlo Park, CA. Presented at WIFT'95.

[21] Curry, H. B. & Feys, R. (1958). *Combinatory Logic, Volume I*. North-Holland. Second printing 1968.

[22] Curry, H. B., Hindley, J. R., & Seldin, J. P. (1972). *Combinatory Logic, Volume II*. North-Holland.

[23] Farmer, A., Gill, A., Komp, E., & Sculthorpe, N. (2012). The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Haskell* (pp. 1–12).

[24] Gentzen, G. (1935). Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39, 176–210.

[25] GHC Team (2014). The Glasgow Haskell Compilation System User's Guide, Version 7.8.4. Website:. http://haskell.org/ghc/.

[26] Gill, A. (2009). Type-Safe Observable Sharing in Haskell. In *Proceedings of the Second ACM SIGPLAN Haskell Symposium*, Haskell '09 (pp. 117–128). New York, NY, USA: ACM.

[27] Girard, J.-Y. (1971). Une Extension De ĬInterpretation De G odel a ĬAnalyse, Et Son Application a ĬElimination Des Coupures Dans ĬAnalyse Et La Theorie Des Types. In J. Fenstad (Ed.), *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics* (pp. 63 – 92). Elsevier.

[28] Gordon, M. (1982). Representing a Logic in the LCF Metalanguage. In D. N'eel (Ed.), *Tools and Notions for Program Construction* (pp. 163–185).: Cambridge University Press.

[29] Gordon, M. (1985). *HOL : A Machine Oriented Formulation of Higher Order Logic.* Technical Report UCAM-CL-TR-68, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500.

[30] Gordon, M. (1991). Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware.

[31] Gordon, M. (2000). From LCF to HOL: A Short History. In *Proof, Language, and Interaction* (pp. 169–186).

[32] Gordon, M. J. C. (1989). HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle & P. A. Subrahmanyam (Eds.), *Current Trends in Hardware Verification and Automated Theorem Proving* (pp. 73–128). Springer-Verlag.

[33] Gordon, M. J. C. & Melham, T. F., Eds. (1993). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* New York, NY, USA: Cambridge University Press.

[34] Gordon, M. J. C., Milner, R., & Wadsworth, C. P. (1979). *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer.

[35] Haftmann, F. (2010). From higher-order logic to haskell: There and back again. In J. P. Gallagher & J. Voigtländer (Eds.), *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010*: ACM.

[36] Haftmann, F. & Wenzel, M. (2007). Constructive Type Classes in Isabelle. In T. Altenkirch & C. McBride (Eds.), *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science* (pp. 160–174). Springer Berlin Heidelberg.

[37] Hall, C. V., Hammond, K., Peyton Jones, S. L., & Wadler, P. L. (1996). Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2), 109–138.

[38] Halling, B. & Alexander, P. (2013). Verifying a Privacy CA Remote Attestation Protocol. In G. Brat, N. Rungta, & A. Venet (Eds.), *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science* (pp. 398–412). Springer Berlin Heidelberg.

[39] Harper, R., Honsell, F., & Plotkin, G. (1993). A Framework for Defining Logics. *J. ACM*, 40(1), 143–184.

[40] Harrison, J. (1995). Inductive definitions: Automation and application. In E. Thomas Schubert, P. Windley, & J. Alves-Foss (Eds.), *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science* (pp. 200–213). Springer Berlin Heidelberg.

[41] Harrison, J. (1996). Hol Light: A Tutorial Introduction. In M. Srivas & A. Camilleri (Eds.), *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science* (pp. 265–269). Springer Berlin Heidelberg.

[42] Hindley, R. (1969). The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146, 29–60.

[43] Homeier, P. (2009). The HOL-Omega Logic. In S. Berghofer, T. Nipkow, C. Urban, & M. Wenzel (Eds.), *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science* (pp. 244–259). Springer Berlin Heidelberg.

[44] Hudak, P. (1996). Building Domain-Specific Embedded Languages. *ACM Comput. Surv.*, 28(4es).

[45] Huffman, B. (2012a). Formal Verification of Monad Transformers. In *ICFP* (pp. 15–16).

[46] Huffman, B. (2012b). Type Constructor Classes and Monad Transformers. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Tycon.shtml`, Formal proof development.

[47] Huffman, B., Matthews, J., & White, P. (2005). Axiomatic Constructor Classes in Isabelle/HOLCF. In J. Hurd & T. Melham (Eds.), *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science* (pp. 147–162). Springer Berlin Heidelberg.

[48] Hurd, J. (2001). Predicate Subtyping with Predicate Sets. In R. Boulton & P. Jackson (Eds.), *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science* (pp. 265–280). Springer Berlin Heidelberg.

[49] Hurd, J. (2009). OpenTheory: Package Management for Higher Order Logic Theories.

[50] Kaes, S. (1992). Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92 (pp. 193–204). New York, NY, USA: ACM.

[51] Kaufmann, M. & Moore, J. S. (2008). *The ACL2 User's Manual*. `http://www.cs.utexas.edu/users/moore/acl2/#User's-Manual`.

[52] Kleene, S. (1952). *Introduction to Metamathematics*. North-Holland Publishing Company. Co-publisher: Wolters–Noordhoff; 8th revised ed.1980.

[53] Klein, G., Derrin, P., & Elphinstone, K. (2009). Experience Report: seL4: Formally Verifying a High-performance Microkernel. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09 (pp. 91–96). New York, NY, USA: ACM.

[54] Klein, G., Sewell, T., & Winwood, S. (2010). Refinement in the Formal Verification of the seL4 Microkernel. In D. S. Hardin (Ed.), *Design and Verification of Microprocessor Systems for High-assurance Applications* (pp. 323–339). Springer US.

[55] Köksal, A., Kuncak, V., & Suter, P. (2011). Scala to the Power of Z3: Integrating SMT and Programming. In N. Bjørner & V. Sofronie-Stokkermans (Eds.), *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science* (pp. 400–406). Springer Berlin Heidelberg.

[56] Krauss, A. (2010). Defining Recursive Functions in Isabelle/HOL.

[57] Kreitz, C. (2002). *The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide*. Department of Computer Science, Cornell University.

[58] Leroy, X. (2006). Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. *SIGPLAN Not.*, 41(1), 42–54.

[59] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., & Vouillon, J. (2014). *The OCaml system (release 4.02): Documentation and user's manual.* Institut National de Recherche en Informatique et en Automatique.

[60] Letouzey, P. (2003). A New Extraction for Coq. In *Proceedings of the 2002 International Conference on Types for Proofs and Programs*, TYPES'02 (pp. 200–219). Berlin, Heidelberg: Springer-Verlag.

[61] Liang, S., Hudak, P., & Jones, M. (1995). Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95 (pp. 333–343). New York, NY, USA: ACM.

[62] Marlow, S. (2010). Haskell 2010 Language Report.

[63] McBride, C. (2005). Epigram: Practical Programming with Dependent Types. In *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP'04 (pp. 130–170). Berlin, Heidelberg: Springer-Verlag.

[64] Melham, T. (1993). *Higher order logic and hardware verification.* New York, NY, USA: Cambridge University Press.

[65] Milner, R. (1972). *Logic for Computable Functions: Description of a Machine Implementation.* Technical report, Stanford University, Stanford, CA, USA.

[66] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 348 – 375.

[67] Milner, R., Tofte, M., & Macqueen, D. (1997). *The Definition of Standard ML.* Cambridge, MA, USA: MIT Press.

[68] Müller, O., Nipkow, T., von Oheimb, D., & Slotosch, O. (1999). HOLCF. In *J. Funct. Program.* (pp. 191–223).

[69] Nipkow, T., Wenzel, M., & Paulson, L. C. (2002). *Isabelle/HOL: a proof assistant for higher-order logic.* Berlin, Heidelberg: Springer-Verlag.

[70] Norell, U. (2009). Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08 (pp. 230–266). Berlin, Heidelberg: Springer-Verlag.

[71] Paulson, L. C. (1987). *Logic and Computation: Interactive Proof with Cambridge LCF*. New York, NY, USA: Cambridge University Press.

[72] Peano, G. (1889). *Arithmetices Principia Novo Methodo Exposita*. Bocca.

[73] Pfenning, F. (1993). On the Undecidability of Partial Polymorphic Type Reconstruction. *Fundam. Inf.*, 19(1-2), 185–199.

[74] Pierce, B. C. (2002). *Types and Programming Languages*. Cambridge, MA, USA: MIT Press.

[75] Reynolds, J. C. (1974). Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation* (pp. 408–423). London, UK, UK: Springer-Verlag.

[76] Rushby, J. (1997). Subtypes for specifications. In M. Jazayeri & H. Schauer (Eds.), *Software Engineering — ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science* (pp. 4–19). Springer Berlin Heidelberg.

[77] Sculthorpe, N., Farmer, A., & Gill, A. (2014). Making a Century in HERMIT. Submitted to peer-review for consideration for publication in the post-proceedings of IFL 2014.

[78] Sculthorpe, N., Frisby, N., & Gill, A. (2013). The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. Submitted to the Journal of Functional Programming.

[79] Shaw, J. (2013). The Happstack Book: Modern, Type-Safe Web Development in Haskell.

[80] Sheard, T. & Jones, S. P. (2002). Template Meta-programming for Haskell. In *In Proceedings of the ACM SIGPLAN Workshop on Haskell* (pp. 1–16).: ACM.

[81] Slind, K. & Norrish, M. (2008). A Brief Overview of HOL4. In O. Mohamed, C. Muñoz, & S. Tahar (Eds.), *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science* (pp. 28–32). Springer Berlin Heidelberg.

[82] Sonnex, W., Drossopoulou, S., & Eisenbach, S. (2012). Zeno: An automated prover for properties of recursive data structures. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12 (pp. 407–421). Berlin, Heidelberg: Springer-Verlag.

[83] Sørensen, M. H. & Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. New York, NY, USA: Elsevier Science Inc.

[84] Sozeau, M. & Oury, N. (2008). First-Class Type Classes. In O. Mohamed, C. Muñoz, & S. Tahar (Eds.), *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science* (pp. 278–293). Springer Berlin Heidelberg.

[85] Sternagel, C. & Thiemann, R. (2014). Certification Monads. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Certification_Monads.shtml`, Formal proof development.

[86] Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., & Donnelly, K. (2007). System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07 (pp. 53–66). New York, NY, USA: ACM.

[87] Team, C. (2012). The Coq Proof Assistant Reference Manual. `http://coq.inria.fr`.

[88] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42), 230–265.

[89] Turing, A. M. (1937). Computability and $\lambda$-definability. *Journal of Symbolic Logic*, 2, 153–163.

[90] Völker, N. (2007). Hol2p - A System of Classical Higher Order Logic with Second Order Polymorphism. In K. Schneider & J. Brandt (Eds.), *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science* (pp. 334–351). Springer Berlin Heidelberg.

[91] Vytiniotis, D., Peyton Jones, S., Claessen, K., & Rosén, D. (2013). Halo: Haskell to logic through denotational semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13 (pp. 431–442). New York, NY, USA: ACM.

[92] Ward, J., Kimmell, G., & Alexander, P. (2005). Prufrock: A framework for constructing polytypic theorem provers. In *Proceedings of the Automated Software Engineering Conference (ASE'05)* Long Beach, CA.

[93] Wiedijk, F. (2009). Stateless HOL. In *TYPES* (pp. 47–61).

[94] Wiedijk, F. (2012). Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285(0), 85 – 100. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10).

[95] Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., & Magalhães, J. P. (2012). Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12 (pp. 53–66). New York, NY, USA: ACM.

# A Select Code from the Verification Plugin

Listing A.1: High-Level Translation Transformations

```haskell
trans :: TransformH CoreTC HOLTerm
trans = promoteBindT (transform $ \ c -> liftM snd . applyT transBind c)

transBind :: TransformH CoreBind (HOLTerm, HOLTerm)
transBind = nonRecT transVar transExpr (\ (Tm bv) (Tm e) -> (bv, e))

transVar :: TransformH Var HOLSum
transVar = transform $ \ c v ->
    let name = pack . unqualifiedName $ varName v in
      if isTKVar v then return . Ty . fromRight . mkSmall $ mkVarType name
      else do (Ty ty) <- applyT transType c (varType v)
              return . Tm $! mkVar name ty

transExpr :: TransformH CoreExpr HOLSum
transExpr =
       varT transVar
      <+ appT transExpr transExpr transApp
      <+ lamT transVar transExpr transLam
      <+ letT transBind transExpr transLet
      <+ caseT transExpr transVar transType (const transAlt) transCase
      <+ typeT transType
```

Listing A.2: Type and Low-Level Translation Transformations

```haskell
transType :: TransformH Type HOLSum
transType =
         tyVarT transVar
      <+ appTyT transType transType transOp
      <+ funTyT transType transType
           (\ (Ty x) (Ty y) -> Ty $ mkFunTy' [x, y])
      <+ forAllTyT transVar transType
           (\ (Ty x) (Ty y) -> Ty . fromRight $ mkUType x y)
      <+ tyConAppT transTyCon (const transType)
           (\ x ys ->
              if null ys
              then let (_, n) = destTypeOp x in
                     if n == 0 then Ty $ tyApp' x []
                     else TyOp x
              else Ty . tyApp' x $ map (\ (Ty y) -> y) ys)

transOp :: HOLSum -> HOLSum -> HOLSum
transOp (Ty (TyVar _ x)) (Ty ty) =
    Ty $ tyApp' (mkTypeOpVar x) [ty]
transOp (Ty (TyApp op tys)) (Ty ty) =
    Ty . tyApp' op $ tys ++ [ty]
transOp _ _ = error "transOp"

transTyCon :: TransformH TyCon TypeOp
transTyCon = contextfreeT $ \ op ->
    if isFunTyCon op then return tyOpFun
    else let name = pack . unqualifiedName $ tyConName op
             arity = tyConArity op in
           return $! newPrimitiveTypeOp name arity

transAlt :: TransformH CoreAlt (Text, [HOLTerm], HOLTerm)
transAlt = altT transAlt' (const transVar) transExpr
             (\ x y z -> (x, map fromTm y, fromTm z))
  where transAlt' :: TransformH AltCon Text
        transAlt' = contextfreeT $ \ ac ->
            case ac of
              DEFAULT -> return "DEFAULT"
              DataAlt dc -> return . pack . unqualifiedName $ dataConName dc
              LitAlt _ -> fail "transAlt: literals are not handled currently."

        fromTm :: HOLSum -> HOLTerm
        fromTm (Tm x) = x
        fromTm _ = error "fromTm"
```

## Listing A.3: Term Translation Transformations

```
transApp :: HOLSum -> HOLSum -> HOLSum
-- force type applications if we can
transApp (Tm x@(HC.Var name (UType xty@(TyVar _ bname) ty))) y =
    case y of
      TyOp op -> Tm . mkVar name $ typeSubst [(mkTypeOpVar bname, op)] ty
      Ty ty' -> Tm . mkVar name $ typeSubst [(xty, ty')] ty
      Ty ty' -> Tm $ mkTyComb' x ty'
      Tm y' -> Tm $ mkComb' x y'
transApp (Tm x) (Ty y) = Tm $ mkTyComb' x y
-- erase type class arguments rather crudely
transApp (Tm x@(HC.Var name ty)) (Tm y@(HC.Var cls _))
    | T.take 2 cls == "$d" =
        let (_, ty') = fromJust $ destFunTy ty in
          Tm $ mkVar name ty'
    | otherwise = Tm $ mkComb' x y
transApp (Tm x) (Tm y) = Tm $ mkComb' x y
transApp _ _ = error "transApp: type-level application at term level."

transLam :: HOLSum -> HOLSum -> HOLSum
transLam (Ty bv) (Tm bod) = Tm . fromRight $ mkTyAbs bv bod
-- erase type class arguments
transLam (Tm bv@(HC.Var name _)) (Tm bod)
    | T.take 2 name == "$d" = Tm bod
    | otherwise = Tm $ mkAbs' bv bod
transLam _ _ = error "transLam: type-level abstraction at term level."

transLet :: (HOLTerm, HOLTerm) -> HOLSum -> HOLSum
transLet (bv@(HC.Var name _), be) (Tm bod)
    -- erase type class arguments
    | T.take 2 name == "$d" = Tm bod
    | otherwise = Tm mkLet'
  where mkLet' :: HOLTerm
        mkLet' =
            let tyLend = mkFunTy' [ty, ty]
                lend = mkComb' (mkVar "LET_END" tyLend) bod
                lbod = mkGabs' bv lend
                tyAbs = typeOf lbod
                (ty1, ty2) = fromJust $ destFunTy tyAbs
                tyLet = mkFunTy' [tyAbs, ty1, ty2] in
              foldl mkComb' (mkVar "LET" tyLet) [lbod, be]
          where ty = typeOf bod

        mkGabs' :: HOLTerm -> HOLTerm -> HOLTerm
        mkGabs' tm1@HC.Var{} tm2 = mkAbs' tm1 tm2
        mkGabs' tm1 tm2 =
            let fTy = mkFunTy' [ty1, ty2]
                fvs = frees tm1
                f = variant (frees tm1 ++ frees tm2) $ mkVar "f" fTy
                bodIn = foldr (mkBinder' "!") (mkGeq' (mkComb' f tm1) tm2) fvs
                tyGabs = mkFunTy' [mkFunTy' [fTy, tyBool], fTy] in
              mkComb' (mkVar "GABS" tyGabs) $ mkAbs' f bodIn
          where ty1 = typeOf tm1
                ty2 = typeOf tm2
transLet _ Ty{} = error "transLet: types not allowed as bodies of let terms."
transLet _ _ = error "transLet: binding must be a variable."

transCase :: HOLSum -> HOLSum -> HOLSum -> [(Text, [HOLTerm], HOLTerm)]
          -> HOLSum
transCase (Tm match) (Tm asVar) (Ty resTy) alts = flip evalState 0 $
    do alts' <- mapM mkPat alts
       let ty' = mkFunTy' [asTy, resTy, tyBool]
           tmSeq = mkVar "_SEQPATTERN" $ mkFunTy' [ty', ty', ty']
           clauses = foldr1 (\ s t -> mkComb' (mkComb' tmSeq s) t) alts'
           tmMatch = mkVar "_MATCH" $ mkFunTy' [asTy, ty', resTy]
           res = mkComb' (mkComb' tmMatch match) clauses
       return $! Tm res
  where asTy = typeOf asVar

        mkPat :: (Text, [HOLTerm], HOLTerm) -> State Int HOLTerm
        mkPat (con, bvs, res) =
            do x <- pgenVar asTy
               y <- pgenVar $ typeOf res
               let patTm = mkVar "_UNGUARDED_PATTERN" $
                             mkFunTy' [tyBool, tyBool, tyBool]
                   tys = map typeOf bvs
                   conTm = foldr (flip mkComb')
                             (mkVar con $ mkFunTy' (tys ++ [asTy])) bvs
                   bod = mkComb' (mkComb' patTm (mkGeq' conTm x)) (mkGeq' res y)
               return $! mkAbs' x (mkAbs' y (foldr (mkBinder' "?") bod bvs))

        pgenVar :: HOLType -> State Int HOLTerm
        pgenVar ty =
            do n <- get
               put $ succ n
               return $! mkVar (pack $ "_GENPVAR" ++ show n) ty
transCase _ _ _ _ = error "transCase: expecting terms, got types."
```

Listing A.4: Primitive HOLTerm Transformations

```
type TransHOL thry a b = Transform (TheoryPath thry) IO a b

liftHOL' :: MonadIO m => TheoryPath thry -> HOL Proof thry a -> m a
liftHOL' ctxt x = liftIO $ runHOLProof True x ctxt

htyvarT :: TransHOL thry Bool a -> TransHOL thry Text b
        -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLType c
htyvarT fl i f = transform $ \ c e ->
    case e of
      TyVar b t -> do b' <- applyT fl c b
                      t' <- applyT i c t
                      liftHOL' c $! f b' t'
      _ -> fail "not a type variable."

htyappT :: TransHOL thry TypeOp a -> TransHOL thry HOLType b
        -> (a -> [b] -> HOL Proof thry c) -> TransHOL thry HOLType c
htyappT op ty f = transform $ \ c e ->
    case e of
      TyApp o tys -> do o' <- applyT op c o
                        tys' <- mapM (applyT ty c) tys
                        liftHOL' c $! f o' tys'
      _ -> fail "not a type level application."

hutypeT :: TransHOL thry HOLType a -> TransHOL thry HOLType b
        -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLType c
hutypeT t1 t2 f = transform $ \ c e ->
    case e of
      UType bv bod -> do bv' <- applyT t1 c bv
                         bod' <- applyT t2 c bod
                         liftHOL' c $! f bv' bod'
      _ -> fail "not a type quantification."

hvarT :: TransHOL thry Text a -> TransHOL thry HOLType b
      -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLTerm c
hvarT i ty f = transform $ \ c e ->
    case e of
      Var v t -> do v' <- applyT i c v
                    t' <- applyT ty c t
                    liftHOL' c $! f v' t'
      _ -> fail "not a variable."

hconstT :: TransHOL thry Text a -> TransHOL thry HOLType b
        -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLTerm c
hconstT i ty f = transform $ \ c e ->
    case e of
      Const v t -> do v' <- applyT i c v
                      t' <- applyT ty c t
                      liftHOL' c $! f v' t'
      _ -> fail "not a constant."

hcombT :: TransHOL thry HOLTerm a -> TransHOL thry HOLTerm b
       -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLTerm c
hcombT t1 t2 f = transform $ \ c e ->
    case e of
      Comb e1 e2 -> do e1' <- applyT t1 c e1
                       e2' <- applyT t2 c e2
                       liftHOL' c $! f e1' e2'
      _ -> fail "not an application."

habsT :: TransHOL thry HOLTerm a -> TransHOL thry HOLTerm b
      -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLTerm c
habsT t1 t2 f = transform $ \ c e ->
    case e of
      Abs bv bod -> do bv' <- applyT t1 c bv
                       bod' <- applyT t2 c bod
                       liftHOL' c $! f bv' bod'
      _ -> fail "not an abstraction."

htycombT :: TransHOL thry HOLTerm a -> TransHOL thry HOLType b
         -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLTerm c
htycombT t1 t2 f = transform $ \ c e ->
    case e of
      TyComb tm ty -> do tm' <- applyT t1 c tm
                         ty' <- applyT t2 c ty
                         liftHOL' c $! f tm' ty'
      _ -> fail "not a type abstraction."

htyabsT :: TransHOL thry HOLType a -> TransHOL thry HOLTerm b
        -> (a -> b -> HOL Proof thry c) -> TransHOL thry HOLTerm c
htyabsT t1 t2 f = transform $ \ c e ->
    case e of
      TyAbs ty tm -> do ty' <- applyT t1 c ty
                        tm' <- applyT t2 c tm
                        liftHOL' c $! f ty' tm'
      _ -> fail "not an type abstraction."
```

Listing A.5: Term and Type Replacement Transformations

```
replaceTerm :: [(Text, HOLTerm)] -> TransHOL thry HOLTerm HOLTerm
replaceTerm tmmap = repTerm
  where repTerm :: TransHOL thry HOLTerm HOLTerm
        repTerm =
               contextfreeT (\ tm@Const{} -> return tm)
            <+ hcombT repTerm repTerm (\ x -> liftO . mkComb x)
            <+ habsT (contextfreeT return) repTerm (\ x -> liftO . mkAbs x)
            <+ htycombT repTerm (contextfreeT return) (\ x -> liftO . mkTyComb x)
            <+ htyabsT (contextfreeT return) repTerm (\ x -> liftO . mkTyAbs x)
            <+ hvarT (contextfreeT return) (contextfreeT return) repVar
               -- EvNote: has the potential to cause issues if a bound var has
               --         a name shared with a mapped constant

        repVar :: Text -> HOLType -> HOL Proof thry HOLTerm
        repVar i ty =
            (tryFind (\ (key, Const name ty') ->
                      if key == i && isJust (typeMatch ty' ty ([], [], []))
                      then mkMConst name ty
                      else fail "") tmmap)
              <|> (return $! mkVar i ty)

replaceType :: [(Text, TypeOp)] -> TransHOL thry HOLTerm HOLTerm
replaceType tymap = repTerm
  where repTerm :: TransHOL thry HOLTerm HOLTerm
        repTerm =
               hvarT (contextfreeT return) repType (\ x -> return . mkVar x)
            <+ contextfreeT (\ tm@Const{} -> return tm)
            <+ hcombT repTerm repTerm (\ x -> liftO . mkComb x)
            <+ habsT repTerm repTerm (\ x -> liftO . mkAbs x)
            <+ htycombT repTerm repType (\ x -> liftO . mkTyComb x)
            <+ htyabsT repType repTerm (\ x -> liftO . mkTyAbs x)

        repType :: TransHOL thry HOLType HOLType
        repType =
               contextfreeT (\ ty@TyVar{} -> return ty)
            <+ htyappT (contextfreeT return) repType repTyApp
            <+ hutypeT (contextfreeT return) repType
                 (\ x -> liftO . mkUType x)

        repTyApp :: TypeOp -> [HOLType] -> HOL Proof thry HOLType
        repTyApp op tys =
            do op' <- repOp
               liftO $! tyApp op' tys
           where repOp :: HOL Proof thry TypeOp
                 repOp =
                     case op of
                       (TyPrimitive i arity)
                           | i == "fun" -> return op
                           | otherwise ->
                               (tryFind (\ (key, op') ->
                                         let (_, arity') = destTypeOp op' in
                                           if key == i && arity' == arity
                                           then return op'
                                           else fail "") tymap)
                                 <|> (return $! mkTypeOpVar i)
                       _ -> return op
```

Listing A.6: Verification Plugin Implementation

```haskell
ctxt :: TheoryPath HaskellType
ctxt = ctxtHaskell

liftHOL :: HOL Proof HaskellType a -> HPM a
liftHOL = liftHOL' ctxt

plugin :: Plugin
plugin = hermitPlugin $ \ _ -> firstPass $
    do liftIO $ putStrLn "Parsing configuration files..."
       tyMap <- prepConsts "types.h2h" $ liftHOL types
       tmMap <- prepConsts "terms.h2h" $ liftHOL constants
       opts <- liftIO $ optParse "config.h2h"
       let getOpt :: String -> String -> HPM String
           getOpt lbl err =
             maybe (fail err) return $ lookup lbl opts
--
       liftIO $ putStrLn "Translating from Core to HOL..."
       target <- getOpt "binding" "<binding> not set in config.h2h."
       tm <- at (bindingOfT $ cmpString2Var target) $ query trans
--
       let pass :: HOLTerm -> HPM HOLTerm
           pass x = liftIO $
               do x' <- applyT (replaceType tyMap) ctxt x
                  applyT (replaceTerm tmMap) ctxt x'
       cls <- getOpt "bindingType" "<bindingType> not set in config.h2h."
       tm' <- if cls == "ConsClass"
              then do cls' <- getOpt "class" "<class> not set in config.h2h."
                      consClassPass cls' tm pass
              else let (bvs, bod) = stripTyAbs tm
                       (bvs', bod') = stripAbs bod in
                     do x <- pass bod'
                        bvs'' <- mapM pass bvs'
                        liftHOL $ flip (foldrM mkTyAll) bvs =<<
                          listMkForall bvs'' x
--
       liftIO $ putStrLn "Proving..."
       prfType <- getOpt "proofType" "<proofType> not set in config.h2h."
       prfMod <- getOpt "proofModule" "<proofModule> not set in config.h2h."
       prfName <- getOpt "proofName" "<proofName> not set in config.h2h."
       tac <- liftHOL $ if prfType == "HOLThm"
                        then do thm <- runHOLHint prfName [prfMod]
                                return (tacACCEPT (thm::HOLThm))
                        else runHOLHint ("return " ++ prfName)
                               [prfMod, "HaskHOL.Deductive"]
       liftHOL $ printHOL tm'
       liftHOL $ printHOL =<< prove tm' tac

consClassPass :: String -> HOLTerm -> (HOLTerm -> HPM HOLTerm) -> HPM HOLTerm
consClassPass cls tm pass =
    do liftIO $ putStrLn "Translating Arguments..."
       let (_, args) = stripComb tm
       args' <- mapM trans' args
--
       liftIO $ putStrLn "Building Class Instance..."
       monad <- liftHOL $ mkConstFull (pack cls) ([],[],[])
       maybe (fail "Reconstruction of class instance failed.") return $
         foldlM mkIComb monad args'
  where trans' :: HOLTerm -> HPM HOLTerm
        trans' x@(Var name _) = pass =<< query
            (do lp <- lookupBind $ unpack name
                case lp of
                  Nothing -> return x
                  Just res -> localPathT res trans)
        trans' x = pass x

        lookupBind :: String -> TransformH CoreTC (Maybe LocalPathH)
        lookupBind x = catchesT [ liftM Just . bindingOfT $ cmpString2Var x
                                , return Nothing
                                ]

prepConsts :: String -> HPM (Map Text b) -> HPM [(Text, b)]
prepConsts file mcnsts =
    do cmap <- liftIO $ parse file
       cnsts <- mcnsts
       let cnsts' = catMaybes $
                      map (\ (x, y) -> do y' <- mapLookup y cnsts
                                          return (x, y')) cmap
       return (cnsts' ++ mapToList cnsts)
```