HaskHOL: A Haskell Hosted Domain Specific Language for Higher-Order Logic Theorem Proving

BY

Evan Christopher Austin

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Chairperson Dr. Perry Alexander

Dr. Andy Gill

Dr. Arvin Agah

Date Defended: July 26, 2011

The Thesis Committee for Evan Christopher Austin
certifies that this is the approved version of the following thesis:

HaskHOL: A Haskell Hosted Domain Specific Language for Higher-Order Logic Theorem Proving

---

Chairperson Dr. Perry Alexander

Date Approved: July 26, 2011

Abstract

HaskHOL is an implementation of a HOL theorem proving capability in Haskell. Motivated by a need to integrate theorem proving capabilities into a Haskell-based tool suite, HaskHOL began as a simple port of HOL Light to Haskell. However, Haskell's laziness, immutable data, and monadic extensions both complicate an implementation and enable a new feature class. This thesis describes HaskHOL, its motivation and implementation. Its use to implement a primitive, interactive theorem prover is explored and its performance is evaluated using a collection of intuitionistically valid problems.

Acknowledgements

Dr. Perry Alexander has served as a constant inspiration, mentor, and friend since very early in my college career. His guidance and the myriad of opportunities he has afforded me have kindled a passion for formal methods research that I can honestly say I would not have had otherwise.

Dr. Andy Gill was the first professor I had at KU who challenged me to go above and beyond what was required for a course. I owe him for much of the success that I have achieved, for I would have likely aimed much lower if not for him pushing me to always do my best.

I owe perhaps the most to Dr. Arvin Agah, without whom I would likely be wasting away in a cubicle somewhere instead of enjoying academia. It was Dr. Agah's guidance and advice during my senior year that convinced me to pursue graduate school, and it was again his help that made the transition to being a master's student a painless one.

I would like to also extend my gratitude to all of my fellow students whom I've had the pleasure to work with over the last few years. Thank you to the other members of the Systems Level Design Group, Nicolas Frisby, Megan Peck, Wesley Peck, and Mark Snyder; to members of the greater Computer Systems Design Laboratory, Andrew Farmer and Michael Jantz; and to the former KU and ITTC students, Garrin Kimmell and Kevin Matlage. I've always believed that you are only as good as the company you keep, so I feel honored to share, or have shared, a lab with such fine people.

Lastly I would like to thank my family, because my mom would hurt me if I didn't.

Contents

Chapter 1

Introduction

The notion of proof is something that everyone has struggled with at one point in time or another. Whether it was in high school geometry class trying to figure out how you could show that triangles always have 180°, or at a bar arguing with your friends about how many NBA teams Jacque Vaughn has played for[1], each of us has either formally or informally tried to prove something. At its heart, a proof is a very simple concept; it is an argument that some statement is true. This simplicity is reflected in the examples mentioned above, with our proofs consisting of trivial applications of the ”laws of triangles” or a quick lookup on the Elias Sports Bureau website, the official record keeper of the NBA. Don’t be mistaken, though, proving can rapidly become a very complicated activity, especially when talking about the domain of computer software or hardware. How can we show that a body of code with millions of lines or a processor with billions of transistors is correct? Is that verification something that we want to do by the unassisted human hand?

[1] Five

The first question is still a relatively open one, however, the answer to the second is a resounding no. If there is one thing humans are great at, it is making mistakes, especially when dealing with problems of intractable size. Thankfully, advances in computer science have spawned a wide array of formal systems for computer aided design and verification, so we no longer are forced to reason about such complex topics on our own. Once such class of systems is automated theorem proving, a branch of theoretical computer science whose goal is to assist and automate proof as discussed above. Even within this class there are numerous other subclasses, separating the systems even further based on their principle logics, levels of interactivity, or domains of problems they are designed to reason over.

This thesis in particular will explore the implementation of a new member of the Higher-Order Logic (HOL) theorem proving family. HOL has a rich history of being used for system verification that dates back to Mike Gordon's 1986 paper, "Why higher-order logic is a good formalism for specifying and verifying hardware" [12]. Since then, research has spawned a wide variety of materials covering verification with HOL, including everything from textbooks [28] to research papers detailing verification targets ranging in size from single algorithms [18–20] to large software systems [22]. Given the success HOL has had verifying such a diverse set of topics, when it came time to connect a verification formalism to Rosetta [3,4], a system level design language in development at The University of Kansas, HOL seemed to be the logical choice.

1.1 Higher-Order Logic and Rosetta

There have been a number of popular and successful tools developed in the HOL family, so selecting among them should be as easy as finding the one that matched our requirements. Many of the tools, such as HOL4 [33], PVS [8] and Isabelle/HOL [29], require the Rosetta specification to be "thrown over the wall" and embedded in the prover with the verification work ultimately occurring in the formal tool's environment and not Rosetta's. This is non-ideal for two reasons. First, it requires translating informational messages among the Rosetta tool suite and the prover. Our experience with VSPEC [6] demonstrates the difficulty of trying to restructure prover outputs to be meaningful in the context of the original specification. Second, it requires the users to become proficient with two very different interfaces, one for Rosetta and one for the formal tool. These two issues lead to the desired properties for a formal tool: it must be lightweight and it must be able to interface with the existing Rosetta environment.

HOL Light [17], a lightweight implementation of a HOL system designed to run in a terminal shell, satisfies these goals, but not without introducing new problems of its own. Namely, it piggybacks upon the OCaml interpreter and requires the use of checkpointing software to prevent having to spend several minutes loading the theorems of the tool every time it is run. Admittedly these are not issues for the majority of HOL Light users, however, the Rosetta tool suite is developed in Haskell, not OCaml, on machines where no reliable checkpointing software exists. Given this, the use of HOL Light introduces another dependency to our tool chain, increase the difficulty of development, and potentially decrease the portability of the Rosetta tool suite.

1.2 Formal Reasoning and Haskell

There have been several efforts to connect formal reasoning systems, even HOL, to Haskell. Recent work has been completed by Florian Haftmann to connect HOL and Haskell via translations between specifications written in Isabelle and executable Haskell source [16]. The generation of code from an Isabelle specification is presented as an established and mature tool, however, the tool for verifying Haskell artifacts, Haskabelle, depends on a large external tool base.

Agda [2], a combination of a dependently typed language and proof assistant, is another attempt to connect formal reasoning to Haskell. Because both pieces utilize concrete syntax that is heavily inspired by Haskell, it is possible for Agda to translate code produced from compiling Haskell into an equivalent Agda specification that can be reasoned about in its proof assistant. This is similar to the approach taken by Haskabelle, the primary differences being the logical foundations and proof techniques associated with each tool.

In addition to the various attempts to connect Haskell with external tools, there is at least one major attempt to bring these reasoning capabilities to Haskell intensionally. Ivor [7] is a type theory based theorem proving library that provides an API for embedding theorem proving capabilities inside of Haskell applications. Instead of dedicating itself to one fixed logical system, like HOL, Ivor aims to be an extensible theorem proving framework with the goal of implementing a variety logical systems and tactic languages that can change based on the application. There is also a major difference compared to the other tools in that Ivor is designed to be used in conjunction with generative programming techniques instead of directly by humans.

1.3 HaskHOL

All of the formal systems mentioned above fail to satisfy the previously established requirements for one reason or another. Given that, it was decided that the best solution for pairing a Haskell-based prover with the Rosetta tool suite was to develop one of our own. The first attempt to do so was Prufrock [38]. Similar to Ivor, Prufrock provided a prover framework portable across both logics and language. The Prufrock experiment was successful in both, but proved too inefficient for a primary verification tool. Specifically, Prufrock did port to multiple syntaxes effectively – including the TPTP problem library [36] – making it an effective prototyping language. However, evaluation with respect to the TPTP library demonstrated inefficiencies leading to the decision to implement a smaller, more targeted proof tool.

With great naivety, I suggested what I thought would be a simple solution to all of the above problems: *why not port HOL Light to Haskell*? While the work began originally as a direct port of HOL Light, it became immediately clear that Haskell was a viable implementation language for a HOL system that was worthy of more in depth exploration. What was found was that many of Haskell's language features, specifically its laziness, purity, and advanced type system, promoted alternative implementation techniques compared to more traditional implementations utilizing the ML family of languages. The results of these efforts is HaskHOL, a Haskell hosted domain specific language for HOL theorem proving. While originally designed to help a single research group, HaskHOL has since grown to become a platform for exploring the challenges and benefits of implementing a HOL system in Haskell with the goal of contributing to a much wider audience than the single research group it was originally designed to help.

1.4 Organization

The organization of this thesis is as follows:

- Chapter 2 – A preliminary introduction to Higher-Order Logic provided to better understand this thesis.
- Chapter 3 – A preliminary introduction to Haskell provided to better understand this thesis.
- Chapter 4 – A high-level explanation of the primitive implementation techniques of HaskHOL.
- Chapter 5 – An exploration of the use of HaskHOL to implement an interactive theorem prover
- Chapter 6 – An evaluation of HaskHOL's performance.

Chapter 2

Higher-Order Logic

Higher-order logic (HOL) [14] is a logical formalism for theorem proving based on Robin Milner's Logic of Computable Functions (LCF) [13]. As its name implies, HOL separates itself from LCF by providing the ability to reason about higher-order predicates. What is shared, though, is an implementation technique that focuses on a small, trusted logical kernel from which more advanced reasoning features are bootstrapped from. Known colloquially as the "LCF-style," this method reduces the code base that must be checked for soundness and completeness, ultimately making verification of the entire system simpler.

2.1 Terms and Types

The primitive term language of any HOL implementation is a typed lambda calculus [17, 29, 33], whose grammar is shown below:

$$\begin{aligned} \text{Type} ::= \;& x \;\mid\; (c\ [\text{Type}]) \\ \text{Term} ::= \;& x : \text{Type} \;\mid\; c : \text{Type} \;\mid\; (\text{Term Term}) \;\mid\; (\lambda\, x\, .\ \text{Term}) \end{aligned}$$

Types can be either variables or applications of a list of types to a type constant while terms can be either typed variables, typed constants, combinations of two terms, or abstractions of a variable over a term. Some HOL systems supplement this grammar with additional term constructors; others instead rely on HOL's ability to define new constants and definitions in terms of the existing constructors.

In either case, writing terms in HOL should feel immediately comfortable for those familiar with the lambda calculus, especially those proficient in functional programming. For example, take the polymorphic constant to represent a test for equality between two terms, $= : a \to a \to \mathit{bool}$. Assuming that we have existing representations for our types, `aTy` and `boolTy` respectively, a term expressing equality between two boolean terms, `x` and `y` can be expressed as follows:

```
(("=":("fun" [aTy, ("fun" [aTy, boolTy]])) x:boolTy) y:boolTy)
```

At this point it is painfully clear that writing any significantly large terms using the primitive constructors directly is more of a burden than it's worth. For this reason, most HOL implementations include a parser to allow terms to be written in a more human friendly style that more closely resembles traditional logic notation. The implementation details of the parser for HaskHOL will be explained in Chapter 4, but in brief the HOL kernel provides the capacity to extend the parser with definitions for new types of a specified arity and new constants of a specified type, associativity, and precedence level. This allows us to write terms such as $a : \mathit{bool} = (x \wedge y \Rightarrow z)$ directly instead of worrying about how to glue together all of the necessary constructors. This relaxed notation is what will be used for the remainder of the HOL background section.

2.2 Theorems

The principle logical data type of HOL is a theorem, usually expressed with the notation $a_1, ..., a_n \vdash c$. Here the term c represents the conclusion of the theorem and is used to state what we are trying to show is valid. The list of terms $a_1, .., a_n$ represent the assumptions of the theorem and are used to state what must be true in order for the conclusion to also be true. Obviously, given that we care about the truth of these terms, all assumptions and the conclusion must be propositions, terms of type boolean. For example, the theorem $x \wedge y \vdash x$ states that a variable, x, is true under the assumption of the conjunction of x and y. A theorem with an empty assumption list, such as $\vdash x \vee \neg x$, reflects a conclusion that is always true, also known as a tautology.

Of important note is the fact that theorems cannot be manually constructed like terms can. Theorems can only come to existence through the application of one of the HOL system's kernel functions, typically the primitive inference rules. It is through this restriction that the HOL system can maintain its argument that if the kernel is sound and complete then the rest of the tool is too. The primitive inference rules of HaskHOL will be discussed in more detail in Section 2.3.

The kernel does provide an alternative method of constructing theorems without application of primitive inference rules by allowing the user to introduce new axioms into the proof theory. To construct a new axiom the user supplies a proposition and the system returns a new theorem with an empty assumption list and with that proposition as the conclusion, effectively accepting that term as true without any burden or guarantee of proof. Clearly this feature can be used to introduce inconsistent theorems into the proof theory, breaking the soundness for the rest of the proof tree. For this reason most HOL systems warn the user when axioms are introduced and some even prevent then from being accepted when the system is run at a higher trust level.

2.3 Primitive Inference Rules

HaskHOL was originally born as a direct port of HOL Light to Haskell, and as such, shares the same ten primitive inference rules [21]. Among them are two rules for basic equality reasoning, two rules for congruence of combinations and abstractions, a beta reduction rule, three rules for deduction of new theorems from existing ones, and two rules for instantiation of type and term variables. Each rule's logical semantics will be presented along with a brief discussion and small example. For these examples, the syntax ⇝ will be used to indicate the evaluation of a rule and its arguments to a theorem value.

2.3.1 Equality Rules

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

This rule provides reflexivity of terms. It takes as input a term and returns a theorem proving that the term is equal to itself. It has no failure conditions.

Example:

```
REFL x ⇝ ⊢ x = x
```

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

This rule provides transitivity of equality. It takes as input two theorems that have equations as their conclusions and returns a theorem proving the equation of the outside terms under the assumption of the union of the two original assumption lists. It fails when the middle terms are not alpha equivalent or when at least one of the theorems does not have an equation as its conclusion.

Example:

```
TRANS (x = y ⊢ x = y) (y = z ⊢ y = z) ⇝ x = y, y = z ⊢ x = z
```

2.3.2 Congruence Rules

$$\text{MK\_COMB}\frac{A1 \vdash f = g \qquad A2 \vdash x = y}{A1 \cup A2 \vdash f\ x = g\ y}$$

This rule provides congruence of term combination. It takes as input two theorems that have equations as their conclusions, the first of function terms and the second of argument terms, and returns a theorem proving the equation of the respective term combinations under the assumption of the union of the two original assumption lists. It fails when the types of the functions terms and argument terms don't agree, when the first theorem conclusion isn't an equation of function terms, or when at least one of the theorems does not have an equation as its conclusion.

Example:

```
MK_COMB (⊢ (λ x .  x) = (λ x .  x)) (x = y ⊢ x = y) ⇝
   x = y ⊢ (λ x .  x) x = (λ x .  x) y
```

$$\text{ABS}\frac{A \vdash t1 = t2 \qquad x\ not\ free\ in\ A}{A \vdash (\lambda x.t1) = (\lambda x.t2)}$$

This rule provides congruence of term abstraction. It takes as input a variable term and a theorem that has an equation as its conclusion and returns a theorem proving the equation of the abstraction of the variable over the terms of the equation under the original assumptions. It fails when the variable term is free in the assumption list or if the conclusion of the theorem is not an equation.

Example:

`ABS x (⊢ x = x) ⇝ ⊢ (λ x .  x) = (λ x .  x)`

2.3.3 Beta Reduction Rule

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

This rule provides equality between a term and its form after beta reduction. It takes as input a combination term consisting of a function and an argument and returns a theorem proving the equation of the original term and its function body. It fails if the combination term is not a valid application or if the argument term is not equivalent to the abstracted term of the function.

Example:

`BETA_RULE ((λ x .  x ∧ y) x) ⇝ ⊢ (λ x .  x ∧ y) x = x ∧ y`

2.3.4 Deduction Rules

$$\text{ASSUME}\frac{t}{t \vdash t}$$

This rule provides the only mechanism in the kernel for adding assumptions between proof steps. It takes as input a term and returns a theorem proving that term under the assumption of itself. It fails if the term is not a proposition.

Example:

`ASSUME (x ∧ y) ⇝ x ∧ y ⊢ x ∧ y`

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

This rule provides modus ponens reasoning for equality. It takes as input two theorems, the first with an equation of terms and the second with a conclusion of the first term from the equation, and returns a theorem proving the second term under the union of the two original assumption lists. It fails if the terms of the two theorems do not agree or if the conclusion of the first theorem is not an equation.

Example:

`EQ_MP (x = y ⊢ x = y) (x ⊢ x) ⇝ x = y, x ⊢ y`

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

This rule provides the only mechanism in the kernel for removing assumptions between proof steps. It takes as input two theorems and returns a theorem proving an equation between their conclusions under the assumption of the union of the first assumption list minus the second term and the second assumption list minus the first term.

Example:

`DEDUCT_ANTISYM_RULE (x ⊢ x) (x ⊢ x) ⇝ ⊢ x = x`

2.3.5 Instantiation Rules

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

This rule provides instantiation of type variables. It takes as input a type environment of type variable and type pairs and a theorem and returns a theorem identical to the original one where all type variables in the assumption list and conclusion are replaced with the associated type from the environment.

Example:

`INST_TYPE [(A, bool)] (x:A = y:A` $\vdash$ `x:A = y:A)` $\leadsto$

`x:bool = y:bool` $\vdash$ `x:bool = x:bool`

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

This rule provides instantiation of term variables. It takes as input a term environment of term variables and term pairs and a a theorem and returns a theorem identical to the original one where all term variables in the assumption list and conclusion are replaced with the associated term from the environment.

Example:

`INST [(x, z), (y, z)] (x = y` $\vdash$ `x = y)` $\leadsto$ `z = z` $\vdash$ `z = z`

2.4 Forward Proof

A basic proof in HOL is conducted through repeated applications of the primitive inference rules. This is referred to as *forward proof* because the user starts with an empty theory and proceeds forward, sequentially building up new theorems until the proof is complete. Notice that this proof technique suffers from the same issue that was discussed regarding the construction of terms; using the primitive constructors of a data type is unnecessarily burdensome. The answer to this problem, as it was for terms, is to provide more advanced functionality built on the primitive constructors that simplifies the user's interactions. This section will discuss two major advanced functions of forward proof: derived rules and conversions.

2.4.1 Derived Rules

Take, for example, a proof of $\Gamma \vdash r = l$ from the hypothesis $\Gamma \vdash l = r$:

1. $\Gamma \vdash l = r$ [Hypothesis]
2. $\vdash (=) = (=)$ [Reflexivity applied to $(=)$]
3. $\Gamma \vdash ((=)l) = ((=)r)$ [Congruence of Combs. applied to 2 and 1]
4. $\vdash l = l$ [Reflexivity applied to l]
5. $\Gamma \vdash (l = l) = (r = l)$ [Congruence of Combs. applied to 3 and 4]
6. $\Gamma \vdash r = l$ [Modus Ponens of Equality applied to 5 and 4]

Note that as long as the hypothesis is of the form $\Gamma \vdash l = r$ this sequence will always return a valid proof. In general, a sequence that returns a valid proof from provided hypotheses is referred to as a *derived rule.* This derived rule just happens to represent a major equality rule missing from our logical kernel, symmetry of equations.

Lines 1 through 3 can also be abstracted out to a derived rule if we generalize it to the following form:

1. $\Gamma \vdash x = y$ [Hypothesis]
2. $\vdash f = f$ [Reflexivity applied to f]
3. $\Gamma \vdash fx = fg$ [Congruence of Combs. applied to 2 and 1]

This derived rule, `AP_TERM`, is used to apply a function to both sides of an equational theorem as long as the types align. This can be used to simplify our derived rule for symmetry, displaying the real benefit of derived rules:

1. $\Gamma \vdash l = r$ [Hypothesis]
2. $\Gamma \vdash ((=)l) = ((=)r)$ [Application of term (=) to 1]
3. $\vdash l = l$ [Reflexivity applied to l]
4. $\Gamma \vdash (l = l) = (r = l)$ [Congruence of Combs. applied to 2 and 3]
5. $\Gamma \vdash r = l$ [Modus Ponens of Equality applied to 4 and 3]

Notice that we could also use a similar derived rule, `AP_THM`, to apply an argument to both sides of an equational theorem to again abstract away lines 3 and 4. However, notice that line 5 requires the theorem built in line 3, causing us to repeat the call to `REFL` if we use the derived rule. Repetition of work is something to be careful to avoid when building or using derived rules because the inefficiency will replicate every time that rule is used. However, for one off proofs the benefits of the added clarity and brevity that derived rules provide may outweigh the cost of the inefficiency.

2.4.2 Conversions

One very important class of derived rules is conversions. A *conversion* is a rule that maps a term to a theorem that concludes the equation of that term and a new term. One example that should be familiar to all of those who have worked with the lambda calculus before is beta conversion:

$$\text{BETA\_CONV}\frac{(\lambda x.u)\ v}{\vdash (\lambda x.u)\ v = u[v/x]}$$

We have already seen the simplest case of beta conversion, where the bound variable is equal to the argument, which was captured with the primitive beta reduction rule that was described in Section 2.3. The `BETA_CONV` conversion leverages the primitive beta rule in combination with the primitive term instantiation rule to handle the other cases.

In general, conversions are used to generate equational theorems to justify the replacement of terms with equivalent ones. Sometimes these theorems are used directly, such as in the implementation of HOL's rewriting tools. Other times, however, it would be nice to skip the theorem step and automate the replacement of the term in an existing theorem. HOL provides this functionality with the `CONV_RULE` derived rule:

$$\text{CONV\_RULE}\frac{c\ where\ c\ t \rightsquigarrow A' \vdash t' \qquad A \vdash t}{A \cup A' \vdash t'}$$

Example:

`CONV_RULE BETA_CONV` $((\lambda\ \texttt{x}\ .\ \ \texttt{x} \wedge \texttt{y})\ \texttt{z} \vdash (\lambda\ \texttt{x}\ .\ \ \texttt{x} \wedge \texttt{y})\ \texttt{z}) \rightsquigarrow$

$(\lambda\ \texttt{x}\ .\ \ \texttt{x} \wedge \texttt{y})\ \texttt{z} \vdash \texttt{z} \wedge \texttt{y}$

HOL also provides the functionality to construct new conversions out of existing ones using conversion combining operators called conversionals. For example, what if we want to prove an equivalence of a term of form $(\lambda x_1....x_n.u)\ v_1...v_n$ with its reduced form $u[v_1/x_1...v_n/x_n]$. We could manually apply `BETA_CONV` n times over the operator of the application, chaining the equivalences together with `TRANS`, but it would be nicer to automate this process. The conversion library provides two terminal conversions, `NO_CONV` and `ALL_CONV`, roughly equivalent to failure and identity functions respectively, and several conversionals for sequencing and trying conversions, such as `THENC`, `ORELSEC`, and `REPEATC`. Also provided are subterm conversionals that target a conversion to a specific portion of a term, such as `RATOR_CONV` and `RAND_CONV` which apply a conversion to the operator or operand of a combination accordingly.

These conversionals allow us to specify the new conversion, `BETAS_CONV` that was described above. All that is necessary is a basic inspection on the structure of the term we are applying the conversion to. If the term is a single application then we simply call the original conversion: `BETA_CONV`. If the term is a nested application then we build a new conversion consisting of a recursive call followed by a call to `BETA_CONV` and apply that the operator of the term: `RATOR_CONV (BETAS_CONV THENC BETA_CONV)`.

2.5 Backwards Proof

For many problems forward proof is too primitive to feel natural or appropriate. An alternative, much more robust, proof technique is goal directed, or backwards, proof based on the notion of tactics, an invention of Milner's from the 1970s [15]. Whereas in forward proof we start with nothing and work towards our goal theorem, in backwards proof we start at our goal and work in reverse, asking ourselves what we need to prove to get to that point. At that point we have effectively split the problem into subproofs that can be examined and solved independently of each other utilizing the same technique. This process then repeats until we no longer have any subproof obligations to complete, the final result being a proof of how the summation of all of our subproofs is sufficient to prove our original goal theorem.

2.5.1 Tactics

Tactics, in short, provide the plumbing to logically formalize the process explained above. More specifically, a tactic is responsible for two things, defining how a goal is split into subgoals and keeping track of the justification for why solving the subgoals solves the original goal. The collection of the new subgoals and the justification is tracked along with some other plumbing in what is called the goal state. This makes a tactic a function of type `Goal` → `GoalState`.

As an example, suppose that we wanted to prove the goal $A \vdash X \iff Y$. We know that $X \iff Y$ is logically equivalent to $(X \Rightarrow Y) \wedge (Y \Rightarrow X)$, so it would appear to be sufficient to prove the subgoals $A \vdash X \Rightarrow Y$ and $A \vdash Y \Rightarrow X$ in order to prove the original goal. In fact, this reasoning is exactly what is capture by the implication anti symmetry rule:

$$\text{IMP\_ANTISYM\_RULE}\frac{A1 \vdash t1 \Rightarrow t2 \qquad A2 \vdash t2 \Rightarrow t1}{A1 \cup A2 \vdash t1 \iff t2}$$

The tactic for the above example, therefore, must be able to deconstruct the original goal to build the two new subgoals as well as provide a justification using `IMP_ANTISYM_RULE`. The tactic in question, `EQ_TAC`, already exists as part of most standard HOL system tactic libraries and is shown below:

```
EQ_TAC (asl, w) =
  let (l, r) = dest_eq w
      tm1    = mk_imp l r
      tm2    = mk_imp r l in
    ([(asl, tm1), (asl, tm2)],
     \ [th1, th2] -> IMP_ANTISYM_RULE th1 th2)
```

A goal is roughly the same type as a theorem, consisting of both an assumption list of terms and a conclusion term. Because of this we can use the same term destruction and construction functions that we do for derived rules, making the construction of the subgoal terms trivial in most cases. On a similar note, the justification constructed by a tactic is just a derived rule that provides the forwarding reasoning from the solution of the subgoals to the solution of the original goal, hence why there is a one-to-one relationship between subgoals and argument theorems to the justification.

Note that EQ_TAC does not actually solve the example goal, it just gets us one step closer. In fact, rarely will a single application of a tactic solve a goal; much more common is the case where a sequence of tactics must be applied to reach a solution. For this reason, tactics have a rich tactical language that is very similar to the conversional language discussed in Section 2.4.2. Also provided is a tactic constructor, CONV_TAC, that creates a tactic from a conversion. This allows very easy definition of new tactics from existing conversions, such as a tactic for beta reduction, BETA_TAC = CONV_TAC (REDEPTH_CONV BETA_CONV).

Chapter 3

Haskell

The previous chapter made numerous references to the lambda calculus and functional programming without concretizing the language we were working with. That language is Haskell, a pure, lazy, functional programming language known for its widespread adaptation in academia and borderline esoteric language features. When we say Haskell we are actually informally referring to the language supported by the Glasgow Haskell Compiler [1]; this covers the Haskell 2010 Standard [25] and numerous other language extensions that augment the syntactic and type systems. For the sake of brevity, the following sections assume a relatively solid level of understanding about functional programming; namely, features that are prevalent in most functional languages will not be explained. Instead, this chapter will focus on explaining the more advanced or novel features of Haskell that are used in the implementation of HaskHOL.

3.1 Type Classes

Most, if not all, functional languages admit parametric polymorphism through the use of either explicit or inferred universal quantifications in their type signatures; for example, `a -> a -> Bool`. On the other hand, the techniques for implementing ad-hoc polymorphism can vary widely from language to language, assuming that they admit it at all. The path that Haskell has taken is the use of *type classes*, a technique that allows the programmer to group related methods together and specify their implementations for a collection of types. The perennial example is a class framing an equality test for two expressions of the same type:

```
class MyEq a where
    eq :: a -> a -> Bool
```

This class can be instantiated for any type that satisfies the kind checking of `a` (`*` in this case). For example:

```
data NotQuiteNat = Zero | Succ NotQuiteNat

instance MyEq NotQuiteNat where
    Zero ‘eq‘ Zero     = True
    Succ x ‘eq‘ Succ y = x ‘eq‘ y
    _ ‘eq‘ _           = False
```

We can now use this newly defined `eq` method in other code we write; for example, checking if an expression is an element of a list.

```
myElem x xs = or $ map (eq x) xs
```

In the example above the `$` operator is used to evaluate the expression on its immediate right and then pass the result to the expression on its immediate left. This is a fantastic way to avoid needing to use parentheses and can lead to cleaner and clearer code. As such, it is an operator that I will use a significant amount from this point on.

When assigning this function a type, the first instinct is to use a parametrically polymorphic type of `a -> [a] -> Bool`, but recall that `eq` is only defined for types that belong to the `MyEq` class. In other words, parametric polymorphism is too strong to state what we want. Therefore, we need to weaken the type by adding the assumption of membership in the class to the type context, like so:

```
myElem :: MyEq a => a -> [a] -> Bool
```

The behavior of the `myElem` function is changed every time we define a new instance of the `MyEq` type class, providing behavior similar to function overloading. That is not to say that the value of type classes is limited to use in type signatures; they can be used in a variety of other ways as well.

They can be used in the contexts of instances to instantiate type constructors...

```
instance MyEq a => MyEq [a] where
    (x:xs) ‘eq‘ (y:ys) = x ‘eq‘ y && xs ‘eq‘ ys
```

In the contexts of classes to provide the notion of inheritance and class extension...

```
class MyEq a => MyOrd a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
```

Or in the contexts of data types to weaken the set of permissible of arguments...

```
data MyEq a => EqProof a = Refl a a
```

3.2 Monads

Perhaps the most famous, and arguably the most intimidating, type class in Haskell is the *Monad* class:

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

As mentioned previously, Haskell is a pure language in that it does not admit mutable variables or other side-effectful language features. In short, monads are how Haskell captures these computational effects without losing its purity. Before trying to explain how the above definition actually accomplishes that, it's beneficial to examine what the methods of the monad class actually do.

We look to the `Identity` monad as an example given its simplicity:

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

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

The definition of `Identity` uses Haskell's record syntax to provide both a constructor, `Identity :: a -> Identity a`, and a destructor, `runIdentity :: Identity a -> a`, succinctly. Additionally, because there is only one constructor, the `newtype` mechanism is used to avoid any overhead that would normally be incurred by using the `data` mechanism to define new types. Both of these are common tricks in Haskell, especially in the implementation of monadic types.

Looking at this definition it is clear that the purpose of `return` is to serve as a polymorphic "boxing" function for the `Monad` class, lifting a pure expression into any monadic container. The purpose of `>>=` is hopefully equally clear; it serves as a sequencing operator, applying the value of the first monadic computation to the functional, second argument. When this second argument is wrapped in a lambda expression, for example `return 4 >>= (\ x -> return $ x * x)`, it has the effect of binding the first argument to a name, hence the name of the `>>=` operator, bind. In the event that you do not care about the value of the first monadic computation, typical with "effect-only" computations, you can use the `>>` operator. This operator is simply an alias to the special case use of the bind operator, `x >>= \ _ -> y`.

This binding behavior is made even more clear in the sugared `do` notation for monadic expressions. In this notation, `x >>= (\ x' -> f x)` can be replaced with the following code:

```
do x' <- x
   f x'
```

Again, when you would like to ignore the value of a computation, all that is required in this syntax is to omit the sugared binding operator:

```
do prereq
   x' <- x
   f x'
```

It should be noted that if the value of the computation `prereq` is not of the type `()` then the compiler will produce a warning. In these cases it is preferable to be explicit that you are ignoring the value by binding it to a wild card:

```
do _  <- prereq
   x' <- x
   f x'
```

This alternative syntax makes it significantly easier to write complicated chains of monadic computations, and, as such, is the preferred way to write monadic code that cannot be succinctly expressed in a single line.

In addition to `do` notation, the base Haskell libraries include a robust `Monad` library that contains everything from monadic generalizations of list functions to functions designed to lift pure functions into monadic ones. Several items from this library will be used in the remainder of this chapter and in the main body of the thesis, however, they will not be discussed here for the sake of brevity. For more information about them, it would be best to consult either Philip Wadler's "Comprehending Monads" paper [37] or the Monad Transformer Library documentation[1]. That being said, what will be discussed in the following subsections are instances of the `Monad` class that are more interesting than `Identity`.

3.2.1 State Monad

The `State` monad is used to capture computations that require the notion of global state.

```
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
    return a = State $ \s -> (a, s)
    m >>= k  = State $ \s -> let
        (a, s’) = runState m s
        in runState (k a) s’
```

From this definition we can see that a stateful computation is represented as a function that takes an initial state value and returns a pairing of the computation value and final state value. Again, like the `Identity` monad, `State` is defined with a record type such that the expression `runState m init` means run the stateful computation `m` with initial state `init`. The bind operator is also written in a similar way, such that both the resultant state and value from the first monadic computation is threaded through to the second. As an interesting aside, `State`’s binding behavior makes its instance of the `>>` operator behave almost identically to the `;` operator from most imperative languages.

The previous definition on its own is not enough to truly represent stateful computations; obviously missing are the capabilities to access and modify the state. Without these, `State` is little more than a specialized version of the `Identity` monad. These methods are defined in an extension of the `Monad` class, `MonadState`.

[1] http://hackage.haskell.org/package/mtl-2.0.1.0

```
class (Monad m) => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()
```

Class extension is hopefully something that is familiar given its introduction earlier in this chapter. This is example is slightly different, though, in that the class takes multiple parameters and requires a functional dependency (`m -> s`. All that this extra information specifies is that `m` and `s` may vary independently and that `s` can be uniquely determined from `m`. This allows us to write a single instance that covers all possible types of `State`.

```
instance MonadState s (State s) where
    get   = State $ \s -> (s, s)
    put s = State $ \_ -> ((), s)
```

As one might be able to derive from the names, `get` returns the state as the value of a computation and `put` takes a state value as input and sets it as the state of a computation. These methods allow us to write stateful functions, like a fresh name generator:

```
freshName :: String -> State Int String
freshName base =
    do count <- get
       let name = base ++ show count
           next = succ count
       put next
       return name
```

This function depends on an integer counter being maintained in the state value to keep track of what the next value to append to the string is, thus allowing it to append the string representation of the counter, obtained via the `show` function, to a base string provided by the user before incrementing the counter and returning the resultant name. This way a guarantee can be made that any two strings generated by this function that are in the same `State` computation will be unique, even if they use the same base string. For example, the expression `evalState ( mapM freshName . take 5 $ repeat "abc")0` will return the list `["abc0","abc1" ,"abc2","abc3","abc4"]`.

Also defined for the `State` monad are a number of convenience functions that capture common uses and combinations of `runState`, `get`, and `put`. Examples include `evalState` and `execState` which look at only the resultant value or state of a computation respectively, and `modify` which accepts a state transforming function rather than making the user explicitly make calls to `get` and `put` to make state modifications. Again, please look to Haskell's Haddock documentation for more information about what the `State` library provides.

3.2.2 IO Monad

To borrow a tired expression usually reserved for talking about one's bed, the `IO` monad is "where the magic happens." Aptly named for input and output, the `IO` monad is used to captures computations that require communicating with sources outside of Haskell. These communications can be anything from printing to standard output to reading input from a file. In fact, the ongoing development of Haskell has led to a large number utilities not typically associated with input and output to also be included in the `IO` monad, such as a rich exception system and support for explicit memory reference management. Given this, rather than describe what the `IO` monad does, it is often easier to dismissively ask, "What doesn't it do?"

Notice that absent so far from the discussion of the `IO` monad is its instance of the `Monad` class. This is because the `IO` monad cannot actually be written in Haskell and instead is implemented as a collection of language primitives, such as `bindIO` and `returnIO`. The instance of the `Monad` class for `IO` simply calls these primitives directly, leaving any knowledge about how `IO` works as compiler magic.

We can reason about what the `IO` monad is and does, though, by describing it as an instance of the `State` monad where the state value is the "real world." Experienced Haskell users may recognize this for the little white lie that it is, however, it makes introductory discussion about `IO` possible. Under this assumption we can interact with the `IO` monad in every way we can with `State` with one major exception; there is no `runIO` function. The argument for this is that the only way to maintain type correctness and safety with effectful `IO` computations is to maintain a strict ordering of them; once you "escape" from the "real world," any guarantee about the ordering of these effects is lost. Therefore, the only way to run an `IO` computation is to bind it to a special function defined in your compiled program, `Main.main :: IO ()`, or to have it called directly or indirectly from some point in the `main` function.

3.2.3 Monad Transformers

Sometimes it is desirable to be able to combine the effects of two different monads. Rather than having to write a new monad that captures the combination of those effects, Haskell provides a *monad transformer* library. The idea behind a monad transformer is that rather than having run function that returns a pure value, like `runState` does, it will instead return a computation for that value in a different monad. To accomplish this, the old, non-transformer definition is modified to accept an additional parameter for this new monad return type, effectively wrapping the new monad transformer definition around the return monad. In this way, repeated applications of monad transformers builds up a stack of monads with the base of the stack being some non-transformer monad. Likewise, repeated applications of the appropriate transformer run functions peel layers off of the stack until the bottom is reached where either a non-transformer run function is called to return a pure value or an IO computation is reached that can be bound to or called from main.

To relate this concept to material we've already seen before, the definition for the State monad transformer is shown below:

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

instance (Monad m) => Monad (StateT s m) where
    return a = StateT $ \s -> return (a, s)
    m >>= k  = StateT $ \s -> do
        ~(a, s') <- runStateT m s
        runStateT (k a) s'
    fail str = StateT $ \_ -> fail str
```

Note that the beauty of this implementation is that it is indifferent to what monad StateT is stacking itself upon. As long as the type m is an instance of the Monad class, then pure values can be boxed with return and let bindings can be replaced with monadic bindings without any other knowledge about what lies beneath StateT on the monad stack.

Defining get and put are equally easy, at least when StateT is at the top of the stack:

```
instance (Monad m) => MonadState s (StateT s m) where
    get   = StateT $ \s -> return (s, s)
    put s = StateT $ \_ -> return ((), s)
```

But what if we want to use other monad's methods when StateT is at the top of the stack, for example any IO computation? In effect, what we would like to do is reach into an arbitrary point of the stack, grab a method, and lift it to the top so that we can use it. We can do this with another specialized monad type class, MonadTrans, whose purpose is to convert a monadic computation into an equivalent computation contained within a given transformer type.

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a

instance MonadTrans (StateT s) where
    lift m = StateT $ \s -> do
        a <- m
        return (a, s)
```

Recall, though, that IO is somewhat of a magic monad in that it is entirely defined by compiler primitives. It makes sense then that it has its own lifting function and type class:

```
class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a

instance MonadIO IO where
    liftIO = id
```

This lifting type class lets us to write code similar to the following example. In this case, we are building a monad stack that is two layers deep, StateT on top of IO, so that we can grab the global state value and then print it to standard output. We can then use evalStateT to peel the StateT layer off, leaving an IO computation that we can bind to main.

```
printState :: StateT Int IO ()
printState =
    do st <- get
       liftIO . putStrLn $ show st

main :: IO ()
main = evalStateT printState 10
```

It should be noted that monad transformers are so powerful that they have subsumed most of the standard monads from the old versions of Haskell's base libraries. For example, there is no longer a definition of State like we saw previously, instead `State s` is defined as an alias to `StateT s Identity`.

3.3 Dynamics

Dynamic typing is an incredibly powerful and useful concept in practice because it admits numerous other features, such as heterogeneous containers. Even though Haskell is a statically typed language, it is possible to provide a basic dynamic typing interface using the techniques covered in Section 3.1. The standard implementation of *dynamics* in Haskell is contained in the `Data.Dynamic` library and is implemented using the `Typeable` type class.

The purpose of the `Typeable` type class is to provide a mechanism to reify any type to a universal type representation. This type representation can be constructed, destructed, and compared just like any other value in Haskell. The definition for the `Typeable` type class is shown below:

```
class Typeable a where
  typeOf :: a -> TypeRep
```

An important fact that can't be seen from this definition is that the `typeOf` method ignores the value passed to it. This makes it possible to accept an undefined argument with a scoped type variable to reify any type without requiring that a valid value of that type be accessible at that point in time. What can be seen from the definition is that `typeOf` only accepts arguments of a monomorphic type. To get the type representation of a polymorphic value it must first be ascribed a monomorphic type, for example `typeOf (undefined :: Bool)`.

Most commonly, `typeOf` is used for the purpose of comparing two different type representations before a cast or coercion is made. This is precisely how dynamics are implemented in `Data.Dynamic`. When a static value is injected into a dynamic container, the type representation is stored along with a coerced version of the object:

```
toDyn :: Typeable a => a -> Dynamic
toDyn v = Dynamic (typeOf v) (unsafeCoerce v)
```

When converting back to a static value, the type representation of a default value is checked against the stored type representation; if the representations are equal then the stored object is again coerced back to its static version, otherwise the default value is returned:

```
fromDyn :: Typeable a => Dynamic -> a -> a
fromDyn (Dynamic t v) def
  | typeOf def == t = unsafeCoerce v
  | otherwise       = def
```

As mentioned, this primitive dynamic type interface makes it possible to create containers like a heterogeneous list:

```
type HetList = [Dynamic]

hetCons :: Typeable a => a -> HetList -> HetList
hetCons x xs = (toDyn x) : xs

example = (1::Integer) ‘hetCons‘ (True ‘hetCons‘ [])
```

3.4 Template Haskell

One of the most powerful extensions that GHC provides is *Template Haskell*, a compile-time metaprogramming library [32]. The goal of Template Haskell is to provide the ability to reify concrete Haskell syntax to an abstract syntax tree that itself can be modified with Haskell, all in a type safe way. This is accomplished through the two major syntactic extensions provided by Template Haskell, splicing and quoting.

Splicing is used to take an abstract syntax tree prepared by Template Haskell and ”splice” it back in with regular Haskell code, effectively providing a translation from the abstract syntax to the concrete syntax. The splice operator, `$(X)` will accept any abstract syntax for an expression, type, or list of declarations in place of `X`. For example, the splice `$(litE (IntegerL (1 + 2)))` will return the concrete Haskell expression `3`.

Quoting is used to take concrete Haskell syntax and convert it to its abstract representation. The quote operators, `[e|...|], [t|...|], [d|...|], and [p|...|]`, accept expressions, types, list of declarations, or patterns accordingly. The operator `[|...|]` is provided as a shorter alternative for the expression quoter since it is the most commonly used. The framework for the abstract syntax for the previous example was derived by quoting an integer expression, `[| 1 |]`, which provides the syntax `LitE (IntegerL 1)`.

Note the change in case between `litE` and `LitE`. This difference has to do with a detail that has been ignored in the explanation so far; Template Haskell has its own computation monad, `Q`, that is closely related to the `IO` monad. As far as the two main operations are concerned, the values returned from quoting are contained within the `Q` monad and arguments to splicing must be `Q` monad computations. This relationship allows quoting to be nested within splicing, or vice versa.

The ultimate goal of Template Haskell is to expose functionality that would not be possible without the metaprogramming paradigm. One basic example is implementing a generic selection function that works for tuples of all sizes. In ordinary Haskell it is impossible to write a single function that works for tuples of all sizes because the size of a tuple directly dictates its type; this means that that you must write a separate selection function for each tuple size. Rather than do this, it is a much cleaner solution to implement a single Template Haskell function that constructs the appropriate selection function at compile time. The code to do this is shown below:

```
sel i n = lamE [pat] rhs
    where pat = tupP (map varP as)
          rhs = varE (as !! (i - 1))
          as  = [ mkName $ "a" ++ show j | j <- [1..n] ]
```

While this may look intimidating to those who do not recognize the internal constructors for Haskell, it is actually a relatively simple function. The `sel` function constructs a list of names, `as`, that is equal in length to the size of the tuple, `n`. From there a pattern for a tuple of that size is constructed, `pat`, and the name at the index `i` in that pattern is identified, `rhs`. Then the function returns a new function that accepts a tuple that satisfies this pattern as an argument and returns the indexed value. This allows the user to write expressions like `$(sel 2 3)(1, 2, 3)` and `$(sel 1 2)(1, 2)` after having only written the `sel` function once.

Chapter 4

HaskHOL

As alluded to in the title, HaskHOL is implemented as a Haskell hosted DSL. The primary technique for developing an embedded DSL in Haskell is well known [11]. The process begins by identifying the functionality of the primitive combinators of the DSL and unifying it around a set of abstract data types. From there, a monadic computation model is identified and a structure around it is implemented that provides a usable interface and types for the programmer. The primitive combinators are then implemented using this structure and the standard monadic techniques of Haskell.

Those familiar with the LCF theorem prover style may immediately recognize how analogous it is to the process just described. The HOL family of formal systems, having their roots in the LCF, follows this style by implementing a small logical kernel that advanced features are bootstrapped from to reduce the burden of proof of soundness and completeness. This commonality is what makes the embedded DSL approach such a natural and obvious choice for the implementation HaskHOL.

Because HaskHOL started as a direct port of HOL Light, it maintains roughly the same logical kernel. The ten primitive inference rules and definitional extension functions map directly to the primitive combinators of HaskHOL, with the primitive types also translating directly. In fact, largely the only difference between the HaskHOL and HOL Light kernels is that HOL Light eschews the monadic computation model in favor of the effectful features of OCaml, like global references, as is standard when using impure languages. The following sections describe the implementation of the HaskHOL kernel, explaining the ramifications of using the monadic implementation style.

4.1 Types, Terms, and Theorems

At the lowest level HaskHOL is based on a sound and complete set of ten primitive inference rules that, when applied sequentially, construct a theorem that serves as proof of a conclusion term. These terms are based on a typed version of Church's λ-calculus, as explained in Section 2.1. As might be expected, HaskHOL implements these primitive data types using Haskell's abstract data types:

```
                            data HOLTerm
data HOLType                    = Var   String  HOLType
    = TyVar String              | Const String  HOLType
    | TyApp String [HOLType]    | Comb  HOLTerm HOLTerm
    deriving (Eq, Ord)          | Abs   HOLTerm HOLTerm
                                deriving (Eq, Ord)
```

Likewise, the theorem data type can be encoded as a constructor taking two arguments, an assumption list of terms and a conclusion term:

```
data Theorem = Thm [HOLTerm] HOLTerm deriving (Eq, Ord)
```

There is nothing particularly novel about the implementation of these primitive types, HOL Light does so in exactly the same way. However, the data types are worth introducing for those unfamiliar with the logical system and to help clarify the types of other features later on.

4.2 The HOL Monad

The purpose of a DSL's monad is to provide a model of computation for its primitive combinators. As Haskell is a pure language, we cannot rely on the use of destructive side effects to achieve this goal in the style of other HOL systems. While there are alternative methods and techniques available, the use of monads to model effects is standard practice among Haskell programmers. In the case of HaskHOL there are primarily two such effects to be concerned about: extension of the proof context and proper exception handling. Those familiar with earlier versions of HaskHOL [5] will recognize that the majority of refinements made are directly related to supporting these effects in more trusted and efficient ways.

Monad transformers, as discussed in Section 3.2.3, are used to combine the `State` and `IO` monads to provide the desired effects. The resultant type for the `HOL` monad, along with its associated run functions, is shown below:

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

runHOL :: HOL a -> IO a
runHOL (HOL a) = evalStateT a initCtxt

runHOLCtxt :: HOLContext -> HOL a -> IO (a, HOLContext)
runHOLCtxt ctxt (HOL a) = runStateT a ctxt
```

The `runHOL` function is used to evaluate a `HOL` monad computation using the predefined initial proof context. The proof context will be explained in more detail in the next subsection, but for now it is sufficient to understand that it roughly represents the notion of the current working theory for a HOL system with the initial context specifying the base theory. Comparatively, the `runHOLCtxt` function is used to run a HOL monad computation using a context provided by the user. It should be noted that this run function has the potential to introduce unsoundness to the system if the supplied context is invalid or poorly constructed. That being said, the power that is provided by the runHOLCtxt function is necessary in certain cases, such as the suspension and resumption of a monadic computation during interactive proof. It should be noted that this problem is analogous to the issue exposed by the use of the put method from the State monad, discussion of which will happen in the next subsection.

4.3 Extensibility

Extensibility is an important feature because it allows the users to define their own types, terms, axioms, etc. HaskHOL carries this information in a context threaded through computations using the State monad, as explained in Section 3.2.1. The context is implemented as a record type as shown below:

```
data HOLContext =
    Ctxt { tmcounter     :: !Int
         , typeConstants :: ![(String, Int)]
         , termConstants :: ![(String, HOLType)]
         , axs           :: ![Theorem]
         , defns         :: ![Theorem]
         , loadedLibs    :: ![String]
         , debug         :: !Bool
         , extState      :: !(Map String ExtState)
         }
```

Any data we need to store in the future is encoded directly as its own field; for example, axioms are stored in the axs :: ![Theorem] field. It is not uncommon, though, for libraries and prover extensions beyond the kernel to require the ability to store their own data. Since information about this data may not be known at the time that HOLContext is defined, the context itself must be extensible.

This contextual extensibility is provided using a technique similar to dynamics, as explained in Section 3.3. The principle difference in HaskHOL's approach is that the extensible data type has a dependency on the `Typeable` class rather than having it store an explicit representation of the original type. In a sense, this is an unboxed version of Haskell's standard `Dynamic` type, with the advantage being that it allows for additional methods to be defined for all dynamic types in a superclass of `Typeable`. This is a common technique among Haskell programmers and is present in several large libraries, such as XMonad [34].

Shown below is the data definition and associated type class for this dynamic data type.

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

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

Defining an instance of this type is trivial thanks largely in part to the GHC specific extension `DeriveDataTypeable` which allows for automatic derivation of `Typeable` instances. This is demonstrated in the example below, a counter used to generate fresh type variable names during term elaboration.

```
newtype TyCounter = TyCounter Int deriving Typeable
instance ExtClass TyCounter where
    initValue = TyCounter 0
```

Note that instead of simply defining an instance of `ExtClass` for `Int`, the integer value is instead wrapped in a `newtype` declaration first; the reason for this twofold. First, it provides documentation at the type level to indicate what that integer value is used for. The second reason has to do with how these dynamic types are contained, as a map of dynamic types indexed by a serialization of their static type. Without first wrapping types in distinct `newtype` declarations it would be impossible to store more than one value of the same base type without overwriting old data. Types could alternatively be wrapped in data declarations, however, in cases like this it is more common to use newtype in combination with GHC's GeneralizedNewtypeDeriving extension for performance reasons.

Storing a value in the extensible context is as easy as serializing its type representation to a String using show and then inserting the value into the extState map using that string as the index. The code to do this is shown below:

```
changeExt :: (Map String ExtState -> Map String ExtState) -> HOL ()
changeExt f = modify $ \ st -> st { extState = f (extState st) }

putExt :: (ExtClass a) => a -> HOL ()
putExt val = changeExt . insert (show . typeOf $ val) $ ExtState val
```

The putExt function does not check to see if a value of the same type already exists before inserting the new value. Implementing this check would not make much sense, given that there is always a value for any valid extension type, even if it is only initValue defined in the ExtClass type class. In this sense, putExt operates very similarly to the State monad's put method. This presents some problems that will be discussed later in this section, but most of these dangers can be mitigated by preventing the user from manually constructing their own values. The easiest way to do this has already been discussed above; wrap extensible data types in a newtype declaration, hiding the internal constructor beyond the module where it was defined. It will still be possible to misuse or abuse putExt within the module where the data type was defined, so the burden to ensure correctness falls on the author of said module.

Retrieving a value from the extensible context is not quite as easy because we have no value to serialize a type from, therefore, we have no way to index into the map. To get around this, we use GHC's ScopedTypeVariables extension to universally quantify the return type of the function so that we can create an undefined, dummy value of the same type. From there the type of the dummy value is serialized and used as the lookup key. As mentioned above, In the event that no value for that type is found in the map the initial value specified in the type class instance, `initValue`, is used. If a value is found the `ExtState` data wrapper which provides the dynamic behavior must be stripped away. This process prevents inference of the equivalence between the original type and the desired return type, so the unboxed value must be cast. The casting function used is the type safe version provided by the `Typeable` library which allows an error to be thrown in the event that casting were to fail. Again, the code to do this is shown below:

```
getExt :: forall a. (ExtClass a) => HOL a
getExt =
    do v <- gets $ lookup (show . typeOf $
                   (undefined :: a)) . extState
       case v of
         Just (ExtState val) ->
             case cast val of
               Just b -> return b
               Nothing -> throwError $ HOLException "getExt"
         Nothing -> return initValue
```

There are issues with using methods that operate like `put` from the `State` monad. In general, exposing these functions directly provides the user with the capability to invalidate the context by supplying a new context that was not properly constructed. This is roughly the same problem that is introduced by `runHOLCtxt` as described in the last section given that `\ x -> liftIO . runHOLCtxt x $ return ()` ≈ `put`. To prevent this, HaskHOL wraps the `StateT` transformer in a `newtype` and avoids deriving instances of the `MonadState` and `MonadIO` classes. Instead, the necessary methods of `MonadState` and `MonadIO` are defined external to the class, forgoing the provided polymorphism in favor of more direct control over how the methods are exported. For example, the new definitions of put and liftIO are shown below:

```
put :: HOLContext -> HOL ()
put ctxt = HOL . StateT $ \ _ -> return ((), ctxt)

liftIO :: IO a -> HOL a
liftIO m = HOL . StateT $ \ s -> do a <- m
                                    return (a, s)
```

The Haskell module system is used to hide these methods and the field constructors of the HOLContext record type outside of the kernel. Again, the goal of these actions is to prevent the user from being able to manually construct values, incorrect or not. This is analogous to the approach of HOL Light where the OCaml module system is used to hide the global references and expose only the definitional extension functions. HaskHOL also hides the internal constructor of the HOL monad type to prevent users from defining their own HOL values as well. In addition to providing the guarantee that hidden methods cannot be redefined later on, hiding the constructor makes a type level guarantee that any value with the HOL type that is constructed safely, that is to say without the use of functions like unsafePerformIO, can be reduced to a combination of only primitive kernel combinators and pure code. In essence, this guarantee goes above and beyond what HOL Light, or any system built upon an impure language, can provide by limiting the possible effects in a proof to those enumerated in the kernel.

4.4 Exception Handling

Exception handling in HaskHOL is performed by using the IO monad and the associated Control.Exception library. Haskell does provide an Error monad as an alternative way to implement exception handling, however, it is little more than a limited interface built around the `Either` data type. Using the `Control.Exception` library provides numerous advantages over the `Error` monad, including a richer interface, the ability to throw errors in pure as well as monadic code, and in many cases a performance boost. What is shared in common are `throwError` and `catchError` methods that operate similar to those in any other language.

HaskHOL follows this trend by basing its exception handling on three main functions: a way to throw errors in pure code, a way throw errors in `HOL` monadic code, and a way to catch both sets of errors in `HOL` monadic code; the code to implement these functions is shown below:

```
throwPureError :: Exception e => e -> a
throwPureError = throw

throwError :: Exception e => e -> HOL a
throwError = liftIO . throwIO

catchError :: Exception e => HOL a -> (e -> HOL a) -> HOL a
catchError job errcase =
    do ctxt <- get
       (a, s') <- liftIO $ runHOLCtxt ctxt job `E.catch` \ e ->
                            runHOLCtxt ctxt (errcase e)
       put s'
       return a
```

Even though errors can be thrown in pure code, suspiciously absent is the ability to catch them in pure code. This is a consequence of inheriting Haskell's exception handling methods that dictate that errors can only be caught in the `IO` monad. This restriction complicates the implementation of `catchError` in that `HOL` computations must be run to the `IO` level for the errors to be handled. When forcing computations to run with `runHOLCtxt`, care must be taken to appropriately handle the input and output proof contexts. Incidentally, the passing around of the proof context is inversely proportional to the performance of the `catchError` function;

the smaller the context the faster the function runs. A large amount of data is currently stored in the proof context so at present this is somewhat of an issue. Plans to alleviate this will be detailed in the Future Work section near the end of this thesis.

HaskHOL also provides several common logical operators related to handling errors, such as the alternate operator, `<|>`. This operator is used in cases where the user would like to run a second `HOL` computation in the event that the first one fails. The implementation of this operator could follow similarly to the implementation of `catchError`, utilizing lifting and projecting functions. Easier yet is to define the operator using `catchError` itself, as shown below.

```
(<|>) :: forall a. HOL a -> HOL a -> HOL a
job <|> errcase = job ‘catchError‘ ignore
    where ignore :: SomeException -> HOL a
          ignore _ = errcase
```

The only challenge with this implementation path is that the type of `catchError` dictates a constraint about what type of exception must be found on the right hand side. Since the alternate operator ignores the exception completely, this creates an unresolvable ambiguity for the type checker. The solution is to make the type of the ignored error explicit. Simply adding the same constraint to this functions type will not help given that you cannot show equivalence between the two. Scoped type variables can sometimes be used to solve this problem, but not in this case given that the type of the error is not present in the top level type signature. Instead, we rely on a dynamic type, `SomeException`, from the `Control.Exception` library to capture all possible exception types without having to introduce a constraint. This type, `SomeException`, works almost identically to our `ExtState` type from the previous section that is used to capture all possible extensible data types.

At first it appears that all that is happening with HaskHOL's exception handling is that it is being shifted into the implementation of the system, slightly edging the point of trust away from the host language's run time system. The real benefit, though, comes from implementing exceptions in a monadic model. Impure languages, like members of the ML family, have either undefined or counter intuitive evaluation orders which can greatly affect the ordering of effects, like throwing exceptions. In a monadic model, this problem disappears because the explicit sequencing of effects is given through application of the bind operator. This provides the guarantee that exceptions are both thrown and caught in the order that they occur, something that is incredibly important as interdependencies are built in a proof tree.

4.5 The Kernel

HaskHOL uses the primitive data types and computation monad mentioned above to build a kernel almost identical to that of HOL Light. Shared are the basic functions for construction, destruction, and observation of types, terms, and theorems. These base functions are used to implement the same ten primitive inference rules and three primitive definition extension functions for axioms, basic types, and basic terms. The HaskHOL implementation of one of the primitive inference rules, `INST`, is shown below along with the original HOL Light implementation for point of comparison.

HaskHOL Implementation of `INST`

```
ruleInst :: [(HOLTerm, HOLTerm)] -> Theorem -> HOL Theorem
ruleInst env (Thm a t) =
    let instFun = vsubst env in
      do a’ <- termImage instFun a
         t’ <- instFun t
         return $! Thm a’ t’
```

HOL Light Implementation of `INST`

```
let INST theta (Sequent(asl,c)) =
  let inst_fun = vsubst theta in
  Sequent(term_image inst_fun asl,inst_fun c)
```

Two things are worth noting about this code. First, excluding minor differences due to varying language syntax and programming styles, it demonstrates that the monadic computation model is sufficient for implementing the primitive logical rules in a way that maintains the conciseness and clarity of the original HOL Light version. Second, as mentioned in the previous section, the monadic computation model makes explicit the sequencing of events, a detail that can be at times obscured or ignored in the HOL Light implementation. As HaskHOL is extended beyond the kernel, a topic that will be discussed in section 4.7, these observations of the monadic model become more beneficial as the code grows to be more and more complicated.

Similar to how the monadic computation model was folded into the trusted core, the last major difference between HaskHOL's kernel and HOL Light's kernel is the inclusion of other effectful combinators. Sections 4.2 and 4.3 have already discussed in detail the dangers of allowing users to construct their own `HOL` values, so any functions that require this, even to carry out seemingly mundane effects, must be defined in the kernel. A perfect example of this are combinators to support printing debug tracing statements, HaskHOL's implementation of which is shown below.

```
turnDebugOn :: HOL ()
turnDebugOn = modify $ \ ctxt -> ctxt { debug = True }

turnDebugOff :: HOL ()
turnDebugOff = modify $ \ ctxt -> ctxt { debug = False }
```

```
printDebug :: String -> HOL a -> HOL a
printDebug str x =
    do ctxt <- get
       if debug ctxt
         then (liftIO . putStrLn) str >> x
         else x
```

Users of Haskell may immediately recognize how similar `printDebug` is to the `trace` function. The primary difference between the two is that because the HaskHOL monad stack is built upon the `IO` monad, the print statements can be properly sequenced to maintain referential transparency. Functions like these are typically defined in the library of a HOL system, outside of the kernel. Given `printDebug`'s dependency on `liftIO`, this is impossible in HaskHOL without modifying it to use `unsafePerformIO`, in effect making it identical to `trace` and breaking referential transparency. This is the constantly considered trade off in HaskHOL, whether it is more appropriate to grow the size of the trusted kernel code or opt for a potentially unsafe library implementation.

4.6 Term Parsing

Another major difference between HaskHOL and HOL Light is the way HaskHOL deals with the construction of long or complicated terms. To do so using the primitive data type constructors would be extremely burdensome. HaskHOL, much like HOL Light, attempts to alleviate this problem by providing a collection of parsing and elaboration functions to allow the user to write terms and types in a more natural string representation:

```
-- Parsing
holParser     :: String -> HolContext -> Either ParseError PreTerm
holTypeParser :: String -> HolContext -> Either ParseError PreType
showErrors :: ParseError -> String
```

```
-- Elaboration
ty_elab :: PreType -> HOL HOLType
elab    :: PreTerm -> HOL HOLTerm
```

These parsing functions are implemented as expression parsers in Parsec [23] using information from the proof context to build the operator tables. They were designed to be as flexible as possible to allow them to be used in a variety of ways, however, they are most commonly used in HaskHOL in combination with the `TermRep` type class, shown below:

```
class TermRep a where
    toHT :: a -> HOL HOLTerm
```

The goal of `TermRep` is to provide a method to reduce any valid representation of a term to its primitive data type representation. This type class allows us to rewrite the type of a combinator to allow it to be called with any representation that has a class instance declared for it. For example, the following code allows us to call the primitive rule for beta reduction with either a string or data type constructor representation:

```
instance TermRep String where
    toHT x = do ctxt <- get
                case holParser x ctxt of
                  Left err -> throw $ "toHT: " ++ showErrors err
                  Right tme -> elab tme

instance TermRep HOLTerm where
    toHT = return

ruleBeta :: TermRep t => t -> HOL Theorem
ruleBeta = ruleBeta' <=< toHT
```

Here we see that `ruleBeta` acts more as a wrapper for the old version of the rule rather than a complete rewrite. This is purely an implementation choice to separate the logic for potential reuse or alternative application; it presents no significant performance difference when compared with a rule rewritten to use `toHT` directly.

Parsing terms at runtime like this is a computationally expensive process that adds up very quickly. As an alternative to the `TermRep` type class, preliminary work has begun on utilizing Template Haskell and its quasi-quotation capabilities to move parsing to compile time [24]. The main two functions used to implement this, along with the basic string quoter, are shown below:

```
baseParse :: String -> HOL a -> TH.Q HOLTerm
baseParse str ld = TH.runIO $ runHOL work
    where work :: HOL HOLTerm
          work = do ld
                    ctxt <- getCtxt
                    case holParser str ctxt of
                      Left err -> throw $ showErrors err
                      Right ptm -> elab ptm

baseQuoter :: HOL a -> QuasiQuoter
baseQuoter ld = QuasiQuoter quoteBaseExp nothing nothing nothing
    where quoteBaseExp str = dataToExpQ (const Nothing) =<<
                               baseParse str ld
          nothing _ = fail "quoting here not supported"
```

The `baseQuoter` quasi-quoter works by accepting a `HOL` computation that will load the appropriate theory before performing any parsing. This computation is passed to the `baseParse` function which handles the interaction with `holParser` and any necessary error handling. Writing a quasi-quoter that is theory specific is as simple as defining an alias to `baseQuoter` supplied with the corresponding load computation. For example, assuming the existence of a computation `loadBoolLib`, the definition of the boolean theory quoter can be written shown below:

```
bool :: QuasiQuoter
bool = baseQuoter loadBoolLib
```

In addition to supplying term quasi-quoters, HaskHOL also provides a basic string quasi-quoter. This quoter is used in cases where the user would like to write strings that contain special characters without having to escape them; such as defining term constants. For example, when introducing the definition for implication, instead of having to write `newBasicDefinition "(==>)= \\p q. p /\\ q <=>p"` one could write the much more human readable `newBasicDefinition [s| (==>)= \p q. p /\ q <=>p |]`. The quoters provided by HaskHOL only work over expressions. There is value in also providing quoters for pattern matching, however, there is still a very complicated issue to consider of whether soundness can be maintained when doing term examination and deconstruction in a quoter rather than using logical kernel primitives. This problem is being actively considered along with many other possible compile time improvements that will be discussed in the future work section.

4.7 Extending HaskHOL

HaskHOL supports libraries and feature extension through the use of the context modifying functions exposed by the kernel. In HOL Light these functions are called at the top level of OCaml modules where they are executed when the main `hol` module is loaded. The result is that every parser extension, user definition, and constant theorem are evaluated before the prover is usable, a process that can take several minutes on some machines, a detail previously pointed out when discussing some of the downsides of HOL Light. HaskHOL takes an alternative approach, explicitly stating when these functions are called by containing them within a wrapper function, `loadLib`, whose type is shown below:

```
loadLib :: String -> HOL a -> HOL ()
```

The arguments to loadLib are a label for the library and the monadic computation that contains all of the context modifying functions that need to be evaluated. The function first checks the context to make sure that another library with the same name hasn't been loaded already. This is done largely to avoid duplicating work by attempting to load the same library twice, but also to alert the user they made be loading an alternative version of a library that has already been loaded. Then it simply evaluates the monadic computation for the library and returns the unit value.

An example use of this is shown for the classical logic library:

```
loadClassicalLib :: HOL ()
loadClassicalLib =
    loadLib "Classical"
      (do loadBoolLib
          loadProofsLib
          addBinders ["@"]
          new_constant "@" "(A->bool)->A"
          aETA_AX
          aSELECT_AX
          extend_basic_rewrites =<< sequence [pSELECT_REFL])
```

Two things are important to note in this example. First, the classical library depends on the the boolean library and the proofs library. Both libraries could in turn have dependencies of their own. Because the loadLib function checks to see if the library dependencies have already been loaded, it is safe to call their load functions in the monadic computation for the classical library without having to worry about a loss of performance or invalidating the proof context. Second, it is hopefully clear that there is no limitation on what code may be contained within the monadic computation for the library. In this example the parser is extended, new axioms are defined, and the basic rewrite engine is extended with a new theorem. In fact, any Haskell code can be used here as long as it is encapsulated within the `HOL` monad. This might seem too open ended, but recall that `HOL` represents a type level guarantee that the code contained within the computation is well defined by HaskHOL's logical kernel.

Chapter 5

Using HaskHOL

While HaskHOL has been described as a DSL for HOL theorem proving, perhaps more accurately it should be referred to as a DSL for implementing HOL theorem proving tools. It is certainly possible to use HaskHOL directly to construct proofs, however, the process is non-interactive and constrained by the Haskell development tools being used. The suggested method of proof with HaskHOL is to build a tool exposing the user interface that best facilitates your verification strategy that utilizes HaskHOL as its logical foundation or prelude.

5.1 HaskHOLi

The first attempt at building such a tool is HaskHOLi, a general interactive theorem prover based on HOL Light's subgoal module. This interactive style is based on the notion of a goal stack, allowing an initial goal to be set that can be expanded to subgoals via tactics. It also provides the functionality for undoing a proof step, reordering the goal stack to allow the subgoals to be solved in a specified order, and printing the goal stack's status at any point in time. Like

HOL Light, HaskHOLi piggybacks on its implementation language's interpreter, in this case GHCi. Doing so greatly reduces the amount of code that needs to be written to provide a familiar REPL environment, allowing HaskHOLi to be implemented in only about one hundred lines of code.

The greatest challenge of implementing HaskHOLi was that GHCi is not a stateful interpreter and, therefore, does not provide its users with a mechanism for storing data to be shared between evaluation of expressions. This made it impossible to store either the proof context or the goal stack without first extending the capabilities of GHCi. After several different attempts, it was decided to store these values using the same method that GHC uses to store its internal variables, via a combination of the `global` function from the GHC API and judicious use of the C preprocessor:

```
#define USER_STATE(name,value,ty)     \
{-# NOINLINE name #-};                \
name :: IORef (ty);                   \
name = Util.global (value);           \
get_/**/name :: IO (ty);              \
get_/**/name = readIORef name;        \
set_/**/name :: (ty) -> IO ();        \
set_/**/name val = writeIORef name val;

USER_STATE(ctxt,initCtxt,HOLContext IO)
USER_STATE(goal_stack,GStack [],GoalStack)
```

The `global` function is implemented using `unsafePerformIO` which provides a backdoor to the `IO` monad to create an `IORef` at run time. In most cases the use of `unsafePerformIO` should be avoided because it is just that, potentially unsafe. In this case, though, it can be reasonably argued that its use is safe given that `name` is never inlined, common sub expression elimination is turned off, and in both cases the initial value of the `IORef` is built through a data constructor and not an effectful function. To be extra safe, though, these functions are hidden from the user and a pure interface to them is exposed instead. This interface is composed of the same functions as HOL Light's subgoal module and a new function, `extendCtxt`:

```
extendCtxt :: HOL a -> IO ()
extendCtxt m =
    do ctxt <- runval $ m >> getCtxt
       set_ctxt ctxt
       set_goal_stack $ GStack []
```

This function runs a HaskHOL monadic computation and sets HaskHOLi's proof context to the resultant context from the computation. It should be noted that it also resets the goal stack to prevent cases where the new context may have invalidated proof steps already taken.

5.2 An Example

Here we examine a short proof in HaskHOLi to illustrate how HaskHOL looks to a user. To begin we launch GHCi and load the HaskHOLi module and the boolean logic library that together prepare our context for propositional logic proofs:

```
Prelude> :m HaskHOLI
Prelude HaskHOLI> extendCtxt loadBoolLib
...
Loading package HaskHOL-0.1 ... linking ... done.
Loading package HaskHOLI-0.1 ... linking ... done.
```

The example proof shows that $x \wedge y \Rightarrow x$ is a tautology, so we set that as our initial goal:

```
Prelude HaskHOLI> g [bool| x /\ y ==> x |]
Free vars in goal: x, y
1 subgoal(s) (1 total)

-----
(x /\ y) ==> x
```

We can now use the expand function in combination with the discharge tactic to create a new subgoal from the antecedent of the implication, $x \wedge y$:

```
Prelude HaskHOLI> e tacDisch
1 subgoal(s) (1 total)
0 x /\ y
-----
x
```

In addition to using the subgoal functions and tactic language, HaskHOLi can still construct individual theorems to assist with the proof. Here we construct the theorem $x \wedge y \vdash x$ which matches with the current subgoal to use later in the proof.

```
Prelude HaskHOLI> thm <- runval $ ruleConjunct1 =<< ruleAssume [bool|
     x /\ y |]
[x /\ y] |- x
```

One of the greatest advantages of working interactively is being able to see when you make a mistake immediately so that you can quickly correct it and proceed. Here we accidentally use the theorem constructed in the last step with the wrong tactic, introducing an extra assumption to the current subgoal rather than simplifying it. When the error is spotted the backup function is called returning the goal stack to the state it was in before:

```
Prelude HaskHOLI> e $ tacAssume thm
1 subgoal(s) (1 total)
0 x /\ y
1 x
-----
x
Prelude HaskHOLI> b
1 subgoal(s) (1 total)
0 x /\ y
-----
x
```

The correct tactic is applied now, resulting in a goal stack with no subgoals indicating a completed proof. If the resultant theorem is to be used in a subsequent proof its value can be return with the `top_thm` function:

```
Prelude HaskHOLI> e $ tacAccept thm
No subgoals
Prelude HaskHOLI> top_thm
|- (x /\ y) ==> x
```

Primitive proof replay and checking capabilities of interactive proofs can be had simply saving the proof commands to a text file and piping it to GHCi via standard input. As long as the last command in the file is `top_thm` a failed proof will result in a GHC exception containing the HaskHOL error message being thrown and a successful proof will result in a theorem being returned. Whether this theorem is the intended one is a check left for the user. It can either be compared against the expected result visually, or the theorem can be deconstructed with its conclusion being compared against the expected term value programmatically.

This simple interactive session illustrates how HaskHOL is used in an interactive mode. HaskHOL also provides an execution feature that runs non-interactive proof scripts generated by hand or automated tools. Such proof scripts are simply

Haskell programs executed by the interpreter and represent the probable interaction between the prover and associated Rosetta tools.

Chapter 6

Evaluation

At the time of this writing, the most advanced library for HaskHOL that is considered both stable and complete is the Intuitionistic library. This library contains both a derived rule, `ruleITAUT`, and a tactic, `tacITAUT`, that take a term as input and attempt to prove that it is a tautology using intuitionistic first-order logic. This process is done via proof search using derived rules and tactics built in previous HaskHOL libraries and extensions (Bool, Equal, Rules, and Tactic). It is worth noting that `ruleITAUT` and `tacITAUT` are identical to HOL Light's `ITAUT` and `ITAUT_TAC` respectively. They also provide similar to functionality to that of HOL4's `TAUT_TAC` with the principle difference being that HOL4 checks tautologies via SAT solver proof replay and admits classical logic principles like the Law of the Excluded Middle or the Law of Double Negation.

Given that the Intuitionistic library incorporates such a large portion of almost all of the other HaskHOL libraries and extensions, it represents an ideal target for evaluation of HaskHOL as a whole. The Intuitionistic Logic Theorem Proving (ILTP) library [30] was chosen as the problem framework to perform this evaluation. Similar to the Thousands of Problems for Theorem Provers (TPTP) library [35], the ILTP library provides a large collection of problems specifically designed for testing automated theorem provers. The principle difference between the two is that the ILTP library, as the name would suggest, specifically limits itself to including problems to test automated theorem provers for intuitionistic logic, whereas the TPTP library contains many additional classes of problems.

6.1 Evaluation Formula Classes

Unlike most automated theorem provers dedicated to intuitionistic logic solving, HaskHOL's Intuitionistic library does a very poor job of identifying terms that are not valid under intuitionistic logic. In fact, supplying either `ruleITAUT` or `tacITAUT` with a term that is not an intuitionistic tautology will generally result in the proof system entering an infinite loop. For this reason, the Intuitionistic library is generally only used for bootstrapping purposes to prove known valid theorems to aid in the construction of more advanced HOL features.

Knowing this, a subset of the ILTP library problems had to be selected that were known to be intuitionistically valid so that the evaluation of HaskHOL mirrored its true use case. The problems chosen were inspired by Roy Dyckhoff's work [9] and represent six classes of scalable problems collected to help evaluate not only a prover's correctness, but also its performance as more and more complicated terms are introduced. Each of these six classes will be briefly introduced and discussed with the final results of HaskHOL's performance summarized at the end of this chapter.

6.1.1 The de Bruijn Class

The first class of problems arrises from Dyckhoff's personal communications with N. G. de Bruijn [9]. de Bruijn provided an example formula that is intuitionistically provable, but is intended as a particularly difficult exercise for students to prove by natural deduction:

$((b \iff c) \Rightarrow (a \wedge b \wedge c)) \wedge ((c \iff a) \Rightarrow (a \wedge b \wedge c)) \wedge ((a \iff b) \Rightarrow (a \wedge b \wedge c)) \Rightarrow (a \wedge b \wedge c)$

In general this class of formulae can be expressed as:

$((\bigwedge_{i=0}^{n} p_i \iff p_{(i+1)(mod\, n)}) \Rightarrow \bigwedge_{i=o}^{n} p_i) \Rightarrow \bigwedge_{i=0}^{n} p_i$

where n is the number of unique atoms in the formula and p_i represents the i^{th} unique atom in the formula. It should be noted that members of this class of problems with an even number of variables are known to be classically unprovable, and as such intuitionistically unprovable, so we restrict HaskHOL's evaluation to problems with an odd number of variables.

6.1.2 The Pigeonhole Class

The pigeonhole principle is something that all computer scientists should be familiar with. In short, it states that if there are $n + 1$ pigeons and n holes then at least one hole will contain two pigeons.

In general this class of formulae can be expressed as:

$\bigwedge_{i=1}^{n+1} \bigvee_{j=1}^{n} occ(i, j) \Rightarrow \bigvee_{i=1, j=1, k=j+1}^{n, n+1, n+1} occ(j, i) \wedge occ(k, i)$

where $occ(x, y)$ indicates that pigeon x is occupying hole y. This property is intuitionistically provable for any number of pigeons, so there is no restriction on HaskHOL's evaluation for this class of problems.

6.1.3 The N-Many Contractions Class

Franzen provided examples in his work [10] of formulae that required n-many contractions for their proof:

$\neg\neg(\neg p1 \vee \neg p2 \vee \neg p3 \vee (p1 \wedge p2 \wedge p3))$

Most modern provers can trivially solve formulae of this form by leveraging Glivenko's Theorem which states that $\neg\neg P$ is a theorem in intuitionistic logic if and only P is a theorem in classical logic and all formulas are quantifier-free. Given this, Dyckhoff replaced any instance of a negated atom with an instance of that atom implying false, transforming $\neg p$ to $p \Rightarrow F$. Additionally, the largest disjunctive clause, the conjunction of all of the free variables, was moved to the front to punish provers who naively grabbed that portion first.

In general, the resultant class of formulae can be expressed as:

$$((\bigwedge_{i=1}^{n} p_i \vee \bigvee_{i=1}^{n} p_i \Rightarrow F) \Rightarrow F) \Rightarrow F$$

where n is the number of negated atoms in Franzen's original formula. Again, this property is intuitionistically provable for any number of atoms, so no restrictions for the evaluation of HaskHOL is necessary.

6.1.4 The Big Normal Natural Deductions Class

Schwichtenberg [31] mentions a class of formulae that have no normal natural deduction proof of size less than an exponential function. Dyckhoff provides a general form for this class of formulae as:

$$(p_n \wedge \bigwedge_{i=1}^{n} p_i \Rightarrow p_i \Rightarrow p_{i-1}) \Rightarrow p_0$$

where n is one less than the number of unique atoms in the formula and bounds the proof size as an exponential function of n.

This class of problems is particularly interesting for HaskHOL's evaluation because most other provers can decide formulae from this class very fast but with some space issues. Given that HaskHOL is built upon a garbage collected runtime system, a space allocation issue could directly result in a serious slowdown of proof speed.

6.1.5 The Korn and Krietz Class

Similar to the work of Franzen, Korn and Kreitz propose a class of formulae that can be proved quite efficiently by classical reasoning using Glivenko's Theorem:

$\neg(\neg a_0 \wedge ((b_n \Rightarrow b_0) \Rightarrow a_n) \wedge (\bigwedge_{i=1}^{n}((b_{n-1} \Rightarrow a_i) \Rightarrow a_{i-1})))$

Again, Dyckhoff transforms any negated atom to an implication of false to remove this classical logic shortcut. Additionally, two different orderings of the antecedents of the implications in the formulae are prepared and conjuncted to augment testing in case a prover is ordering dependent.

The resultant general form is then given as:

$((a_0 \Rightarrow F) \wedge ((b_n \Rightarrow b_0) \Rightarrow a_n) \wedge (\bigwedge_{i=1}^{n}((b_{n-1} \Rightarrow a_i) \Rightarrow a_{i-1})) \Rightarrow F) \wedge$

$((\bigwedge_{i=1}^{n}((b_{n-1} \Rightarrow a_i) \Rightarrow a_{i-1})) \wedge ((b_n \Rightarrow b_0) \Rightarrow a_n) \wedge (a_o \Rightarrow F))$

where n is one half of the number of unique atoms in the formula.

6.1.6 The Equivalences Class

The final class of problems is composed of formulae containing nothing but bi-implications of atoms. These formulae are designed with the generally agreed upon notion that bi-implications can cause difficulty for even non-intuitionistic solvers. There are a number of ways to prepare this class, however, Dyckhoff settles for the simplest form that he finds adequate, a bi-implication of all atoms in both forward and reverse order themselves bi-implicated.

In general this is given as:

$(\Longleftrightarrow_{i=1}^{n} p_i) \Longleftrightarrow (\Longleftrightarrow_{i=n}^{1} p_i)$

where n is the number of unique atoms in the formula.

6.2 Results

Table 6.1 below shows the results of using `tacITaut` to solve three increasingly difficult problems from each of the above evaluation classes[1].

Table 6.1. HaskHOL Evaluation Results

Class	**N**	**Solved?**	**Time (sec)**
de Bruijn	1	YES	6.024
	2	YES	272.528
	3	YES	1872.035
Pigeon Hole	1	YES	0.014
	2	YES	0.577
	3	YES	21.659
N-Contractions	1	YES	0.076
	2	YES	1.086
	3	YES	28.742
Big Natural Deductions	1	YES	0.046
	2	YES	0.745
	3	YES	3.679
Korn and Krietz	1	YES	0.223
	2	YES	3.573
	3	YES	9.664
Equivalences	1	YES	0.005
	2	YES	0.044
	3	YES	0.233

[1] Running on OS X 10.6.6, 2.2 GHz Intel Core 2 Duo, 4 GB 667 MHz DDR2 SDRAM. Compiled with `ghc -O2. Averaged over ten iterations.`

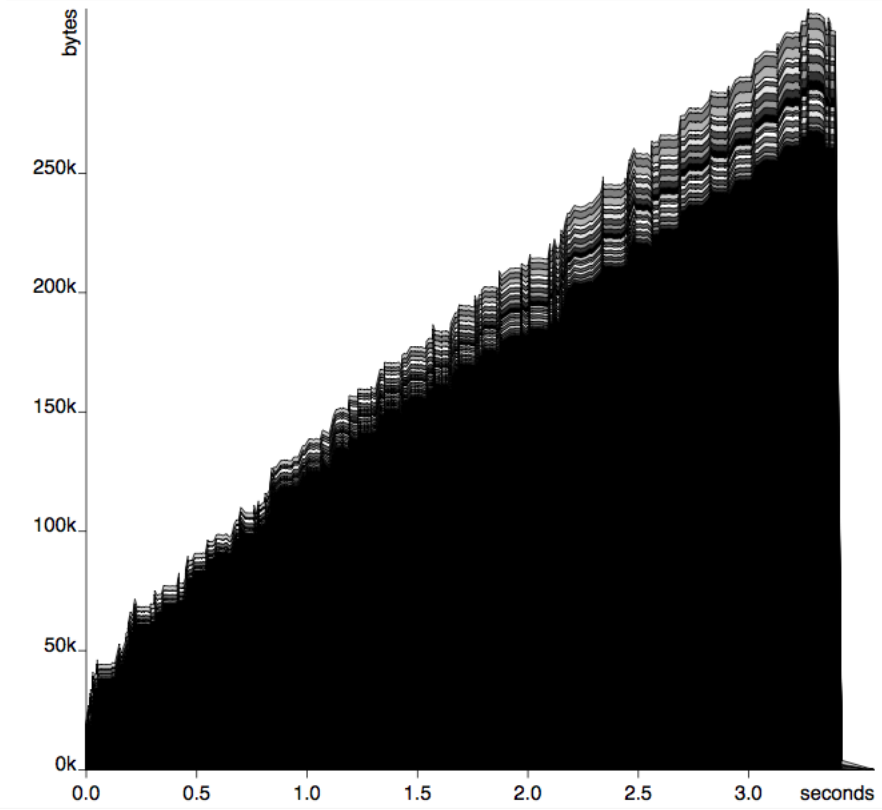
The first result worth mentioning is that HaskHOL was capable of solving each problem, a very promising sign. Unfortunately there are still some classes of problems that present a challenge for HaskHOL, specifically the de Bruijn class. Beyond that class, though, HaskHOL was able to solve all problems of complexity $N = 1$ exceptionally fast, another very promising sign. In addition to these generalities, a few more specific observations were made:

- Excluding de Bruijn, Korn and Krietz is the slowest class in all cases, but scales the best.
- The Pigeon Hole and N-Contractions classes scale terribly.
- The equivalences class was thought to be a hard one, yet HaskHOL appears to handle it extremely well.

Two other specific observations were made that are worthy of more detailed discussion, given that they indicate possible avenues for improvement for HaskHOL. The first, as was expected, is that the Big Natural Deductions class of problems did exhibit a space leak during execution. This is clearly documented in Figure 6.1.

Space leaks are a relatively common problem in Haskell and are usually indicative of areas in the code that are "too lazy." Further heap profiling during the execution of this problem class should help to pinpoint these troublesome areas, hopefully identifying areas where improvement may benefit all classes of problems. The second observation relates to the abysmal performance of the de Bruijn problem class. After building call graphs from the profiling information of the tests, it was noted that the de Bruijn class depends more heavily on the

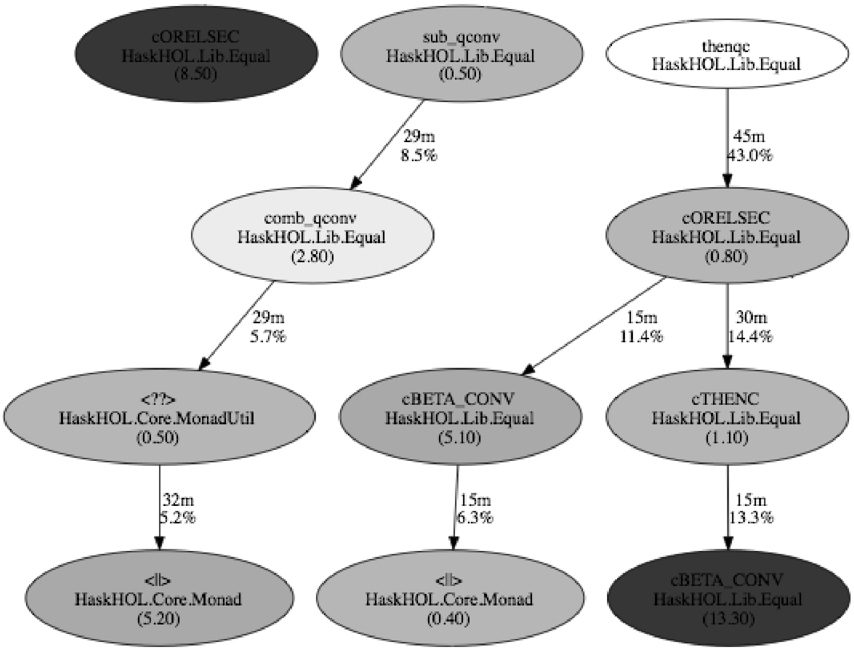
Figure 6.1. Heap profile for Big Natural Deductions, N=3



conversion language than the other classes do. These conversions represent a major bottleneck in the solution of this problem, as indicated by a subset of the call graph pictured in Figure 6.2.

Again, this information points directly to troublesome areas that are deserving of further attention, with the goal of improving the performance for all problem classes. Unfortunately, the slowdown here is not as simple or well understood as a space leak; improvement will most likely have to come in the form of an alternative implementation for the conversion language. This will be discussed in more detail in the future work section.

Figure 6.2. Subset of call graph for de Bruijn, N=3



Chapter 7

Conclusions and Future Work

HaskHOL has already shown great promise in progressing towards its goal of implementing a full HOL system in Haskell. That being said, as is true of most things, there is always room for improvement. Future advancements planned for HaskHOL fall within three main domains:

- Improving the performance of current features.
- Advancing HaskHOL's feature set.
- Integrating HaskHOL with the Rosetta tool suite.

Items two and three and very much tied together; the order in which new theories are developed for HaskHOL will very much be dictated by the needs of the Rosetta tool suite. As such, it will be hard to speak about the future work associated with these topics because they will follow the ever changing demands and ideas presented by the Systems Level Design Group. There are numerous points of improvement that can be discussed for current features, though.

As noted in Section 4.4, HaskHOL still contains a few implementation choices that are less than ideal. The immediate focus of the future work is to fix the most glaring of these issues, the bloat of the proof context that inhibits efficient error handling. Currently, when a theorem is proved and is intended to be reused it is naively cached in the proof context, leading to a rapid expansion of the size of the context as more and more theories are loaded. At this point in time there appears to be two possible solutions to this problem. The first is to find a more space efficient way to store the information contained within a theorem before caching it. Yet to be considered is whether the cost of translating to and from this alternative representation will outweigh the overall benefit. The second is to push as much proof as possible to compile time using Template Haskell.

The use of Template Haskell with HaskHOL has already been explored for the purpose of compile time quasi-quotation of terms as explained in Section 4.6. As mentioned, the quasi-quotation functionality itself could be extended to support pattern matching, but it exposes the bigger issue of combining theorem proving and metaprogramming; how can you be assured that you maintain soundness and completeness? As is, HaskHOL provides several guarantees about the correctness of a proof based on the explicit ordering of the effects used to construct it. The use of Template Haskell to perform more complicated term quoting or compile time proof essentially ”cuts in line,” skipping to a specific point in this sequence of effects. In order to maintain the correctness guarantees there has to be some way to reason that the result of the actions taken by Template Haskell is equivalent to the result had the normal ordering of effects occurred. In a sense this is what HaskHOL attempts to do now with basic term quoting, tying a quasi-quoter to a theory by having it perform that theory's load function before any parsing occurs. For something as minor as term parsing, this hand waving reasoning is acceptable. However, when you expand the notion to compile time proof where an entire session can be invalidated by the introduction of an inconsistent axiom, this connection must be formalized. This issue must be address before Template Haskell's metaprogramming capabilities can be leveraged for their true power.

Remaining reimplementation work will focus on transforming existing code to leverage more of Haskell's language features. The most apparent ”low hanging fruit” is the redesign of HaskHOL's conversionals and tactics as type restricted monads. All three objects serve roughly the same purpose, acting as data types that describe computations. Thus, to re-express the first two using Haskell's implementation of monads would appear to make sense. Andrew Martin and Jeremy Gibbons have already conducted similar research [26] interpreting Angel, a generic tactic language [27] in Haskell. In the case of HaskHOL, interpreting conversionals and tactics as monads would allow many of the sublanguage connectives, such as `then`,`fail`, and `or_else`, to be replaced with existing Haskell monad combinators. This should lead to a dramatic reduction in code size and increase in clarity for large tactics, like those for solving intuitionistic or first-order logic. It may also provide a secondary benefit of improving the performance of conversionals and tactics which would lead to significant improvements in the results of the test suite from Section 6.

There also remains the exploration of the tiny changes brought with the recent release of GHC 7. This is a particularly exciting release because it includes the movement to the Haskell 2010 standard and several new extensions to try out. One of the major ones, `RebindableSyntax`, presents a way to overload or replace syntax bound in the GHC prelude. Specifically, this extension was provided to allow for the traditional `if ... then ... else` syntax to be used in conjunction with monadic computations without having to first bind the result of the condition computation. This alone should lead to a significant code cleanup in HaskHOL. Also included with GHC 7 is improved support for new compilation flags and backends, like LLVM. These represent new avenues to adventure down in search of performance improvement with little to no modification of the code itself.

Finally, before HaskHOL can be formally released in any trusted code base its logical kernel must be verified for soundness. In the Prufrock work the TPTP libraries served as a source of test cases. The same was done for HaskHOL, allowing for potential comparison with Prufrock and other provers as well as providing evidence of soundness. I would like to take the verification a step further, though, and provide a more formal argument for the correctness of the HaskHOL kernel. Ultimately this will require opening the flood gates of reasoning about monadic code, however, I think it is the necessary and proper next step.

References

[1] The Glasgow Haskell Compiler. Website:. `http://haskell.org/ghc/`.

[2] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying haskell programs using constructive type theory. In *In Haskell'05*. ACM Press, 2005.

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

[4] P. Alexander. Rosetta: Standardization at the System Level. *IEEE Computer*, 42(1):108–110, January 2009.

[5] E. Austin and P. Alexander. Haskhol: A haskell hosted domain specific language representation of hol light. In *Trends in Functional Programming Draft Proceedings (TFP 2010)*, 2010.

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

[7] E. Brady. Ivor, a proof engine. Draft Paper:. `http://www.cs.st-andrews.ac.uk/~eb/drafts/ivor.pdf`.

[8] J. Crow, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*. SRI International, Menlo Park, CA, June 1995. Presented at WIFT'95.

[9] R. Dyckhoff. Some benchmark formulae for intuitionistic propositional logic. Website:. `http://www.cs.st-andrews.ac.uk/~rd/logic/marks.html`.

[10] T. Franzen. Algorithmic aspects of intuitionistic propositional logic. Technical Report SICS R87010, Swedish Institute of Computer Science, 1987.

[11] A. Gill. A haskell hosted dsl for writing transformation systems. In *IFIP Working Conference on Domain Specific Languages*, 07/2009 2009.

[12] M. Gordon. *Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware*. North-Holland, 1986.

[13] M. Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[14] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 73–128. Springer-Verlag, 1989.

[15] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[16] F. Haftmann. From higher-order logic to haskell: There and back again. In J. P. Gallagher and J. Voigtländer, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010*. ACM, 2010.

[17] J. Harrison. Hol light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCADÕ96), volume 1166 of Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[18] J. Harrison. Floating point verification in hol light: The exponential function. *Form. Methods Syst. Des.*, 16:271–305, June 2000.

[19] J. Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 217–233, London, UK, 2000. Springer-Verlag.

[20] J. Harrison. Formal verification of ia-64 division algorithms. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '00, pages 233–251, London, UK, 2000. Springer-Verlag.

[21] J. Harrison. The hol light system reference (2.2), 2011.

[22] G. Klein, P. Derrin, and K. Elphinstone. Experience report: sel4: formally verifying a high-performance microkernel. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 91–96, New York, NY, USA, 2009. ACM.

[23] D. Leijen and E. Meijer. Parsec: A practical parser library, 2001.

[24] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.

[25] S. Marlow. Haskell 2010 language report, 2010.

[26] A. Martin and J. Gibbons. A monadic interpretation of tactics, 2002.

[27] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus Ñ abridged version. *Formal Aspects of Computing*, 8:479–489, 1996. 10.1007/BF01213535.

[28] T. Melham. *Higher order logic and hardware verification.* Cambridge University Press, New York, NY, USA, 1993.

[29] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[30] T. Raths, J. Otten, and C. Kreitz. The iltp problem library for intuitionistic logic, release v1.1. *Journal of Automated Reasoning.*

[31] H. Schwichtenberg. Termination of permutative conversions in gentzen's sequent calculus. 1997.

[32] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *In Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM, 2002.

[33] K. Slind and M. Norrish. A brief overview of hol4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 28–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] D. Stewart and S. Sjanssen. Xmonad. In *Haskell*, pages 119–119, 2007.

[35] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[36] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[37] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[38] J. Ward, G. Kimmell, and P. Alexander. Prufrock: A framework for constructing polytypic theorem provers. In *Proceedings of the Automated Software Engineering Conference (ASE'05)*, Long Beach, CA, November 2005.