Haskell + HOL = HaskHOL

Evan Austin, Perry Alexander

The University of Kansas
Information and Telecommunication Technology Center
2335 Irving Hill Rd, Lawrence, KS 66045
{ecaustin,alex}@ittc.ku.edu

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 paper describes HaskHOL, its motivation and implementation.

1 Introduction

Higher-order logic (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" [10]. Since then, research has spawned a wide variety of materials covering verification with HOL, including everything from textbooks [20] to research papers detailing verification targets ranging in size from single algorithms [14, 16, 15] to large software systems [17]. Given the success HOL has had verifying such a diverse set of topics, when it came time to connect a verification formalism to Rosetta [4, 3], a system level design language in development at The University of Kansas, HOL seemed to be the logical choice.

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 [23], PVS [7] and Isabelle/HOL [21], 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 [5] 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 us to our desired properties for a formal tool: it must be lightweight and it must be able to interface with the existing Rosetta environment.

HOL Light [13], 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.

With great naivety, the first author suggested what he 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. 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.

What follows is a high-level explanation of the primitive implementation techniques used to develop HaskHOL with comparisons and contrasts to HOL Light, its closest relative in the HOL family tree. Additionally, a preliminary performance evaluation of the HaskHOL DSL is described, and a tentative plan of future work is outlined discussing the major advancements HaskHOL would like to make, thanks largely in part to the ever improving Glasgow Haskell Compiler (GHC) [1].

2 HaskHOL

The primary technique for developing an embedded domain specific language (DSL) in Haskell is well known [9]. 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 Logic of Computable Functions [11], follows this style by implementing a small logical kernel with advanced features bootstrapped from it to reduce the burden of proof of soundness and consistency.

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 subsections describe the implementation of the HaskHOL kernel, explaining the ramifications of using the monadic implementation style.

2.1 Types, Terms, and Theorems

At the lowest level HaskHOL is based on a sound and consistent 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, where each type can be either a variable or an application of types to a constructor and each term can be either a typed variable, a typed constant, an application of two terms, or an abstraction of two terms. As might be expected, HaskHOL implements these primitive data types using Haskell's abstract data types:

```
data HolType
    = TyVar String
    | TyApp String [HolType]
    deriving (Eq, Ord)
```

```
data HolTerm
    = Var   String  HolType
    | Const String  HolType
    | Comb  HolTerm HolTerm
    | 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 = Seq [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.

2.2 The HOL Monad

As was explained previously, 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.

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

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

runHOL :: HOL a -> IO a

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

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. This problem is analogous to the issue exposed by the use of the `put` method from the `State` monad, discussion of which will also happen in the next subsection.

2.3 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. This context is implemented as a parameterized 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. 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[24].

```
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. 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, the wrapper is stripped away, and the result is casted to the correct type. 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. As a point of reference, the types of these context extension functions are shown below:

```
putExt :: (ExtClass a) => a -> HOL ()

getExt :: forall a. (ExtClass a) => HOL a
```

Notice the parallels between `putExt` and the `put` method from the `State` monad. In general, the problem with exposing these functions directly is that they provide the user with the capability to invalidate the context by supplying new context data 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 ()` $\approx$ `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 definition of `put` is shown below:

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

The Haskell module system is used to hide these methods and the field constructors of the `HOLContext` record type, or the internal constructor of an extension type, outside of the module where they are defined. Again, the goal of these actions is to prevent the user from being able to manually construct and insert context 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.

2.4 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 errors in `HOL` monadic code. Again, as a point of reference, the types of these functions is shown below:

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

throwError :: Exception e => e -> HOL a

catchError :: Exception e => HOL a -> (e -> HOL a) -> HOL 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 size of the proof context being passed around 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, however, plans to alleviate this will be detailed in the Future Work section.

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 unspecified 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.

2.5 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 2.6, 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 2.2 and 2.3 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. The other option would be to use functions like `unsafePerformIO` to force these effects into code outside of the kernel. 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.

2.6 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. This is 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 may 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 ["@"]
      newConstant "@" "(A->bool)->A"
      axEta
      axSelect
      extendBasicRewrites =<< sequence [proofSelectRefl])
```

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 the `HOL` type represents a type level guarantee that the code contained within the computation is well defined by HaskHOL's logical kernel.

3 Performance 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[22] was chosen as the problem framework to perform this evaluation. Similar to the Thousands of Problems for Theorem Provers (TPTP) library[25], 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.

4 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[8] 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 and their general formulations are shown below:

4.1 The de Bruijn Class

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

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.

4.2 The Pigeonhole Class

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

where n is the number of holes and $occ(x, y)$ 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.

4.3 The N-Many Contractions Class

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

where n is the number of negated atoms, all represented indirectly as $p \Rightarrow F$.

Again, this property is intuitionistically provable for any number of atoms, so no restrictions for the evaluation of HaskHOL is necessary.

4.4 The Big Normal Natural Deductions Class

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

4.5 The Korn and Krietz Class

$((a_0 \Rightarrow F) \wedge ((b_n \Rightarrow b_0) \Rightarrow a_n) \wedge (\bigwedge_{i=1}^{n}((b_{n-1} \Rightarrow a_i) \Rightarrow a_{i-1})) \Rightarrow F) \wedge$
$((\bigwedge_{i=1}^{n}((b_{n-1} \Rightarrow a_i) \Rightarrow a_{i-1})) \wedge ((b_n \Rightarrow b_0) \Rightarrow a_n) \wedge (a_o \Rightarrow F))$
where n is one half of the number of unique atoms in the formula.

4.6 The Equivalences Class

$(\Longleftrightarrow_{i=1}^{n} p_i) \Longleftrightarrow (\Longleftrightarrow_{i=n}^{1} p_i)$
where n is the number of unique atoms in the formula.

5 Results

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

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.

[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.`

Table 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

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, verified by basic heap profiling. 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 conversion language than the other classes do. These conversions represent a major bottleneck in the solution of this problem. 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.

6 Related Work

In addition to HOL Light, to which comparisons have been made throughout the paper, there are other related HOL implementations that attempt to integrate with Haskell. Recent work has been completed by Florian Haftmann to connect HOL and Haskell using a different method. Rather than trying to implement a native representation of HOL in Haskell, Haftmann presents two tools which work together to provide a translation between specifications written in Isabelle and executable Haskell source [12]. The generation of code from an Isabelle specification is presented as an established and mature tool, however Haskabelle is primarily for verifying Haskell artifacts where HaskHOL operates on artifacts implemented in Haskell.

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 besides HaskHOL to bring these reasoning capabilities to Haskell intensionally. Ivor [6] 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 HaskHOL has done by selecting the 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 in the target user of the two tools, with Ivor being designed to be used in conjunction with generative programming techniques instead of directly by humans.

Our first attempt at integrating a Haskel-based prover into the Rosetta tool suite was Prufrock [26]. Prufrock differed substantially from HaskHOL in both intent and implementation. While HaskHOL implements a HOL prover directly in an efficient manner, 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 – making it an effective prototyping language. However, evaluation with respect to the TPTP library demonstrated inefficiencies leading to the decision to implement HaskHOL as a smaller, more targeted proof tool.

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 2.4, 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. 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 consistency? 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 conversions 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 [19] interpreting Angel, a generic tactic language [18] in Haskell. In the case of HaskHOL, interpreting conversions 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 conversions and tactics which would lead to significant improvements in the results of the test suite from Section 3.

There also remains the exploration of the many changes brought with the recent release of GHC 7, as well as its subsequent minor version updates. These are particularly exciting releases because they include the movement to the Haskell 2010 standard, as well as several new extensions to try out. One of the major extensions, Safe Haskell, has the goal of allowing users to safely execute untrusted code by restricting the Haskell functionality that is permitted within said code. This safe subset of Haskell may ultimately become the same subset used to implement HaskHOL given the guarantees provided by the restrictions. For example, the use of the `GeneralizedNewtypeDeriving` extension, one used generously by HaskHOL, can be used to violate constructor access control; Safe Haskell disallows this extension entirely to avoid this problem. Safe Haskell also disallows the use of Template Haskell, though, which runs orthogonal to previously discussed goals, so a compromise between these two implementation paths still needs to be found.

References

1. The Glasgow Haskell Compiler. Website:. `http://haskell.org/ghc/`.
2. Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *In Haskell05*. ACM Press, 2005.
3. P. Alexander. Rosetta: Standardization at the System Level. *IEEE Computer*, 42(1):108–110, January 2009.
4. Perry Alexander. *System Level Design with Rosetta*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
5. P. Baraona, J. Penix, and P. Alexander. VSPEC: A Declarative Requirements Specification Language for VHDL. In Jean-Michel Berge, Oz Levia, and Jacques 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.
6. Edwin Brady. Ivor, a proof engine. Draft paper, available at `http://www.cs.st-andrews.ac.uk/~eb/drafts/ivor.pdf`.
7. Judy Crow, John Rushby, Natarajan Shankar, and Mandayan Srivas. *A Tutorial Introduction to PVS*. SRI International, Menlo Park, CA, June 1995. Presented at WIFT'95.

8. Roy Dyckhoff. Some benchmark formulae for intuitionistic propositional logic. Website:. http://haskell.org/ghc/.
9. Andy Gill. A haskell hosted dsl for writing transformation systems. In *IFIP Working Conference on Domain Specific Languages*, 07/2009 2009.
10. M. Gordon. *Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware*. North-Holland, 1986.
11. Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.
12. Florian Haftmann. From higher-order logic to haskell: There and back again. In John P. Gallagher and Janis 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.
13. John Harrison. Hol light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD96), volume 1166 of Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
14. John Harrison. Floating point verification in hol light: The exponential function. *Form. Methods Syst. Des.*, 16:271–305, June 2000.
15. John 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.
16. John 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.
17. Gerwin Klein, Philip Derrin, and Kevin 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.
18. A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus abridged version. *Formal Aspects of Computing*, 8:479–489, 1996. 10.1007/BF01213535.
19. Andrew Martin and Jeremy Gibbons. A monadic interpretation of tactics, 2002.
20. T. Melham. *Higher order logic and hardware verification*. Cambridge University Press, New York, NY, USA, 1993.
21. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
22. Thomas Raths, Jens Otten, and Christoph Kreitz. The iltp problem library for intuitionistic logic, release v1.1. *Journal of Automated Reasoning*.
23. Konrad Slind and Michael 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.
24. Don Stewart and Spencer Sjanssen. Xmonad. In *Haskell*, pages 119–119, 2007.
25. 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.
26. 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.