Haskell + HOL = HaskHOL

Evan Austin
University of Kansas - ITTC
Lawrence, KS, U.S.A.
ecaustin@ittc.ku.edu

Perry Alexander
University of Kansas - ITTC
Lawrence, KS, U.S.A.
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" [11]. Since then, research has spawned a wide variety of materials covering verification with HOL, including everything from textbooks [24] to research papers detailing verification targets ranging in size from single algorithms [15, 17, 16] to large software systems [21]. Given the success HOL has had verifying such a diverse set of topics, when it came time to connect a verification formalism to Rosetta [6, 5], 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 [28], PVS [8] and Isabelle/HOL [25], 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 [7] 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 [14], 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 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 [18, 19, 10]. 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 [12], 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 monad, HOL, explaining the ramifications of using the monadic implementation style.

2.1 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.

Shown below is the definition of the HOL monad that captures these effects:

```
newtype HOL thry a b =
    HOL {runHOLCtxt :: (HOLContext thry) -> IO (b, HOLContext thry)}


instance Monad (HOL thry a) where
    return x = HOL $ \ s ->
        return (x, s)

    m >>= k = HOL $ \ s ->
        do (b, s') <- runHOLCtxt m s
           runHOLCtxt (k b) s'

    fail str = HOL $ \ _ ->
        E.throwIO $ HOLException str
```

Those familiar with Haskell may recognize this as the flattening of a State and IO monad stack. The HOL monad's adoption of the State monad's shape is intentional, as it removes the necessity of explicitly passing IORef, or other state representation, values from computation to computation. In other words, it hides the concept of state from the user, such that they can focus on the predominant concern: reasoning. Implicitly, the HOL monad carries the proof context as an instance of the HOLContext data type, details of which will be explained in Section 2.2.

The runHOLCtxt function is used to run a HOL monad computation using one of these contexts 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 or the shifting of proof efforts to compile time. 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 Section 2.4.

2.2 Extensibility

Extensibility is an important feature because it allows the users to define their own types, terms, axioms, etc. As described in the previous section, HaskHOL carries this information in a context threaded through computations using the `HOL` monad. This context is implemented as a parameterized record type as shown below, where the parameter is a phantom type variable. This variable can be instantiated with a type value that indicates what theory the data in the context represents, such that when this type is reflected in the HOL monad it can be used to tie computations to their prerequesite theories.

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

Any data we need to store in the context is encoded directly as its own field; for example, axioms are stored in the axs :: [(String, 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 principal 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 other subclasses. This is a common technique among Haskell programmers and is present in several large libraries, such as XMonad[29].

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

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

Defining an instance of the ExtState data 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 is 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 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. Furthermore, inference of these wrapper types drives how values are stored and retrieved, as shown by the type signatures of the related functions below:

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

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

2.3 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. Mimicking the design of the Control.Exception library, HaskHOL has two primary exception handling methods, throwError and catchError, around which a richer interface is developed:

```
throwError :: Exception e => e -> HOL thry a b
throwError e = HOL $ \ _ -> E.throwIO e

catchError :: Exception e => HOL thry a b -> (e -> HOL thry a b) ->
              HOL thry a b
catchError job errcase = HOL $ \ s ->
    runHOLCtxt job s `E.catch` \ e ->
    runHOLCtxt (errcase e) s
```

Suspiciously absent is a method for throwing exceptions 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. While there does exist ways to throw errors in pure Haskell code, such as the popular error function, to guarantee that they are caught and handled in the correct order they must be evaluated with an IO computation. In practice, this results in a severe performance degradation when compared to using result tags via Maybe or Either data types. As such, HaskHOL avoids including a method for throwing pure exceptions.

Another performance consideration worth mentioning involves the implementation of catchError where 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.

2.4 Protecting Safety

In the implementations discussed in the previous two subsections, numerous parallels are drawn between HaskHOL methods and already existing Haskell analogs. This raises the obvious question of "Why is HaskHOL reinventing the wheel?" In general, the problem is that the existing Haskell methods present the opportunity for the safety of the proof system to be destroyed. For example, note the similarities between putExt and the put method from the

State monad. The concern with exposing put functions like these 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 () ≈ put. To solve this specific issue, HaskHOL avoids deriving instances of the MonadState and MonadIO classes for the HOL monad. 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.

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.

That is not to say that the authors present HaskHOL as a bulletproof system. In fact, there still exists a number of ways to break the safety of not only HaskHOL, but the underlying Haskell runtime system as well; manually deriving bad Typeable instances is one commonly cited example. To do so would be akin to getting into a car, putting on the seat belt, adjusting all of the rear view mirrors, and then intentionally crashing head first into a wall at 100 MPH. In short, it is our goal to provide the safest system possible along with a collection of best practices to avoid any remaining safety issues. If users chose to ignore those practices, there is not much more that we can do.

3 Performance Evaluation

Shown below are a collection of benchmarks comparing the current implementations of HaskHOL and HOL Light. The test problems used are selected from the Intuitionistic Logic Theorem Proving (ILTP) library [27]. Similar to the Thousands of Problems for Theorem Provers library [30], the ILTP library is designed to test automated theorem provers, specifically those restricted to handling only intuitionistically true problems. The restriction to intuitionistic tautology checking was selected as the authors feel it represents the perfect "worst case scenario" to test. While not typically used for proof directly, the ITAUT rule is critical in bootstrapping later first-order logic theories. Given this, an efficiency bottleneck in the intuitionistic theory would necessarily be reflected in later theories, potentially with an exponential explosion in slowness.

The problems selected from the ILTP library are those belonging to Roy Dyckhoff's benchmark library [9]. These problems are expressed as classes of formulae that are scalable in complexity and have variations that are both intuitionistically valid and invalid. By selecting the appropriate members of these classes, namely those intuitionistically true with complexity of n=3, we can guarantee both termination of the ITAUT rule and a significant enough run

time to be profiled. The results of running these tests are shown in Table 1.

Table 1: Comparative Performance of Related Provers

Class	Prover	Time (sec)
de Bruijn	HaskHOL	16.124
	HOL Light	10.932
Pigeon Hole	HaskHOL	0.117
	HOL Light	0.104
N-Contractions	HaskHOL	0.228
	HOL Light	0.187
Big Natural Deductions	HaskHOL	0.00235
	HOL Light	0.0374
Korn and Krietz	HaskHOL	0.141
	HOL Light	0.137
Equivalences	HaskHOL	0.00127
	HOL Light	0.0202

All results were gathered on the same machine[1] averaging five executions of each test. HaskHOL was tested with a framework built using the latest Platform Haskell [2] and Criterion benchmarking library [26] releases. HOL Light was tested less rigorously, using its built in time function.

For all tests, HaskHOL was able to perform at roughly the same level as HOL Light, even managing to surpass it in two cases. The biggest gap between the two provers occurred with the test for the de Bruijn class of problems, where HaskHOL was slightly over five seconds slower. This problem class was shown to be the most difficult for both provers, likely because of both the size of the de Bruijn term and the amount of backtracking that the proof required. It is the belief of the authors that the inferior performance of HaskHOL is because of this backtracking, with the problem being rooted in the inefficiency of the catchError function indicated in Section 2.3.

For the other tests, given the short run times and potentially different overheads for the varying test frameworks, it's impossible to make definitive claims about the comparative performances. However, at least for these classes of problems, it is not an unreasonable claim to state that HaskHOL's performance is now at a level where its speed alone is not reason enough to deter its use.

Further testing is planned to both gauge the performance of more advanced theories and to compare HaskHOL against other members of the HOL theorem proving family. In order to achieve both, the authors are targeting a future integration of HaskHOL with Joe Hurd's OpenTheory theory library for HOL systems [20]. Given that OpenTheory is authored in ML, where as HaskHOL is authored in Haskell, there still remains some questions about how this integration will work. As such, it is unknown when this additional testing data will be available.

4 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.

[1] 2.2 GHz Core 2 Duo, 4 GB 667 MHz DDR2 SDRam, OSX 10.7.2, GHC 7.0.4, OCaml 3.11.1

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 [13]. 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 [3], 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.

HOL Zero is another relatively new member of the HOL theorem prover family who shares the goal of improving the guarantee of safety [4]. Like the more traditional members of the family tree, HOL Zero still relies pervasively on side effects for its implementation. It also differs from HaskHOL in that it is not trying to position itself as a general purpose theorem prover, but rather as a proof checker used to confirm the validity of proofs from other systems. The HOL Zero project certainly appears to have merit, especially given that it's already been used to identify a number of unsoundness issues in prover implementations. Interestingly enough, the majority of these issues seem to have their roots in the use of OCaml as an implementation language, perhaps further motivating the HaskHOL work, if only for its choice of a different implementation language.

Our first attempt at integrating a Haskel-based prover into the Rosetta tool suite was Prufrock [31]. 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.

5 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. As noted in Section 2.3, 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. There are two possible paths that can be followed to achieve this goal. The first would be to reformulate the definition of HaskHOL's kernel to minimize the amount of information that needs to be carried in the context in the first place. This is similar to the Stateless HOL work of Freek Wiedijk [32]. It is worth noting that Wiedijk indicates that there is a slight performance loss when using this technique, so it is yet to be seen if the increase in exception handling performance would offset it for a net gain. The second path would be to use Template Haskell to construct the context as pointers to top level values that store the actual data.

The use of Template Haskell with HaskHOL has already been explored in depth for the purpose of compile time quasi-quotation of terms and compile time proof. Both features expose 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 by tagging HOL computations with numerous additional type variables. There remains works to be done, though, mainly to minimize the enormous amount of code bloat that this technique causes.

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 [23] interpreting Angel, a generic tactic language [22] 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 checking intuitionistic or first-order logic propositions for tautologies. 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.

References

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

[2] The Haskell Platform. Website:. http://hackage.haskell.org/platform/.

[3] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, Ulf Norell, and Ulf Norell. Verifying haskell programs using constructive type theory. 2005.

[4] Mark Adams. Introducing hol zero - (extended abstract). In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 142–143. Springer, 2010.

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

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

[7] 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.

[8] 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.

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

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

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

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

[13] Florian Haftmann. From higher-order logic to haskell: there and back again. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 155–158, New York, NY, USA, 2010. ACM.

[14] John Harrison. Hol light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96), volume 1166 of Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

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

[16] 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.

[17] 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.

[18] Paul Hudak. Building domain-specific embedded languages. *ACM COMPUTING SURVEYS*, 28, 1996.

[19] Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[20] Joe Hurd. The opentheory standard theory library. In *NASA Formal Methods*, pages 177–191, 2011.

[21] 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.

[22] A. P. Martin, Paul H. B. Gardiner, Jim Woodcock, and Jim Woodcock. A tactic calculus-abridged version. pages 479–489, 1996.

[23] Andrew Martin, Jeremy Gibbons, and Jeremy Gibbons. A monadic interpretation of tactics. 2002.

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

[25] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[26] Bryan O'Sullivan. Criterion, a new benchmarking library for haskell. Website:. http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/.

[27] Thomas Raths, Jens Otten, Christoph Kreitz, and Christoph Kreitz. The iltp problem library for intuitionistic logic. pages 261–271, 2007.

[28] 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.

[29] Don Stewart and Spencer Sjanssen. Xmonad. In *Haskell*, pages 119–119, 2007.

[30] 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.

[31] 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.

[32] Freek Wiedijk. Stateless hol. In *TYPES*, pages 47–61, 2009.