HaskHOL: A Haskell Hosted Domain Specific Language Representation of HOL Light

Evan Austin and Perry Alexander

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

Abstract. Traditionally, members of the higher-order logic (HOL) theorem proving family have been implemented in the Standard ML programming language or one of its derivatives. This paper presents a description of a recently initiated project intended to break with tradition and implement a lightweight HOL theorem prover library, HaskHOL, as a Haskell hosted domain specific language. The goal of this work is to provide the ability for Haskell users to reason about their code directly without having to transform it or otherwise export it to an external tool. The paper also presents a verification technique leveraging popular Haskell tools `QuickCheck` and `Haskell Program Coverage` to increase confidence that the logical kernel of HaskHOL is implemented correctly.

Key words: Haskell, HOL, HOL Light, Theorem Prover

1 Introduction

Modern higher-order logic (HOL) theorem provers have a rich history dating back to when Michael Gordon first modified Cambridge LCF, a system based on Robin Milner's Logic for Computable Functions, back in the late 1980s[10]. Starting with HOL90, a reimplementation of the first stable release of a HOL system (HOL88), these theorem provers have all shared more than their logical basis; they were all implemented in Standard ML or one of its derivatives. This is a trend that has continued to this day, leaving users of other functional programming languages with few to no native representations of a HOL system.

HaskHOL aims to correct this deficiency for Haskell by providing a hosted domain specific language (DSL) representation of a lightweight HOL theorem prover that users can leverage to reason about their code without having to leave the Haskell universe. The design and implementation of HaskHOL is heavily influence by HOL Light, a popular member of the HOL theorem proving family developed by John Harrison, sharing its logical kernel and data type representations[12]. The HOL Light system was selected as the basis of HaskHOL because it has a much simpler logical kernel when compared to other HOL provers while still maintaing comparable proving power and demonstrating an impressive track record of successful verifications of industrial problems.

This paper will present the challenges that arose when implementing a logical system that has previously relied on a strict and impure functional language, OCaml, in a lazy and pure functional language, Haskell, and how the HaskHOL implementation overcomes them. Specifically, we explore the use of Haskell type classes to replicate the original HOL Light's quotation parsing functionality and the use of monads to represent the various side effects that the HOL Light kernel requires. Additionally, the paper documents the use of two Haskell tools, `QuickCheck`[5] and `Haskell Program Coverage`[9], to raise assurance that HaskHOL's implementation of HOL Light is correct.

2 Motivation

The lack of a simple and lightweight Haskell theorem proving library is a problem that has recently manifested itself in the research of the System Level Design Group (SLDG) at The University of Kansas. The SLDG is leading the development of Rosetta, a specification language for system-level hardware and software co-design[1]. Part of this development involves implementing and maintaing a robust toolset to aid in the design and verification process. One of these tools provides a transformation from Rosetta to PVS, a specification language and verification toolset developed by SRI International[15], to prove properties specified within system components. Often the targeted properties are simple enough that they could be proved without requiring the full level of specification and interaction that PVS provides. For example, a property that states that the concatenation of two bit vectors results in a new bit vector with a length equal to the sum of the individual bit vectors' lengths can be easily proved without having to translate an entire component from Rosetta to a PVS specification. Instead, it might be preferred to interface with a lightweight theorem proving system that will either prove the property trivially or pass it to a decision procedure solver to do the heavy lifting, in either case handling the proof automatically without user interaction.

Additionally, there may be times when we desire to reason about the Haskell code of the tools directly, or to extend the tools themselves with reasoning power, like a type correctness condition (TCC) discharger for the Rosetta type checker. For this reason, we used a theorem prover library that can be leveraged at the Haskell level. There have been several attempts to bridge a connection between Haskell code and external reasoning tools, several of which are mentioned in more detail in the related work section (Section5). All of these attempts take an approach similar to the Rosetta to PVS transliteration in that they require outside tools to move between the code universe and the reasoning universe and still require a degree of user interaction once inside the reasoning universe. HaskHOL attempts to bypass these restrictions by representing the theorem proving library as a hosted DSL allowing the programmer to reason about their code without ever leaving Haskell. Furthermore, because of its implementation as a library, applications can leverage the reasoning power without relying on external tools. This is similar to how Haskell applications can include the Parsec library, gaining parsing capabilities without having to rely on an external parser.

3 Implementation

As was mentioned in the introduction, HaskHOL is heavily influenced by HOL Light, sharing its logical kernel and term, type, and theorem representations. Because of this, it should be noted that, where appropriate and possible, names of constructors and functions in the implementation of HaskHOL are the same or similar to those selected in the original implementation of HOL Light. This was an intentional design decision to help those familiar with HOL Light quickly acclimate themselves to HaskHOL. HaskHOL began as a direct port of HOL Light and was gradually refined to take advantage of Haskell-specific features. As we demonstrate in the following subsections, sometimes this was done because Haskell lacks the impure features of OCaml that HOL Light relies on, and other times it was done because it allowed a duplication of a HOL Light feature in a simpler way or in a way that could shrink the trusted computing base of the prover comparatively. As HaskHOL matures beyond the kernel, the use of these Haskell-specific features increases the distance between itself and HOL Light, an observation that will be discussed in more detail in the conclusion and future work section (Section 6).

3.1 Terms and Types

At the lowest level, the HOL Light logical system is based on a typed version of Church's λ-calculus. Each term can be expressed as a variable, constant, application, or abstraction, and each term has a unique type that is either a type variable or an application of types to a constructor. HaskHOL, like the original HOL Light implementation in OCaml, captures this concept with recursive data types as shown below:

```
data HolType
    = TyVar String
    | TyApp String [HolType]

data HolTerm
    = Var   String  HolType
    | Const String  HolType
    | Comb  HolTerm HolTerm
    | Abs   HolTerm HolTerm
```

Given these constructors, the boolean type can be expressed as `(TyApp "bool" [])` and the truth term can be expressed as `(Const "T" (TyApp "bool" []))`. A potential burden to the user does exist when trying to express larger terms using these primitive constructors. To combat this, HOL Light provides extra syntax for quotation parsing which allows a user to express a term in a more "human friendly" string representation.

HaskHOL approaches this problem in a similar manner, exposing more expressive external data types, `Term` and `Type`, parsing functions to translate from strings to `Term` and `Type`, and an elaboration function to transform from `Term` to `HolTerm` and `Type` to `HolType`. A Haskell type class, `TermRep`, is used to define a function to translate any one of these term representations to a `HolTerm`. Using this type class in the context of a wrapper function allows the user to call a function that requires a `HolTerm` input with any of the previously mentioned term representations that they would like. This is demonstrated in the code below (Note: `HolM` is a monadic type and will be explained in Section 3.4 and `<=<` is simply a monadic composition operator.):

```
class TermRep a where
    toHT :: a -> HolM HolTerm

someRule :: (TermRep t) => t -> HolM a
someRule = someRule' <=< toHT

someRule' :: HolTerm -> HolM a
someRule' tm = return undefined

-- String representation
ex1 = someRule "x:bool"

-- Term representation
ex2 = someRule (As (EVar "x") TyBool)

-- HolTerm representation
ex3 = someRule (Var "x" (TyApp "bool" []))
```

This technique of using a type class with a wrapper function provides similar functionality to quotation parsing without having to increase the trusted computing base of the prover system by relying on an additional pre-processor, like camlp5 for OCaml, or a compile-time metaprogramming library like Template Haskell that provides quasi-quotation capabilities.

3.2 Logical Kernel

The logical kernel of HOL Light is based on the traditional notion of theorems, a data type that represents a logical formula, or boolean term, that has been proved. The notation $\vdash c$ represents a theorem that says c is proved and $a_1, \ldots, a_n \vdash c$ represents a theorem that says c is proved under the assumption that $a_1, \ldots, a_n$ are all also proved. HaskHOL represents theorems with the following data type and type synonym:

```
data Sequent a c = Seq [a] c

type Theorem = Sequent HolTerm HolTerm
```

In HaskHOL, a `Theorem` should only be created through the application of one of the primitive inference rules of the HOL Light logical system. The primitive rules themselves are not interesting in the context of this paper, however, their implementation for HaskHOL is.

As mentioned in the introduction, traditionally HOL theorem provers have been implemented in a version of Standard ML, a strict and impure functional programming language. HaskHOL breaks from tradition in that it is implemented in Haskell, a lazy and pure functional programming language. The challenges presented by moving from a strict/impure language to a lazy/pure language become apparent immediately in the implementation of the primitive inference rules. For example, the original OCaml implementation of one of the rules, `ASSUME`, and the direct Haskell translation is given below:

Original HOL Light Code

```
let ASSUME tm =
    if Pervasives.compare (type_of tm) bool_ty = 0
    then Sequent([tm],tm)
    else failwith "ASSUME: not a proposition"
```

Equivalent Haskell Code

```
cASSUME :: HolTerm -> Theorem
cASSUME tm = if (type_of tm) == tybool
             then Seq [tm] tm
             else error "cASSUME: not a proposition"
```

The issue with the presented Haskell code is that `error` is not a very robust way to report errors in a program. Haskell's lazy evaluation means that the exceptions thrown by `error` due to a non-well-typed term or a failure earlier in a chain of inference rule applications can potentially remain hidden or unaddressed instead of being handled immediately like it would be in a strict language. Furthermore, the only way to catch an exception thrown by `error` is to do so in the `IO` monad. Given these limitations, HaskHOL instead uses the `Error` monad to take advantage of its `throwError`/`catchError` functions and its strictness in dealing with whether or not a previous monadic computation resulted in an exception. The updated Haskell code is shown below:

Updated Haskell Code With Error Monad

```
cASSUME :: (MonadError String m) => HolTerm -> m Theorem
cASSUME tm =
    do { ty <- type_of tm
       ; if ty == tybool
```

```
          then return $ Seq [tm] tm
          else throwError "cASSUME: not a proposition"
        }
```

The use of monads to represent side effects, like the error exceptions in this case, is common in Haskell programs and is something that will again be utilized in later sections.

3.3 Context

The previous section presented the definition of a theorem as a_1, ..., $a_n \vdash c$. Perhaps more accurately, the definition should be Γ a_1, ..., $a_n \vdash c$ where Γ is a context holding all of the types, constants, definitions, and axioms that have been loaded and the environments for type and term variables necessary to prove the theorem. The original HOL Light implementation takes advantage of OCaml's global references to keep track of all of these pieces of the context, as demonstrated below for constants:

```
let the_term_constants =
     ref ["=",Tyapp("fun",[aty;Tyapp("fun",[aty;bool_ty])])]

let get_const_type s = assoc s (!the_term_constants)
```

HaskHOL instead represents the concept of a context with a new datatype using record syntax as shown below:

```
data HolContext = Ctxt {constants::[(String, HolType)]}

initCtxt =
  Ctxt [("=", TyApp "fun" [aty, TyApp "fun" [aty, bool_ty]])]

lookupConstType :: String -> HolContext -> Maybe HolType
lookupConstType x env = lookup x (constants env)

get_const_type :: (MonadError String m) =>
                   String -> HolContext -> m HolType
get_const_type x env =
  case lookupConstType x env of
    Nothing -> throwError "get_const_type"
    Just ty -> return ty
```

The obvious challenge is how to make the this context available to the functions and inference rules that need it given that Haskell has no notion of global state. As is standard in Haskell programs, this is done using the `State` monad. Implementations of the `mk_const` function in OCaml and Haskell are shown below for comparison:

Original HOL Light Code

```
let mk_const(name,theta) =
    let uty = try get_const_type name with Failure _ ->
      failwith "mk_const: not a constant name" in
    Const(name,type_subst theta uty)
```

Haskell Code Utilizing State Monad

```
mk_const :: (MonadState HolContext m, MonadError String m) =>
             String -> HolTypeEnv -> m HolTerm
mk_const name tyenv =
  do ctxt <- get
     catchError (do uty <- get_const_type name ctxt
                    return $ Const name $ type_subst tyenv uty
                (\ _ -> throwError "mk_const")
```

3.4 Putting It All Together

As became evident with the `mk_const` example, the combination of using the `Error` and `State` monads can lead to a relatively large context for the type of a HaskHOL rule or function. To address this concern, HaskHOL includes a type synonym that uses monad transformers to create a new monad type, `HolM`. This type and an associated run function is shown below:

```
type HolM m = StateT HolContext (ErrorT String m)

runHolT x = runErrorT (evalStateT x initCtxt)
```

Putting this together with what has been demonstrated in the previous sections we have HaskHOL's final representation of HOL Light's primitive inference rules and associated functions, as demonstrated by the updated `mk_const` and `cASSUME` examples shown below:

HaskHOL Code

```
mk_const :: (Monad m) =>
             String -> HolTypeEnv -> HolM m HolTerm
mk_const name tyenv =
  do ctxt <- get
     catchError (do uty <- get_const_type name ctxt
                    return $ Const name $ type_subst tyenv uty
                (\ _ -> throwError "mk_const")


cASSUME :: (Monad m, TermRep a) => a -> HolM m Theorem
cASSUME = cASSUME' <=< toHT

cASSUME' :: (Monad m) => HolTerm -> HolM m Theorem
```

```
cASSUME' tm =
    do { ty <- type_of tm
       ; if ty == tybool
         then return $ Seq [tm] tm
         else throwError "cASSUME: term not a proposition"
       }
```

4 Verification with `QuickCheck`

The ultimate goal of a theorem prover is to provide a verification that some property is correct, or true. The question arrises, though, how do you know when the prover itself is correct? Most LCF style provers, including HOL Light and HaskHOL, reduce all inference rules down to a combination of primitive rules implemented by a much smaller logical kernel. The question still remains, though, how do you know when the kernel is correct? HaskHOL attempts to answer this question by verifying the kernel using `QuickCheck`, a tool specifically designed for automated testing of Haskell programs.

4.1 Brief Overview of `QuickCheck`

`QuickCheck` works by having the user define a property about their code that should hold true, and then generating a large, random set of data to test this property. For example, if the user would like to verify that their definition of addition is commutative, the following property could be written:

```
prop_add_com :: Int -> Int -> Bool
prop_add_com x y = (x + y) == (y + x)
```

The `quickCheck` function could then be called with the desired user settings, including values like the total number of tests to run and the number of discarded inputs allowed before failure. It would randomly generate values for x and y and confirm that with each pair of values the property holds. Testing the kernel of HaskHOL is not this simple however, because, as was shown in Section 3.4, all of the primitive inference rules are monadic in nature.

To handle this, `QuickCheck` provides an extension to test monadic code[6]. Four main functions are added to help convert a monadic property (`PropertyM`) to a regular property. For reference purposes, their names and types are shown below:

```
pre :: Monad m => Bool -> PropertyM m () Source
run :: Monad m => m a -> PropertyM m a
assert :: Monad m => Bool -> PropertyM m () Source
monadicIO :: PropertyM IO a -> Property
```

The `pre` function converts a regular boolean to a `PropertyM` to express a precondition, `run` converts a monadic computation to a `PropertyM`, and `assert` converts a regular boolean to express a postcondition. Once all of the pieces of the property are converted to PropertyM computations, a function like monadicIO will convert this monadic property into a regular property which quickCheck can operate over. This technique is the one applied to verify HaskHOL's logical kernel and is explained in more detail in Section 4.3.

4.2 Term Generation

In order for QuickCheck to be able to generate the random inputs to test properties, an instance of the Arbitrary type class must be defined for the type of input that needs to be generated. Most typically this is done using combinators and functions predefined in the QuickCheck library to help guarantee that the inputs generated fit the random nature desired. For a simple recursive data type like HolTerm or HolType this is a relatively straightforward and well documented process. Depending on the property that is to be tested, though, simple random generation may not be appropriate. For example, a few of of the primitive inference rules of HaskHOL expect a well typed term or proposition as input. Rather than generating an extremely large number of random inputs in search of these terms, it might be preferred to generate only those terms to begin with. HaskHOL includes several of these more specialized generators and associated property quantifiers, including a generator that creates a term from a given type, term_from_type, and a generator and quantifier for well typed propositions shown below:

```
genWellTypedProp :: Int -> Gen HolTerm
genWellTypedProp n = do tm <- term_from_type tybool n
                        return tm

forWellTypedProp :: Testable b => (HolTerm -> b) -> Property
forWellTypedProp = forAll $ sized genWellTypedProp
```

4.3 Verifying Inference Rules

Verifying an inference rule with QuickCheck is a simple, two-step process. First a proposition must be written that captures the expected behavior of the rule. This involves identifying the inputs to be randomized, applying the rule to these inputs, and comparing the result to the expected output. For example, the proposition to test the cASSUME rule is shown below:

```
assume :: (Monad m) => HolTerm -> m Bool
assume t = do res <- runHolT $ cASSUME t
              let ans = Seq [t] t in
                case res of
                  Left _ -> return False
                  Right thm -> return $ thm == ans
```

In this case we want to supply a `HolTerm` to `cASSUME` and expect to get back a a `Theorem` that shows that term proved under the assumption of itself, captured by `Seq [t] t`. After the application of the rule, `False` is returned if there is a failure, otherwise, the comparison between the result and the expected answer is returned.

The second step involves identifying all of the cases where this rule may succeed or fail and developing a property to test for each case. For example, `cASSUME` should always succeed when supplied with a well typed proposition and should always fail when not. The property quantifier from Section 4.2 can be used to guarantee the first condition and can be combined with the monadic functions from Section 4.1 to produce the property shown below:

```
prop_assume1 =
  forWellTypedProp (\ t -> monadicIO $ do res <- assume t
                                          assert res)
```

Likewise, a property quantifier that generates well typed terms of a random type can be combined with a precondition that the term is not a boolean type to produce a property to test for failure, again as shown below:

```
prop_assume2 =
  forWellTypedTerm (\ (ty, t) ->
                       monadicIO $ do pre (ty /= tybool)
                                      res <- assume t
                                      assert $ not res)
```

4.4 Measuring Test Coverage With `Haskell Program Coverage`

The obvious question presented by verification of this style is how does one know when all of the possible success and failure conditions have been identified. HaskHOL attempts to answer this question by using `Haskell Program Coverage`, a tool-kit specifically designed for this purpose. `Haskell Program Coverage` works by compiling a Haskell program with the `-fhpc` flag and then during the execution of the program keeping track of which expressions, alternatives, and local declarations are used as well as the coverage of boolean conditions in the program. This information can be reported in a short summary format using the `hpc report` tool as demonstrated by the following terminal dump:

```
$ ./main
Testing cASSUME Success
+++ OK, passed 1000 tests.
Testing cASSUME Failure
+++ OK, passed 1000 tests.
$ hpc report main
  7% expressions used (290/4135)
  1% boolean coverage (1/56)
```

```
     0% guards (0/18), 18 unevaluated
     2% 'if' conditions (1/38), 1 always False, 36 unevaluated
   100% qualifiers (0/0)
 5% alternatives used (21/357)
 2% local declarations used (1/48)
10% top-level declarations used (28/259)
```

Unfortunately, the way that `Haskell Program Coverage` works is that this report actually contains statistics for every library imported by the program, not necessarily just the functions that are being targeted by the testing. There are flags to reduce the reporting to a per-module basis or to exclude certain modules. However, it is still difficult to see how these statistics relate to the Haskell code that has been written. For this reason, `Haskell Program Coverage` also provides the `hpc markup` tool. This tool produces annotated HTML files that use a color-coding system to indicate which portions of the code have been visited and which booleans evaluate to always true or always false, potentially indicating a poorly written boolean condition or test of that condition.

At this point, HaskHOL leverages this combination of `Haskell Program Coverage` and `QuickCheck` to verify its logical kernel to provide assurance as to the correct implementation of the already widely used and highly trusted logical kernel of HOL Light. Furthermore, it is planned to extend this testing procedure to any extensions made to the HaskHOL kernel including derived rules and the tactic language.

5 Related Work

Recent work has been completed by Florian Haftmann to provide higher-order logical reasoning in 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, another popular member of the HOL theorem proving family, and executable Haskell source[11]. The generation of code from an Isabelle specification is presented as an established and mature tool, however, Haskabelle, the tool which provides a translation in the opposite direction, still appears to be a young and not fully realized utility. Because Haskabelle more closely matches the intended use case of HaskHOL, proving properties of previously written Haskell code, it will be interesting to see how this tool develops and what advantages and disadvantages appear when comparing the two different approaches.

Agda is another tool that attempts to allow formal reasoning about existing Haskell code utilizing its own dependently typed language and proof assistant whose concrete syntax is heavily inspired by Haskell[2]. Because of the similarities between the two languages, it is possible for Agda to translate code produced from compiling Haskell into an equivalent Agda specification which can be reasoned about using Agda's 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 reason about Haskell programs with external tools, there is at least one major attempt besides HaskHOL to bring these reasoning capabilities to Haskell programs directly. Ivor is a type theory based theorem prover library that provides an API for embedding theorem proving capablilites inside of Haskell applications[4]. Instead of dedicating itself to one fixed logical system, like HaskHOL has done by selecting the HOL Light kernel, Ivor aims to be more of 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 difference in the separation level between the Haskell code and the theorem prover, with HaskHOL taking the hosted DSL approach and Ivor taking the separate program with exposed API approach.

Regarding verification of the HOL Light kernel, John Harrison initiated work to self-verify an imperfect model of HOL Light (lacking definitional mechanisms). Harrison's work is particularly interesting because it appears to be a direct verification of the HOL Light semantics whereas HaskHOL's verification is over a more abstract specification captured by user-written properties.

6 Conclusions and Future Work

HaskHOL at this point encompasses a verified, and, therefore, assumed accurate implementation of the HOL Light logical kernel. Additionally, there has been success extending this kernel to allow for derived rules and definitions for propositional logic and equality reasoning, term conversion, and term rewriting. There has also been work to extend HaskHOL with a tactic language, however, this work has been delayed to instead investigate if the computational model for HaskHOL can be reduced to the applicative functor level, making reasoning about tactics and further extensions easier.

The next immediate target for work with HaskHOL is to build a linkage between it and a decision procedure solver. The goal is to target the SMT-Lib Language[16] with a specialized HaskHOL proof tactic, similar to the work linking Yices to Isabelle[7] and linking Isabelle to more general SMT solvers[3].

There are also several additional ongoing projects at the University of Kansas which would be potentially interesting to link with HaskHOL. Notably, Andy Gill's group is currently developing ChalkBoard, a DSL for image generation, processing, and animation[14]. A potential linkage between ChalkBoard and HaskHOL would present a new and interesting way to display and animate proofs for the purposes of demonstration and teaching. Also coming out of Andy Gill's group is Kansas Lava, a hardware description DSL following the Lava design pattern[8]. A potential linkage between Kansas Lava and HaskHOL would allow formal reasoning about the circuits being designed, again without having to leave the Haskell universe. Hopefully, given more time, HaskHOL can continue to mature and present itself as a viable library for higher-order logic proving within Haskell for all sorts of other, as yet unseen, applications.

References

1. Rosetta Specification Language, `http://rosetta-lang.org/`.
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. Damian Barsotti, Leonor Prensa Nieto, and Alwen Tiu. Verification of clock synchronization algorithms: experiments on a combination of deductive tools. *Form. Asp. Comput.*, 19(3):321–341, 2007.
4. Edwin Brady. Ivor, a proof engine. Draft paper, available at `http://www.cs.st-andrews.ac.uk/~eb/drafts/ivor.pdf`.
5. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.
6. Koen Claessen and John Hughes. Testing monadic code with quickcheck. In *IN PROC. ACM SIGPLAN WORKSHOP ON HASKELL*, pages 65–77, 2002.
7. Levent Erkk and John Matthews. Using yices as an automated solver in isabelle/hol. In *In Automated Formal Methods08*, pages 3–13. ACM Press, 2008.
8. Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.
9. Andy Gill and Colin Runciman. Haskell Program Coverage. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2007.
10. Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.
11. 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*, 2010.
12. 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.
13. John Harrison. Towards self-verification of hol light. In *In International Joint Conference on Automated Reasoning*, pages 177–191. Springer-Verlag, 2006.
14. Kevin Matlage and Andy Gill. ChalkBoard: Mapping functions to polygons. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.
15. S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system, 1992.
16. Silvio Ranise, Loria, and Cesare Tinelli. The smt-lib standard: Version 1.2. Technical report, 2006.