Stateless Higher-Order Logic with Quantified Types

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. There have been numerous extensions to classical higher-order logic, but not all of them interact non-trivially. Two such extensions, stateless HOL and HOL extended with quantified types, generate an interesting conflict in the way that type operator variables are implemented and handled. This paper details a proposed solution to that conflict and explores the key impacts to the logical kernel. A prototype system, implemented in GHC Haskell, is provided along with a discussion of how type classes can be used to simplify the logic from the user's perspective.

1 Introduction

Higher-order logic (HOL) has been in existence for a little over three decades [9]. In that time, the original proof system has spawned an impressive family tree with each descendent imparting their own spin on the HOL design. This paper focuses on the confluence of the logics of two such systems: Freek Wiedijk's Stateless HOL [25] and Norbert Vöelker's HOL2P [24]. The intersection of these logics is problematic in that each introduces the notion of type operators in different ways to serve different purposes.

Taking a step back, it's important to address why these extensions are attractive to users in the first place. There's a growing desire in the Haskell community to formally verify software written in the language. Following from the "eat your own dog food" attitude, there's an equally strong inclination for those tools to implemented in Haskell itself. There have been numerous attempts to meet these goals that we are aware of, but ultimately they all rely on throwing the verification over the wall to an external tool at some point in the process [10, 14]. It is our goal to simplify that workflow by providing a general purpose theorem prover written in Haskell for Haskell.

We have targeted HOL as the base logic for our proof system given its success in verifying a wide variety of problem domains, both in hardware [8, 16] and software [15]. We were also attracted by the large number of HOL based theorem provers with active communities, including Isabelle/HOL [17], HOL4 [21], and HOL Light [11]. Each system has an impressive body of verification work to draw from to compliment our efforts.

When implementing provers in the LCF family tree some variant of the ML programming language is typically used. This presents a challenge when trying to implement a related system in Haskell as it does not offer the side-effectful features that are so pervasively leveraged when working with ML. The authors are painfully aware of this fact after a less than successful attempt to naively translate HOL Light directly into Haskell [3]. This is why we personally are interested in the Stateless HOL system; we view it as a large leap towards a pure and total logical kernel that aligns more closely with Haskell's ideology.

We encounter an analogous problem when moving in the opposite direction. Haskell's numerous type system extensions, especially those included in the Glasgow Haskell Compiler [1], greatly surpass what is traditionally representable in HOL. We look to the HOL2P system for guidance on how to move to a polymorphic lambda calculus in an attempt to capture more of these features directly.

In this paper we make the following contributions. In Section 2 we review the primitive logics for the systems mentioned above, focusing on changes that are relative to our desired features. Then, in Section 3 we identify the inconsistency that occurs with the melding of these systems and propose a modified logic that solves the problem. Finally, in Section 4 we discuss a prototype system based on this logic, HaskHOL, that is implemented using GHC Haskell.

2 Background

The foundation of both Stateless HOL and HOL2P is John Harrison's HOL Light. The goal of HOL Light is to provide a full powered HOL proof assistant with a logical kernel that is simpler compared to those of related systems. The simplified kernel paired with an embedded domain-specific language implementation approach gives the entire system a very lightweight feel, as the name would imply.

Central to the logical kernel is the representation of HOL types and terms. HOL Light's elected representation, shown below [1], maps almost directly to the simply-typed lambda calculus.

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

data HOLTerm
    = Var   String  HOLType
    | Const String  HOLType
    | Comb  HOLTerm HOLTerm
    | Abs   HOLTerm HOLTerm
```

[1] The implementation shown here is written in Haskell to make comparison with the material in Section 4 more direct. It is worth noting, though, that HOL Light, Stateless HOL, and HOL2P are all actually implemented in OCaml, not Haskell.

The only significant between the two is that, rather than fixing the set of base types, HOL Light supports type extension through its `TyApp` constructor. The first field of the constructor is a string identifier for a constant with the second field containing a list of type arguments to apply to the constant.

For example, HOL Light includes two primitive type constants mapping to boolean and function types respectively:

```
tyBool :: HOLType
tyBool = TyApp "bool" []

tyFun :: HOLType -> HOLType -> HOLType
tyFun ty1 ty2 = TyApp "fun" [ty1, ty2]
```

Any other type constants must be introduced through HOL Light's theory extension mechanisms. This restriction is necessary to guarantee that a user can not provide conflicting definitions of a constant within the same working theory. In order to facilitate this design decision, HOL Light leverages global memory references to both track constants as they are introduced and store any information associated with the constant.

2.1 Stateless HOL

Stateless HOL modifies the logical kernel of HOL Light in an effort to sever the dependence on global state for the previously mentioned primitive extension mechanisms. The author of the system cites this as being primarily useful for allowing definitional "undo" functions. These methods would provide the user with the ability to change or remove constants from the current working theory without requiring a complete reloading of the system. As mentioned in the introduction, though, we see value in the stateless approach because of its increased purity.

The principal idea behind the stateless modification is a simple one: embed properties of the kernel types directly such that you no longer need to query global state in order to retrieve them. The implementation is roughly the same regardless of which kernel type you're working with, so we'll focus on the embedding of type constant information since it's the use most relevant to our work.

```
data HOLType
    = TyVar String
    | TyApp TypeOp [HOLType]

data TypeOp
    = TyPrim    String Int
    | TyDefined String Int Theorem
```

The above two data types provide the complete definition of HOL types in the Stateless HOL system. Note that, compared to the original HOL Light system, the first field in the `TyApp` constructor has been changed from `String` to a new auxiliary data type, `TypeOp`, representing type operators.

There are two possible cases for type operators: primitive operators, such as `bool` and `fun`, and defined operators, which are those introduced via theory extension. In either case the `TypeOp` instance carries both the operator identifier and its arity, where as previously the user would have to call a stateful function, `getTypeArity`, in order to retrieve the second piece of information. Instances of defined type operators also carry their definitional theorem which can be used to differentiate between operators of the same name.

2.2 HOL2P

Excluding the redefinition of the kernel types, the logic of Stateless HOL is largely unchanged compared to that of the original HOL Light system. This is not the case with HOL2P, as it looks to extend HOL Light's logic rather than reimplement it in an alternative manner. The 2P, in this case, refers to the move from a simply-typed lambda calculus to a second-order, polymorphic lambda calculus.

Added to the logic are universal types, type abstractions and combinations, and type operator variables, as made familiar by numerous System F based programming languages. Thanks to Coquand, however, we know that the combination of HOL and System F is inconsistent [6] To avoid this problem HOL2P introduces a "smallness" constraint on bound types: universal types cannot abstract over other universal types or type variables that are otherwise unconstrained.

Great care was taken by the system's author to make the changes introduced by HOL2P as minimally invasive to the original HOL Light logic as possible. HOL2P makes only one modification to the implementation of HOL types; the addition of a constructor for universal types:

```
data HOLType
    = TyVar String
    | TyApp String [HOLType]
    | UType HOLType HOLType
```

All other type features of the system are added via auxilliary definition or syntactic distinction in the parser.

The lack of a structural distinction among these elements complicates a number of primitive operations of the logic. For example, the only difference between a type operator variable and a regular type variable is the presence or lack of a `_` character prefixing the variable name. Furthermore, in order to support substitution of type operator variables, these identifiers have to be repeatedly boxed and unboxed by `TyVar` constructors in order to satisfy both the type of the substitution function and the first field in the `TyApp` constructor. Because all

of this must be done in a sound and consistent way formerly trivial methods, such as `mkVarType`, now contain a number of correctness conditions as part of their implementation.

3 Stateless HOL with Type Quantifiers

When combining these logics we first identified what was different between the primitive inference rules of the systems. Excluding the four additional rules that HOL2P added to bring congruence and beta reduction to the type level, the systems share the same ten basic rules of the original HOL Light system. Of these rules, only one had significant differences:

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

Implicit in the `INST_TYPE` rule is the definition of type instantiation; this is where the key change between systems occurs.

In the simplest case, type instantiation is a substitution performed over type variables. For Stateless HOL this substitution is trivially defined by the following rules:

$$\begin{array}{lll} \text{x [ty/x]} & = \text{ty} & \\ \text{y [ty/x]} & = \text{y} & (\text{y} \neq \text{x}) \\ \text{(c } a_1 \text{ ... } a_n\text{) [ty/x]} & = \text{(c } a_1\text{' ... } a_n\text{')} & \end{array}$$

Note that we use a tick notation to indicate recursive application of substitution. The intent here is to show that in the `TyApp` case only the first field is left unchanged. This makes sense from a functional programming perspective where substitution would be defined as a function of type `(HOLType, HOLType)-> HOLType -> HOLType`. Because `c` in this system is of type `TypeOp`, we know that we cannot recursively apply the substitution function to it.

This is not the case for HOL2P as the first argument to a type application may be a type operator variable and, therefore, may be subject to substitution. To handle this possibility we need to add two additional rules. For the sake of completeness, we also show the name capture avoiding substitution rules for universal types:

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

The first rule shown above handles the case when we are substituting a known type constant for a type operator variable. The second rule handles the substitution of a universal type for a type operator variable. In this case we repeatedly perform a beta reduction until we're left with a small type.

As was alluded to in Section 2, these rules are ill-defined. This is due to the fact that, unlike in Stateless HOL, there is no true representation of a type constant or operator outside of the context of a type application. In order to build the substitution argument `[c/_x]`, therefore, we must "fake it" by supplying type variables with identifiers that match those of `c` and `_x`. It is then up to the implementation of the substitution function to perform the correct coercions.

The integration of a stateless approach destroys this work around. Because there is no longer a global state storing information about constants in the kernel, there is no way to convert from a variable identifier to a type operator. This means that the type operator must be constructed before the substitution occurs, changing the argument type from `(HOLType, HOLType)` to `(?, TypeOp)`.

In the above explanation we indicate that the first type of the substitution argument is unknown because it is dependent on the implementation of type operator variables. Since that facet of the type system was not state dependent we could continue to play the coercion games of HOL2P should we want to. However, the change to the second type of the argument is not optional and, more importantly, mandates the separation of the substitution function in to multiple pieces to maintain well-typedness.

As will be discussed in more detail in Section 4, we elect to make the distinction of type operator variables purely structural. As such, we end up with three separate substitution functions. The rules for these functions are shown below along with their associated argument types for clarity's sake:

(HOLType, HOLType) Substitution

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

In all cases, ty must preserve the smallness of x.

(TypeOp, TypeOp) Substitution

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

(TypeOp, HOLType) Substitution

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

It's worth noting that the resultant impact to the logical kernel goes beyond having to add more primitive inference rules to accommodate the additional substitution functions. Because type substitution is such a fundamental operation it's used in number of places, both directly and indirectly. Any use that is fully polymorphic, which is to say it could be any of the above rule sets, requires three separate versions of the calling function. The result, therefore, is a linear explosion in code, both for users and implementers, not just in the kernel, but the entire proof system. We address how we combat this growth in complexity with our prototype implementation, HaskHOL, in the next section.

4 HaskHOL

As has been mentioned multiple times, HaskHOL is our prototype implementation of the logic described in the previous section. Going a step beyond Stateless HOL, we've striven to make the logical kernel of HaskHOL both pure and total. Using monads, as is the tradition when simulating side-effects in Haskell, we then built a stateful layer on top of that kernel. The resulting interface that is exposed the user is nearly identical in form and function to those of the HOL systems that have inspired HaskHOL's development.

The implementation of the kernel types in HaskHOL follows closely from the intersection of Stateless HOL and HOL2P's primitive types. Only two significant changes have been made, both with the goal of moving away from using syntax to distinguish between the various kinds of variables.

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

data TypeOp
    = TyOpVar String
    | TyPrim String Int
    | TyDefined String Int Theorem
```

The first change follows from the discussion in the previous section. We have elected to implement type operator variables as a new constructor in the `TypeOp` data type. This allows us to implement the `(TypeOp, TypeOp)` and `(TypeOp, HOLType)` substitution functions as near direct transcriptions of the rules that define them. Similarly, the second change is made to the structure of non-operator type variables. The distinction between small and unconstrained variables is now carried in an additional boolean argument to the constructor rather than in the variables' names.

While both of these changes greatly simplify the implementation of the type substitution functions, we still have to deal with the code explosion problem discussed at the end of previous section. The first solution we thought of involved using heterogeneous data structures to fold the multiple substitution functions

back into one. Ultimately we decided against going down this path because we couldn't find a data structure that felt satisfying to use. The structures all seemed to be split into two categories, those that were implemented with questionable language hacks and those that were overly burdensome to use.

Instead, we decided to utilize Haskell's type class system to provide an ad hoc polymorphic view of the substitution function. There would still be three separate functions to implement, but we could rely on Haskell to implicitly pass in the correct one at compile time. This way, at least from a user's perspective, the logic of the system is no more complex than it needs to be.

Our solution consists of a multiparameter type class with two methods, one to filter the argument list to only valid pairs and one to perform the substitution. While testing this approach we found that we had a related hiccup with type instantiation. Depending on the type of the instantiation list, name capture issues introduced by type abstractions needed to be handled in different ways, requiring yet another type class. To simplify the types of the methods in our kernel we elected to implement our substitution class as a superclass of our instantiation class. The signatures for both are shown below:

```
class TypeSubst a b where
    validSubst :: (a, b) -> Bool
    typeSubst :: [(a, b)] -> HOLType -> HOLType

class TypeSubst a b => Inst a b where
    instTyAbs :: [(HOLTerm, HOLTerm)] -> [(a, b)] -> HOLTerm
                 -> Either HOLTerm HOLTerm
```

This gives us the following types for our type instantiation function and `INST_TYPE` rule:

```
inst :: Inst a b => [(a, b)] -> HOLTerm -> HOLTerm

primINST_TYPE :: Inst a b => [(a, b)] -> Theorem -> Theorem
```

Using these methods requires no additional work by the user with one small exception. Haskell will always infer the most general type it can for a value. It will also try and select the most general type class instance it can for a value. In the event that an instance does not exist for that general type Haskell will fail with a type inference error. Within the context of our system, this happens whenever the user tries to call one of these functions, either directly or indirectly, with an empty list. This is easily remedied by giving the list any suitable type annotation, as is done in this example: `mkConst "="([]::[(HOLType, HOLType)])`.

At the time of this submission, a partial release of the source code for the HaskHOL system is being made available on the first author's website at `http://people.eecs.ku.edu/~eaustin/`. This release is partial in that it stops at the the stateful layer of the proof system, such that the provided code focuses primarily on the features relevant to the content of this paper. A more complete release is planned on the public Hackage server in the near future and is also available from the authors by request.

5 Related Work

HaskHOL is not alone in trying to bring a general purpose theorem prover to the Haskell community. Agda [5], a combination of a dependently typed programming language and proof assistant, has been attempting to grow a sizable user base for the past few years. While not exactly fitting the "in Haskell for Haskell" paradigm that HaskHOL is striving to fill, Agda gets close thanks largely to the fact that its syntax and design was heavily inspired by Haskell itself. Because of this, it is possible to translate between Haskell code and Agda specification with relative ease which seems to be its main appeal to the Haskell community.

MProver [19] is another proof system developed in Haskell. Like HaskHOL, it is a relative newcomer to the field and still appears to be in an early, developmental stage. The system shows great promise, though, especially with its approach to tackling verification of lazy code through equational reasoning. Rather than selecting an established logic as its base, the foundation of MProver appears to be a new, purpose built logic designed to target the more interesting aspects of the Haskell language.

On the HOL side of things, HaskHOL is hardly unique in its attempt to develop a different and improved proof system. One system, HOL Zero [2], has been drawing a significant amount of praise lately for its approach to higher-order logic. Rather than focusing on improving a user's ability to develop proofs, HOL Zero's driving goal is to make the system as sound and trustworthy as possible. The purpose of doing so is two fold. First, it allows HOL Zero to act as a trusted proof checker for other less reliable or well known systems. Second, it allows HOL Zero to serve as a good example of what a simple and correct theorem prover should be.

HOL Omega [13] is another HOL system that we're keeping a close eye on. We're doing so largely in part because it is similarly influenced by HOL2P, though it chooses to go a different direction by making logical extensions to HOL4 rather than HOL Light. The unique portion of HOL Omega's logic that we're really interested in is the inclusion of a kind system similar to the one found in System Fω. As Haskell users we're no strangers to using kinds in our day to day work, so we're excited to see if a proof system can capture that style of programming in a natural and straightforward way.

6 Conclusions and Future Work

The goal of HaskHOL has always been to be a general purpose theorem prover used in support of Haskell based projects. With the current iteration of HaskHOL's logic described in this paper we feel closer to achieving that goal than we ever have before. This is obviously largely thanks to the authors of the HOL systems that have inspired us up to this point. However, like any good research project, Haskell represents a moving target that seems to have an ever increasing velocity.

Recently, the work on kind promotion has had a serious impact on Haskell's type system and core language [26]. This change opens the door to a wide variety of useful programming techniques that HaskHOL will remain unable to verify until it adds a compatible kind system to its logic. HOL Omega was intentionally mentioned last in the related work section for this reason. It comes very close to what we desire for the next evolutionary step of HaskHOL. We're hoping to one day leverage a similar approach to integrating a kind system with HOL in order to close the capability gap between HaskHOL and Haskell as much as we can.

While we tackle that problem, we hope to simultaneously work with the incredible team behind the HERMIT transformation system [7]. Our hope is that this will be a mutually beneficial relationship. For us, HERMIT represents the path of least resistance to using HaskHOL to verify Haskell at the core level, a desirable target for a number of potential projects. For them, we're hoping that HaskHOL represents a viable method for generating proof objects to provide trust in advanced and complex language transformations. It is our goal to have at least a theoretical connection between the two tools done by this summer.

References

1. The Glasgow Haskell Compiler. Website:. http://haskell.org/ghc/.
2. Hol zero. Website:. http://www.proof-technologies.com/holzero/.
3. Evan Austin and Perry Alexander. Haskhol: A haskell hosted domain specific language representation of hol light. In *Trends in Functional Programming Draft Proceedings*, 2010.
4. Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors. *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.
5. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer Berlin Heidelberg, 2009.
6. Thierry Coquand. A new paradox in type theory. In *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.
7. Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The hermit in the machine: a plugin for the interactive transformation of ghc core language programs. In Voigtländer [23], pages 1–12.
8. M. Gordon. *Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware*. North-Holland, 1986.
9. Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.
10. 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.

11. 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.
12. Tom Hirschowitz, editor. *Proceedings Types for Proofs and Programs, Revised Selected Papers*, volume 53 of *EPTCS*, 2009.
13. Peter V. Homeier. The hol-omega logic. In Berghofer et al. [4], pages 244–259.
14. Brian Huffman. Formal verification of monad transformers. In Thiemann and Findler [22], pages 15–16.
15. 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.
16. T. Melham. *Higher order logic and hardware verification*. Cambridge University Press, New York, NY, USA, 1993.
17. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
18. Benjamin C. Pierce, editor. *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*. ACM, 2012.
19. Adam Procter, William L. Harrison, and Aaron Stump. The design of a practical proof checker for a lazy functional language. In *Trends in Functional Programming Draft Proceedings*, 2012.
20. Klaus Schneider and Jens Brandt, editors. *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.
21. 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.
22. Peter Thiemann and Robby Bruce Findler, editors. *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. ACM, 2012.
23. Janis Voigtländer, editor. *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. ACM, 2012.
24. Norbert Völker. Hol2p - a system of classical higher order logic with second order polymorphism. In Schneider and Brandt [20], pages 334–351.
25. Freek Wiedijk. Stateless hol. In Hirschowitz [12], pages 47–61.
26. Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In Pierce [18], pages 53–66.