A Monadic Approach to the LCF-Style

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. The predominant, root design among current proof assistants, the "LCF-style," is traditionally realized through impure, functional languages. Languages that eschew side-effects in the name of purity, however, present a largely untapped platform for novel experimentation in the implementation of theorem provers. The work in this paper breaks with tradition by detailing a pure, monadic approach to the LCF-style that can be utilized by such languages, e.g. Haskell. A new, HOL system that utilizes this technique, HaskHOL, is introduced and some of the remaining, open problems are discussed.

1 Introduction

There is an ideological split in the functional programming community regarding the role and importance of purity in a language's design. Implementors of interactive theorem provers, however, have almost unanimously elected to rely on impure languages to build their systems. Browse through the documentation and source repositories of the most popular proof tools and you will find some variation of the usual suspects, Lisp and ML. There is another shared trend to be noticed; an implementation style that dates back to early LCF systems [8].

This ubiquitous pairing of impure languages with the LCF-style has spawned an impressive number of successful theorem prover systems; among them: Isabelle [16], Coq [14], Nuprl [2], and every member of the HOL family [9]. Yet, as pure languages, namely GHC Haskell [1], emerge as hotbeds for language theory research, we can not help but feel that analogous opportunities to involve purity in theorem prover research are lying untapped.

Not only are pure languages going largely unused in research related to LCF-style provers, but the pervasive use of impure side-effects in the implementation of these provers has made it almost an unintentional, secondary characteristic of the style. When we first attempted to build our own proof assistant, we broke with this tradition, as we needed a monadic system that could easily integrate with a larger project we were developing. Not only did we fail to find anyone else attempting similar work, but we struggled to translate many of the popular features of these proof systems to a pure implementation. In an effort to document our work, in addition to hopefully stimulating discussion about the applicability of pure languages in the theorem prover realm, we present our monadic approach to the LCF-style.

The work in this paper is structured as follows. Section 2 provides a brief characterization of the LCF-style. Comparisons are drawn to parallel implementation techniques in the functional programming realm, and a basic monadic approach to the LCF-style is presented in Section 3. Section 4 refines this approach by tackling the most difficult proof system side-effect to simulate: global, extensible state. The contents of Section 5 further refine the approach by presenting a strategy for optimization of monadic computations related to stateful proofs. A hybrid attempt at a proof system meta-languages that utilizes meta-programming is explored in Section 6. Finally, the paper closes in Section 7 with a brief introduction to our system, HaskHOL, paired with discussion of some open problems with the system's implementation.

2 Characterizing an LCF-Style Prover

The central tenet of the LCF approach is an abstraction of theorems to a type whose construction is precisely constrained. When this constraint is obeyed, it forces a bootstrapping approach to deriving new methods of construction not found in the core logic. Stated more concretely, when following this implementation technique, the advanced proof capabilities of a system must be reducible to a composition of that system's primitive inference rules. Thus, the soundness of an entire system can be assumed, provided that its logical kernel is itself shown to be sound; this is the essence of the LCF-style.

Critically paired with the LCF-style is a meta-language that can faithfully implement the necessary restrictions on the construction of theorems. At minimum, this language must be strongly typed and have a mechanism for controlling the visibility of data type constructors. Typically, this mechanism is provided in the form of the language's module system, its design of abstract data types, or a combination of the two. Implementation of a proof system, therefore, can proceed directly by mapping its core logic to functions in the meta-language that construct and destruct theorems appropriately.

In functional programming nomenclature, the LCF-style can be generalized as an implementation of an embedded domain specific language (EDSL). As was mentioned in the introduction, the LCF-style is traditionally facilitated through impure features, however, the process for implementing EDSLs in pure languages is both equally common and well understood [11]. The only additional required step is that effects that were implicitly introduced by a proof system's meta-language must now be explicitly structured with a computational monad written in an EDSL's host language. Identifying these effects and figuring out how to accurately simulate them via a monad is the crux of the presented work.

It's worth noting that the LCF-style does not itself mandate the use of impure side-effects. For example, Freek Wiedijk has presented a stateless version of a HOL system whose kernel can trivially be made pure [21]. However, Stateless HOL still elects to implement a stateful layer on top of its kernel in the name of practicality and performance. It is this computational layer that we are targeting for translation to a monadic implementation.

3 A Monadic Approach to the LCF-Style

When simulating the side-effects of an LCF-style theorem prover via a monad, it's important to recognize that a portion of the trusted code base is being shifted from the meta-language to the prover itself. Depending on where the definition of the monad is introduced to the proof system, a variety of potential issues could arise. If the implementation of the monad exists at the logical kernel level, then the guarantee of soundness provided by the LCF approach is potentially at risk, as the monad is now lies in the critical path to theorem construction. Even if the monad implementation resides at a higher level, as it might in a monadic variant of a system similar to the previously mentioned Stateless HOL, there is still a possibility of unintentionally making the system inconsistent.

Protecting the soundness of a monadic kernel can be achieved by extending the LCF-style's core tenet to additionally constrain the construction of arbitrary monadic computations. Obscuring the internal constructors of the monad and its argument type(s), just as would be done for the abstract theorem type, sufficiently implements this requirement. This process is trivially straight-forward, so the remaining discussion is focused on a harder problem: preserving the consistency of a monadic system.

One potential path to a monad implementation is through the use of monad transformers. These transformers implement one class of effects each, such that they can be combined in a stack-like manner to form a single monad displaying a closed set of effects [13]. Take for example a basic `State` and `IO` monad transformer stack that could be used to model the effects in a HOL system:

```
-- Type of the theory context
type Context = ...

-- Type of the monad
type HOL = StateT Context IO
```

This process promotes the reuse of standard library definitions, which can be beneficial, however, it's important to understand the resultant consequences.

In order to facilitate arbitrary transformer stacks, a monad's effects are defined via non-proper morphisms contained within type classes associated with its transformer. In the above example, the `StateT` transformer is associated with the `MonadState` class which provides, most notably, the `get` and `put` morphisms. Note that these morphisms serve as duals and, when using the standard transformer libraries, it is impossible to expose one for use without also exposing the other.

This presents an issue in the stated example, as `put` can be used to inject an inconsistent theory context into a computation. This can happen in a variety of ways, though the simplest is a restoration of an old context after a proof is performed:

```
bad1 :: HOL HOLThm
bad1 =
    do ctxt <- get -- store old theory context
       newAxiom ax -- introduce an axiom
       th <- someProof -- requires ax to succeed
       put ctxt -- restore the old context, sans axiom
       return th
```

This problem is not unique to the morphisms provided by the `MonadState` class. The above definition of `HOL` also utilizes the `IO` monad which itself has an associated class, `MonadIO`. This class provides the `liftIO` method that can used to lift an arbitrary I/O computation into any transformer stack that is rooted with the `IO` monad. Again, this can be used to introduce inconsistency to the system by discarding theory context updates:

```
bad2 :: HOL HOLThm
bad2 =
    do ctxt <- get
       liftIO $ evalState bad2' ctxt
       -- 'evalState' discards the modified context
  where bad2' = newAxiom ax >> someProof
```

As a general rule of thumb, it can be assumed that any transformer that provides a method for lifting computations or destructively updating any of its argument types exposes a pathway to inconsistency.

Regrettably, providing an indirect monad definition via a `newtype` wrapper is not sufficient to protect against these issues. The combination of the GHC extensions `StandaloneDeriving` and `GeneralizedNewtypeDeriving` allows a user to circumvent this wrapper to generate a type class instance, even where one was not previously provided[1]:

```
newtype HOL a = HOL'(StateT Context IO a)
    deriving Monad

deriving instance (MonadState Context) HOL
```

An indirect definition that instead uses a `data` wrapper provides the desired protection, but not without negatively affecting the performance of all computations that use the monad.

The preferable approach is to provide a manually constructed, flattened version of the desired transformer stack. This monad's non-proper morphisms can then be defined individually, such that their visibility can be controlled just like any other top-level definition:

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/deriving.html`

```
module HOL (get) where

newtype HOL a =
    HOL { runHOL :: Context -> IO (a, Context) }

get :: HOL Context
get = HOL $ \ s -> return (s, s)

-- Not exposed external to the HOL module
put :: Context -> HOL ()
put s = HOL $ \ _ -> return ((), s)
```

4 Type-Directed Extensible State

Note that the type of theory contexts was left undefined in the examples from the previous section. This was intentional, as it is not immediately obvious how to model these contexts in the presented, monadic approach. Even among LCF-style theorem provers in the same family, there are differences in implementation. Using members of the HOL family as an example:

– HOL Light [10] – Follows a pragmatic approach, modeling the theory context as an implicit collection of top-level, mutable references.
– HOL4 [19] – Models the theory context as a symbol table maintained as a binary map data structure defined in its pre-kernel system.
– Isabelle/HOL – Models the theory context using the very robust definition of generic proof contexts provided by Isabelle's meta-language, Pure [17].

Shared among these approaches, though, is the notion that the theory context is both heterogenous and extensible outside of the logical kernel. Unfortunately, while GHC provides a standard analog for most of the commonly occurring side-effects leveraged by LCF-style proof systems, it lacks an agreed upon technique for modeling extensible state. Presented in this section is an approach similar to the one utilized by HOL4. The idea of a central symbol table that carries configuration data is seen in other large Haskell systems, such as XMonad [20]. However, as will be discussed in Section 7.1 we are not entirely convinced that this is the optimal technique.

At the heart of the presented approach is a standard Haskell technique for heterogeneity, an existentially typed abstract data type:

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

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

The `ExtState` constructor can be used to box values of different types, provided they have an instance of the `ExtClass` class, making their collection well typed. The inclusion of the `ExtClass` class constraint serves two purposes. First, it acts as a subclass to other type class constraints we need to introduce. Second, it provides a mechanism to define initial values for individual pieces of the theory context. Defining initial values for context extensions allows static configuration information to be passed in a dictionary manner, rather than requiring it be marshaled around by a stateful monad, which can help with performance.

The theory context is modeled as a map whose indices are serializations of the types of the context's extensions. These serializations are produced via the `Typeable` class, as introduced through the `ExtClass` class. Destructive updating of the context, therefore, takes the following form, where `val` is the value to insert or update and `ctxt` is the context map:

```
insertMap (show . typeOf $ val) (ExtState val) ctxt
```

Similarly, context retrieval can be written using the option monad, `Maybe`, to guard the type-safe casting from the existential type to the target type, `a`:

```
fromMaybe initValue $
  do (ExtState val) <- lookup (show $
                          typeOf (undefined :: a)) ctxt
     cast val
```

Note that this approach mandates that each piece of the theory context has a unique type to prevent future extensions from overlapping the index of an old extension. This uniqueness can easily be enforced by utilizing `newtype` wrappers for context types that are not exported outside of the module they are defined.

An example of all of these pieces in play is shown below. Briefly explained, a new unique type is defined for binder operator tokens and the necessary class instance is provided. This allows the system implementor to write methods for adding and retrieving binder operators to be used during term parsing:

```
newtype BinderOps = BinderOps [String]
    deriving Typeable

instance ExtClass BinderOps where
    initValue = BinderOps ["\\"]

parseAsBinder :: String -> HOL ()
parseAsBinder op =
    modifyExt (\ (BinderOps ops) ->
                   BinderOps $ op `insert` ops)

binders :: HOL [String]
binders =
    do (BinderOps ops) <- getExt
       return ops
```

In the above example, `modifyExt` and `getExt` are primitive monadic computations that follow from the general forms of context map manipulation previously discussed.

5 Optimizing Monadic Proof

As was demonstrated in the previous sections, it is certainly possible to provide a monad definition that sufficiently simulates the side-effects required in an LCF-style proof system. This implementation is not necessarily performant, though. Perhaps the most commonly cited criticism of using monads is their impact on computational efficiency within a single thread of execution when compared to equivalent, impure programs. The potential problem is demonstrated below with a small, example program:

```
main :: IO ()
main = print . (flip evalState) 35 $
   do x <- f
      y <- f
      return (x, y)
  where f :: State Int Int
        f = gets fib
```

The important thing to note in the above code is that there is an expensive sub-computation, `f`, that is repeated. This computation depends on the monadic state value, which can visually be identified as being constant between evaluations.

Unfortunately, the compiler cannot make the same observation, thus `f` is evaluated twice. The program *can* be manually optimized, though, by sinking the point of evaluation within the `where` clause:

```
main :: IO ()
main = print $
   let x = f
       y = f in
     (x, y)
  where f :: Int
        f = evalState (gets fib) 35
```

Because `f` now fully evaluates to a pure value, the compiler is able to perform the appropriate inlining and subexpression elimination, effectively cutting runtime time in half.

The above example can be shifted to the domain of theorem proving by imagining `f` as a lemma used within a larger proof. Recall from Section 3 that forcing the evaluation of a proof computation and then using the resultant theorem can raise inconsistency issues. The general problem is that context modifications critical to a proof could potentially be discarded. This can be protected against by enforcing two properties:

1. Only non-context-modifying computations can be forcefully evaluated.
2. A theorem can only be used within a context consistent with the one in which it was originally proved.

Both of these properties can be witnessed statically at the type-level by tagging monadic proof computations with phantom type variables [7]:

```
newtype HOL cls thry a =
  HOL {runHOL :: Context thry -> IO (a, Context thry)}

evalState :: HOL cls thry a -> Context thry -> IO a
evalState m = liftM fst $ runHOL m
```

The first type variable in the definition of `HOL`, `cls`, records a tag for the classification of a computation. This variable is inhabited by one of two possible empty `data` declarations: `Theory`, for theory context-modifying computations; or `Proof`, for effect free, proof computations. The classification type of a computation is inferred from its component, primitive computations. For example, the previously shown `modifyExt` method would be classified as a `Theory` computation given that it is used to update a context extension value. The classification of effect-free computations are left fully polymorphic to prevent type inference from disallowing them to be mixed with `Theory` computations. The `Proof` tag, therefore, is only explicitly used when a witness to the first property is needed.

The second type variable, `thry`, records a tag for the working theory required by a computation. These tags are unique, empty data declarations that should be generated for each theory context checkpoint that is associated with a library. For example, the proof computation for the truth theorem, `|- T`, would carry a tag of `BoolThry` indicating that it requires the definition of `T` from the Boolean logic library. Note that the `thry` type variable also haunts the `Context` type definition to guarantee that theory context values stay tightly coupled to their respective tags.

With these tags in place, it's trivial to define an alias to the evaluation function that provides a guarantee of the first property. For the sake of simplified discussion, assume that there exists a method, `runIO`, that can be used internal to this evaluation function to safely escape the `IO` monad [2]:

```
safeEval :: HOL Proof thry a -> Context thry -> a
safeEval m = runIO . evalHOL m
```

Note, though, that this definition provides no guarantee of the second property. Once `safeEval` returns a pure value, it can be lifted into any computation via the `return` function, regardless if it is consistent with the theory context or not.

Rather than return an unprotected pure value, the resultant value should also be tagged with the theory context used to compute it. This process can made general for all possible values by using an open type family that defines morphisms for both protecting and using protected data safely. One possible implementation is shown below that, when paired with `safeEval`, provides guarantees for both of the desired properties:

[2] This assumption holds as long as primitive computations with non-benign `IO` effects are correctly tagged as `Theory` computations

```
class Protected a where
    data PData a thry
    protect :: HOLContext thry -> a -> PData a thry
    serve :: PData a thry -> HOL cls thry a

instance Protected HOLThm where
    data PData HOLThm thry = PThm HOLThm
    protect _ = PThm
    serve (PThm thm) = return thm

safeProof :: HOL Proof thry HOLThm -> Context thry
          -> PData HOLThm thry
safeProof mthm ctxt = protect ctxt $ safeEval mthm ctxt
```

The `safeProof` method can be used to safely optimize monadic proof in a number of convenient ways, ranging from run-time memoization to staged, compile-time computation. When paired with the system of theory context tags as described above, though, it too strongly enforces the second property. This combination forces type inference to find a monomorphic value for `thry`, such that a tagged value can only be used in the context in which it was computed.

In the absence of primitive "undefinition" methods, LCF-style theory contexts can be assumed to be monomorphic. Thus, a theory context is always consistent, not only with itself, but also with any new context formed through extension. A proper type-level witness to the second property, therefore, would be polymorphic, such that tagged values are allowed to be used in any context subsequent to the one in which it was computed.

As was the case with the basic definition of theory contexts, even within the same family of LCF-style provers there are differences in how the hierarchy of contexts is represented. Again, HOL Light takes a very pragmatic approach and leverages its host language, OCaml's, script-like execution scheme to build a strict, linear ordering of theory contexts. HOL4 and Isabelle/HOL, on the other hand, both have much more complex representations that more closely resemble a semi-lattice structure.

In either case, the method for generating a polymorphic value for the `thry` variable is a translation of the context hierarchy structure to a type class representation. In the interest of simplifying the following discussion, a linear ordering of theories is assumed. Thus, rather than storing only the most recently seen theory checkpoint, theory context tags should store all checkpoints witnessed up to that point. For example, the type of the context associated with the Boolean logic library is shown below:

```
ctxtBool :: Context (ExtThry BoolThry BaseThry)
```

Theory membership can trivially be checked, therefore, by constructing a set of type class instances that iterate over this linear type, much as one would a list. Again using the Boolean logic library as the example, the following instances check a theory tag for the existence of the `BoolThry` type, indicating that the library is required to evaluate a computation.

```
class BaseContext a => BoolContext a
instance BaseCtxt b => BoolContext (ExtThry BoolThry b)

instance BoolContext b => BoolContext (ExtThry a b)
```

In the above code, the first two lines assert a correct linear relationship between the Boolean context and its parent, the Base, or kernel, context. The last line performs the iterative search if the `BoolThry` type is not found at the head of the type. No instance is defined for the base tag type, `BaseThry`, which provides a terminating condition for the type checker.

The `BoolContext` class can now be used as a constraint to indicate that the Boolean logic library is required for a computation. When paired with the same definition of `safeProof` that was shown above, the desired, polymorphic behavior is observed:

```
thmTRUTH :: BoolContext thry => HOL Proof thry HOLThm

pTRUTH :: BoolContext thry => PData thry HOLThm
pTRUTH = safeProof thmTRUTH ctxtBool
```

The optimized version of the truth theorem, `pTRUTH`, can now be safely reused in any context that includes the information from the Boolean logic library.

6 Meta-Programming as a Meta-Language

The previous two sections have detailed how to enhance a basic monadic approach, but both sets of improvements come at the cost of fairly complicated type system wizardry. From the point of view of a system implementor familiar with pure, functional languages, this is burdensome, but not unexpected. However, from the point of view of a theorem prover user, it is clear that the amount of host language code required to be written to develop new theories goes well beyond what is typically called for by a proof system.

Thankfully, the majority of this required code is boilerplate that can be automatically derived. Take for example, the introduction of a new extension to the theory context as described in Section 4. To facilitate the described, type-directed approach to extensible state, the user needs to provide, minimally, two pieces of code:

- A unique `newtype` wrapper for the extension type.
- An instance of the `ExtClass` class that defines the initial value for the extension type.

We have already seen an example of this in the definition of binder operator tokens:

```
newtype BinderOps = BinderOps [String]
    deriving Typeable

instance ExtClass BinderOps where
    initValue = BinderOps ["\\"]
```

In the above code, note that only two key pieces of information need to be provided by the user: the name of the `newtype` wrapper and the initial value; the remainder of the code can be thought of as a template:

```
newtype <name> = <name> <type of initial value>
    deriving Typeable

instance ExtClass <name> where
    initValue = <name> <initial value>
```

GHC provides a library for compile-time meta-programming, Template Haskell [18], that allows the user to generate and splice templated code anywhere in the source of their program. This includes the capability for splicing in top-level declarations, as would be required with the above example. A snippet of the Template Haskell code used to generate a splice for new extensions is shown below:

```
newExtension :: String -> ExpQ -> Q [Dec]
newExtension ext val =
    do val' <- val
       case val' of
         SigE e eTy ->
             let name = mkName ext
                 ty = ... -- newtype declaration
                 extCls = ... -- instance declaration
               in return [ty, extCls]
         _ -> fail "newExtension"
```

Using this general splice, the name of the wrapper is provided as a `String` and the initial value is provided as a Template Haskell expression, `ExpQ`. The latter type is necessary in order to easily handle initial values whose inferable type is polymorphic; for example, a case where an initial value is an empty list. By providing the initial value as a quote containing a type ascription, Template Haskell can easily deconstruct the expression to extract both the value and its intended type. The boilerplate code for binder operators shown above, can now be replaced with the simple, single line splice shown below; a major improvement for the user:

```
newExtension "BinderOps" [| ["\\"] :: [String] |]
```

This process can be specialized for anywhere a sizable piece of boilerplate or template code needs to be used. Among the numerous possibilities:

- Defining new context extensions or configuration flags.
- Controlling theory context construction.
- Marshaling compile-time proof optimizations.
- Safely extracting static information from a context, i.e. definitions or axioms.

When used in this way, Template Haskell, or a similar meta-programming language, begins to act as the meta-language of the proof system at a level above what the host language can provide. The result is a proof system of complexity lying somewhere between lightweight systems, like HOL Light, and more robust systems, like Isabelle, Coq, and Nuprl.

7 HaskHOL

Following the techniques described in this paper, we have implemented our own HOL theorem prover system, HaskHOL, based on a stateless higher-order logic with quantified types [3]. This system is currently available on the public package repository for Haskell, Hackage[3]. At the time of this writing, only the core of the system is publicly available. Users who have GHC Haskell installed along with a copy of the Cabal package management system can easily install HaskHOL with the following command:

```
cabal install haskhol-core
```

Additional, experimental libraries are available by request, however, we are awaiting the official release of GHC 7.8 before stabilizing them for public release.

7.1 Open Problems in HaskHOL

As was hinted at in Section 4, there are a few issues with the implementation of HaskHOL that we view as open problems.

Most of these issues involve the design pattern at the heart of HaskHOL's extensible state mechanism. A perceived advantage of the technique presented in Section 4 was a simplified path to context checkpointing. By requiring context extensions to have an instance of the `Lift` class, a theory context value could be easily spliced to a top-level definition using Template Haskell. This value can be pre-compiled, both improving run-time performance and opening up new optimization opportunities. This system works great, until an extension is introduced that falls into one of the following two classes:

- Values for which an instance of the `Lift` class cannot be derived.
- Values that require a large number of `Exp` constructors to implement a `Lift` instance.

[3] `http://hackage.haskell.org`

The first problem class is most commonly encountered when attempting to store data in the context that is related to a proof system's rewrite engine. Functionally typed objects, such as conversions or tactics, cannot be represented with Template Haskell's `Exp` type, as Haskell lacks the ability to inspect function construction. Instead, these objects must go through a defunctionalization pass before being stored in the context. This process is not only burdensome for the user, it also introduces a fair amount of run-time overhead.

The second problem class is representative more of a perceived weakness of Template Haskell, than the design of HaskHOL's extensible state mechanism. The dominating factor in compile-time performance when using Template Haskell is the typing and renaming of splices. When attempting to splice large values, the maximum memory residency of the GHC compiler skyrockets and the size of the resultant object files become noticeably larger.

We have found a reasonable workaround to these issues that involves constructing pure values by escaping the `IO` monad with `unsafePerformIO` instead of Template Haskell. This solution is obviously non-ideal, however, it seems preferable the extremely long or failed compilations that would occasionally be encountered when building some of HaskHOL's more advanced libraries. Furthermore, the "damage" from using this non-ideal solution is mitigated by restricting its application to only the very large splices that choke the GHC compiler.

It is our belief, though, that the probable solution to these problems is to scrap the idea of theory checkpointing via Template Haskell altogether. A number of people have pointed to the OpenTheory [12] system as a possible alternative, however, we have yet to be able to invest the time to see if it is compatible with our monadic approach.

8 Related Work and Conclusions

There is a small overlap between the functional programming and theorem prover worlds where pure languages do seem to have a place: the development of dependently typed languages that blur the line between proof assistant and programming language. Agda [4], a language based on intuitionistic type theory, is perhaps the most well known system in this domain. Edwin Brady's dependently typed languages, Idris [6] and the now deprecated Ivor [5], are closely related, though their focus appears to be on producing efficient code rather than providing a robust prover environment.

Outside of system implementations, there has been an interesting, recent pairing of monads and higher-order logic. Myreen et. al presented some early work towards a verifiable implementation of HOL Light [15]. This work included a monadic definition of HOL Light's kernel written in HOL4, though, the described monad seemed designed to be easy to verify rather than applicable for real world use. We are especially excited to see how this work develops, as to the best of our knowledge it is the work most closely related to our own efforts.

It is our hope that pure languages find increased traction in the realm of theorem provers. We are certainly not making the claim that current systems are flawed in any way, just that there exist unexplored research opportunities that could potentially benefit proof tools, current and future alike. Our perceived first, logical step in this exploration, a monadic re-envisioning of the LCF-style, has been presented along with a prototype system implementation. Ideally this system, HaskHOL, would serve as both a platform for further research and possibly, if given enough time to mature, a viable, real-world proof system.

References

1. The Glasgow Haskell Compiler. Website:. http://haskell.org/ghc/.
2. StuartF. Allen, RobertL. Constable, Rich Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl Open Logical Environment. In David McAllester, editor, *Automated Deduction - CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 170–176. Springer Berlin Heidelberg, 2000.
3. Evan Austin and Perry Alexander. Stateless Higher-Order Logic with Quantified Types. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 469–476. Springer Berlin Heidelberg, 2013.
4. 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.
5. Edwin Brady. Ivor, a Proof Engine. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*, pages 145–162. Springer Berlin Heidelberg, 2007.
6. Edwin C. Brady. Idris: General Purpose Programming with Dependent Types. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 1–2, New York, NY, USA, 2013. ACM.
7. MATTHEW FLUET and RICCARDO PUCELLA. Phantom Types and Subtyping. *Journal of Functional Programming*, 16:751–791, 11 2006.
8. Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF. 1979.
9. Mike Gordon. From LCF to HOL: A Short History. In *Proof, Language, and Interaction*, pages 169–186, 2000.
10. John Harrison. Hol Light: A Tutorial Introduction. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer Berlin Heidelberg, 1996.
11. Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.*, 28(4es), December 1996.
12. Joe Hurd. OpenTheory: Package Management for Higher Order Logic Theories. pages 31–37.
13. Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

14. The Coq development team. *The Coq Proof Assistant Reference Manual.* LogiCal Project, 2004. Version 8.0.
15. MagnusO. Myreen, Scott Owens, and Ramana Kumar. Steps towards Verified Implementations of HOL Light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 490–495. Springer Berlin Heidelberg, 2013.
16. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.
17. Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. *CoRR*, cs.LO/9301106, 1993.
18. Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.
19. Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In OtmaneAit Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer Berlin Heidelberg, 2008.
20. Don Stewart and Spencer Sjanssen. Xmonad. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 119–119, New York, NY, USA, 2007. ACM.
21. Freek Wiedijk. Stateless HOL. In *TYPES*, pages 47–61, 2009.