Challenges Implementing an LCF-Style Proof System with Haskell

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

***Abstract*—The predominant, root design among current proof assistants, the LCF style, is traditionally realized through impure, functional languages. Thus, languages that eschew side-effects in the name of purity collectively represent a largely untapped platform for exploring alternate implementations of LCF-style provers. The work in this paper details the challenges we have encountered in the development of one such implementation, a monadic approach to the LCF style tailored to the Haskell programming language. The resultant proof system, HaskHOL, is introduced and our current work with it is briefly discussed.**

I. Introduction

There is an ideological split in the functional programming community regarding the role and importance of purity in a language's design. Comparatively, implementors of interactive theorem provers 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 – a boot-strapping implementation style that dates back to early LCF systems [1].

This ubiquitous pairing of impure languages with the LCF style has spawned an impressive number of successful theorem prover systems; among them: Isabelle [2], Coq [3], and every member of the HOL family [4]. Yet, as pure languages, namely Haskell [5], emerge as hotbeds for language theory research, we can not help but feel that analogous opportunities to involve them in theorem prover research are lying untapped.

Parallel to its success in academic research, Haskell is gaining more and more acceptance as a language suitable for industrial application. Its increased "real world" usage has brought about a growing demand for tools that can both help reason about the correctness of Haskell programs and provide reasoning power to Haskell-based systems. The original motivation behind the work documented in this paper anecdotally reflects this demand; we needed a lightweight proof system that could easily integrate with an existing tool suite implemented with Haskell.

When we began designing a monadic proof system suitable to Haskell's standard methodologies, we noted an ancillary trend amongst the previously mentioned provers that were now serving as our inspiration. The pervasive use of impure side-effects in the implementation of LCF-style provers had made it an almost unintentional, secondary characteristic of the approach. As such, not only did we struggle to translate many of the defining features of these provers to Haskell, but we failed to find anyone else who had succeeded at similar attempts.

In an effort to document the challenges we have faced in our work, in addition to hopefully stimulating discussion about the applicability of pure languages for the development of theorem provers and other formal reasoning tools, we present our monadic approach to the LCF style. The work in this paper is structured as follows. Section II 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 III. Section IV refines this approach by tackling the most difficult proof system side-effect to simulate: global, extensible state. Finally, the contents of Section V further refine the approach by presenting a generalized technique for the optimization of monadic computations related to stateful proofs.

II. Characterizing an LCF-Style Prover

The central tenet of the LCF style 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 host 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 target language that construct and destruct theorems appropriately.

The LCF style has a more general analog in functional programming: domain specific languages. As was mentioned in the introduction, the LCF style is traditionally realized through impure features; the process for implementing comparable DSLs in pure languages is both equally common and well understood [6].The only additional required step by a pure approach is that effects that were previously introduced implicitly by the host language must now be explicitly structured with a computational monad.

As a brief aside, it is worth noting that the LCF style does not itself mandate the use of impure side-effects, nor does it dictate how the implementation of a proof system must be structured. Although they have a shared heritage, the systems listed in the introduction differ quite greatly both in design and degree of impurity:

- Isabelle is implemented with Standard ML (SML) [7], a language that is typically cited as having impure aspects as opposed to being labeled entirely impure. Isabelle makes great distinction between its host language and the various meta-languages it provides, e.g. Isabelle/HOL, by defining them in terms of an intermediate logical framework, Isabelle/Pure. The use of the impure features of SML in Isabelle is mostly constrained to the implementation of this framework.
- HOL4 [8] is the most-direct descendent of the early LCF and HOL systems that originally motivated the development of ML, so it should be of no surprise that it too relies on SML as its implementation language. In addition to admitting only a single meta-language, HOL4 differs from Isabelle in that it is less concerned with restricting the use of impure SML features in its implementation.
- Coq is a proof system of complexity similar to that of Isabelle/HOL and HOL4. The main difference between these systems, beyond their differing foundational logics, is that OCaml [9] is used for Coq's implementation language. Compared to SML, OCaml is significantly more impure. Coq takes significant advantage of this impurity in the implementation of its underlying infrastructure, utilizing languages features ranging from mutable data structures to unsafe type coercions.
- HOL Light [10] is yet another proof system that elects to use OCaml as its implementation language. As is implied by its name, this system is notable for its comparatively lightweight implementation. The meta-language exposed to the user is simply OCaml extended with quasi-quotations for concrete syntax. As such, the use of impure language features can be found everywhere in this system, including the implementation of logical theories.
- Stateless HOL [11] is a modification of HOL Light that attempts to bound its use of side-effects. Unlike Isabelle or Coq this system is designed to remove impurity from the logical kernel, rather than constrain it there. The goal of this approach is to make the use of mutable state a matter of convenience, rather than necessity; beyond the kernel, the overall purity of the system lies unchanged.

Given that our original motivation required a lightweight system, the presented monadic approach follows closely from the design of HOL Light and its extensions/modifications. This is an important fact to keep in mind when reading the following sections, as a number of critical implementation choices were made specifically with this "lightweight" goal in mind. The resultant system, HaskHOL, is essentially a shallow embedding of HOL theorem proving in Haskell, rather than a full-fledged proof system like Isabelle, HOL4, or Coq.

III. A Monadic Approach to the LCF Style

When simulating the side-effects of an LCF-style theorem prover via a monad, it is important to recognize that a portion of the trusted code base is being shifted from the host language to the prover itself. For example, instead of being able to rely on a compiler's implementation of references in good faith, it is now upon the monad to correctly structure and simulate mutable state. Depending on where the definition of this monad is introduced in the proof system, a variety of potential issues could arise.

Implementing the computational monad inside the logical kernel of a system puts it in the critical path of constructing theorems, i.e. the primitive inference rules become monadic computations:

```
primTRANS :: SomeMonad m => HOLThm -> HOLThm
          -> m HOLThm
```

The guarantee of soundness provided by the LCF approach is now at risk, as the monad represents a potential mechanism for bypassing the constraints on theorem construction. If the monad implementation instead resides at a higher level of the system, as it might in a monadic variant of the previously mentioned Stateless HOL, there are still a number of ways to make the system inconsistent, intentionally or otherwise.

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

One potential technique for implementing a monad is with 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 providing a closed set of effects [12]. Take for example a basic `State` and `IO` monad transformer stack that could be used to model the effects in a HOL system, as shown in Figure 1. This process promotes the reuse of standard library definitions which can be beneficial, however, it is important to understand the resultant consequences.

In order to facilitate arbitrary transformer stacks, a monad's effects are defined by methods contained within type classes associated with each transformer. In this example, the `StateT` transformer is associated with the `MonadState` class that provides, most notably, the `get` and `put` methods.

Fig. 1. The `HOL` Monad Through Transformers

```
-- Type of a Theory Context
type Context = ...

type HOL = StateT Context IO

runHOL :: HOL a -> Context -> IO (a, Context)
runHOL = runStateT
```

Note that these methods serve as duals such that, when using the standard transformer libraries, it is impossible to expose one for use without also exposing the other. This presents a potential issue, as `put` can be used to inject an inconsistent theory context into a computation. This can happen in a variety of ways, although the simplest is a restoration of an old context after a proof is performed:

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

This problem is not unique to the methods provided by the `MonadState` class. The definition of the `HOL` monad shown in Figure 1 also utilizes the `IO` monad that itself has an associated class, `MonadIO`. This class provides the `liftIO` method that can be 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
       -- 'evalState' discards the modified context
       liftIO $ evalState bad2' ctxt
  where bad2' = newAxiom ax >> someProof
```

As a general rule of thumb, it can be assumed that any transformer that provides methods for destructively updating its notion of environment/state or lifting computations 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 language 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
```

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

Fig. 2. The `HOL` Monad Through Manual Construction

```
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)
```

An indirect definition that instead uses a `data` wrapper provides the desired protection, but not without negatively affecting the performance of the monad. The preferable approach, shown in Figure 2, is to manually construct a flattened version of the desired transformer stack. The requisite methods can then be defined individually, such that their visibility can be controlled just like any other kernel method.

IV. 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 for a 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 – Models the theory context pragmatically as an implicit collection of top-level, mutable references.
- HOL4 – Models the theory context as a global symbol table maintained by the pre-kernel system of its underlying infrastructure.
- Isabelle/HOL – Models the theory context as an Isabelle/Pure generic proof context[2], again created and otherwise maintained by Isabelle's underlying system infrastructure.

Shared among these approaches is the notion that the theory context is both heterogenous and extensible outside of the logical kernel. Unfortunately, while Haskell 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. HOL Light's approach *can* be simulated, however, it would require the use of `unsafePerformIO` that is at best a code smell and at worst a serious threat to the type safety of the system.

[2]These proof contexts are a symbol table for a theory, much like the one stored by HOL4, extended with a number of other values and functions to facilitate stateless, parallel proof.

Instead, this section presents an approach similar to the one utilized by HOL4 and Isabelle/HOL. Comparable uses of a central symbol table to carry configuration data are seen in other large Haskell systems. These tables are implemented using a standard Haskell technique for heterogeneity – constrained, generalized algebraic data types (GADTs):

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

data ExtState where
    ExtState :: ExtClass a => a -> ExtState
```

The `ExtState` constructor shown above can be used to box values of different types, provided they have an instance of the `ExtClass` class. Wrapping all state values with this constructor homogenizes their collection, making it well typed.

The inclusion of the `ExtClass` class constraint serves two purposes. First, it acts as a subclass to other necessary class constraints; in this case, `Typeable`. Second, it provides a mechanism for defining initial values for individual elements of the theory context. This allows static configuration information to be provided at in a dictionary-passing manner, rather than via a computation's state, which can help with performance.

A 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. Context retrieval is implemented as a guarded, type-safe casting from the GADT to the type of the target context extension. 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; again, the visibility of the constructors for this type should be suitably controlled.

An example of these pieces in play is shown in Figure 3. Briefly explained, a new, unique type for binder operator tokens and the necessary class instance are defined. This allows the system implementor to write methods for adding and retrieving binder operators to be used during term parsing. In this example, `modifyExt` and `getExt` are primitive monadic computations that follow from general map manipulations, modified to suit our design of theory contexts as described in the previous paragraph.

Fig. 3. Extensible State Through GADTs

```
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
```

A. Practical Implications of Stateful Monads

Compound monadic computations expand to sequences of binds that enforce an ordering of effects. For example, the `binders` computation from Figure 3 desugars to the following, where `(>>=)` is Haskell's monadic bind operator:

```
getExt >>= \ x ->
  case x of (BinderOps ops) -> return ops
```

The `(>>=)` operator is defined such that its left-hand argument is evaluated first in order to provide the result to its right-hand argument, simulating an imperative execution scheme. As such, a proof that makes explicit use of all the context modifications it depends on is guaranteed to construct a satisfying theory context during its evaluation. Figure 4 contains once such example where a proof of the truth theorem, `thmTRUTH`, depends only on the definition of the truth constant, as introduced to the working theory context by `defT`. This proof will always succeed, assuming a bug-free implementation of the system up to and including its definition.

At first glance the other proof computation shown, `thmI`, appears to exhibit the same guarantee. However, it implicitly relies on a number of context modifications to parse a term (requires the definition of the universal quantifier, `(!)`) and then prove it through rewriting (requires extending the set of basic rewrites). This proof could be changed to make these dependencies explicit, however that is an intractable solution in general. Furthermore, doing so would defeat one of the main purposes of using a mechanized prover which is to lessen the amount of work involved in a proof.

A proof like `thmI` can remain implicit in its dependencies provided that we guarantee the construction of a satisfying context before the proof is attempted. The simplest way to do this is to lift the set of context modifications for a theory to a separate, top-level computation that is evaluated before any proof effort.

Fig. 4. Explicit and Implicit Stateful Effects

```
defT :: HOL HOLThm
defT = newBasicDefinition "T"
  [str| T = ((\ p:bool . p) = (\ p:bool . p)) |]

defI :: HOL HOLThm
defI = newDefinition "I" [str| I = \x:A. x |]

thmTRUTH :: HOL HOLThm
thmTRUTH =
    do tm <- toHTm [str| \p:bool. p |]
       tdef <- defT
       either fail return $
         do th1 <- ruleSYM tdef
            primEQ_MP th1 (primREFL tm)

thmI :: HOL HOLThm
thmI = prove "!x:A. I x = x" $ tacREWRITE [defI]
```

An example of this technique tailored to the evaluation of `thmI` is shown below:

```
loadCtxt :: HOL ()
loadCtxt = defFORALL >> extendBasicRewrites [...]

runHOL (loadCtxt >> thmI) ctxtBase
```

This technique is a monadic approximation of the "implementation of theories as scripts" approach employed by REPL-style proof systems, such as HOL Light. Those familiar with such systems can point out a serious drawback of their use – a significant amount of time is spent preparing the proof environment every time the system is launched. This delay is amplified by the monadic approach described above, as contexts are rebuilt not just once per session, but for each and every evaluation.

More complex systems, such as Isabelle, HOL4, and Coq, avoid this problem by compiling or otherwise storing their theories in a way that allows them to be used on an as-needed basis without requiring repeated work. The implementations of lightweight systems typically provide an embedding that is too shallow to mimic this approach. In HaskHOL, for example, theories are simply a collection of Haskell functions, such that we lack the ability to inspect or otherwise interact with their construction. We can, however, serialize just the values associated with theories, i.e. the previously discussed theory contexts, through a simple adaptation of the approach to extensible state presented earlier in this section.

Most Haskell libraries for persistent data utilize a method of type-directed serialization compatible with the base techniques of this approach. To concretize discussion, the following examples utilize the acid-state library from the Happstack project [13], although the general ideas should be applicable to other libraries.

Fig. 5. Persistent State Through Existential Types

```
newtype HOL a = HOL { runHOL :: FilePath -> IO a }

openLocalStateHOL ast =
    HOL $ \ fp -> openLocalStateFrom fp ast

binders :: HOL [String]
binders =
    do acid <- openLocalStateHOL
                (initValue :: BinderOps)
       binds <- queryHOL acid GetBinders
       closeAcidStateHOL acid
       return binds
```

Data is serialized by acid-state as a collection of binary files whose access is carefully guarded to provide guarantees of the ACID properties. Its API, inspired by database management systems, provides variations on four basic functions: opening a connection to a data store, querying its value, updating its value, and closing the connection. A reimplementation of the `binders` computation using acid-state is shown in Figure 5.

The primary change in this reimplementation is that the state of the `HOL` monad is now a file path where a theory context is stored. The `openLocalStateHOL` method opens a connection to the desired theory context value guided by this file path and an initial value, reusing `initValue` from our `ExtClass`. Once connected to, this value can be queried, as is done in the example, or updated: `updateHOL acid (InsertBinder op)`. The `GetBinders` and `InsertBinder` constructors used here are defunctionalizations of user-written access methods that are automatically generated by the acid-state library. For example, `GetBinders` maps to a method that looks similar to the previous implementation of `binders`, further demonstrating the compatibility of persistent data libraries and the previous approach:

```
getBinders :: Query BinderOps [Text]
getBinders =
    do (BinderOps ops) <- ask
       return ops
```

Contexts are written to disk during the evaluation of their "loading" computations, e.g. `loadCtxt`. By pairing these computations with the context they extend and their associated file path, we can make this creation a dynamic process:

```
data Context = Context
    { ctxtFP :: FilePath
    , ctxtBase :: Maybe Context
    , ctxtLoad :: HOL ()
    }

ctxtNew = Context "new" (Just ctxtBase) loadCtxt

runHOL' thmI ctxtNew
```

In this example code, `runHOL'` is a wrapper to the previously seen `runHOL` function. This new evaluation method checks for the existence of a theory context before using it. If it exists, its file path is passed to `runHOL` and evaluation proceeds as before, otherwise it is created by first evaluating its `ctxtLoad` computation. This ensures that the effort to create a context is never repeated, giving us a lightweight approximation of the compilation process performed by more complex proof systems.

V. Optimizing Monadic Proof

As was demonstrated in the previous sections, it is certainly possible to construct a monad definition that sufficiently simulates the side-effects required in an LCF-style proof system. This implementation is not necessarily performant. 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 this example is that there is an expensive sub-computation, `f`, that is repeated. This computation depends on the monadic state value that we can visually identify as remaining constant between evaluations.

Unfortunately, the compiler cannot make the same observation, thus `f` is evaluated twice. The program *can* be manually optimized 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` is now a pure binding, rather than a monadic one, the compiler is able to perform the appropriate inlining and subexpression elimination, effectively cutting runtime time in half. This example can be shifted to the domain of theorem proving by imagining `f` as a lemma used within a larger proof.

Recall from Section III 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 properties can be witnessed statically at the type-level by tagging monadic proof computations with phantom type variables:

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

evalHOL :: HOL cls thry a -> Context thry -> IO a
evalHOL 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 `updateHOL` 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, otherwise type inference would disallow their mixing with `Theory` computations. Therefore, the `Proof` tag 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 truth from the Boolean logic library. Note that the `thry` type variable also haunts the `Context` type to guarantee that theory context values stay tightly coupled to their respective tags. With these tags in place, it is trivial to define an alias to the evaluation function that provides a guarantee of the first property.

Fig. 6. Protecting Context-Sensitive Data

```
class Protected a where
    data PData a thry
    protect :: Context 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
```

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[3]:

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

However, 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` method, 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 be made general for all possible values by using an open type family that defines methods for both protecting and using protected data safely. One possible implementation is shown in Figure 6 that, when paired with `safeEval`, provides guarantees for both of the desired properties:

A. Polymorphic Protection

The `safeProof` method included in Figure 6 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 encoding of theory context tags as described in the previous section it too strongly enforces the second, enumerated, safety property. With their current formulation, for each call to `protect` the type checker can infer only a single, possible type for the theory tag, such that protected values can be reused *only* in the context in which they were computed.

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

In the absence of primitive "undefinition" methods, LCF-style theory contexts can be assumed to be monotonic. Thus, a theory context is always consistent, not only with itself, but with any new context formed through its extension. Therefore, a protected value should be reusable in the context in which it was computed *and* any of its subsequent extensions. To safely permit this, a theory tag must be polymorphic, with its possible instantiations constrained to the appropriate set of contexts.

As was the case with the abstract representations of theory contexts themselves, the HOL family of provers differs in how they model context hierarchies. Again, HOL Light takes a pragmatic approach and leverages its host language'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 resemble semi-lattice structures. In either case, constraining the theory tag follows directly from translating a context hierarchy to an equivalent type representation.

The technique described in Section V reified theory contexts to the type-level by recording only the most recently seen checkpoint. In order to construct a sufficiently polymorphic view of contexts, we must instead record information about all the checkpoints seen in the construction of a theory, including their relationships. With this information available, constraints on the `thry` type can be expressed as tests for the existence of requisite checkpoints in the type representation of a theory context.

In the interest of simplifying the following discussion, a linear ordering of theories, like HOL Light's, is assumed. Under this assumption, the type of a theory context can be encoded directly by promoting the ordered list of its theory checkpoints to the type level. We perform this promotion by mimicking the syntax of list construction, using the type `ExtThry c t` to denote an extension of theory `t` with the checkpoint `c` and the type `BaseThry` to denote the unextended theory of our logical kernel. Thus, the existence of a checkpoint can be tested for by recursively searching this linear type, much as one would a list.

Using Boolean logic as the example theory, Figure 7 shows the pertinent types and instances for the construction of its theory type constraint, `BoolCtxt`. The crux of the presented solution lies in the closed type family definition of `BoolContext`. The two instances for this family collectively define the recursive search pattern for theory types, with the first instance serving as the terminating, base case.

Fig. 7. Constructing Polymorphic Theory Constraints

```
data BoolThry
type instance BoolThry == BoolThry = 'True
type BoolType = ExtThry BoolThry BaseThry

type family BoolContext a :: Bool where
    BoolContext BaseThry = 'False
    BoolContext (ExtThry a b) =
        (a == BoolThry) || (BoolContext b)

type family BoolCtxt a :: Constraint where
    BoolCtxt a = (BaseCtxt a, BoolContext a ~ 'True)

type instance PolyTheory BoolType b = BoolCtxt b
```

This search pattern relies on a promotion of boolean values to the type level in order to define type-level functions for both equality, `(==)`, and disjunction, `(||)`. Applications of these functions are reduced by a constraint solver which uses type family instances as rewrite rules. Each checkpoint tag has its reflexive equality asserted by a type instance, such that when a constraint is applied to a satisfying theory type it will reduce to the promoted value `'True`. Constraints that are not satisfied will statically fail with one of the following error messages:

- `Couldn't match type ''False' with ''True'` - The base theory context was provided and it failed to satisfy the constraint.
- `Couldn't match type 'A == B' with ''True'` - A theory context was provided with the last seen checkpoint `A`, however, the constraint requires the checkpoint `B` or later, e.g. `Couldn't match type 'BoolThry == SimpThry' with ''True'`.

In order to constrain a theory tag to require it to contain the Boolean logic checkpoint, an equality constraint of the form `BoolContext thry ~ 'True` is introduced to the type context. The `BoolCtxt` type both provides a shorthand for this constraint and asserts the presence of the theory that the Boolean logic library extends, `BaseCtxt`. For cases where multiple checkpoints are required, e.g. `(BoolCtxt thry, TheoremsCtxt thry, SimpCtxt thry)`, enforcing their linear ordering inside type constraints allows us to write only the most inclusive constraint, i.e. `(SimpCtxt thry)`, as it necessarily implies the others.

The `PolyTheory` type is used to relax a monomorphic theory tag to its corresponding polymorphic constraint. This allows us to redefine `safeProof`, as shown in Figure 8.

Fig. 8. A Refinement of `safeProof`

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

pthmTRUTH :: BoolCtxt thry => PData thry HOLThm
pthmTRUTH = safeProof (...) ctxtBool

-- These will succeed
testGood1 = evalHOL (serve pthmTRUTH) ctxtBool
testGood2 = evalHOL (serve pthmTRUTH) ctxtSimp

-- This will fail
testBad = evalHOL (serve pthmTRUTH) ctxtBase
```

Protected values, such as `pthmTRUTH`, are now assigned the correct, polymorphic type when they are constructed. This theorem can now be safely served in the context it was constructed(`ctxtBool`) and any of its extensions (`ctxtSimp`), but not in a context lacking the Boolean logic checkpoint (`ctxtBase`).

VI. Related Work

When we first began our work, we were unaware of any other groups pursuing monadic implementations of LCF-style proof systems. While attending the 2013 conference on Interactive Theorem Proving this changed, as we had the pleasure of being introduced to Myreen et. al's early work towards a verifiable implementation of HOL Light [14]. This work has since matured [15] to a fully mechanized proof of soundness for a stateless version of HOL Light's logical kernel. Critical to this mechanization process is a monadic implementation of the kernel that can be translated to CakeML, a subset of SML with a verified compiler [16].

Though not explicitly shown in the cited papers, the monad used by this work is a state-exception monad similar to the one contained in Figure 1[4]. Sections III through V detailed a number of reasons why using such a monad in the implementation of a prover intended for real world use is impractical. Myreen et. al do not need to concern themselves with this impracticality, though, as the monadic implementation exists only as an intermediate step on the way to a stateful CakeML implementation; a language that is apparently quite performant according to their benchmarks. We are curious if this work could be modified to use our monadic implementation instead.

[4]Myreen et. al's implementation of exceptions would be more accurately modeled in Haskell with an `Either` construction rather than with `IO`'s extensible exceptions.

However, we are not sure if HOL4 and CakeML have the capability to respectively reason about and implement the I/O effects we rely on.

The only other attempt at implementing a HOL system with Haskell that we could find also used a state-exception monad [17]. In this case, state was simulated with a combination of mutable `IORef`s stored in the `Reader` portion of a monad transformer stack. The technique for type-directed extensible state from Section IV could be adapted to more closely match this approach, storing a collection of `IORef`s for a context rather than its values directly, though it is not immediately clear if there would be any perceptible benefit. This work was apparently abandoned shortly after conception, though there is no documentation or publication indicating why. The last commit to the work indicates that the author was attempting an implementation of the tactic system which is, interestingly enough, the same point in our work that we first noticed the potential issues brought upon by a naive monadic approach. To state that these issues were the reason the author quit his implementation would be pure speculation, though.

VII. Conclusions and Future Work

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 [18]. Stable versions of the core of this system are currently available on the public package repository for Haskell, Hackage[5]. Users who have GHC Haskell [19] installed along with a copy of its Cabal package management system can easily install HaskHOL's logical kernel with the following command:

```
cabal install haskhol-core
```

More recent, experiemental versions of the core system and a number of additional libraries and related tools are available from the first author's Github at https://github.com/ecaustin.

Currently, we are exploring the application of HaskHOL for automatically verifying properties of Haskell programs. HaskHOL's implementation as a Haskell-hosted DSL allows us to easily, and natively, integrate it with the GHC system as a compiler plugin. By operating as a plugin, we can use GHC's simplification passes to desugar source to a core, intermediate language that maps almost directly to HaskHOL's term language.

[5]http://hackage.haskell.org

Examples are available as part of the *haskhol-haskell* repository from the Github account linked above. More information regarding this work, including a draft paper that details its implementation, is available at http://haskhol.org.

REFERENCES

[1] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, "Edinburgh LCF," 1979.

[2] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[3] The Coq development team, *The Coq Proof Assistant Reference Manual*, LogiCal Project, 2004, version 8.0. [Online]. Available: http://coq.inria.fr

[4] M. Gordon, "From LCF to HOL: A Short History." in *Proof, Language, and Interaction*, 2000, pp. 169–186.

[5] S. Marlow, "Haskell 2010 Language Report," 2010.

[6] P. Hudak, "Building Domain-Specific Embedded Languages," *ACM Comput. Surv.*, vol. 28, no. 4es, Dec. 1996. [Online]. Available: http://doi.acm.org/10.1145/242224.242477

[7] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[8] K. Slind and M. Norrish, "A Brief Overview of HOL4," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, O. Mohamed, C. Muñoz, and S. Tahar, Eds. Springer Berlin Heidelberg, 2008, vol. 5170, pp. 28–32. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71067-7_6

[9] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, *The OCaml system (release 4.02): Documentation and user's manual*, Institut National de Recherche en Informatique et en Automatique, September 2014. [Online]. Available: http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf

[10] J. Harrison, "Hol Light: A Tutorial Introduction," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, M. Srivas and A. Camilleri, Eds. Springer Berlin Heidelberg, 1996, vol. 1166, pp. 265–269. [Online]. Available: http://dx.doi.org/10.1007/BFb0031814

[11] F. Wiedijk, "Stateless HOL," in *TYPES*, 2009, pp. 47–61.

[12] S. Liang, P. Hudak, and M. Jones, "Monad Transformers and Modular Interpreters," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 333–343. [Online]. Available: http://doi.acm.org/10.1145/199448.199528

[13] J. Shaw, "The Happstack Book: Modern, Type-Safe Web Development in Haskell," 2013.

[14] M. Myreen, S. Owens, and R. Kumar, "Steps towards Verified Implementations of HOL Light," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds. Springer Berlin Heidelberg, 2013, vol. 7998, pp. 490–495. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39634-2_38

[15] R. Kumar, R. Arthan, M. Myreen, and S. Owens, "HOL with Definitions: Semantics, Soundness, and a Verified Implementation," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, G. Klein and R. Gamboa, Eds. Springer International Publishing, 2014, vol. 8558, pp. 308–324. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08970-6_20

[16] M. O. Myreen and S. Owens, "Proof-producing translation of higher-order logic into pure and stateful ML," *J. Funct. Program.*, vol. 24, no. 2-3, pp. 284–315, 2014. [Online]. Available: http://dx.doi.org/10.1017/S0956796813000282

[17] T. Elliott, "hol-light-haskell," Website, https://github.com/elliottt/hol-light-haskell.

[18] E. Austin and P. Alexander, "Stateless Higher-Order Logic with Quantified Types," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds. Springer Berlin Heidelberg, 2013, vol. 7998, pp. 469–476. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39634-2_35

[19] , "The Glasgow Haskell Compiler," Website, http://haskell.org/ghc/.