Chasing Sound, Efficient Proof with a Monadic, HOL System

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. This paper proposes an alternative approach to implementing a Higher-Order Logic (HOL) theorem prover that follows a pure, monadic style. Key to this implementation is an extensive use of compile-time meta-programming techniques in an attempt to come closer to the run-time performance of existing, side-effectful systems. Starting from a naive implementation, a step-wise refinement is provided where each change is motivated by improving performance without sacrificing soundness or safety. The ultimate goal of this work is not to argue that using purity is the necessary and best choice. Rather, the objective is to explore if purity can reveal a path towards a higher level of trust in proof systems that does not detract from their real-world usability. In order to test this hypothesis, the techniques presented in this paper are used to implement a HOL system with the Glasgow Haskell Compiler.

1 Introduction

Higher-Order Logic (HOL) theorem provers and the ML programming language have enjoyed a rich history together dating back to the logic's earliest days [12]. Even as new functional languages rose to popularity, HOL systems continued to rely on varying flavors of ML. The fact that the HOL community never suffered an ideological split from ML is of no great surprise. For one, there is a tradition among HOL provers where new versions draw from previous code bases quite heavily, a practice that obviously runs orthogonal to a stylistic or linguistic rewrite. Also, the ML language was tailor-made to implement HOL's predecessor, the Logic for Computable Functions.

This paper presents work lying along a road less traveled, an implementation of a HOL system written in GHC Haskell [1, 18]. The objective behind the switch to Haskell is to replace existing HOL systems' pervasive reliance of side-effects with a pure, monadic computation model. The authors believe that this monadic model presents an opportunity to potentially increase trust in the proof system, both by structuring effects with more formality and by providing a hook into GHC's advance type system extensions.

The consequence of this change is that impact of run-time effects can no longer be passively ignored. If a function has a potential effect, it will be stated explicitly at both the type and term level by the presence of a monadic computation. This necessitates two major differences when compared with more traditional implementations. First, theory contexts must shift from the memory level to the term level such that they can be passed around as values between computations, either implicitly or explicitly. Second, computations must be reevaluated every time they are bound in a new scope in the event that a change in context would lead to a change in effect.

In an attempt to minimize both user burden and performance implications that arise from these changes, the system takes extensive advantage of compile-time meta-programming techniques. These techniques are introduced in the paper as iterative refinements to an initial, naive, monadic system. The resultant theorem prover, HaskHOL, lies somewhere between the lightweight, interpreted approach of HOL Light [13] and compiled systems with larger footprints, such as HOL4 [24] or Isabelle/HOL [19].

2 Comparing Lightweight Approaches

The lightweight implementations mentioned in the introduction are labeled as such for their desire to model as much of the prover theory and syntax in their host programming languages as possible. By projecting to the host level, the proof system is able to to take advantage of existing libraries, reducing burden of implementation and increasing trust in the resultant code. This approach is not unique to theorem provers and is, in fact, just an example of shallowly embedding a domain specific language (DSL) in a host language.

When developing HaskHOL, the lightweight approach was selected given the great success that has been achieved using monads to implement DSLs [14, 15, 11]. In short, the purpose of the monad is to provide a model of computation for the language's primitive combinators. For a HOL system, these primitive combinators map directly to the methods provided by the prover's logical kernel. The details of HaskHOL's `HOL` monad are not particularly important in the context of this paper as long as we operate under the assumption that it can be used to faithfully and accurately implement a logical kernel without introducing unsoundness.

To provide a convenient point to begin comparison, Figure 1 shows two separate implementations of HOL's `TRUTH` derived rule. The first is taken from HOL Light, a lightweight HOL system written in OCaml. The second is also taken from a lightweight HOL system, this time written in Haskell with a monadic style, best representative of a very primitive version of HaskHOL. As alluded to in the introduction, the principle differences between these implementations focus on how the context of theories are handled and how side-effectful expressions are bound. These differences are present in nearly every piece of code that implements a proof, but can perhaps be most simply demonstrated by examining how the respective systems handle parsing the same HOL term.

Fig. 1. Comparative Implementations of `TRUTH`

HOL Light

```
let TRUTH = EQ_MP (SYM T_DEF) (REFL `\p:bool. p`);;
```

Primitive HaskHOL

```
thmTRUTH :: HOL Theorem
thmTRUTH =
    do lth <- ruleSYM =<< defT
       ruleEQ_MP lth =<< ruleREFL =<< parseHOLTerm "\\p:bool. p"
```

In the case of HOL Light, the Camlp5 [7] extension is used to provide a convenient back-tick notation for a tokenizing stream parser for HOL terms. The trick behind this parser is that type inference and term validity checking is enabled by examining token membership in the relevant parts of the theory context, such as lists of acknowledged binders and operators. These pieces of the context are stored in memory references that are manipulated by globally accessible methods provided to the user, making the parser one of an extensible nature. For example, to correctly parse the term from above, `\p:bool. p`, it is required that `\` be an accepted binder and `bool` be a defined type constant. Given OCaml's imperative, interpreter evaluation style [17], this necessitates that the appropriate method calls are made by the user before the parsing is to happen, however, once the term is parsed it can be reused without concern for what the current context is.

Conversely, the primitive HaskHOL implementation must check the current context nearly every time the parsed term is to be used. This is a consequence of the monadic model where computations are reran every time they are bound in a new scope. Reparsing terms so frequently presents a significant issue when working with an extensible parser, like the one in HOL systems. First, because no guarantees can be made about where in the chain a call to the parser will be made, any context modifying computations must be lifted to the start of the chain to guarantee that they will be ran before any parsing occurs. Second, HaskHOL's parser, like the rest of the system, is written in a monadic style that can be extremely slow based on how much backtracking it needs to do. There are tricks that can be used to increase its speed, like building a specialized expression parser for the provided context before parsing begins, but given that the context could change every time we parse this still leads to parsing dominating proof code in performance metrics.

Early versions of HaskHOL attempted to minimize the impact of these two issues by grouping the context modifying computations of a theory together into a load function that could be bound before a proof. For example, the code below would start with the initial theory context and load the boolean logic theory before running `thmTRUTH`:

```
runHOLCtxt ctxtKernel $ loadBoolThry >> thmTRUTH
```

Obviously this solution is far from perfect given that if the user neglects to bind the correct load functions the result will be, at best, a confusing parser error or, at worst, a loss of soundness.

3 From Run-Time to Compile-Time

Ideally, HaskHOL would be able to mimic HOL Light's "parse once, use many" behavior while also bundling the necessary parser extensions with the appropriate parsing computations. Template Haskell, a compile-time meta-programming library for Haskell [22], provides a facility, quasiquotation, that does just this. The general idea behind quasiquotation is to provide a mechanism that allows the programmer to write a portion of their code in the syntax of their DSL. This syntax is then parsed and manipulated as directed by the programmer before automatically being projected back into Haskell's core abstract syntax. The result can then be injected back into the original source code at compile time, such that the parsed term can be used as a pure value at run time. In a sense, quasiquotation provides the same functionality as Camlp5's preprocessor, with the added benefit that all work is done at compile time.

Quasiquoters, like the rest of Template Haskell, allow for splicing into a variety of locations. For parsing, we only care about splicing into expressions. Therefore, all we need to construct our parsing quasiquoter is to provide a function of the type `String -> Q Exp` where `Q` is Template Haskell's monad and `Exp` is the data type for Haskell's abstract syntax for expressions. If we rely on Haskell's generic programming libraries and GHC's `DeriveDataTypeable` extension we can simplify that significantly by letting Haskell automatically derive the translation between the`HOLTerm` and `Exp` data types.

Note that this resultant function type, `String -> Q HOLTerm`, nearly matches the type of our monadic parser, with the exception that the result is returned in the `Q` monad instead of the `HOL` monad. We can provide a path from `HOL` to `Q` using a combination of the previously shown `runHOLCtxt` and Template Haskell's `runIO`. Forcing evaluation with `runIO` can be thought of as the compile-time analogue of using `unsafePerformIO`, a Haskell function known not to be type safe. The key difference between the two is that with `runIO` still ties execution back to the `Q` monad, which can't be escaped by the user, guaranteeing the ordering of the effects in a single `Q` computation. This means that any use of `runIO` will be safe as long as the splices that rely on it are explicitly dependent and staged appropriately, a requirement enforced by Template Haskell, or the ordering of their execution does not matter. Parsing, in our case, falls within the second classification as long as we can construct the appropriate context outside of the parsing function.

Piecing this all together, we are left with the the following definitions for HaskHOL's base quasiquoter:

```
baseParse :: String -> HOLContext -> Q HOLTerm
baseParse s ctxt = runIO . liftM fst $ runHOLCtxt ctxt work
    where work :: HOL HOLTerm
          work = case holParser s ctxt of
                   Left err -> throw $ showErrors err
                   Right ptm -> elab ptm

baseQuoter :: HOLContext -> QuasiQuoter
baseQuoter ctxt = QuasiQuoter quoteBaseExp nothing nothing nothing
    where quoteBaseExp x = dataToExpQ (const Nothing) =<< baseParse x
          ctxt
          nothing _ = fail "quoting here not supported"
```

To define a quasiquoter for a specific theory we simply provide the corresponding context; for example: `bool = baseQuoter ctxtBool`. Using this technique, the example for `thmTRUTH` from Section 2 can be refined to use the new `bool` quasiquoter:

```
thmTRUTH :: HOL Theorem
thmTRUTH =
    do lth <- ruleSYM =<< defT
       ruleEQ_MP lth =<< ruleREFL [bool| \p:bool. p |]
```

The run-time performance of this computation is markedly improved compared to its previous iteration, but not without introducing a new concern. Specifically, the term is parsed with a context that is not guaranteed to be the same, or even compatible, with the context used to run the rest of the computation. We punt on this issue for now to revisit it in detail in Section 5.

4 From Terms to Theorems

The key concept from the previous section was the use of a composition of `runIO` and `runHOLCtxt` to force a monadic computation at compile-time. We can easily extend this practice to cover not only parsing, but any `HOL` computation that has a result that can be lifted into the Haskell core syntax:

```
runCompileTime :: Lift a => HOLContext -> HOL a -> Q Exp
runCompileTime ctxt m =
    do (res, _) <- runIO . runHOLCtxt ctxt m
       lift res
```

This splicing function could be used to prove `thmTRUTH` at compile-time, allowing us to treat it as a pure value, much like HOL Light does:

```
thmTRUTH :: Theorem
thmTRUTH =
    $(runCompileTime ctxtBool $
      do lth <- ruleSYM =<< defT
         ruleEQ_MP lth =<< ruleREFL [bool| \ p:bool . p |])
```

The problem becomes that we could just as easily prove an equivalent theorem by introducing a new axiom:

```
thmTRUTH :: Theorem
thmTRUTH =
    $(runCompileTime ctxtBool $ newAxiom [bool| T |])
```

Because the splice only returns the resultant value of the computation and not its resultant context, this reliance on an axiom is never reflected to the user.

The more general statement framing this problem is that computations forced at compile-time must have a notion of "constantness" to them. That is to say, they should have a deterministic, read only relationship with the provided context. In order to guarantee this, we need to differentiate between the primitive combinators that are used to extend theory contexts and those that are benign and used exclusively for proof. The easiest way to do this is to adjust the definition of the `HOL` monad to include a phantom type [6] that can be used to tag a computation's behavior at the type level.

The key notion behind phantom types is that they represent a way to superficially differentiate two expressions at the type level without affecting their term level value. The perennial example is using phantom types to guarantee well-formedness in a basic expression interpreter:

```
data Expr a = Expr String

mkConst :: Show a => a -> Expr a
plus :: Expr Int -> Expr Int -> Expr Int
```

Note that in the definition of the `Expr` data type, the type variable `a` is not used as an argument to any of the constructors; this is what makes it a phantom type. When expressions are built using the smart constructors shown this type is filled in automatically, such that the term `mkConst 1 ‘plus‘ mkConst 2` would type-check, but `mkConst 1 ‘plus‘ mkConst True` would not.

In HaskHOL, we concretize the phantom types of any context modifying computations in the kernel, similar to the smart constructors above. For example, the type of `newAxiom` becomes `HOLTerm -> HOL Theory Theorem`. Additionally, we change the type of `runCompileTime` to indicate that it can only accept proof computations:

```
runCompileTime :: Lift a => HOLContext -> HOL Proof a -> Q Exp
```

Every other occurrence of the `HOL` monad's phantom type should be left polymorphic so that type inference does not prevent us from mixing these computations in other areas of our code.

This solution prohibits the above example where context modifying computations are intentionally used to circumvent sound proof techniques. Interestingly enough, it also prohibits a class of non-malicious, but technically incorrect, proofs as well. Take, for example, an attempt to prove at compile time the definition of a constant which is known to exist in the provided context:

```
defT' :: HOL Theory Theorem
defT' = newBasicDefinition ...

loadBoolLib :: HOL Theory ()
loadBoolLib = loadLib $ ... >> defT' >> ...

ctxtBool :: HOLContext
ctxtBool = $(... loadBoolLib ...)

defT :: Theorem
defT = $(runCompileTime ctxtBool defT')
```

In this case, even though the redefinition is benign, a static typing error will be thrown for `defT` because of the conflicting phantom types. The way around this is to define extraction functions for desired portions of the context. In the case of definitions, we provide a function that searches the context for definitions with a left-hand side that matches a provided term and splices the matching theorem in:

```
defT :: Theorem
defT = $(extractDefinition ctxtBool [bool| T |])
```

The user is still protected statically with this approach, as an error is thrown if the supplied context does not contain an appropriate theorem. However, once again the crux of the problem is showing that the context that `defT` is extracted from is compatible with the context used at run time.

5 Tying Contexts to Computations

Expanding upon the core idea from the previous section, we add yet another phantom type, this time to the definition of the `HOLContext` data type. The purpose of this variable is to provide a type-level reification of the context that is being used to run a computation, so that this knowledge may be inferred in other places. After pushing this new variable through to the `HOL` monad and its run function, we have the following definitions, taken directly from the current version of HaskHOL:

```
newtype HOL t a b = HOL (StateT (HOLContext t) IO b) deriving Monad

runHOLCtxt :: HOLContext t -> HOL t a b -> IO (b, HOLContext t)
runHOLCtxt ctxt (HOL a) = runStateT a ctxt
```

Following similarly, we can protect theorems proved or extracted at compile time by tagging them with the context they were generated with:

```
newtype PTheorem thry = PThm Theorem

protect :: HOLContext thry -> Theorem -> PTheorem thry
protect _ thm = PThm thm
```

While this protection is little more than a wrapper to the existing theorem value, it serves two significant purposes. First, it provides a mechanism through which construction and destruction of protected theorems can be restricted via the module system. Second, it differentiates proven theorems at the type level such that they can not be used at run-time without first being lifted back into the `HOL` monad via the corresponding `serve` method:

```
serve :: PTheorem thry -> HOL thry a Theorem
serve (PThm thm) = return thm
```

Assuming that there exists a tag for the boolean theory, `BoolThry`, and that the compile-time proof and extraction functions are written to utilize the above protection mechanism, we can modify the example from the previous section as shown below:

```
defT :: PTheorem BoolThry
defT = $(extractDefinition ctxtBool [bool| T |])

thmTRUTH :: PTheorem BoolThry
thmTRUTH =
    $(proveCompileTime ctxtBool $
      do lth <- ruleSYM =<< serve defT
         ruleEQ_MP lth =<< ruleREFL [bool| \ p:bool . p |])
```

The compile-time extraction of `defT` from the `ctxtBool` context necessitates that it is tagged with the `BoolThry` type. Given that, the type of the internal monadic computation for `thmTRUTH` can be inferred to be `HOL BoolThry Proof Theorem` following from its use of `defT`. The function `proveCompileTime` checks this type against what is required for compile-time proof with the boolean theory, resulting in `thmTRUTH` also being tagged with the `BoolThry` type, the expected and correct value.

Notice that unlike the previous section, all of the phantom variables are being assigned concrete types instead of being left polymorphic. This solution works when using protected theorems generated from the same context, but when mixing different contexts things quickly fall apart. Take for example, the following:

```
th1 :: PTheorem BoolThry
th2 :: PTheorem ClassThry

exProof :: HOL thry Proof Theorem
exProof =
    do th1' <- serve th1
       th2' <- serve th2
       ruleTRANS th1' th2'
```

In this case, Haskell can not infer a type for `exProof` because its monadic computation has two binds, `th1'` and `th2'`, of different types. This makes perfect sense from a typing perspective, but is confusing from a logical perspective where we know that classical logic subsumes boolean logic.

6 Expressing More About HaskHOL Contexts

In order to provide protected theorems with a more expressive type to solve the problem from the last section, we need to take a step back and examine how contexts in HaskHOL are formed. Recall from Section 2 that the side-effects that construct a HOL theory are collected and simulated with a single monadic computation. In the case of advanced theories, frequently there is either an implicit or explicit dependency on a previously loaded theory. For example, the definition of falsity in the tactics theory is written in terms of propositional false from the boolean theory. The construction of the tactics theory context, therefore, can be expressed as the result of running a chain of computations containing these two load functions with the kernel theory context:

```
runHOLCtxt ctxtKernel $ loadBoolThry >> loadTacticsThry
```

Given the above construction, we know that the tactics theory contains the entirety of the boolean theory and any theorem protected with the boolean context should be serveable within a computation ran with the tactics context. Generalizing this statement, any context can be expressed as a chain of load computations ran with the kernel context. If protection for a theorem is necessitated by use of knowledge in a context generated by a load computation, `L`, then that theorem is serveable within a computation run with any other context that has `L` in its own construction chain. As a quick note, the above statement depends on the assumption that all context modifying computations are monotonic; for example, we can only add definitions, we can never remove them. In practice this is a bit of a white lie, usually due to the inclusion of methods of convenience, as is the case with "unparse" commands in HOL Light. For that reason, we acknowledge that the following modification of HaskHOL's protection mechanism only guarantees soundness to same level that Section 5's implementation does if the system is used in the intended way.

Where the type tags from the previous section faltered is they only reified the last load computation used to construct a context. In order to check whether a specific load function is used, as described above, we need to record the entirety of the computation chain. To implement this, we follow a technique similar to the one described in the *Data types á la carte* functional pearl [26]. The key difference is that because there is a strict linear ordering implied by theories' dependencies on each other, we don't need the full expressiveness of Swierstra's sum type. For example, the type of the two contexts mentioned above can be expressed as follows:

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

ctxtTactics :: HOLContext (ExtThry TacticsThry (ExtThry BoolThry
    BaseThry))
```

In order to keep the types of protected theorems polymorphic, each theory has a type class associated with it that reflects that it has been loaded. Instances of these type classes are defined through inductive inspection of a context type, a

process that relies on GHC's `OverlappingInstances` extension. Given the following definition of the boolean theory's type class, and assuming a similar definition exists for the classical theory, we can refine the types from last section's example:

```
class BoolCtxt a

instance BoolCtxt (ExtThry BoolThry b)
instance BoolCtxt b => BoolCtxt (ExtThry a b)

th1 :: BoolCtxt thry => PTheorem thry
th2 :: ClassCtxt thry => PTheorem thry

exProof :: (BoolCtxt thry, ClassCtxt thry) => HOL thry Proof Theorem
exProof =
    do th1' <- serve th1
       th2' <- serve th2
       ruleTRANS th1' th2'
```

As can be seen with the type of `exProof`, the constraints of a protected theorem are reflected in any other computation that uses it. This process continues, such that the type of all top level computations reflect exactly what contexts are required to run them. The result of all of this is monadic theorem values that can be used with the same efficiency as pure theorem values with the added benefits that all proof work is done at compile time and reuse is protected by static type checking.

7 Evaluation

In order to quantify the impact that compile-time proof has, we compare the performance of Primitive HaskHOL and HaskHOL over a shared problem set. We also compare the performance of HOL Light, HaskHOL's closest relative in the HOL prover family, to give an indication of how close this technique brings us to the run-time performance of a side-effectful implementation. The test problems used are selected from the Intuitionistic Logic Theorem Proving (ILTP) library [21]. Similar to the Thousands of Problems for Theorem Provers library [25], the ILTP library is designed to test automated theorem provers, specifically those restricted to handling only intuitionistically true problems.

In our tests, the restriction to intuitionistic tautology checking is important for two reasons. For one, the development of Primitive HaskHOL stopped at the intuitionistic theory, making it the most advanced theory available for testing with that system. Primarily, though, the intuitionistic theory and its associated `ITAUT` derived rule represent the perfect worst case scenario to test. While not typically used for proof directly, the `ITAUT` rule is critical in bootstrapping later first-order logic theories. Given this, an efficiency bottleneck in the intuitionistic theory would necessarily be reflected in later theories, potentially with an exponential explosion in slowness.

The problems selected from the ILTP library are those belonging to Roy Dyckhoff's benchmark library [8]. These problems are expressed as classes of formulae that are scalable in complexity and have variations that are both intuitionistically valid and invalid. By selecting the appropriate members of these classes, namely those intuitionistically true with complexity of `n=3`, we can guarantee both termination of the `ITAUT` rule and a significant enough run time to be profiled. The results of running these tests are shown in Table 1.

Table 1. Comparative Performance of Related Provers

Class	Prover	Time (sec)
de Bruijn	Primitive HaskHOL	13237.59
	HaskHOL	16.124
	HOL Light	10.932
Pigeon Hole	Primitive HaskHOL	146.014
	HaskHOL	0.117
	HOL Light	0.104
N-Contractions	Primitive HaskHOL	161.960
	HaskHOL	0.228
	HOL Light	0.187
Big Natural Deductions	Primitive HaskHOL	24.318
	HaskHOL	0.00235
	HOL Light	0.0374
Korn and Krietz	Primitive HaskHOL	61.062
	HaskHOL	0.141
	HOL Light	0.137
Equivalences	Primitive HaskHOL	0.578
	HaskHOL	0.00127
	HOL Light	0.0202

All results were gathered on the same machine[1] averaging five executions of each test. Primitive HaskHOL and HaskHOL were tested using the same framework built using the latest Platform Haskell [2] and Criterion benchmarking library [20] releases. HOL Light was tested less rigorously, using its built in `time` function.

In all cases, the improvement in run-time performance from Primitive HaskHOL to HaskHOL was significant. For the de Bruijn test case, the application of compile-time proof shaved more than three and a half hours off of the run-time, providing the most impressive reduction in terms of absolute time. In terms of relative time, performance increased anywhere from 433 times, in the Korn and Krietz test case, to a staggering 10348 times, in the big natural deductions test case. Furthermore, in all cases the performance of HaskHOL was brought within

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

the same order of magnitude as HOL Light, even surpassing it in two instances: equivalences and big natural deductions. That being said, given the short run times and potentially different overheads for the varying test frameworks, it's impossible to make the claim that HaskHOL is as fast, or faster than, HOL Light. However, at least for these classes of problems, it is not an unreasonable claim to state that HaskHOL's performance is now at a level where its speed alone is not enough to deter use in a real world application.

8 Related Work

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

Outside of the world of HOL, there has recently been a focus on improving trustworthiness of extensions to interactive theorem provers in general. In 2010 Matt Kaufmann, Konrad Slind, and Mike Gordon organized a workshop to discuss just this topic [3]. Topics at this workshop ranged from foundations of trust in existing proof systems to how to integrate external tools with provers in a trusted way. Of the work presented, Slind's and Natarajan Shankar's talks are perhaps the most relevant because they focus on the notions of kernels of truth and extending those kernels following the LCF style. Ideally, the work in this paper can be reduced to the minimum size and be presented as part of a monadic HOL kernel, following from the concepts of these talks.

Using well-founded techniques to implement a proof system is obviously a great idea, however, the trustworthiness of the system will always be dependent on the implementation being faithful to the authors' intentions. In short, you must be able to reason about the correctness of your code. For many people, the appeal of functional programming is that it is relatively easy to reason about equationally, such that a path from problem specification to solution can be shown to be correct [16]. Unfortunately, a lot of that ease disappears when discussing functional code that relies on side effects. Recent work has been completed to correct that issue for pure functional programs that frame effectful computations with monads [10]. While not targeted at increasing the trustworthiness at theorem provers directly, this work potentially opens the door to doing so for any provers written in a monadic style.

9 Conclusions and Future Work

First, at this point in time, the protection mechanism described in this paper works only for theorems; it can, and *should*, be extended to support terms as well. There are two possible ways to do so: by making the current methods and types polymorphic via a type family, or by creating new methods specifically for terms. Each approach has its own advantages. If a polymorphic implementation is selected, users only have to learn one set of methods, making the entire process familiar and approachable. If a separate implementation is selected, it opens up the protected types for terms to reify more information than just what context they are valid under. One possible example would be tracking the free variables in a term, such that an error could be thrown if they are redefined as constants later on.

The other realm to explore is to see just how much more of the prover can be lifted into the type system via the theory type classes. Currently these classes are empty, but it's not a stretch to consider using them to hold theory specific context information. Alternatively, they could act as superclasses for type classes that carry more general information. In either case, the goal would be to eliminate the need to use the `State` monad, at least for pre-constructed theories, or at the very least minimize the amount of information it carries. This would greatly reduce the memory footprint required to run HaskHOL proofs and would lead to increases in performance across the board, especially for any computations that require frequent backtracking.

The goal of this work was to provide evidence that a pure, monadic implementation of a HOL system was not only viable, but could potentially match the performance of a side-effectful implementation. With the results presented in Section 7 we feel we have accomplished this. Furthermore, we feel that we've made a strong case that GHC's type extensions are a good mechanism for increasing trustworthiness by lifting aspects of a systems soundness to the type level where they can be checked statically. While the work presented in this paper is a rough first step at best, we hope it will inspire others to travel down the same investigative path with us.

References

1. The Glasgow Haskell Compiler. Website:. `http://haskell.org/ghc/`.
2. The haskell platform. Website:. http://hackage.haskell.org/platform/.
3. Workshop on trusted extensions of interactive theorem provers. Website:. http://www.cs.utexas.edu/ kaufmann/itp-trusted-extensions-aug-2010/.
4. Mark Adams. Introducing hol zero - (extended abstract). In Fukuda et al. [9], pages 142–143.
5. Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors. *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. ACM, 2011.
6. James Cheney and Ralf Hinze. Phantom types, 2003.
7. Daniel de Rauglaudre. Camlp5. Website:. http://cristal.inria.fr/ ddr/camlp5/.

8. Roy Dyckhoff. Some benchmark formulae for intuitionistic propositional logic. Website:. http://www.cs.st-andrews.ac.uk/ rd/logic/marks.html.
9. Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors. *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, volume 6327 of *Lecture Notes in Computer Science*. Springer, 2010.
10. Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In Chakravarty et al. [5], pages 2–14.
11. Andy Gill. A haskell hosted dsl for writing transformation systems. In *IFIP Working Conference on Domain Specific Languages*, 07/2009 2009.
12. Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.
13. 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.
14. Paul Hudak. Building domain-specific embedded languages. *ACM COMPUTING SURVEYS*, 28, 1996.
15. Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
16. Graham Hutton. The countdown problem. *J. Funct. Program.*, 12(6):609–616, 2002.
17. Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 3.12. Website:. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.
18. Simon Marlow. Haskell 2010 language report.
19. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
20. Bryan O'Sullivan. Criterion, a new benchmarking library for haskell. Website:. http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/.
21. Thomas Raths, Jens Otten, and Christoph Kreitz. The iltp problem library for intuitionistic logic, release v1.1. *Journal of Automated Reasoning*.
22. Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *In Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM, 2002.
23. Konrad Slind. Trusted extensions of interactive theorem provers: Workshop summary. Website:. http://www.cs.utexas.edu/ kaufmann/itp-trusted-extensions-aug-2010/summary/summary.pdf.
24. 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.
25. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
26. Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.