An ICFP Pearl Submission

Protect and Serve

Policing Your Monadic Computations

Evan Austin Perry Alexander

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

Abstract

Pure, functional programs that are structured with monads can frequently run slower than comparable impure implementations due to repeated evaluations of intermediate monadic computations. When these computations are the dominating factor in the time complexity of a program this slowdown quickly changes from being an annoyance to a serious problem. In this paper we identify a class of monadic computations that exposes a methodology for safely forcing evaluation in a way that can be used to mimic impure languages' inlining of effectful expressions. We demonstrate the application of this technique in the HaskHOL theorem prover where it is used to police the soundness of compile-time proof.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.2.4 [*Software/Program Verification*]: Formal methods

General Terms Languages, Verification

Keywords Haskell, monads, Higher-Order Logic, HaskHOL

1. Introduction

If Jerry Seinfeld ever did a bit about functional programming, he'd probably open with the line, "What's the deal with monads?" While it's questionable if even a master comedian could pay off that setup, we can take solace in the fact that there's at least a number of serious answers to the question; among the most popular – monads provide a mechanism for simulating side-effects in pure languages. Take the following snippets of code as an example:

```
(* OCaml example using global references *)
let state = ref 0;;
let x = !state > 0;;
let comp =
  let x1 = x
  and x2 = state := !state + 1; x in
    (x1, x2);;
```



```
{- Haskell example using State monad -}
x :: State Integer Bool
x = do val <- get
       return (val > 0)

comp :: (Bool, Bool)
comp = flip evalState 0 $
  do x1 <- x
     x2 <- modify (+ 1) >> x
     return (x1, x2)
```

To the left, we have a piece of OCaml code that performs a simple comparison against the value of a global reference, both before and after it is incremented. Above, we have a Haskell version that utilizes the `State` monad to simulate the same behavior. Or does it? If we run both programs, OCaml returns `(false, false)` and Haskell returns `(False, True)`. So what's going on?

In short, OCaml is evaluating and inlining `x` as it would any other expression, regardless of the fact that it is side-effectful. The result is that the value of `x` in the binding for `x2` is calculated before the state is incremented, giving us the unexpected, but technically correct, behavior we observed. So why is this important, aside from reminding us to be careful about how we use languages with side-effects? The answer is that sometimes we want to mimic this impure behavior in a pure language.

To demonstrate this point, we slightly modify the previous Haskell snippet:

```
y :: State Integer Integer
y = do val <- get
       return (fib val)

comp2 :: (Integer, Integer)
comp2 = flip evalState 50 $
  do y1 <- y
     y2 <- y
     return (y1, y2)
```

Two things are important about this change: a) `y` is now the dominant factor in the time complexity of our program and b) the state of the computation does not change between the evaluations of `y`. Knowing this, it is our hope that the compiler will optimize the program by performing an inlining operation similar to what was seen in the OCaml example. It doesn't, though, and the reason it doesn't is a good one – in general, the desired program transformation is not sound.

That leaves us with two basic options. We could utilize GHC Haskell's rewrite engine to encode program or computation specific optimizations. Alternatively, we could manually refactor the code to evaluate complex, but constant, computations before they are used. Our claim, though, is that for large systems both of these techniques make it increasingly difficult to prove that the resultant

program is correct and functionally equivalent to the original. In this paper, rather than trying to provide a new, general optimization strategy, we focus on identifying a class of monadic computations for which the correctness of a refactoring approach can be much more easily guaranteed.

We introduce this class of computations in Section 2, explaining their most important property with an example involving a modified `State` monad. The next two sections demonstrate how we guarantee the safety of our transformation, with the titular `protect` and `serve` methods being introduced at the end of Section 4. We then discuss a non-trivial application of these techniques in Section 5 where they're used to support compile-time operations in a Higher-Order Logic (HOL) proof system. For those interested in the system specific details, Section 6 shows the implementation of `protect` and `serve` in the HaskHOL theorem prover. Finally, we close with a discussion section where we expound upon some open questions and concerns lingering in our minds.

2. The Monotonic State Monad

The chief requirement for any optimization strategy, including our attempt to mimic an inlining transformation for monadic code, should be that it is behavior preserving. Specifically, we want to be sure that we only lift evaluated computations back into the monad when they're in a context where the value of the resultant binding would be equivalent to a non-optimized version. By that definition, our first example is a poor target for optimization while our second example is a good one. It is worth pointing out, though, that the syntax and types of both examples are nearly identical. This leaves little information to indicate that a computation is suitable for optimization outside of the programmer's own intuition. This is the basis of our main argument against the manual refactoring approach; mixing complex systems and programmers' intuitions is an excellent recipe for bugs.

In the specific case of the `State` monad, a lot of what clouds our decision making is that it is impossible to quantify a general relationship between the states of a computation. This is due to there being only one restriction on the non-proper `put` morphism of the `MonadState` class: the new state value must be the same type as the old state value; aside from that, anything goes. Within the body of `comp` in our first example, the call to `modify (+ 1)` could have just as easily been a call to a computation of unknown or undeterminable behavior. Furthermore, `x` itself could have been a state modifying computation, a point that we'll touch on in the next section.

To solve this problem, we present an alternative implementation of the `State` monad which we have dubbed the monotonic state monad, or `MonoState`:

```
class (Enum s, Monad m) =>
      MonadMonoState s m | m -> s where
  monoGet :: m s
  monoSucc :: m ()

newtype MonoState s a =
  MonoState { runMonoState :: s -> (a,s) }

instance Monad (MonoState s) where
  return x = MonoState $ \s -> (x,s)
  m >>= k = MonoState $ \s ->
    let (x, s') = runMonoState m s in
      runMonoState (k x) s'

instance Enum s =>
         MonadMonoState s (MonoState s) where
  monoGet = MonoState $ \s -> (s, s)
  monoSucc = MonoState $ \s -> ((), succ s)
```

Note that the only difference between `MonoState` and `State` is that `put` has been replaced with `monoSucc`. We rely on the `Enum` class to implement this new method such that the only possible state modifying computations consist of one or more calls to `succ`. With this restriction, its trivial to show that the states of any `MonoState` computation are monotonically increasing during evaluation.

With this relationship in place, we can begin to guarantee the sound reuse of evaluated computations provided that they were constructed with the intention of leveraging our monad's monotonicity principle. For example, a computation that performs an equality test on the value of the state would not be fitting as it requires constancy of the state, a relationship that monotonicity is too weak to guarantee. However, a computation that performs an ordering test on the value of the state would be a good choice as there always exists a starting value for our state that will make such a computation constant.

Let's reimplement our first example with this concept in mind:

```
x' :: Bool
x' = flip evalMonoState 1 $
  do val <- monoGet
     return (val > 0)

comp' :: (Bool, Bool)
comp' = flip evalMonoState 0 $
  do x1 <- return x'
     x2 <- monoSucc >> return x'
     return (x1, x2)
```

Note that, as its return value would imply, the computation `x'` becomes constant for all state values greater than zero. Therefore, we pick a satisfying state to evaluate `x'` with and lift it back into the main computation with `return`. In this way we preserve the overall shape and type of `comp'` while gaining the optimization we're looking for.

The challenge now shifts to making sure that the state used to evaluate the overall computation also satisfies the condition we used to pick the optimizing state. Unfortunately, we can see that in this case it does not. The good news, though, is that this problem is relatively easy to solve in general. This is precisely the purpose of the `protect` and `serve` functions that have been alluded to up to this point. The policing that the subtitle refers to is the checking and rechecking of a computation's starting value against its optimization condition. More details regarding the implementation of these functions will be discussed in Section 4.

3. The Tagged State Monad

Before we discuss the details of `protect` and `serve`, we stop to investigate a brief aside from the previous section. Recall that we mentioned a potential issue when the computation to be optimized is itself state-modifying. If a computation has an effect that would influence the execution of the rest of the program we don't want to make it constant as we would lose any repetitions of said effect. However, some effects, like the retrieval of global state via `get`, are benign outside of the context in which they're used. The first step, therefore, is to identify which effects we should be concerned about and which we can safely ignore.

For the sake of discussion, we will refer to the set of effects that have the potential to leave a lasting impression as active and the rest as passive. Each monad's unique behavior, and thus its set of possible effects, is defined by an interface that is formed from the collection of its primitive, non-proper morphisms. In the case of the `State` monad, we have two such methods: `get`, containing a passive effect, and `put`, containing an active effect. If we classify these primitive computations we can then infer the correct classification for any more complex computations built from them.

When tagging effects with their classification, ideally we'd like to able to achieve two things. First, we want to be able to check the tag statically. Second, we don't want the tag to change the behavior of the underlying effect. The easiest way we've found to satisfy both of these goals is to introduce the tag at the type level by wrapping the target monad with a `newtype` containing a phantom type variable [2]. We demonstrate this below with yet another modification of the `State` monad, the tagged state monad:

```
data Passive
data Active

newtype TagState tag s a = Tag (State s a)
  deriving Monad

tagGet :: TagState tag s s
tagGet = Tag get

tagPut :: s -> TagState Active s ()
tagPut = Tag . put
```

In the above definition of `TagState`, `tag` is the phantom type variable used to carry the effect classification. This variable is inhabited by one of two empty data declarations, `Passive` or `Active` . All primitive morphisms containing active effects have aliases that tag them as such with the remaining morphism aliases left polymorphic. This allows us to combine elements from either set, such as in `State`'s derived method `modify`, with the type checker inferring the `Active` tag only when it's necessary. The `Passive` tag, therefore, is only used in places where we want to explicitly exclude the use of active effects.

If we use this technique to guide the evaluation of monadic computations we should take note that we can only infer passivity from the top down. In other words, top-level computations can know that the entirety of their body is passive, but intermediate computations cannot attest to the classification of the context they will used in if they're left polymorphic. A reimplemented version of our second example demonstrates this notion well:

```
evalPassive :: TagState Passive s a -> s -> a
evalPassive (Tag m) = evalState m

y' :: TagState tag Integer Integer
y' = do val <- tagGet
        return (fib val)

comp2' :: (Integer, Integer)
comp2' = flip evalPassive 2 $
  do y <- y'
     y <- y'
     return (x1, x2)

comp2b' :: TagState Active Integer
           (Integer, Integer)
comp2b' = do x1 <- y'
             tagPut x1
             x2 <- y'
             return (x1, x2)
```

In the above code we have left `y'` polymorphic such that it can be used in a passive, top-level computation, `comp2'`, and an active one, `comp2b'`. Either `comp2'` or `comp2b'` can make statements about their classification relative to the entire code base but, unfortunately, `y'` cannot.

To further this point, we have defined an alias to `evalState`, `evalPassive`, that allows the evaluation of passive computations only. While it would be possible to evaluate `y'` with `evalPassive`, it would result in incorrect behavior should we try and lift the result back into `comp2b'`. Regrettably, there's no way for the `TagState` monad to catch a case like this. It can be used to prevent the evaluation of `comp2b'`, though, with a compile-time error documenting the type mismatch between `Active` and `Passive`. For that reason, this technique is best applied in a defensive manner to guard against the accidental evaluation of an active computation.

4. The Sealed State Monad

Up to this point, we have identified a class of monadic computations that we'd like to optimize as well as a strategy and technique for doing so in a safe way. The remaining half of the problem left to solve is how to guarantee that this optimization is only applied when it results in a sound transformation. Recall from Section 2 that this is done by checking that the starting value used to evaluate a top-level computation satisfies the optimization conditions of all of its intermediate computations.

Our explanation starts with the simplest possible case, optimization conditions with only one satisfying value. The check, therefore, becomes a direct comparison between the values used to evaluate computations. Just as in the previous section, phantom type variables are used to tag these values with the information necessary to compare them. Because we need this information to persist after evaluation, though, simply tagging the monad type itself is not enough; we must also tag results of evaluations with the value used to produce them.

One final modification of the `State` monad, the sealed state monad, is shown below:

```
newtype Env lbl s = Env s
newtype Res lbl a = Res a

seal :: State (Env lbl s) a -> (Env lbl s) ->
    Res lbl a
seal m = Res . evalState m

unseal :: Res lbl a -> State (Env lbl s) a
unseal (Res a) = return a
```

Modification is perhaps a misnomer; the `seal` and `unseal` methods are probably better thought of as enhancements to the `State` monad. Their term level functionality is relatively basic, as they act primarily as a way to box and unbox results with the `newtype` wrapper `Res`. The real heavy lifting is done in the type level, where these methods carry the phantom type variable `lbl` is from computation to result and vice versa.

The trick to this approach is that we need a type level reification of the values we want to compare. In all of the above examples, we've been dealing with positive integers, so we rely on a standard type-level representation of naturals:

```
data Zero
data Succ n

zero :: Res Zero ()
zero = seal (return ()) $ Env 0

one :: Res (Succ Zero) ()
one = seal (return ()) $ Env 1

onesOnly :: State (Env (Succ Zero) ()) ()
onesOnly =
  do x <- unseal zero -- type error
     return x
```

We have defined two intermediate computations above that have both had their results sealed. As the names would imply, `zero` has been sealed with the starting state 0 and `one` has been sealed with the starting state 1. The actual comparison of starting values occurs when results are unsealed in a top-level computation. A type mismatch error will be thrown when unsealing `zero` in `onesOnly` due to its explicit annotation of requiring a starting state of 1.

More complicated optimization conditions can be checked provided that they can also be reified to the type level. Given that we want to accept any satisfying value, we use type classes with carefully defined instances to provide the necessary level of polymorphism:

```
class GTZ a
instance GTZ (Succ n)

one' :: GTZ n => Res n ()
one' = seal (return ()) $ Env 1

noZeros' :: State (Env (Succ Zero) ()) ()
noZeros' =
  do x <- unseal one' -- type ok
     return x
```

Above we have introduced the condition `(val > 0)` as the type class `GTZ`. We have also defined an instance of this class that satisfies any tag constructed with a `Succ` data type on the outermost level, matching the inductive representation of naturals with values of 1 or greater. If we use this class to annotate a polymorphic type for a result it gives us the ability to seal and unseal with different values provided that both of their tags have instances of the class.

The last step in reaching our desired pair of methods, `protect` and `serve`, is to separate the acts of boxing and evaluation in `seal`:

```
seal' :: State (Env lbl s) a -> (Env lbl s) ->
         Res lbl a
seal' m s = protect s $ evalState m s

protect :: Env lbl s -> a -> Res lbl a
protect _ = Res

serve :: Res lbl a -> State (Env lbl s) a
serve (Res a) = return a
```

This separation motivates our decision to tag the state values directly rather than having the tag in the monad type; it allows for the flexibility of protecting pure values generated from non-monadic sources. It also allows for inspecting and observing a monadic result at its original type between evaluation and protecting, something that `seal` prevented. Finally, we pair this separation with renaming of `unseal` to `serve` to create the corniest police pun since `CHiPs` went off the air.

5. Protecting HOL and Serving the Proof

The HOL proof system has been around since the late 1980s [3]. In that time it has inspired a number of direct descendants and close relatives including HOL Light, HOL4, HOL Omega, Isabelle/HOL, PVS, and many others. We are currently trying to introduce our own branch to that family tree with HaskHOL, a HOL theorem prover implemented as an embedded domain specific language (DSL) in Haskell [1]. Like most DSLs, HaskHOL uses a monad to structure its computations; in this case with the aptly named `HOL` monad. This is a break from the HOL tradition, though, where proof systems leverage the effectful features of their implementation language, usually a derivative of the ML language.

This has created an issue for HaskHOL analogous to the one framed in the introduction section. There are a number of features in a HOL system, such as proof search tactics, that may cause the same theorem to be called upon hundreds or thousands of times in a single proof. While the other provers are able to get by with a "prove once" methodology, HaskHOL and its monadic implementation style must pay the price of reevaluating a theorem every time it is used. For that reason we have targeted it as a potential application for the `protect` and `serve` technique.

The last three sections have each identified a major factor to be used when deciding if a monadic computation is a suitable candidate for our optimization:

- The values of the monad's parameter type must form a monotonic relationship during evaluation.
- The primitive effects of the monad must all be classifiable as active or passive.
- The values of the monad's parameter type and any optimizing conditions must all have an accurate type level reification.

To understand why HaskHOL's `HOL` monad satisfies all of the above properties, we must understand the LCF implementation style that is at the heart of all HOL systems.

The basic idea behind the LCF style is that a proof system starts with a small, trusted logical kernel from which more complex features can be bootstrapped. In order to do this, a prover maintains the notion of a current working theory, a collection of all knowledge introduced to the system at that point in time. The key to maintaining soundness with a bootstrapping approach is that knowledge can only ever be added to this theory, never removed. Given this, we can easily show that the stateful portion of the `HOL` monad used to model the current working theory is monotonically increasing.

Because the construction of the current working theory is so carefully controlled, it makes classifying the primitive morphisms of the `HOL` monad fairly easy. We tag any method that extends the current working theory as `Theory`, a more fitting version of `Active` for this application, and use the `Proof` tag to enforce passive, proof computations. It should be pointed out, though, that there are a number of methods that fall in a grey area between these classifications. The methods for turning debugging on and off, for example, do so by modifying the state of the `HOL` monad which technically makes them active effects. We choose to classify them as passive, though, since they only affect the intermediate output presented to the user and not the proofs themselves. This decision was made in an effort to avoid overcomplicating the system with safety, as we believe there's no point in designing a bulletproof car if it ends up too heavy to drive.

The LCF design also enables a straightforward method for assigning type level tags to values of the current working theory. Because it can grow quite large, the current working theory is frequently check pointed at key places. All of the expressions required to build the theory to that point are gathered into a single module, along with any relevant theorems, rules, or other items, which is labeled to indicate the logic or feature that it defines. For example, the "Bool" module introduces basic, propositional logic to the proof system. In our system, these labels are converted directly to data types so that they can be used in conjunction with `protect` and `serve`.

As we mentioned in the introduction, our desired application of `protect` and `serve` in HaskHOL is to support compile-time operations. The idea of performing computations at compile-time is not itself a novel idea, as many HOL systems already include either a pre-processor or quasi-quoter that allows users to write logical terms in a more human readable form. The difference is that these systems are fairly cavalier in the way they use these terms, rarely if ever checking that the theory used to parse them agrees with the current working theory.

The `protect` and `serve` technique originated as a way to solve this problem. By signing quasi-quoted terms with the theory used to parse them we had a way to guarantee their sound reuse. While implementing this functionality we realized that it was a very robust approach with a number of other potential applications within the system. We have since added methods to extract and reuse knowledge from theories and evaluate proof computations at compile

time, all of which are policed by `protect` and `serve`. The result is a system that has a lightweight feel, much like the system that originally inspired HaskHOL, HOL Light [4], while maintaining the efficiency of a more complex, pre-compiled system, like HOL4 [6].

6. Compile-Time Proof with HaskHOL

At the heart of HaskHOL is the `HOL` monad, a flattened stack of `State` and `IO`:

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

This combination allows us to simulate the stateful nature of a HOL system's current working theory while also providing basic exception handling and message printing capabilities. The `HOL` type is parameterized by three type variables: `cls`, a phantom type variable used to classify the passivity of a computation's effects, `thry`, a (almost) phantom type variable used to label a computation's associated theory, and `a`, the return type of a computation.

The details of the state type, `HOLContext`, are relatively uninteresting beyond noting that it is parameterized by the phantom type variable that holds the theory label. Hence our parenthetical use of the word almost in the last paragraph; we were just one layer off. These labels are constructed by representing the linear module structure found in many HOL systems as a type-level list of theory types:

```
data BaseThry = BaseThry deriving Typeable
data ExtThry a b = ExtThry a b
  deriving Typeable

-- Type for the Boolean Logic Theory Label
type BoolType = ExtThry BoolThry
                (ExtThry EqualThry BaseThry)
```

Each theory label also has an associated type class that is used when a computation needs to check that the theory has been loaded as part of its optimization condition. This membership check is performed via overlapping instances that traverse the list of theories, working similarly to the `elem` list operation. The theory's type class is also used to assert any pre-requisite theories, giving a strict ordering to how theories are loaded:

```
class EqualCtxt a => BoolCtxt a
instance EqualCtxt b =>
         BoolCtxt (ExtThry BoolThry b)
instance BoolCtxt b => BoolCtxt (ExtThry a b)
```

For compile-time operations, we utilize Template Haskell [5] to automatically annotate the correct optimization condition for a computation. We do this by defining a type class, `DerivedCtxt`, that explicitly links a theory label to the `Name` of its associated type class. Because of the strict, linear ordering we established above, we can safely select the type class of the head of our theory label list as it represents the most recently loaded theory:

```
class DerivedCtxt a where
  contextName :: a -> Name

instance DerivedCtxt BaseThry where
  contextName _ = ''BaseCtxt

instance DerivedCtxt a =>
         DerivedCtxt (ExtThry a b) where
  contextName _ = contextName (undefined :: a)

instance DerivedCtxt BoolThry where
  contextName _ = ''BoolCtxt
```

The last piece of the puzzle is to define HaskHOL's specialized versions of `protect` and `serve`. As we mentioned in the previous section, these methods are used to protect a variety of types. We want to differentiate between the wrapping constructors of each type, so we elect to implement the policing mechanism as a type family to achieve our desired level of polymorphism:

```
class Lift a => Protected a where
  data PData a thry
  protect :: HOLContext thry -> a ->
             PData a thry
  serve :: PData a thry -> HOL cls thry a
  liftTy :: a -> Name
  protLift :: PData a thry -> Q Exp

instance Protected Theorem where
  data PData Theorem thry = PThm Theorem
  protect _ = PThm
  serve (PThm thm) = return thm
  liftTy _ = ''Theorem
  protLift (PThm thm) =
    conE 'PThm `appE` lift thm

type PTheorem thry = PData Theorem thry
```

The two methods in the `Protected` type class that we have not discussed before, `liftTy` and `protLift`, are used to simplify the Template Haskell wizardry behind automatically deriving the correct type for an optimization condition.

The combination of the `DerivedCtxt` and `Protected` type classes are all that we need to provide the basis for of our compile-time operations. This foundation consists of just three major functions:

```
buildThryType :: forall a thry.
  (Protected a, DerivedCtxt thry) =>
    PData a thry -> Type

liftProtectedExp :: forall a thry.
  (Protected a, DerivedCtxt thry) =>
    PData a thry -> Q Exp

liftProtected :: forall a thry.
  (Protected a, DerivedCtxt thry) =>
    String -> PData a thry -> Q [Dec]
```

The `buildThryType` function automatically derives the optimization condition for a computation in the manner described in the previous paragraphs. This type is then paired with a protected result and lifted into the source code at compile time via a Template Haskell splice. This can be done either as an annotated expression, built with `liftProtectedExp`, or a top level declaration, built with `liftProtected`.

Shown below is the application of `protect` and `liftProtected` in our compile-time proof operation:

```
proveCompileTime :: DerivedCtxt thry =>
  HOLContext thry -> String ->
  HOL Proof thry Theorem -> Q [Dec]
proveCompileTime ctx lbl th =
  do thm <- runIO $
              do putStr $ "proving: " ++ lbl
                 thm <- flip evalHOLCtxt ctx $
                          (turnDebugOn >> th)
                 putStrLn "...proved."
                 return thm
     liftProtected lbl $ protect ctx thm
```

```
-- Example specialization of proveCompileTime
proveBool :: String ->
             HOL BoolType Proof Theorem ->
             Q [Dec]
proveBool = proveCompileTime ctxtBool
```

In HaskHOL, we tend to specialize `proveCompileTime` for any library that contains a check pointed theory. We find that this reduces syntactic burden and helps to clarify our intent in files that have a large number of theorems constructed at compile time. The function `proveBool` is one such specialization for the boolean logic library.

```
-- Example splice from the Boolean library
proveBool "thmTRUTH" $
  do lth <- ruleSYM =<< serve defT
     rth <- liftM primREFL $
              toHT [str| \p:bool. p |]
     tryEither $ primEQ_MP lth rth
```

The splice shown in the code above, `thmTRUTH`, produces a very simple theorem in HOL, `|- T`. Put simply, this theorem states that the truth term, `T`, is always true. Obviously such a fundamental theorem will be called on numerous times in the course of a proof, hence our desire to mimic a "prove once" evaluation style.

Note that within the body of `thmTRUTH` we see a use of `protect` 's counterpart, `serve`, to lift the value `defT` into the proof. This value contains the theorem that defines `T` and was protected after being extracted from the boolean theory, `ctxtBool`, elsewhere in the library. This is a common pattern in HaskHOL libraries as previous knowledge is frequently used in the construction of new knowledge and theorems. The result of this chain of `serve`s is a top-level computation that infers an optimization condition requiring multiple libraries to be loaded, giving you an exact representation of what logic is required in order complete a proof, something most other systems don't provide.

7. Discussion

From a design standpoint we are exceptionally happy with the way that `protect` and `serve` turned out. However, its practical application in HaskHOL leaves a few things to be desired. Our first complaint is that automatically deriving optimization conditions currently requires what we consider to be a ridiculous amount of boilerplate code. Ultimately, it should be a very straight forward process, but we're forced to simulate simple list operations at the type level with a mess of overlapping type class instances. These instances can be constructed for new theories automatically with Template Haskell, so it's of minimal inconvenience for a user of the system, but as an implementor of the system it's somewhat of pain to support and maintain.

We're excited about the new data kind promotion work, though, as it appears to be the perfect answer to our cries [7]. Our short term goal is to rewrite the generation of optimization conditions in HaskHOL such that they use actual type-level lists, rather than simulated versions. It's yet to be seen if this approach will generate new issues of its own, but our hope is at the very least it will reduce the overall number of auxiliary definitions currently required in the core of HaskHOL.

Our second concern is less of a complaint and more of an observation; splicing HaskHOL theorems with Template Haskell can take a really, really long time. This isn't necessarily indicative of a problem with either system, rather it's just an unfortunate interaction between the two. Much of the lightweight feel of HaskHOL that we mentioned earlier is thanks to an extremely primitive representation of logical terms in the kernel of the system. This means that theorems with large terms in their conclusions are actually the application of hundreds of thousands of constructors.

When you add in the constructors needed to build an instance of Template Haskell's `Exp` data type and the complex type annotations for the optimization conditions you end up with a nightmare for GHC's type checker. The result is that proofs that may evaluate in a few seconds at run time can take several minutes to evaluate, type check, and splice at compile time. We're waiting with bated breath for the introduction of the long-awaited update to Template Haskell[1] to see if the new approaches to typing splices alleviates this problem at all. In the mean time, we remain happy and optimistic about the system's performance as is.

References

[1] HaskHOL. Website:, 2013. `https://wiki.ittc.ku.edu/sldg_wiki/index.php/HaskHOL`.

[2] M. FLUET and R. PUCELLA. Phantom types and subtyping. *Journal of Functional Programming*, 16:751–791, 10 2006. ISSN 1469-7653. URL `http://journals.cambridge.org/article_S0956796806006046`.

[3] M. Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[4] J. 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.

[5] T. Sheard and S. L. P. Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[6] K. Slind and M. 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. ISBN 978-3-540-71065-3. URL `http://dx.doi.org/10.1007/978-3-540-71067-7_6`.

[7] B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation*, TLDI '12, pages 53–66, Philadelphia, PA, USA, 2012. ACM. ISBN 978-1-4503-1120-5.

[1] `http://hackage.haskell.org/trac/ghc/blog/Template%20Haskell%20Proposal`