Submitted for Consideration for ICFP'15

The Proof is in the Plugin

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

An oft-cited advantage of programming in a pure, functional language is that guarantees of correctness can be made quite directly via equational reasoning. Though this reasoning is in and of itself a fairly simple process, attempts to mechanize its application typically rely on integrating with complex, external proof systems. It is our belief that the burden of working with these systems precludes the average programmer from formally verifying properties that they could otherwise informally demonstrate to be true. The work in this paper discusses an alternative approach to program verification – the integration of a lightweight proof system into the GHC compiler pipeline via its plugin framework. A motivating example is followed to demonstrate how the capabilities of this prover can be used to easily verify properties of Haskell programs.

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

General Terms Languages, Verification

Keywords Haskell, HaskHOL, HERMIT

1. Introduction

Among the most commonly cited advantages of working with purely functional languages is that reasoning about program behavior is significantly easier in the presence of referential transparency. Expected behavior can be expressed by ascribing key pieces of code with properties that collectively form a valid argument of an implementation's correctness. Much like the evaluation strategy of the host language itself, these properties can be rewritten through repeated substitution of function and data type definitions until they are reduced into true or false statements. This is the essence of the equational reasoning approach to program verification.

In Haskell, correctness properties are often self-introduced, whether the programmer realizes it or not. As an example, the de facto approach to structuring effectful computation in Haskell relies on the `Monad` type class. The documentation for this class includes three properties, inherited from the definition of monads found in category theory, that all `Monad` instances should obey.


```
{- The Monad Laws:
Left identity:  return a >>= k  ==  k a

Right identity: m >>= return  ==  m

Associativity:
  m >>= (\x -> k x >>= h) == (m >>= k) >>= h
-}
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Figure 1. The `Monad` Type Class

These properties, along with a minimal class signature, are shown in Figure 1.

Though the trend is slowly changing, the majority of `Monad` instances are implemented without verifying or even stating these laws. Given that the laws double as rewrite rules in most compilers for the language, this is obviously non-ideal for anyone concerned with program correctness. It is our belief that the dearth of correctness proofs, especially for type class properties, is a matter of inconvenience rather than impossibility; they can be, and should be, verified. Continuing with our motivating example, a definition of the `Identity` monad and a proof of its left identity law is shown in Figure 2.

Verification of the monad laws for other instances proceed similarly. The catch is that the effort required for an equational reasoning proof is directly proportional to the complexity of both the proof obligation and its requisite definitions. The `Identity` monad is intentionally targeted for the proof derivation in Figure 2 as it has the simplest possible `Monad` instance. Conversely, the definition of the computational monad of the HERMIT system discussed later in this paper relies on nested record types, smart constructors, and other monads. Attempting a paper proof with its definition, while certainly possible, would be both time consuming and error prone. This is of no surprise to anyone familiar with the motivations behind mechanizing proof theory.

Ideally, the average programmer would be able to throw their code over the wall to a formal reasoning tool that would assist them with their proofs of correctness in the same transparent manner as a type checker. Unfortunately, despite all of the benefits that the functional paradigm provides, it does nothing to assuage a major problem in software verification: marshaling knowledge between implementation and verification environments is, more often than not, just as complex and error prone as attempting proofs by hand. Furthermore, once inside a verification environment, programmers typically find themselves at the bottom of a steep learning curve that only further dissuades them from formality; especially if their intended goal is just to complete a one-off proof.

```
data Identity a = Identity a

runIdentity :: Identity a -> a
runIdentity (Identity a) = a

instance Monad Identity where
    return = Identity
    m >>= k  = k (runIdentity m)
```

Left Identity Law

$$return\ a >>= k = k\ a \tag{1}$$

Rewrite with Definition of $(>>=)$

$$k\ (runIdentity\ (return\ a)) = k\ a \tag{2}$$

Rewrite with Definition of $return$

$$k\ (runIdentity\ (Identity\ a)) = k\ a \tag{3}$$

Rewrite with Definition of $runIdentity$

$$k\ a = k\ a \tag{4}$$

By Reflexivity

$$True \tag{5}$$

Figure 2. A Partial Verification of the `Identity` Monad

The consequence of these problems is that "easy" proofs are typically done either informally or not at all.

The formal verifications of functional programs that do exist are spread among a diverse set of proof assistants, logics, and representation techniques; each with their own pros and cons. Focusing on examples related to monads, proofs of the monad laws frequently arise as ad hoc lemmas in larger efforts [7, 22]. The monads in these verifications are "built to order", such that it is difficult to restructure their proofs to be useful outside of their original context. In other words, these proofs are not of immediate benefit to anyone attempting to verify existing and unrelated implementations.

Brian Huffman has presented work that more generally formalizes Haskell type classes [10], including `Monad`, using Isabelle/HOLCF [16] and its Tycon library [11]. Unfortunately, by the author's own admission, portions of the presented technique are impractical for widespread use. The domain-theoretic model of Haskell's type system is logically distant from what the average functional programmer is familiar with and an advanced working knowledge of the underlying proof system and logic is required for the more complicated cases.

Huffman et. al's early work [12] took place at a time before type classes were common, first-class constructs of proof languages. Since then, at least two popular proof systems, Coq [15] and Isabelle/HOL [17], have been extended with implementations of type classes similar to Haskell's [6, 21]. These systems can be used to more clearly model and verify `Monad` instances; and in the case of working with Isabelle/HOL, there even exists a tool to automate the necessary translations from existing Haskell code [5]. However, Coq and Isabelle are both large, complex systems that offer more reasoning power than is necessary in most cases. Furthermore, they can be overwhelming to work with for new users requiring significant background experience to use effectively.

By instead working with a lightweight and approachable tool that itself is implemented in Haskell, library authors would be able to more easily integrate formal verification into their existing workflows. In this paper, we present a novel approach to program verification that depends on such a tool.

The primary goal of this work is to mitigate or eliminate as many barriers to entry for formal reasoning as possible. Rather than working at the source level, as the previously discussed approaches do, we instead target the intermediate representation of the Glasgow Haskell Compiler (GHC) [4]. Using the compiler to desugar and simplify definitions to a core syntax allows us to translate them directly to an analogous higher-order logic (HOL) that should feel familiar to functional programmers.

By mechanizing this logic as a Haskell-embedded domain specific language (EDSL), we can entirely encapsulate the verification effort within a GHC compiler plugin. This provides us not only with the ease of integration and use that we desire, but as a secondary benefit we can replay verification results with the compiler on demand.

The remainder of the paper is structured as follows:

- Section 2 serves as a brief introduction to the theorem prover EDSL we will use to mechanize our proof efforts.
- Section 3 provides a high-level overview of GHC's plugin architecture and details the other major libraries critical to our work.
- Section 4 presents an informal translation semantics from GHC's core data types to HOL, including the current limitations of this translation.
- Finally, Section 5 works through an example to demonstrate the verification process in practice.

2. Verifying Constructor Classes in HaskHOL

The verification workflow described at the end of the introduction is predicated on the existence of a HOL proof system implemented in Haskell. One such system, HaskHOL, has been under development for the last several years [1]. Inspired collectively by HOL Light [8] and two of its modifications/extensions, Stateless HOL [25] and HOL2P [24], HaskHOL is a lightweight, monadic implementation of a second-order polymorphic HOL [2]. This logic is capable of representing a large subset of Haskell programs, including constructor classes such as `Monad`. We can demonstrate HaskHOL's applicability by using it to verify the introduction example.

Following from GHC's intermediate representation of type classes as dictionary-passing constructors, the `Monad` class is represented in HaskHOL as a constant term whose type is dictated by its argument constructor and operations:

```
forall (m :: * -> *).
  (forall a b. m a -> (a -> m b) -> m b) ->
  (forall a. a -> m a) ->
  Monad m
```

HaskHOL's polymorphism is restricted in that universally quantified type variables may only be instantiated by "small" types, i.e. types that are not themselves quantified. Additionally, type operators are distinct from regular types in the language such that they also cannot be used to instantiate quantified type variables. This leads to the construction of monads shown in Figure 3[1] where the type operator variable to be instantiated by a type constructor, `_M`, is left globally free.

[1] A HaskHOL syntax cheat sheet:
`'A` - Small type variables
`_M` - Type operator variables
`(\ / \\)` - Term-level term/type abstraction
`(! / !!)` - Term-level, universal term/type quantification
`[: 'A]` - Term-level type application
`$` - Type-level universal type quantification
`'A _M` - ML-Style, type-level type application

```
MONAD
  (bind : % 'A 'B. 'A _M -> ('A -> 'B _M) -> 'B _M)
  (return : % 'A. 'A -> 'A _M)  =
((!! 'A 'B.
   ! (a: 'A) (f:'A -> 'B _M).
     bind [: 'A] [: 'B] (return [: 'A] a) f = f a) /\
 (!! 'A.
   ! (m: 'A _M).
     bind m return = m) /\
 (!! 'A 'B 'C.
   ! (m: 'A _M) (f: 'A -> 'B _M) (g: 'B -> 'C _M).
     bind (bind m f) g = bind m (\ x. bind (f x) g)))
```

Figure 3. The Construction of Monads in HaskHOL

```
prove
  [str| MONAD
        (\\ 'A 'B. \ m k. k (RunIdentity m))
        Identity |] $
tacREWRITE [defMONAD] `_THEN` tacCONJ `_THENL`
[ _REPEAT tacGEN_TY `_THEN`
  tacREWRITE [defIdentity, defRunIdentity]
, tacCONJ `_THENL`
  [ tacGEN_TY `_THEN`
    tacMATCH_MP inductionIdentity `_THEN`
    tacREWRITE [defIdentity, defRunIdentity]
  , _REPEAT tacGEN_TY `_THEN`
    tacACCEPT thmTRUTH
  ]
]
```

Figure 4. The `Identity` Monad and its HaskHOL Proof

The `MONAD` constant is the conjunction of the monad laws, translated to HOL propositions. This translation includes making type abstractions and applications in the laws explicit, similar to how the GHC rewrite system requires pattern variables to be explicitly bound in rules. The specific proof obligation for the `Identity` instance can be achieved by supplying appropriate definitions for `bind` and `return`. Assuming that `Identity` and `RunIdentity` are constants in our working theory context, this produces the HOL term shown in Figure 4[2]

Included in Figure 4 is the proof tactic for this term. The term is rewritten using the definition of the `MONAD` constant to bring the instantiated monad laws into view. Each law is separated from the rest via a conjunctive split (`tacCONJ`), its bound types are generalized (`tacGEN_TY`), and the resultant subgoal is proved by rewriting (`tacREWRITE`). The rewriting step incorporates the provided list of definitions with the standard set of rewrites, e.g. beta reduction. Given that the definition of `runIdentity` depends on pattern matching against the `Identity` constructor, proving the left-identity law subgoal requires an extra step. We manually perform rule induction to handle pattern matching by invoking the primitive recursion theorem for the `Identity` type via the `tacMATCH_MP` tactic.

This proof tactic may seem daunting to those unfamiliar with HOL systems. However, it is simply a Haskell value that can be manipulated in the same ways as any other Haskell value. Specifically, we can abstract out the constant values that represent HOL theorems and use tactic combinators to make the structure more general. The function shown in Figure 5 will construct a proof tactic that will work for most constructor class verifications, provided that the proof obligation is derived following the above process.

[2] The `str` quasi-quoter prepares a `String` for HaskHOL's parser, notably removing the need to escape special characters.

```
proveConsClass consDef indThm thms =
    tacREWRITE [consDef] `_THEN` _REPEAT
    (_TRY (tacCONJ `_THEN` _REPEAT tacGEN_TY)
     `_THEN` _TRY (tacMATCH_MP indThm)
     `_THEN` tacREWRITE thms)

proveIDMonad =
    proveConsClass defMONAD inductionIdentity
      [defIdentity, defRunIdentity]
```

Figure 5. A More General Tactic for Constructor Classes

A programmer only needs to be able to identify three pieces of information to complete their proof:

1. The definitional theorem for the constructor class.
2. The recursion theorem to use for induction.
3. The list of additional theorems to use for rewriting.

For proofs not requiring induction, an `undefined` value can be supplied for the recursion theorem to intentionally fail, and therefore skip, that proof step. Proofs requiring more complicated induction schemes, however, will require the user to modify the structure of this tactic or develop a tactic of their own.

For now, we are operating under the simplifying assumption that the items enumerated above are available as part of an existing theory. We will explain this assumption in more detail in the following sections of the paper, but it is primarily due to the immaturity of the systems we are working with. The only item that needs to be constructed for a verification is the proof obligation itself. This obligation is a translation from the intermediate representation of a class instance to HOL, where the instance type constructor is replaced with the HOL constant for the class, e.g. `MONAD` in the example in Figure 4. As mentioned previously, our intention is to automate this translation as part of a compiler plugin.

3. The HERMIT with the KURE

Working at the core level of the compiler is beneficial for a number of reasons. As was mentioned in the introduction, the primary benefit is that the syntax we need to account for in our translation is simpler and maps more directly to our target HOL term language. We have also observed the following other benefits:

- The core syntax is extremely stable compared to the source syntax.
- Terms are safely assumed to be well-typed and correctly constructed before translation.
- Type information is stored locally, simplifying translation.
- Integration with GHC, Cabal, and other system tools is significantly easier.

The final point enumerated above, simplicity of integration, is in large part thanks to GHC's compiler plugin architecture. A GHC plugin modifies the compilation phases of the compiler pipeline with a function of type `[CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]`. In this signature, the abstract type `CoreToDo` represents a compilation pass, such that a plugin author can add, remove, modify, and reorder passes as they please by manipulating the `[CoreToDo]` argument. Knowing this, we can easily make verification a part of the compilation process by inserting a compilation pass of our own design into the pipeline at any point we desire.

The critical piece of any user-defined plugin is the function it uses to interact with a module. This function carries the type `ModGuts -> CoreM ModGuts`.

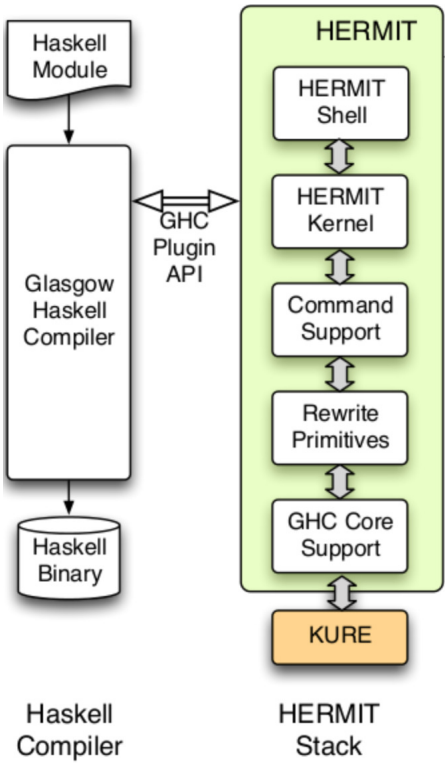

Figure 6. The HERMIT Framework

Aptly named, the `ModGuts` data type stores all of the information contained in the "guts" of the module currently being compiled. There are a number of fields in the `ModGuts` type that a verification plugin may be interested in, however, for this work we are primarily concerned with the `CoreProgram` value stored in the `mg_binds` field. As can be inferred from the name of the field, this data type holds all the top-level bindings in a module, stored as a list of variable and expression pairs. These expressions are values of the core type `CoreExpr` that will be examined in more detail in Section 4.

Ultimately, we require a translation function of type `CoreExpr -> HOLTerm`, where `HOLTerm` is HaskHOL's abstract data type for HOL terms. We could write this function directly using the `GhcPlugins` module provided as part of the GHC API. However, browsing this module's documentation[3] makes it clear that there would be a significant amount of integration work to do before translation could even be attempted. Instead, we rely on the Haskell Equational Reasoning Model-to-Implementation Tunnel (HERMIT) [3] to do most of the heavy lifting for us.

Originally built as a tool for interactively developing new optimization passes, HERMIT has since evolved to serve as a generalized framework for constructing new compilation passes of all forms. Figure 6, courtesy of Farmer et al., shows the key components of HERMIT [3]. At the root of HERMIT is the Kansas University Rewrite Engine (KURE), an EDSL for strategic rewriting [20]. HERMIT specializes the primitive combinators of KURE to provide generic traversals for GHC's core data types, as indicated by the "GHC Core Support" box in the aforementioned figure.

[3] `http://www.haskell.org/ghc/docs/latest/html/libraries/ghc-7.8.4/GhcPlugins.html`

These traversals are structured by KURE's `Transform` data type that abstractly models a transformation from type `a` to monadic values from type `m b`, with a given context of type `c`:

```
data Transform c m a b =
    Transform { applyT :: c -> a -> m b }
```

For HERMIT, `Transform` is specialized by the type synonym `type TransformH a b = Transform HermitC HermitM a b`. The `HermitM` monad is essentially GHC's core monad, `CoreM`, augmented with error handling as provided by KURE. There are additional effects structured by `HermitM`, but they are not utilized by this work.

The context for HERMIT transformations, `HermitC`, tracks all of the bindings in a module from the top-level down. The context also tracks the location inside of a module that a HERMIT computation is manipulating or accessing. These locations are stored with the digital equivalent of a trail of breadcrumbs that leads back to the root of the module. These paths of crumbs are an abstraction, again provided by KURE, that can also be used to dictate how to traverse an expression for transformation or rewriting.

HERMIT provides a dictionary of navigations that can compute a path to a target location in a module. The `bindingOfT` function, shown below, returns the path to the first binding it encounters whose variable satisfies a given predicate.

```
type LocalPathH = LocalPath Crumb

bindingOfT :: (Var -> Bool)
           -> TransformH CoreTC LocalPathH
```

Note that `bindingOfT` is itself a transformation that converts the constructors it encounters to their corresponding `Crumbs`. Given that a path could traverse any type from `ModGuts` all the way down to CoreExpr, HERMIT defines sum types that bundle entire data type hierarchies together. The `CoreTC` type is the most inclusive of these sum types.

When paired with the `cmpString2Var` function, `bindingOfT` makes for a clean and concise way to target definitions of a given name. For example, the following computation would apply a transformation to the binding that carries the intermediate representation of `Identity`'s `Monad` instance:

```
at (bindingOfT . cmpString2Var $ "$fMonadIdentity") $
   query trans
```

The types of the `at` and `query` combinators constrain the possible types of the transformation function, `trans`. Refining the statement from earlier in this section, the translation function we require is not of type `CoreExpr -> HOLTerm`, but rather `TransformH CoreTC HOLSum`, where `HOLSum` is the sum of HaskHOL's primitive types that mirrors `CoreTC`.

For our purposes, we know that we will not be manipulating any types above `CoreExpr` in the hierarchy. Provided we can write a transformation of type `TransformH CoreExp HOLTerm`, we can use KURE's `Injection` class and HERMIT's promotion combinators to bridge the gap from `CoreExpr` to `CoreTC`. We demonstrate the general technique for constructing such a transformation in HERMIT by focusing on a "dummy" translation of variables.

HERMIT's combinator for transforming `CoreExpr` variables is shown at the top of Figure 7. A `Var` is essentially a wrapper for typed identifiers, capable of representing terms and types both. The `varT` combinator lifts a transformation of `Id`s into a transformation of `CoreExpr`s that consist of a single variable occurrence. It unboxes the `Var` value, applies the sub-transformation to its internal and extends the working context with a `Crumb` to indicate that we are now inside of a variable. The `dummy` transformation takes the name of an `Id` and uses it to create a new, variable HOL term of an arbitrary type.

```
varT :: TransformH Id b
     -> TransformH CoreExpr b
varT t = transform $ \ c -> \case
    Var v -> applyT t (c @@ Var_Id) v
    _     -> fail "not a variable."

dummy :: TransformH Id HOLSum
dummy = contextfreeT $ \ id -> Tm $
    Tm $ mkVarType (name v) tyA
  where name = pack . unqualified . varName

trans :: TransformH CoreExpr HOLSum
trans =
     varT dummy
  <+ appT trans trans (\ (Tm x) (Tm y) -> ...)
```

Figure 7. Transforming `Var`s.

When `varT` is applied to `dummy`, a transformation of the desired type is produced. Note that the `varT` combinator will fail if presented with a `CoreExpr` value not constructed with `Var`. We can handle the cases for other constructors by writing transformations for them and then threading everything together with combinators that handle the exceptions. The above example uses the `(<+)` combinator, the KURE equivalent of `Alternative`'s `<|>`. The actual implementation of our translation function follows directly from the informal semantics discussed in the following section.

4. GHC Core

The intermediate language of GHC is an implementation of System $F^{\uparrow}_{C}$ [26], an extension of System F_C [23] that supports data type promotion. System F_C is itself is an extension of the the well known System F [18] that supports type-level equalities through coercion. Given that the foundational logic of HaskHOL is less expressive than System F, and therefore any of its extensions, it can only represent a subset of GHC's core language. The following is the list of assumptions we make about `CoreExpr` values which collectively define the limitations of our system:

1. Types are simply kinded, i.e. $\star$ or $k_1 \rightarrow k_2$, and/or are otherwise safe to ignore.
2. Reducing a type application results in a substitution or instantiation that obeys the restrictions of HaskHOL's polymorphism.
3. Casts are discharged or otherwise removed before translation, e.g. `newtype`s are replaced with equivalent `data` definitions.
4. Binding groups can be reduced to a list of possibly self recursive, but not mutually recursive, expressions.
5. All of the bindings in a group reside within a single module.
6. Constants of the language, primitives; literals; and user-defined alike, all map to analogous constants in an existing HaskHOL theory.

Assumptions 1 and 2 are due to limitations of HaskHOL's type system. Though there are HOL systems that can reason about kinds [9], HaskHOL currently lacks that capability. As for the previously discussed restrictions of HaskHOL's polymorphism, they are in place to maintain consistency of its logic; something that is critical for a proof system but not necessarily important for a type system.

Assumption 3 is primarily due to our unfamiliarity with the implementation of coercions in GHC. In practice, the coercion of `newtype`s cited in the assumption list above is the only type cast we have encountered in our, admittedly limited, testing.

```
data Type
  = TyVarTy  Id
  | AppTy    Type  Type
  | TyConApp TyCon [Type]
  | FunTy    Type  Type
  | ForAllTy Id    Type

data CoreExpr
  = Var  Id
  | App  CoreExpr CoreExpr
  | Lam  Id       CoreExpr
  | Type Type
```

Figure 8. GHC's Core – Simplified

It is our plan to iteratively develop this verification workflow, adding features as we can, so our hope is that coercions will be fully addressed by future work.

Assumption 4 is again due to a limitation of HaskHOL. Following from the logic of HOL Light that it is based on, HaskHOL permits mutually recursive data types, but not mutually recursive function definitions. Again, there are HOL systems that do have this capability [14], but HaskHOL is not currently one of them.

Assumptions 5 and 6 are due to limitations and design choices of HERMIT and the GHC plugin architecture. The only `ModGuts` value available to a plugin is the value for the module currently being compiled. While there is likely a way to store these values while compiling a sequence of modules, this technique would be infeasible for any code that depended on the base library or other libraries where the source code was not immediately accessible. Additionally, HERMIT does not currently provide combinators to transform the `TyCon` values that store type definitions, so translating constants would be supported for terms only without a sizable amount of additional work. While these limitations may be addressed in the future, for now we find it an acceptable compromise to depend on HaskHOL's definitions for constants.

A simplified view of GHC's core data types is shown in Figure 8. The `Cast`, `Lit`, and `LitTy` constructors have been removed following from our previously enumerated list of assumptions. The `Tick` constructor has been removed as its primary purpose is annotating information for profiling and debugging purposes, so it is not relevant to our translation. Finally, the `Case` and `Let` constructors have been removed as both map quite directly to meta-constructs in HaskHOL's term language. This makes their translations both awkward to formalize and comparatively uninteresting.

A corresponding, simplified view of the primitive data types of HaskHOL is shown in Figure 9. Constructors and constructor fields that are not critical to our translation have been converted to simpler types or removed entirely. This has mainly resulted in the removal of types that facilitated HaskHOL's semi-stateless features, leaving behind a term language very similar to that of HOL2P. Additionally, we have defined the `HOLSum` sum type that mirrors HERMIT's `CoreTC` type.

Figure 10 defines the predicate-based translations for the `Id` and `TyCon` types. Term variables translate directly between language and logic. As was mentioned in the assumptions above, when translating type variables we discard their kinds, keeping just their names. When translating type constructors, we map everything to primitive type operators in HaskHOL, making sure to handle the special case of the function type constructor at the time of translation. Primitive type operators are used rather than variable operators so that we can maintain the arity of types.

Figure 11 defines the translations for the constructors of the `Type` data type. Translations not involving type application proceed directly.

```
data HOLSum
    = Tm HOLTerm | Ty HOLType | TyOp TypeOp

data TypeOp
    = TyOpVar   String
    | TyPrim    String Int

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

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

Figure 9. HaskHOL's Primitive Types – Simplified

`Id` to `HOLSum`

$$\frac{varType\ id \longrightarrow ty}{\begin{array}{c} if(isId\ id):\ x = Tm\ (Var\ (varName\ id)\ ty) \\ else:\ x = Ty\ (TyVar\ (varName\ id)) \end{array}}{id \longrightarrow x}$$

`TyCon` to `TypeOp`

$$\frac{\begin{array}{c} if(isFunTyCon\ op):\ x = TyPrim\ \text{“}fun\text{”}\ 2 \\ else:\ x = TyPrim\ (tyConName\ op)\ (tyConArity\ op) \end{array}}{tyCon \longrightarrow x}$$

Figure 10. Translating `Id`s and `TyCon`s

Non-constructor type application translations depend on whether the operator of the application is a type variable or another application. In the case of a type variable, it is converted to a type operator variable and a HOL type application is built. We use a variable type operator variable rather than a primitive type operator for two reasons. First, we do not know the right arity to give to the operator since we have erased kind information. Second, it makes type substitutions performed when translating `CoreExpr` values slightly easier.

Constructor type application translations depend on the length of the argument type list. When passing a constructor as an argument to a type application, GHC will pair it with an empty type argument list in a `TyConApp` to satisfy the type of the `CoreExpr` `Type` constructor. HaskHOL, however, does not permit partial applications of type operators, so must check to see if a type operator is actually nullary or not before translating such applications. If the operator is nullary then we build the appropriate HOL type application, otherwise we just return the type operator itself. Note that this requires adding an additional translation case for `App` constructs to handle applying type operators.

Figure 12 defines the translations for the constructors of the `CoreExpr` data type. The rules for translating `Var` and `Type` are trivial. The rules for translating `App` and `Lam` values, however, are fairly complex. Given that we are currently mapping all constants to their equivalent values in HaskHOL, we do not need to translate type classes or other dictionary values.

`TyVarTy` to `HOLSum`

$$\frac{id \longrightarrow id'}{TyVarTy\ id \longrightarrow id'}$$

`AppTy` to `HOLSum`

$$\frac{id \longrightarrow Ty\ (TyVar\ x) \quad ty \longrightarrow Ty\ ty'}{AppTy\ (TyVarTy\ id)\ ty \longrightarrow Ty\ (TyApp\ (TyOpVar\ x)\ [ty'])}$$

$$\frac{ty1 \longrightarrow Ty\ (TyApp\ op\ tys) \quad ty2 \longrightarrow Ty\ ty2'}{AppTy\ ty1\ ty2 \longrightarrow Ty\ (TyApp\ op\ (tys +\!\!+ [ty2']))}$$

`TyConApp` to `HOLSum`

$$\frac{\begin{array}{c} op \longrightarrow TyOp\ op' \\ \hline if(arity\ op' = 0) : x = Ty\ (TyApp\ op'\ []) \\ else : x = TyOp\ op' \end{array}}{TyConApp\ op\ [] \longrightarrow x}$$

$$\frac{op \longrightarrow TyOp\ op' \quad for\ each\ i \in n,\ ty_i \longrightarrow Ty\ ty'_i}{TyConApp\ op\ [ty_1, ..., ty_n] \longrightarrow Ty\ (TyApp\ op'\ [ty'_1, ..., ty'_n]}$$

`FunTy` to `HOLSum`

$$\frac{ty1 \longrightarrow Ty\ ty1' \quad ty2 \longrightarrow Ty\ ty2'}{FunTy\ ty1\ ty2 \longrightarrow Ty\ (TyApp\ tyOpFun\ [ty1', ty2'])}$$

`ForAllTy` to `HOLSum`

$$\frac{id \mapsto Ty\ id' \quad ty \mapsto Ty\ ty'}{ForallTy\ id\ ty \longrightarrow Ty\ (UType\ id'\ ty')}$$

Figure 11. Translating `Type`s

Each of these constructors, therefore, has a case that essentially erases dictionary arguments or parameters accordingly, adjusting types as needed.

We have additional cases in the rules for `App` to force evaluation of type applications where possible. We do this primarily because of the limitation of HaskHOL's polymorphism that prevents the binding of type operator variables. For example,

```
(x : forall _m. forall a b. _m a b) (->)
```

is an allowable term in GHC's core language and will reduce to:

```
x : forall a b. a -> b
```

The HaskHOL equivalent,

```
(x : % _m. % 'a 'b. (a, b) _m) [: (->)]
```

is malformed for a number of reasons, though we can represent the reduced term fine:

```
x : % 'a 'b. 'a -> 'b
```

While not necessary, we also force the application of non-operator types. In most cases this produces terms that more closely match HaskHOL constants, as the majority were defined with globally free, rather than bound, type variables.

`Var` to `HOLSum`

$$\frac{id \longrightarrow id'}{Var\ id \longrightarrow id'}$$

`App` to `HOLSum`

$$\frac{\begin{array}{c}ty \longrightarrow Ty\ ty' \\ id \longrightarrow Tm\ (Var\ x\ (UType\ ty_1 @ (TyVar\ bv)\ ty_2))\end{array}}{\begin{array}{c}if (ty' = TyOp\ op) : xty = [TyOpVar\ bv \mapsto op]\ ty_2 \\ else : xty = [ty_1 \mapsto ty'] ty_2\end{array}}$$
$$\overline{App\ (Var\ id)\ (Type\ ty) \longrightarrow Tm\ (Var\ x\ xty)}$$

$$\frac{f \longrightarrow Tm\ f' \quad ty \longrightarrow Ty\ ty'}{App\ f\ (Type\ ty) \longrightarrow Tm\ (TyComb\ f'\ ty')}$$

$$\frac{\begin{array}{c}id_2 \longrightarrow Tm\ id_2' \\ id_1 \longrightarrow Tm\ id_1' @ (Var\ name\ (ty_1 \to ty_2))\end{array}}{\begin{array}{c}if (isDict\ id_2') : x = Tm\ (Var\ name\ ty_2) \\ else : x = Tm\ (Comb\ id_1'\ id_2')\end{array}}$$
$$\overline{App\ (Var\ id_1)\ (Var\ id_2) \longrightarrow x}$$

$$\frac{f \longrightarrow Tm f' \quad a \longrightarrow Tm a'}{App\ f\ a \longrightarrow Tm\ (Comb\ f'\ a')}$$

`Lam` to `HOLSum`

$$\frac{\begin{array}{c}id \longrightarrow id' \quad tm \longrightarrow Tm\ tm' \\ if (id' = Ty\ ty) : x = Tm\ (TyAbs\ ty\ tm') \\ else\ let (Tm\ id'') = id'\ in \\ if\ (isDict\ id'') : x = Tm\ tm' \\ else : x = Tm\ (Abs\ id''\ tm')\end{array}}{Lam\ id\ tm \longrightarrow x}$$

`Type` to `HOLSum`

$$\frac{ty \longrightarrow ty'}{Type\ ty \longrightarrow ty'}$$

Figure 12. Translating `CoreExprs`

5. Putting It All Together

In this section we will piece together the concepts from the previous sections to demonstrate a complete, assisted verification of the monad class for the `Identity` data type. Recall that, due to current limitations of the systems we are working with, all relevant definitions must be contained within a single module. The first step to our verification, therefore, is to construct this module. A complete module containing the definition of the `Identity` type, the `Monad` class, and their intersection is shown in Figure 13.

Next we need to make sure that we have a HaskHOL theory that contains the requisite definitions for both constructing our proof obligation and proving it. We have already discussed HaskHOL's formalization of monads in Section 2, which is included in the appropriately named Haskell theory.

```
module Monad where

import Prelude hiding (Monad, return, (>>=))

data Identity a = Identity a

runIdentity :: Identity a -> a
runIdentity (Identity a) = a

class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a

instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

Figure 13. The Monad Module

The following definitional theorems, some of which were previously used but not discussed, are also available as part of this theory:

```
inductionIdentity --
    |- !P.(!a.P (ID a)) ==> (!x.P x)

recursionIdentity --
    |- !f.?fn.!a.fn (ID a) = f a

defIdentity --
    |- Identity = (\\'a.(\ x . ID x))

defRunIdentity --
    |- runIdentity (ID x) = x
```

Together, the `inductionIdentity` and `recursionIdentity` theorems define the HOL equivalent of the Haskell data definition: `data Identity a = ID a`. The `defIdentity` theorem defines a term constant, `Identity`, that acts as a wrapper for the `ID` constructor. This extra level of indirection is required because the method for defining abstract types in HaskHOL is based on HOL Light's first-order polymorphism. Therefore, constructors are given less general types than their Haskell equivalents. Our long term plan is to provide a type definition method that will fully generalize constructors, but for now we must either use this trick or perform a type generalization transformation after the initial translation has completed.

Following the procedure laid out in Section 2, the remainder of our verification is carried out by:

- Translating GHC's intermediate representation of the target type class instance.
- Deconstructing the translated term into definitions corresponding to the class's methods.
- Further translating any locally available bindings that align with these definitions.
- Recombining the final version of the definitions, replacing the dictionary constructor with `MONAD` as the head term of the application.
- Proving the resultant term correct.

The `hermit` executable provided by the HERMIT library is an invaluable tool that can help us at each step of this process. This program compiles a module, launches HERMIT's interactive shell, and populates a context with information pulled from the module's `ModGuts`. An example execution of `hermit` on the `Monad.hs` module is shown in Figure 14.

```
module main:Monad where
  runIdentity :: ∀ a . Identity a → a
  $c>>= :: ∀ a b . Identity a → (a → Identity b) → Identity b
  $fMonadIdentity :: Monad Identity
hermit<0> binding-of '$fMonadIdentity
$fMonadIdentity = D:Monad ▲ $c>>= Identity
hermit<1> info
Node:        Binding Group
Constructor: NonRec

Path:     [prog, prog-tail, prog-tail, prog-head]
Children: [nonrec-rhs]

Free local identifiers:  $c>>=
Free global identifiers: Identity, D:Monad
Free type variables:
Free coercion variables:
Local bindings in scope:
  $c>>= : 2$NONREC
  runIdentity : 1$NONREC
```

Figure 14. `hermit`'s View of the `Monad.hs` Module

In this case, we have identified `$fMonadIdentity` as the name of the binding that corresponds to the type class instance we are targetting. We can expand the binding and examine its definition with the `binding-of` command. Its definition is formed by the application of four terms: the dictionary constructor, the instance type[4], and the two definitions for `Monad`'s methods.

As their names and their locations within this dictionary would imply, the `$c>>=` and `Identity` terms are the definitions for `(>>=)` and `return` accordingly. Querying for information about the binding group with the `info` command reveals that `Identity` is a globally free identifier. This, paired with the fact that its name has a leading capital letter, is a good indication that it is a constructor for a data type. Unfortunately, due to the previously mentioned limitations of HERMIT relating to type constructors, no other information about this identifier can be retrieved, and no other translation work can be done.

The `$c>>=` term, however, refers to another local binding in scope, so there is additional work to be done. If we expand its binding, we can see the expression it contains. We must translate this definition in order to produce the correct proof obligation:

```
$c>>= :: ∀ a b . Identity a → (a → Identity b) → Identity b
$c>>= = λ a b m k → k (runIdentity a m)
```

Note that the translation of the above expansion will match with the `bind` definition we supplied as part of the proof obligation back in Figure 4; a good indication that our verification is on the right path. In fact, translation produces an identical obligation, at least visually. Careful readers probably noted that the translation semantics presented in Section 4 never return constant terms. It is up to the users of the translated terms to substitute in constants for variables where appropriate.

This substitution is relatively easy to program thanks to the combinators and methods provided by HERMIT and HaskHOL. One possible implementation for term substitution is shown in Figure 15. Given that all of the constant names in this example mirror their HaskHOL mappings, we can simply look them up in our theory's list of defined `constants`. However, to account for cases when the names might not line up, the `repVar` function accepts an auxiliary mapping from variable names to constant terms. We search the union of this mapping and the `constants` mapping for the term that corresponds with a given name.

[4] HERMIT's default pretty-printer replaces type applications with green triangles for brevity's sake.

```
repVar :: Map Text HOLTerm -> Text -> HOLType
       -> HOL Proof thry HOLTerm
repVar tmMap i ty =
  do cs <- constants
     case mapLookup i $ cs `mapUnion` tmMap of
       Just (Const i' _) ->
           mkMConst i' ty <?>
             "type mismatch for term constant."
       _ -> return $! mkVar i ty
```

Figure 15. Substituting Constants for Variables

```
trans :: TransformH CoreTC HOLTerm
trans = ...

pass :: TheoryPath thry -> HOLTerm
     -> HPM HOLTerm
pass ctxt tm = liftIO $
    do tm' <- applyT replaceType ctxt tm
       applyT replaceTerm ctxt tm'

trans' :: Bool -> TheoryPath thry -> HOLTerm
       -> HPM HOLTerm
trans' True ctxt (Var name _) =
    pass ctxt =<<
      (at (lookupBinding $ unpack name) $
          query trans)
trans' _ ctxt tm = pass ctxt tm
```

Figure 16. Translation and Replacement

If a matching constant skeleton is found, we can use the `mkMConst` method to construct an instantiation of the constant that matches the provided type. If no match is found we return a variable of the provided name and type. Note that `repVar` fits nicely as a reconstruction function in a KURE transformation of `HOLTerms`.

```
replaceTerm :: Map Text HOLTerm
            -> Transform (TheoryPath thry)
               (HOL Proof thry) HOLTerm HOLTerm
replaceTerm tmMap =
       hvarT (contextfreeT return)
         (contextfreeT return) (repVar tmMap)
    <+ ...
```

Structuring this transformation to use a HaskHOL theory context and the `HOL` monad allows us to safely make calls to `HOL` computations, such as `mkMConst`, during transformation. The definition of other transformations, such as the replacement of type operator variables with type constants, follows similarly.

The entire translation process can be succinctly captured by the three definitions shown in Figure 16 The first definition, `trans`, is the transformation from `CoreTC` to `HOLTerm` defined by our translation semantics. The second definition, `pass`, sequences the replacement of type constants and term constants. The third definition, `trans'`, combines the two previous definitions, making the `trans` pass optional based on a boolean argument. This allows us to use the `trans'` function for both `$c>>=` and `Identity`, providing some uniformity to the definition of our plugin.

The entire definition of our plugin is included in Appendix A as a reference. We will also concisely explain the major steps here though, as they correspond to the itemized bullet-points above.

First we translate GHC's intermediate representation:

```
tm <- at (lookupBind "$fMonadIdentity") $
        query trans
```

Then we deconstruct the translated term:

```
let (_, [bnd, ret]) = stripComb tm
```

We further translate these arguments:

```
bnd' <- trans' True bnd
ret' <- trans' False ret
```

Recombine everything:

```
monad <- mkConstFull "MONAD" ([],[],[])
let tm' = mkIComb (mkIComb monad bnd') ret'
```

And finally we prove our constructed proof term:

```
prove tm'
 (proveConsClass defMONAD inductionIdentity
  [defIdentity, defRunIdentity])
```

As the plugin is completing this work, the user's terminal is keeping them informed:

```
> ghc Monad.hs -O2 -fforce-recomp
      -fplugin=HaskHOL.Haskell.Plugin
...
> Parsing constant mappings...
> Translating from Core to HOL...
> Translating Bind...
> Translating Return...
> Building Monad Instance...
> Proving...
> |- MONAD
       (\\'a 'b.(\ m k . k (runIdentity m)))
       Identity
```

Should you desire to experiment with this verification workflow yourself, all of the requisite HaskHOL packages are available from the first author's Github, `https://github.com/ecaustin/`. The `haskhol-haskell` package on this site contains the plugin itself, as well as a README containing installation and execution instructions. Please note that the development of this verification workflow is active research, such that the implementation of the plugin at the time you download it may differ from the presentation of this paper; hopefully because of improvements. More information regarding the HaskHOL system is also available at `haskhol.org`.

6. Related Work

The closest related work is a recent extension of HERMIT itself. HERMIT has always supported equational reasoning, insomuch that it could be used to mechanize the rewriting of Haskell terms. This reasoning was extremely restricted, though, as it could only be initiated from existing, not arbitrary, terms. Additionally, this reasoning was necessarily destructive because HERMIT had no notion of theorems or saved proofs. A rewrite could be saved as a script, however, the only way to "prove" it correct was to apply it and actually transform a term into its goal state.

When the GHC rewrite rule system was changed to permit inactive rules, it provided HERMIT with a source of core expressions that could be modified without affecting anything else in the compilation environment. HERMIT's rewrite engine was extended to allow direct reasoning over these rules, and a notion of lemmas was formalized to act as proof objects that could be saved and reused as attestations of equivalence [19]. Essentially, HERMIT lemmas were designed and implemented to provide a mechanism for safe and verified term rewriting.

One of the biggest differences between HaskHOL's logic and HERMIT's logic is that HERMIT's implementation of equational reasoning eschews the typical concerns of soundness in favor of practicality. Additionally, the expressivity of HERMIT's lemmas is significantly limited compared to the term languages of HaskHOL and other more general proof systems. For example, HERMIT lemmas can only express equivalences, not implications or any other statements based on non-equational, propositional connectives. That being said, the early work cited above would seem to indicate that HERMIT's extended equational reasoning system works well for its intended purpose.

The Haskabelle tool mentioned in the introduction is probably the next closest piece of related work. Like HaskHOL and HERMIT, the goal of Haskabelle is to facilitate equational reasoning of Haskell programs. The principal difference, though, is that Haskabelle operates at the source level, rather than at the intermediate level. The primary advantage of working at the source level is that the resultant specification and proof terms more closely resemble the original implementation. This advantage comes at a steep cost, though, as the Haskell programming language, or more specifically the GHC implementation of it, is constantly changing and adding new syntax that must be accounted for. Comparatively, the core language of GHC changes at a much slower rate.

The secondary consequence of working with Haskabelle is that it requires your verifications to be performed with the Isabelle system. This is not intended as an insult or backhanded comment about that system, it is simply a statement of fact. Isabelle is an incredibly impressive proof system, however, its reasoning capability is significantly overkill for most verifications that we care about. It is our opinion simple verifications should be completed with simple tools, and that the larger, more complex systems should be reserved for the larger, more complex problems.

Outside of the Haskell universe, there have been a number of other attempts to integrate formal reasoning tools with programming languages. One such example that we are familiar with is Köksal, et. al's work on integrating the Z3 SMT solver with the Scala programming language [13]. This integration differed from our work, in that their goal was to use Z3 to provide Scala with additional reasoning power, rather than use it to verify Scala programs. Additionally, they elected to integrate Z3 as a library using Scala's equivalent of Haskell's foreign function interface, rather than at the compiler level.

Both of these factors make their work much closer to any of the SAT and SMT binding libraries available on Hackage than ours. However, given that a number of these libraries are incomplete, abandoned, or both, it would seem to indicate that there are significant challenges to integrating reasoning tools with this approach. At the same time though, we will be the first to admit that there were significant challenges in reimplementing a formal logic in our language of choice rather than integrating with an existing tool. In either case, we point to the work of Köksal as an example of how beneficial a symbiotic relationship between formal reasoning tool and programming language can be.

7. Conclusions and Future Work

This work was primarily motivated by the desire to find a lightweight and approachable solution for verifying type class properties, specifically the monad laws. When we began our implementation and integration with the HERMIT system we knew the proposed verification workflow would be novel, however, we were not entirely sure it would be useable. Thankfully, we feel that the example presented in this paper has demonstrated the validity of our approach, at least for simple class instances. Before we experiment and see if our process scales to more complicated examples, though, there are a number of issues we feel we need to address with our future work.

First and foremost, as can be easily seen in the source code included in Appendix A, a significant portion of the presented plugin is hardcoded for this single example. Though the translation transformation, constant substitution transformations, and proof tactic are constructed for general use, this plugin will fail to compile and verify any module other than `Monad.hs`, as shown in Figure 13. This is a particularly troubling issue, as the current implementation of GHC plugins require them to be compiled and installed as part of a GHC package before they can be used. Making even a minor or insignificant change to a plugin requires the developer to follow these steps, adding a non-trivial amount of time to the development and testing process.

Some of the rigidity of the plugin can be removed by retrieving command line options rather than hardcoding values. However, it is not immediately apparent to us how to pass complex, polymorphic values, e.g. proof tactics, as command line options. One possible solution that we would like to pursue would be launching a robust, interactive proof environment to replace the current, final step of the plugin. This would allow users to both easily reuse existing proof tactics and develop new ones on the fly.

Beyond that, Section 4 contained a detailed list of limitations of our current implementation. This list was split roughly down the middle, with logical limitations of the HaskHOL system accounting for half of the issues and the immaturity of the implementation accounting for the other half. The past several years spent developing HaskHOL have taught us a valuable lesson that modifying the foundational logic of a proof system is no small undertaking. As such, we will likely endeavor to fix issues due to immaturity first. With these known limitations in mind, we still find the current state of the work exciting and encouraging.

Acknowledgement

We would like to thank Andy Gill and Andrew Farmer. Without their continued feedback and guidance, this work would not be possible.

References

[1] E. Austin. HaskHOL: A Haskell Hosted Domain Specific Language for Higher-Order Logic Theorem Proving. Master's thesis, University of Kansas, Lawrence, KS, 2011.

[2] E. Austin and P. Alexander. Stateless Higher-Order Logic with Quantified Types. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 469–476. Springer Berlin Heidelberg, 2013.

[3] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Haskell*, pages 1–12, 2012.

[4] GHC Team. The Glasgow Haskell Compilation System User's Guide, Version 7.8.2. Website:. http://haskell.org/ghc/.

[5] F. Haftmann. From Higher-Order Logic to Haskell: There and Back Again. In J. P. Gallagher and J. Voigtländer, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010*. ACM, 2010.

[6] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin Heidelberg, 2007.

[7] B. Halling and P. Alexander. Verifying a Privacy CA Remote Attestation Protocol. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 398–412. Springer Berlin Heidelberg, 2013.

[8] J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96), volume 1166 of Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[9] P. Homeier. The HOL-Omega Logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 244–259. Springer Berlin Heidelberg, 2009.

[10] B. Huffman. Formal Verification of Monad Transformers. In *ICFP*, pages 15–16, 2012.

[11] B. Huffman. Type Constructor Classes and Monad Transformers. *Archive of Formal Proofs*, June 2012. `http://afp.sf.net/entries/Tycon.shtml`, Formal proof development.

[12] B. Huffman, J. Matthews, and P. White. Axiomatic Constructor Classes in Isabelle/HOLCF. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin Heidelberg, 2005.

[13] A. Köksal, V. Kuncak, and P. Suter. Scala to the Power of Z3: Integrating SMT and Programming. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer Berlin Heidelberg, 2011.

[14] A. Krauss. Defining Recursive Functions in Isabelle/HOL.

[15] The Coq development team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2004. Version 8.0.

[16] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF. In *J. Funct. Program.*, pages 191–223, 1999.

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

[18] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[19] N. Sculthorpe, A. Farmer, and A. Gill. Making a Century in HERMIT. Submitted to peer-review for consideration for publication in the post-proceedings of IFL 2014, 2014.

[20] N. Sculthorpe, N. Frisby, and A. Gill. The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. Submitted to the Journal of Functional Programming, 2013.

[21] M. Sozeau and N. Oury. First-Class Type Classes. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer Berlin Heidelberg, 2008.

[22] C. Sternagel and R. Thiemann. Certification Monads. *Archive of Formal Proofs*, Oct. 2014. `http://afp.sf.net/entries/Certification_Monads.shtml`, Formal proof development.

[23] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[24] N. Völker. HOL2P - A System of Classical Higher Order Logic with Second Order Polymorphism. In *TPHOLs*, pages 334–351, 2007.

[25] F. Wiedijk. Stateless HOL. In *TYPES*, pages 47–61, 2009.

[26] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

A. The `Monad` Verification Plugin

```
plugin :: Plugin
plugin = hermitPlugin $ \ _ -> firstPass $
    do liftIO $ putStrLn "Parsing constant mappings..."
       let ctxt = ctxtHaskell
           liftHOL = liftHOL' ctxt
       tyMap <- prepConsts "types.h2h" $ liftHOL types
       tmMap <- prepConsts "terms.h2h" $ liftHOL constants
--
       liftIO $ putStrLn "Translating from Core to HOL..."
       tm <- at (lookupBind "$fMonadIdentity") $ query trans
       let (_, [bnd, ret]) = stripComb tm
--
       let pass :: HOLTerm -> HPM HOLTerm
           pass tm = liftIO $
               do tm' <- applyT (replaceType tyMap) ctxt tm
                  applyT (replaceTerm tmMap) ctxt tm'

           trans' :: Bool -> HOLTerm -> HPM HOLTerm
           trans' True (Var name _) =
               pass =<< (at (lookupBind $ unpack name) $
                            query trans)
           trans' _ tm = pass tm
--
       liftIO $ putStrLn "Translating Bind..."
       bnd' <- trans' True bnd
--
       liftIO $ putStrLn "Translating Return..."
       ret' <- trans' False ret
--
       liftIO $ putStrLn "Building Monad Instance..."
       monad <- liftHOL $ mkConstFull "MONAD" ([],[],[])
       let tm' = fromJust $
                    liftM1 mkIComb (mkIComb monad bnd') ret'
--
       liftIO $ putStrLn "Proving..."
       liftHOL $ printHOL =<<
         prove tm'
           (proveConsClass defMONAD inductionIdentity
            [defIdentity, defRunIdentity])
```