Compiler Directed Verification of Haskell Programs

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

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

ABSTRACT

Transitioning between an implementation and verification environment is a prime source for introducing complexity and errors to software verification. When working with a reasoning tool that is logically distant from the implementation language this problem is particularly evident. In this paper, we present a novel approach to verifying properties of Haskell programs that is entirely contained in, and directed by, the compiler. Leveraging GHC Haskell's compiler plugin framework, `Core` expressions are automatically translated to equivalent higher-order logic expressions for use with HaskHOL, a Haskell-based HOL theorem prover. To explore the applicability of the approach, we present an example utilizing this Haskell-in-Haskell verification work flow.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Languages, Verification

Keywords

Haskell, HOL, GHC, HaskHOL

1. INTRODUCTION

Among the most commonly cited advantages of working with purely functional languages is how much easier reasoning about program behavior becomes when it is referentially transparent. Unfortunately, for all of the benefits that the paradigm brings, it does nothing to assuage a major problem in software verification: marshaling knowledge between implementation and verification environments is, more often than not, a complex and error prone process. Our previous experience with the VSPEC [4] and Rosetta [1] tool suites provides anecdotal evidence supporting this claim, as we found it difficult to restructure prover outputs to be meaningful in the context of the original system specification. The consequence of this problem is that, what should be easy verifications of properties of purely functional programs are either done hand-wavingly or not at all.

Take, for example, the three laws ascribed to the `Monad` type class in the standard Haskell libraries:

```
{-
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
```

It is critically important that instances of the `Monad` class obey these properties as they support the correctness of the `do`-notation syntactic sugar. If instances are unfaithful in their implementation, the behavior of any program that utilizes them will be confusing at best and erroneous at worst. Yet, in spite of their importance, most monadic libraries either ignore these laws entirely or informally argue why their implementation is correct; the Haskell equivalent of "the proof is left as an exercise for the reader."

Brian Huffman approached this problem [11] by presenting a technique for verifying constructor class properties in Isabelle/HOLCF [15, 14]. Even under the assumption that a tool existed that could automate the necessary translations, e.g. a Haskabelle [8] equivalent for HOLCF, we found the solution to be less than satisfying. It required the use of a large and complex proof system and the domain-theoretic model of Haskell's type system is logically distant from the polymorphic lambda calculus that functional programmers are already familiar with. Both of these factors present significant barriers for Haskell library authors to overcome should they formally verify their work.

In this paper, we present early work on a novel approach to verifying properties of Haskell programs that will eliminate these barriers. Rather than working at the source level, we instead target GHC Haskell's [7] intermediate `Core` representation, such that we can translate it directly to an equivalent polymorphic, higher-order logic [3]. When implementing this logic as a Haskell-embedded domain specific language (EDSL), we can entirely direct the verification effort as a GHC compiler plugin. This allows a Haskell library author to integrate verification into their standard compilation process and have verification results replayed by the compiler on demand.

2. GHC COMPILER PLUGINS

Inspired by a similar feature in GCC [6] [1], the GHC Team recently added the ability for GHC to load user-written compiler plugins at compile time [2]. These plugins install additional compilation phases into the compiler pipeline through a function of type `[CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]`. In this signature, the abstract type `CoreToDo` represents a single compilation pass. These passes are constructed in a variety of ways to differentiate between phases of different purpose, however, we're primarily concerned with the `CoreDoPluginPass` constructor that structures compilation passes introduced via user-written plugin.

This constructor carries a name for the new pass and a function of type `ModGuts -> CoreM ModGuts`. Aptly named, the `ModGuts` type stores pertinent information contained in the "guts" of the module currently being compiled. Given that this work relies on the compilation plugin framework as a mechanism for automatic translation from implementation language to verification logic, we focus on the `CoreProgram` value stored in a `ModGuts` instance. This value holds all top-level bindings in a module, stored approximately as a list of `Core` expressions, the core data type that is the focus of Section 3.

Abstractly, we desire a translation function with type `Core -> HOLTerm`, where `HOLTerm` is the abstract representation of higher-order terms in our target theorem prover, HaskHOL [2]. We could write this function directly using the `GhcPlugins` module provided as part of the GHC API, however, browsing the source documentation [3] and the formal specification of `Core` developed by Richard Eisenberg [4] exposes a large volume of minutiae surrounding GHC's intermediate data types. Instead, we rely on the Haskell Equational Reasoning Model-to-Implementation Tunnel (HERMIT) [5] 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 1, courtesy of Farmer et al., shows the key components of HERMIT. At the root of HERMIT is the Kansas University Rewrite Engine (KURE), an EDSL for strategic rewriting [17]. 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.

These traversals are structured by the following data type providing a translation function from type `a` to type `b`, given a context of type `c` and structural monad of type `m`:

```
data Translate c m a b =
    Translate { apply :: c -> a -> m b }
```

[1]http://gcc.gnu.org/wiki/plugins
[2]https://www.haskell.org/ghc/docs/7.8.2/html/users_guide/compiler-plugins.html
[3]http://www.haskell.org/ghc/docs/7.8.2/html/libraries/ghc-7.8.2/GhcPlugins.html
[4]https://github.com/ghc/ghc/tree/master/docs/core-spec

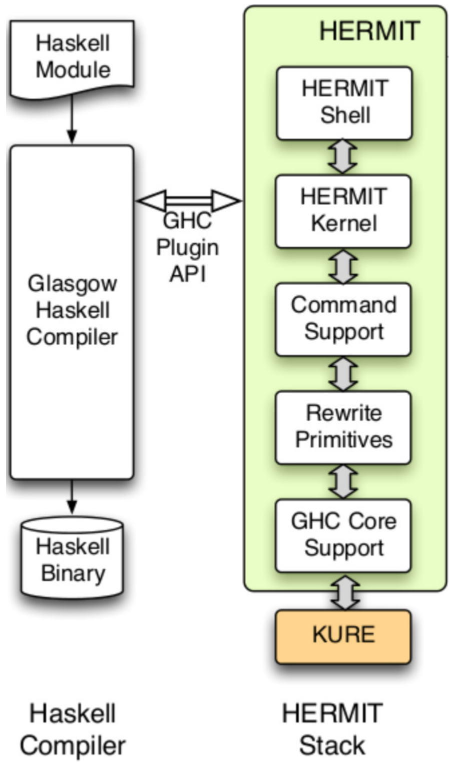

Figure 1: The HERMIT framework.

In HERMIT, this type is specialized to:

```
type TranslateH a b =
    Translate Context HermitM a b
```

where `HermitM` is GHC's core monad, `CoreM`, augmented with error handling. The `Context` for HERMIT translations is constructed in the other critical portion its framework, the "Kernel." All bindings in scope are converted to an abstract syntax tree that HERMIT can operate over, such that translations that require knowledge of other bindings can be made. Refining our earlier statement from this section, we are looking to develop a translation function with type `TranslateH Core HOLTerm` that HERMIT can use to install a new compilation pass for verification of a module at a phase of our choice.

3. GHC CORE

The intermediate language of GHC, `Core`, is an implementation of System $F_C^{\uparrow}$ [20], an extension of System F_C [18] that supports data type promotion. System F_C is itself is an extension of the the well known System F [16] that supports type-level equalities through coercion. Given this is still preliminary work, we make some simplifying assumptions about the possible values of the `Core` types:

- `Type`s are simply `Kind`ed, i.e. $\star$ or $k_1 \rightarrow k_2$.
- `Type` polymorphism is sufficiently restricted to be representable in our target logic, i.e. Rank-2 Types or simpler.

```
type CoreProgram = [Bind]

type Bind = (Id, Core)

data Core
  = Var   Id
  | App   Core Core
  | Lam   Id Core
  | Let   [Bind] Core
  | Case  Core Type [Alt]
  | Type  Type

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

type Alt = (AltCon, Core)

data AltCon
  = DataAlt DataCon
  | DEFAULT

data Id
  = TyVar { varName :: Name }
  | Id    { varName :: Name,
            varType :: Type }
```

Figure 2: The `Core` data type.

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

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

Figure 3: HaskHOL's primitive data types.

- `Cast`s can be discharged before translation of `Type`s, i.e. `newtype`s and `GADT`s can be replaced with equivalent `data` definitions.
- `Binding` groups can be reduced to a list of possibly self-recursive, but not mutually recursive, expressions.
- `Literal`, `TyLit`, `DataCon`, and `TyCon` constants all map trivially to equivalent constants in our target theorem prover.

This results in the simplified view of the `Core` data types shown in Figure 2.

The previously stated assumptions are made both to simplify the translation semantics and to guarantee that our initial examples can be mapped to HaskHOL's 2nd order polymorphic logic. We provide a simplified view of the primitive data types of our prototype implementation of this logic in Figure 3.

Figures 4 and 5 document a semantics for translating from `Core` to `HOLTerm`. Note that we rely on a more human readable, quasi-quoted form in the rules for the `Let` and `Case` constructor cases. This is purely to save space, as the definitions for syntactic sugar in most HOL systems expand to quite large applications of primitive constructors.

`Id` to `HOLType`

$$\frac{}{id \mapsto TyVar\ (varName\ id)}$$

`Core` to `HOLType`

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

`Type` to `HOLType`

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

$$\frac{id \rightsquigarrow op \qquad ty \mapsto ty'}{AppTy\ (TyVarTy\ id)\ ty \mapsto TyApp\ op\ [ty']}$$

$$\frac{ty1 \mapsto TyApp\ op\ tys \qquad ty2 \mapsto ty2'}{AppTy\ ty1\ ty2 \mapsto TyApp\ op\ (tys ++ [ty2'])}$$

$$\frac{con \rightsquigarrow op \qquad ty1 \mapsto ty1' \qquad ... \qquad tyn \mapsto tyn'}{TyConApp\ op\ [ty1, ..., tyn] \mapsto TyApp\ op\ [ty1', ..., tyn']}$$

$$\frac{ty1 \mapsto ty1' \qquad ty2 \mapsto ty2'}{FunTy\ ty1\ ty2 \mapsto TyApp\ tyOpFun\ [ty1', ty2']}$$

$$\frac{id \mapsto id' \qquad ty \mapsto ty'}{ForallTy\ id\ ty \longmapsto UType\ id'\ ty'}$$

Figure 4: Translation semantics for `HOLType`s.

4. MONAD VERIFICATION

We demonstrate the proposed verification pipeline by stepping through a proof of the monad laws for the instance of the `Identity` data type shown below. Note that the syntax of pretty-printed `HOLTerm`s used by HaskHOL is differentiated by the `[hol| ... |]` quasi-quoter for all examples in this section.

```
data Identity a = Identity a

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

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

In `Core`, a parameterized type class, such as `Monad`, is flattened to a higher-order dictionary constructor that accepts arguments for the definitions of its non-proper morphisms. The pretty-printed `Type` for the `Monad` dictionary constructor follows:

`Id` to `HOLTerm`

$$\frac{}{id \longmapsto Var\ (varName\ id)\ (varType\ id)}$$

`Core` to `HOLTerm`

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

$$\frac{\begin{array}{c} f \longmapsto f' \qquad arg \longmapsto arg' \\ let\ C = if\ (isType\ arg)\ then\ TyComb\ else\ Comb \end{array}}{App\ f\ arg \longmapsto C\ f'\ arg'}$$

$$\frac{\begin{array}{c} id \longmapsto id' \qquad tm \longmapsto tm' \\ let\ C = if\ (isTyVar\ id)\ then\ TyAbs\ else\ Abs \end{array}}{Lam\ id\ tm \longmapsto C\ id'\ tm'}$$

$$\frac{bnds \longmapsto [(id1, tm1), ..., (idn, tmn)] \qquad bod \longmapsto bod'}{Let\ bnds\ bod \longmapsto [|let\ id1 = tm1\ and\ ...\ idn = tmn\ in\ bod'|]}$$

$$\frac{alts \longmapsto [(c1, a1), ..., (cn, an)]}{Case\ x\ ty\ alts \longmapsto [|match\ x : ty\ with\ c1 \to a1|\ ...\ |cn \to an|]}$$

Figure 5: Translation semantics for `HOLTerm`s.

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

We model this constructor in HaskHOL by introducing a new term constant to our theory of matching type. All that is required is to provide explicit parameter names for the non-proper morphisms in their correct order:

```
newDefinition [hol|
  MONAD (bind : % 'A 'B. 'A _M ->
                  ('A -> 'B _M) -> 'B _M)
        (return : % 'A. 'A -> 'A _M) = ...
|]
```

Ignoring minor syntactic differences between the representations, we have a nearly identical signature, as one would expect with the translation semantics presented in the previous section. The only significant difference is that the original parameterized type, `m`, is left free in the HOL definition. We elected to make this simplification as we knew we would only be dealing with saturated applications of the `Monad` dictionary, thus its value is easily inferable and, therefore, redundant.

Following the technique presented in the examples of the polymorphic HOL system that influenced HaskHOL, HOL2P [19], we define the `MONAD` constant in terms of the conjunction of its associated properties. These properties are communicated to HERMIT as GHC rewrite rules[5] contained within the module:

```
{-# RULES
  "monad_law1" [~]
    forall a k. return a >>= k = k a
  #-}
```

We rely on the recently added special phase notation, `~`, to indicate to the compiler that we do not want the rule to be active. Thus we gain the advantage of having the rule parsed and typechecked to confirm that it is a valid Haskell expression without it interfering with the compilation of our source code.

In HERMIT, the introduced rule is available as a lemma that we can treat as a pair of `Core` expressions to be translated:

```
forall m a b $dMonad $dMonad a k.
(>>=) m $dMonad a b (return m $dMonad a a) k
==
k a
```

Following a second, similar simplification we can discard quantifications for the parameter type and any dictionary types. Thus, when provided with the requisite mapping from Haskell to HaskHOL identifiers, our translation semantics produces the following term:

```
[hol| !! 'A 'B. ! (a:'A) (k:'A -> 'B _M).
        bind [:'A] [:'B] (return [:'A] a) k =
          k a
|]
```

After translating the remaining rules, our definition of the `Monad` type class in HaskHOL is complete.

We proceed with the verification by translating remaining top-level bindings, introducing them to the theory context using HaskHOL's `defineType` and `define` methods accordingly.

```
  defineType "Identity = Identity A"

  define [hol| runIdentity = \\ 'A. \x.
            match x:'A Identity with
                Identity a -> a
        |]
```

With these definitions in place, we can build a proof obligation by translating the binding GHC generated for `Identity` 's `Monad` instance:

```
$fMonadIdentity :: Monad Identity
$fMonadIdentity =
  D:Monad Identity $c>>= Identity

$c>>= :: forall a b . Identity a ->
         (a -> Identity b) -> Identity b
$c>>= =
  \ a b m k -> k (runIdentity a m)
```

After again discarding any unnecessary type arguments and performing any necessary inlining, we have a fully saturated application of the `MONAD` constructor:

```
prove [hol|
  MONAD
    (\\ 'A 'B. \ m k. k (runIdentity [:'A] m))
    IDTYPE
  |] by
  ...
```

In this case, proof proceeds quite trivially via repeated simplification using the definitional theorems of the constants introduced in the previous steps.

[5]`http://www.haskell.org/ghc/docs/latest/html/users_guide/rewrite-rules.html`

5. CONCLUSIONS AND RELATED WORK

The workflow documented in the previous section is not fully fleshed out and is difficult to accurately evaluate. However, one obvious point to begin discussion with is the simplifying assumptions stated in Section 3. In some cases these simplifications are necessitated by a lack of functionality in the HaskHOL proof system. The restrictions on kinds and higher rank types, specifically, are mandated by HaskHOL's type system; largely inherited from the previously mentioned HOL2P. However, these restrictions could be lifted or lessened if the target logic was sufficiently advanced. The HOL-Omega system [10] is taking steps in this direction, differentiating itself from other members of the HOL family by introducing a universe of kinds that approaches that found in System $F^{\uparrow}_{C}$. Any future advancements to HaskHOL would likely be inspired by this research.

Similarly, the manipulation of `Binding` groups is mandated by the design of HaskHOL's `define` method; again, inherited from an inspirational system, HOL Light [9]. The ability to handle mutually recursive definitions exists in other members of the HOL family, though; notably in Isabelle/HOL [12]. In either case, the lack of functionality speaks more to the youth of HaskHOL than it does a failure or weakness in the proposed technique for compiler directed verification.

Remaining simplifying assumptions are made not with the intention to skirt difficult work, but because we have not fully explored the intricacies of GHC `Core`. As an example, note the differences in the definition of `runIdentity` when the `Identity` type is introduced as a `newtype` rather than a `data` type:

```
-- newtype
runIdentity :: forall a. Identity a -> a
runIdentity = \ a ds -> ds `cast`
     (axiomInst NTCo:Identity 0
      <Representational:a>)

-- data
runIdentity :: forall a. Identity a -> a
runIdentity = \ a ds ->
  case ds of wild a
    Identity a -> a
```

In the first piece of code the type of the argument term is coerced, essentially erasing the ephemeral `newtype` wrapper, without modifying its structure. For this specific example a conversion between the two forms seems immediately obvious, however, it opens the door to the larger question of how to handle `Coercion`s in general. For the time being, we elect defer questions like this in favor of focusing on increasing the level of automation in the proposed verification process rather than the number of language features it supports.

Automation is, of course, critically important to any verification process. So much so, that a number of popular theorem prover tools have added functionality to automatically generate executable source code from specifications verified in their environments. Given that the specification languages of proof systems like Coq [13] and Isabelle are becoming increasingly full-featured, this is arguably a preferable path to verifying *new* programs. However, our work is predominantly focused on verifying, and re-verifying on demand, *existing* programs.

The most closely related research that shares this goal is the previously mentioned Haskabelle tool. Completing the isomorphism between Haskell and Isabelle/HOL, Haskabelle translates existing Haskell source files into verifiable Isabelle specifications. The principal difference between that work and ours is that Haskabelle operates at the source level, where as we aim to operate at the intermediate, compiler level. Ultimately, the goal of our work is provide a translation from Haskell to HOL that is as robust as Haskabelle's, without requiring the added footprint of a major proof system, like Isabelle, and without leaving the GHC environment. It is our belief that by constraining the verification effort to a compiler plugin pass, we enable more automation with fewer errors.

6. REFERENCES

[1] P. Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

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

[3] 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.

[4] P. Baraona, J. Penix, and P. Alexander. VSPEC: A Declarative Requirements Specification Language for VHDL. In J.-M. Berge, O. Levia, and J. Rouillard, editors, *High-Level System Modeling: Specification Languages*, volume 3 of *Current Issues in Electronic Modeling*, chapter 3, pages 51–75. Kluwer Academic Publishers, Boston, MA, 1995.

[5] 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.

[6] Free Software Foundation. GNU Compiler Collection Homepage. Website:. http://gcc.gnu.org/.

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

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

[17] 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.

[18] 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.

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

[20] 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.