Certified `HLints` with Isabelle/HOLCF-Prelude

Joachim Breitner[1]*, Brian Huffman[2], Neil Mitchell[3] and Christian Sternagel[4]†

[1] Karlsruhe Institute of Technology, breitner@kit.edu
[2] Galois, Inc., huffman@galois.com
[3] ndmitchell@gmail.com
[4] JAIST, c-sterna@jaist.ac.jp

Abstract

We present the HOLCF-Prelude, a formalization of a large part of Haskell's standard prelude in Isabelle/HOLCF. Applying this formalization to the hints suggested by `HLint` allows us to certify them formally.

In pure functional languages such as Haskell, equational reasoning is a valuable tool for refactoring, to improve both efficiency and aesthetics. For example, an experienced programmer would replace `reverse ".txt" `isPrefixOf` reverse filename` with the more readable (and more efficient) `".txt" `isSuffixOf` filename`. In this paper we call such a replacement a *rewrite*. We only want to apply rewrites that are valid and thus some natural questions arise: *Is the original expression equivalent to the replaced expression?* With a language like Haskell, this entails the question: *What about when infinite or undefined values are involved?*

To highlight some of the issues, consider another example. Assuming the definition

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

can we safely apply the following rewrite?

$$\texttt{reverse (reverse xs)} = \texttt{xs} \tag{$\star$}$$

Let us try to prove ($\star$) by induction:

- **Base case** (`xs = []`). Just apply the definition of `reverse`.
- **Step case** (`xs = y:ys`). We have:

```
reverse (reverse (y:ys))
  = reverse (reverse ys ++ [y])          (by definition of reverse)
  = reverse [y] ++ reverse (reverse ys)  (using an auxiliary lemma)
  = reverse [y] ++ ys                    (by induction hypothesis)
  = y:ys
```

Such fast-and-loose reasoning [2] is oftentimes useful, but may fail for lazy languages: The above rewrite is neither valid for infinite `xs`, nor when `xs` contains undefined values on the spine. In addition to the above cases, we should have considered the undefined input $\bot$ (pronounced *bottom*) and made sure that the desired property is *admissible* for our setting.[1] These extra requirements can be tricky to follow, so automated assistance would be welcome.

Such assistance is available using higher-order logic for computable functions (HOLCF, [5]). HOLCF is based on the higher-order logic (HOL) instance of the proof assistant Isabelle [7] that provides functions, recursive definitions, (data) types, type classes, etc.; and constitutes a domain-theoretic framework that allows us to generate types in HOL that match the denotation

*Supported by the Deutsche Telekom Stiftung.
†Supported by the Austrian Science Fund (FWF): J3202.
[1] See [5] for a formal definition of admissibility.

of types in Haskell, i.e., with possibly infinite values and explicit bottom values. With these pieces we can define functions using Haskell-like pattern matching with call-by-need semantics which can handle both laziness and infinite data structures. The definition of `reverse` carries over quite naturally:

fixrec *reverse* **where**
  "reverse·[] = []" |
  "reverse·(x:xs) = reverse·xs ++ [x]"

Note that since Isabelle/HOL's default function type represents total functions, there is the special notation '·' for application of Haskell-like (i.e., continuous) functions.

In Isabelle/HOLCF every domain is equipped with a partial order $\sqsubseteq$ whose least element is $\bot$. We say *s is less defined than t*, whenever $s \sqsubset t$. To formalize our proof about `reverse` in Isabelle/HOLCF we switch from equality to $\sqsubseteq$. First we show how reverse "distributes" over list-append:

lemma *reverse_append_below*: *"reverse·(xs ++ ys) ⊑ reverse·ys ++ reverse·xs"*
proof (*induction xs*)
  case (*Cons x xs*)
  have *"reverse·(x:xs ++ ys) = reverse·(xs ++ ys) ++ [x]"* **by** *simp*
  also have *"... ⊑ (reverse·ys ++ reverse·xs) ++ [x]"*
    by (*rule monofun_cfun*)+ (*simp_all add*: *Cons.IH*)
  finally show *?case* **by** *simp*
qed *simp_all*

Then we obtain the desired lemma:

lemma *reverse_reverse_below*: *"reverse·(reverse·xs) ⊑ xs"*
proof (*induction xs*)
  case (*Cons x xs*)
  have *"reverse·(reverse·(x:xs)) = reverse·(reverse·xs ++ [x])"* **by** *simp*
  also have *"... ⊑ reverse·[x] ++ reverse·(reverse·xs)"* **by** (*rule reverse_append_below*)
  also have *"... = x : reverse·(reverse·xs)"* **by** *simp*
  also have *"... ⊑ x : xs"* **by** (*simp add*: *Cons.IH*)
  finally show *?case* **.**
qed *simp_all*

In both cases, the proofs just require induction followed by equational reasoning (*simplification* in Isabelle parlance), where all cases except for the step-case are trivial (i.e., solved automatically).

In order to make Isabelle/HOLCF more useful for the verification of Haskell programs, we have started to formalize some Haskell standard modules [9]. The result is the ongoing open source project Isabelle/HOLCF-Prelude[2] (HOLCF-Prelude for short). Contributions are most welcome and you may obtain the corresponding mercurial repository via

```
hg clone http://hg.code.sf.net/p/holcf-prelude/code holcf-prelude
```

As of version 0.1, it contains theories about booleans, the `Maybe` type, integers, tuples, lists, functions on those types, as well as the type classes `Eq` and `Ord`.

The tool `HLint`[3] (version 1.8.46) suggests improvements to Haskell code. Example suggestions include using more appropriate functions, eliminating redundant language extension pragmas and avoiding excessive bracketing. Many suggestions are rewrites, which are called *hints*. In hints, all single-letter variables are treated as free variables and the expression they match on

[2] http://sourceforge.net/p/holcf-prelude/
[3] http://community.haskell.org/~ndm/hlint/

the left-hand side is substituted on the right-hand side. `HLint` also allows a severity level (like `error` and `warn`ing) and notes to be associated with each hint. Notes are presented to the user along with the hint. Most hints represent equalities and have no note, but some are only true in certain circumstances. The majority of `HLint`'s hints are "obvious," but there are many of them, contributed by a large number of people. Manually checking the hints is error-prone, and several bugs have been reported by end-users (quite possibly after modifying their code in response to a hint).

The certification of these hints is our first application of the HOLCF-Prelude. Consider the hint (of severity level `warn`)

```
warn = reverse (reverse x) ==> x where note = IncreasesLaziness
```

which says that you should replace `reverse (reverse x)` in your code by `x`, but also notes that such a replacement will possibly increase the laziness of the program, meaning that there may be situations in which the original code crashes or does not terminate, while it will not do so after applying the hint. If we also have the file `test.hs` containing

```
output xs = print (reverse (reverse (sort xs)))
```

and run `HLint` on the file, it will respond with:

```
test.hs:1:20: Warning: Use alternative
Found:
  reverse (reverse (sort xs))
Why not:
  sort xs
Note: increases laziness
```

That is, `HLint` suggests a rewrite, and warns the user that the resulting expression is lazier than the original one, so if strictness was the purpose of using `reverse` the replacement may not be desirable.[4]

In order to facilitate the formal verification of such hints, we have modified `HLint` to generate Isabelle/HOLCF lemmas. We can do this for the above hint by running

```
hlint \
  --with='warn = reverse (reverse x) ==> x where note = IncreasesLaziness' \
  --proof=/dev/null --report
```

which generates a file `report.txt` containing the above hint in Isabelle notation:

```
reverse\<cdot>(reverse\<cdot>x) \<sqsubseteq> x
```

This we turn into the lemma (as shown in an Isabelle UI, such as Isabelle/jEdit or Isabelle/Proof General)

lemma *"reverse·(reverse·x) ⊑ x"*

that we have proven above.

Proofs for many of `HLint`'s default hints are already part of the HOLCF-Prelude. During our formalization we uncovered three previously unknown errors in `HLint` (and many missing annotations):

- The hint `take (length x - 1) x ==> init x` introduces a crash on the empty list (and thus was removed from `HLint`'s database).
- The hint `head (drop n x) ==> x !! n` is only true if the index is non-negative (and thus was modified to include this condition).

[4] Anyone wanting a spine-strict list would be better off using the more efficient `length x `seq` x` pattern.

- The hint `take i s == t ==> (i == length t) && (t `isPrefixOf` s)` was found to be erroneous (and thus was removed from `HLint`'s database).

Before starting this formalization effort a handful of hints had laziness annotations, but they were not intended to be complete. With the new scheme we know (for the proved hints) that we did not miss any annotations.

Known Issues. We haven proven some `HLint` hints correct, but while we can have confidence that the hint itself is correct, there are still ways the user can end up with incorrect rewrites.

The validity of any rewrite depends on the definitions of the functions involved. The Haskell standard [10] contains implementation suggestions for many functions, mostly aiming for simplicity and elegance, and we follow these definitions. In real compilers, for example GHC, the actual implementation is often somewhat different (look out for `USE_REPORT_PRELUDE` in the sources). In many cases, the definition is believed to be equivalent – for example `splitAt` in the GHC sources is performed with a single traversal of the list and unboxed `Int#` values, while the standard defines `splitAt` in terms of `take` and `drop`. In other cases the definition is only morally equivalent – for example `elem` is only equivalent for commutative definitions of `Eq`.

Another issue are type classes. Currently HOLCF-Prelude supports `Eq` and `Ord`. While it might be tempting to assume that the former implements an equivalence relation, such properties are not enforced by Haskell. What constitutes a valid instance of `Eq`? For maximal flexibility, we distinguish several cases in our formalization, among them: *Eq* (just syntactic, i.e., functions *eq* and *neq* are available for the type, where the default implementation for *neq* is assumed), *Eq_sym* (assuming *eq* is strict and symmetric) and *Eq_equiv* (extends *Eq_sym* to an equivalence relation). The question arises, when `HLint` suggests a rewrite involving *Eq*, what kind of properties may we assume? Currently we require annotations on all hints requiring properties of `Eq`, translate them to *Eq_sym* for the proofs, and display notes to the user when suggesting such replacements.

Another potential source of errors is in `HLint` itself. While the hint may be true, `HLint` has complicated unification routines tuned for performance, and issues like variable binding and capture have caused errors in older versions. `HLint` also performs various transformations to apply the hint in different circumstances. E.g., ($\star$) may be applied to expressions such as `reverse $ reverse xs`, where the `$` is translated away for matching purposes. Another limitation is that `HLint` does not perform full name resolution, approximating what set of names a particular identifier may refer to when searching for replacements.

Not an issue of validity but of expressiveness is our treatment of $\bot$: While to the Haskell developer, exceptions, pattern-match failures, system crashes, deadlocks and nontermination are very different things, in our semantics of the language, all of them are modeled as $\bot$, which is unique in every type. So for the purposes of the HOLCF-Prelude,

lemma *"head·[] = last·(repeat·1)"*

is a theorem, while no one would want to replace `head []` with `last (repeat 1)` in their code.

Related Work. To formally verify Haskell code there are two main approaches: 1) formalize functions and their desired properties in a proof assistant and then generate Haskell code; or 2) translate Haskell code to the language of a proof assistant and then prove the desired properties.

Generating Haskell code (that is correct by construction) is supported in several proof assistants, for example Isabelle [4] and Coq [6]. Such generated code can be called from normal Haskell code, but must originally be written in the language of the proof assistant, which Haskell programmers may find burdensome.

Translating Haskell code is the approach taken by tools such as Haskabelle [3] which produces Isabelle/HOL specifications. Haskell code can also be manually translated to syntactically

similar languages such as Agda [8]. Many of these approaches work in the setting of a strict language but fail to express propositions about laziness, undefinedness and infiniteness. Results obtained this way still hold in the lazy setting under certain conditions, as explained in [2].

To preserve the precise semantics of Haskell Abel et al. [1] provide a translation to Agda where functions are wrapped in an abstract evaluation monad. However, this yields Agda code that does not immediately resemble the original Haskell code. Our work allows for translating Haskell code to Isabelle/HOLCF specifications in a semantics preserving manner, without obscuring the relationship to the original code.

Conclusion and Future Work. We have presented the HOLCF-Prelude, our formalization of a large part of Haskell's standard prelude in Isabelle/HOLCF. Applying this formalization to the rewrites suggested by `HLint` allows us to provide certified hints. At the time of writing we have certified 143 of the 322 hints in `HLint`'s database. The usefulness of our approach is supported by the flaws found in the database. Most of the hints which remain unproven refer to types or type classes not modeled in our formalization (e.g., `Arrow`, `Functor`, `Monad`) or make statements about things happening at a lower level (e.g., `IO` and exceptions).

As future work, the same techniques may be applied to certify rewrites that are automatically applied by the Haskell compiler GHC [11].

References

[1] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying Haskell programs using constructive type theory. In *the ACM SIGPLAN Haskell Workshop, Haskell'05*, pages 62–73. ACM, 2005. doi:10.1145/1088348.1088355.

[2] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. *SIGPLAN Not.*, 41(1):206–217, 2006. doi:10.1145/1111320.1111056.

[3] F. Haftmann. From higher-order logic to Haskell: there and back again. In *the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM'10*, pages 155–158. ACM, 2010. doi:10.1145/1706356.1706385.

[4] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming, FLOPS'10*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010. doi:10.1007/978-3-642-12251-4_9.

[5] B. Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. PhD thesis, Portland State University, 2012.

[6] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, 2004.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

[8] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[9] S. Peyton Jones. Haskell 98 - Standard Prelude. *Journal of Functional Programming*, 13(1):103–124, 2003. doi:10.1017/S0956796803001011.

[10] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Online version: http://www.haskell.org/onlinereport/.

[11] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *the ACM SIGPLAN Haskell Workshop, Haskell'01*, pages 203–233, 2001.