Translating Haskell programs to Isabelle by shallow embedding

Paolo Torrini, Till Mossakowski, Christian Maeder

Abstract

We present a tool for the translation of Haskell programs to Isabelle HOLCF and Isabelle HOL based on a shallow embdedding approach, implemented as part of the Hets system, on top of the static analysis carried out by Programatica. The translation of monads uses the AWE extension of Isabelle.

It has been suggested that, owing to their relative semantical simplicity, programs written in functional languages are a natural target for verification [Tho94]. Haskell is indeed a strongly typed, purely functional language with lazy evaluation, based on a type system of polymorphic types with type and type constructor classes, where side effects are modelled using monads (introduced as a type costructor class), and allowing for a pseudo-imperative programming style (do notation) [Hud00].

In order to verify a program, we need to associate its code to a logical representation that allows for requirements to be expressed and for proofs to be carried out. There are different ways to do this. The way we carry this out here is to embed Haskell into an expressive higher-order logic, together with providing an implementation of the embedding that allows for an automated translation of Haskell. From the theoretical point of view, the correctness of the embdedding depends on the denotational semantics of the language. From the practical point of view, once the program code can be translated automatically to the logic, all the burden of the verification can rest on the theorem proving.

Depending on the expressiveness of the logic, the translation of the progrmaming language may be carried out at different levels of depth. In general, the deeper is the embedding, the less it relies on meta-level features of the target language. This may be a plus from the point of view of semantic clarity. Taking advantage of meta-level feaures, on the other hand, may be useful in order to make the theorem proving less tedious.

Isabelle is a generic theorem-prover written in SML that allows for the formalisation of a variety of logics [Pau94]. Isabelle-HOL is the implementation of classical higher-order logic based on simply typed lambda calculus, extended with axiomatic type classes. Isabelle-HOLCF [MNvOS99] is HOL extended with a formalisation of the theory of computable functions (LCF) based on axiomatic type classes.

Hets [MMLW03] is a parsing, static analysis and proof management tool designed to interface with various language specific tools, in order to support heterogeneous specification. In particular, for the parsing and static analysis of Haskell programs Hets relies on Programatica [Tea00], a tool that has its own proof management, including a translation of Haskell to Isabelle-HOLCF.

In the following, we present translations of Haskell into some of the Isabelle logics. These translations are implemented as part of Hets, and are based on a shallow embedding approach quite different from that used in Programatica. In our translations Haskell types are mapped to Isabelle types. We are assuming that the type systems of HOL and HOLCF are similar enough to the type system of Haskell, in order to guarantee the semantical correctness of our translations. Moreover, we rely on axiomatic classes to translate Haskell types classes.

A type class in Isabelle is conceptually quite different from one in Haskell. The former is associated to a set of axioms, whereas the latter is associated to a set of function declarations. Isabelle classes may have at most a type parameter. Moreover, Isabelle does not allow for type constructor classes. The last limitation is the most serious of the three, since it makes quite hard to cope with monads. We can get around this obstacle relying on a method for the axiomatic encoding of monads in HOL that uses parametrisation. Parametrisation based on theory morphism has been implemented, as a conservative extension of various Isabelle logics, in AWE, a tool designed to support proof reuse [JLB05].

The translation to HOLCF differs from that to HOL insofar as only the first one relies on a denotational semantics for the full language, keeping into account partiality and lazy evaluation, and associating recursive functions to corresponding fixed points, using on the Fixrec extension of HOLCF. The translation to HOL is more restrictive; it does not keep into account any form of partiality, and it restricts translation of recursive functions to the primitive recursive ones. It would have been possible to give an account of partiality in HOL by using option types. However, we chose to keep the HOL translation as simple as possible for didactic purpose.

These translations are on the overall quite different from other ones based on deep embeddings, particularly from the Programatica translation of Haskell into HOLCF [HMW05], where the type system is modelled at the logical level, as well as from translations of Miranda and Haskell into FOL (Isabelle classical first-order logic) where higher-order features have to be dealt with less directly [Tho94, Tho92].

The hs2thy tool

(suggested change: from h2hf to hs2thy) The tool is designed to support the translation of simple but realistic Haskell programs to HOLCF and, with some restriction, to HOL. Not all the syntactical features of Haskell are covered yet, although most of those used in the Prelude are. The most noticeable limitations are related to built-in types, pattern-matching and local definitions. There are additional and more substantial restrictions in the case of HOL, related to termination and to recursion.

The application can be installed as part of Hets. It requires Haskell GHC and Programatica. It is run by the command *h2hf [h—hc—mh—mhc] file.hs* where the options stand, respectively, for HOL, HOLCF, HOL extended with AWE and HOLCF extended with AWE.

For the sake of maintainance, *Haskell2IsabelleHOLCF.hs* is the module that contains the main translation function, carried out as theory translation after a preliminary step of signature translation, and *IsaSign.hs* contains the internal representation of Isabelle logics.

Type signature

Our general strategy for translating signatures is to map Haskell types to Isabelle defined types. The translation to HOLCF keeps into account the fact that Haskell expressions are evaluated lazily. The translation to HOL does not, and moreover requires that all the functions defined in the Haskell program are total ones...

All types have a sort in Isabelle, defined by the set of the classes of which it is a member. In HOL, all types belong to the class type; in HOLCF, they belong to class pcpo (pointed complete poset).

All Haskell function declarations are translated to Isabelle ones. Type variables are assigned default sort type in HOL and pcpo in HOLCF. Names get translated by a function that preserves them, up to avoidance of clashes with Isabelle keywords.

The only built-in types that are properly covered are booleans, unbounded integers and rationals. These get translated, in the case of HOL, to Isabelle booleans, integers and rationals, respectively. Bound integers and floating point numbers would require low-level modelling, and have not been dealt with; anyway, bounded integers are accepted and simply treated as unbounded by the application.

In the HOLCF translation, all built-in types get lifted to the corresponding domains (pcpos), which by definion include the undefined value, by using the lift HOLCF type constructor. In particular, type boolean get translated to type tr (short for bool lift), and similarly for types integer and rational.

Recursive data-types declarations can be translated to datatype declarations in HOL. In HOLCF they can be translated to domain declarations. The two structure are quite similar, except that in domains, which are pcpo, all the parameters are required to be pcpos as well. Morevover, Isabelle domain declarations require to introduce names for destructors. Mutually recursive Haskell datatypes can be translated to both logics, relying on the specific Isabelle syntax for them.

Function definitions

Haskell function definitions are translated to corresponding Isabelle ones. Non-recursive definitions are translated to ordinary definitions (keyword *defs*), whereas recursive ones require specific keywords. The type of the function, inclusive of class annotation, is included in each of its definitions in order to allow for overloading. In the translation to HOLCF, when the Haskell value is of built-in type, the translated value has the lifted corresponding type — this is either the case of a lifted value (where *Def* is the lifting function) or the case of the undefined value (UU).

As to local definitions, they can be introduced in Haskell by let and where expressions. Let expressions are translated to Isabelle let expressions, whereas where expressions are not covered.

Recursive definitions

Recursive defitions is the topic that really sets HOL and HOLCF apart. In HOLCF, where every type is a domain and every function is continuous, all recursive functions can be defined by fixpoint operator. This is essentially a function that, given as argument the function definition body abstracted of the recursive call name, returns the corresponding recursive function. Coding this directly would be rather cumbersome, particularly in the case of mutually recursive functions, where tuples of definition bodies and tupled abstraction are needed. We can instead rely on the *recdef* package, which allows to handle fixpoint definitions like ordinary recursive ones, offering nice syntax to deal with mutual recursion as well.

The case of HOL is more compplex. There a distinction must be drawn between primitive recursive functions and generic recursive ones. In the formers, termination is guaranteed by the fact that recursive definitions are strictly associated with the datatype structure of a parameter. In the latters, termination needs to be proved. This can be done, in a *recdef* definition, by providing a measure for the function parameter that can be proved to be strictly decreasing for each occurrence of a recursive call in the definition. This requires a generally unbounded amount of ingenuity and cannot be carried out by a simple translation.

For this reason, the translation of recursive functions to HOL is restricted to primitive recursive ones. Mutually recursive functions are allowed under additional restrictions. These are: all the functions should be recursive in the first argument, this should be of the same type for each of them, the constructors should be listed in the same order in each of the case expressions, and also the variable names in the case patterns are expected to be maintained.

The translation of mutually recursive functions $a \to b, \ldots a \to d$ introduces a new function $a \to (b * \ldots * d)$ defined, for each case pattern, as the product of the values correspondingly taken by the original functions.

Pattern matching

Support of pattern-matching in Isabelle is more restricted than in Haskell. Nested patterns are disallowed; case expressions are sensitive to the order of the constructors, which should be the same as in the datatype declaration. In particular, the case variable si required to be a variable. Translation of pattern-matching is potentially laborious. For this reason guarded expressions, list comprehension and nested pattern-matching have been left out; they can be replaced without loss, anyway, using conditional expressions and map functions.

Function definition by top level pattern matching is not allowed in Isabelle with ordinary definitions, but it is with those that are explicitly declared to be recursive ones. However, particularly in HOL primitive recursive definitions patterns are allowed in one parameter only. In order to translate function definitions with patterns in more than one parameter, without resorting to using tuples and more complex definitions (*recdef* instead of *primrec*), our translations turn a multiple definition by top level pattern matching into a definition by case construct.

Classes

Haskell classes and instances are translated to Isabelle, respectively, as classes with empty axiomatisation and as instances with trivial instantiation proof. The translation supports only single-parameter classes — in principle, more than one parameter could be handled using tuples.

Built-in classes — Eq and Ord in particular — are translated to newly defined classes in Isabelle; the corresponding axiomatisation however here is not provided.

Functions declarations associated to classes are translated to Isabelle as independent function declarations with appropriate class annotation. Function definitions associated to instances are translated as overloaded function definitions, relying on class annotation of typed variables.

Monads

Isabelle does not allow for classes of type constructors, hence a problem in representing monads. We can deal with this problem, however, relying on an axiomatisation of monads that allows for the presentation of monadic types as an axiomatic class, as provided in []. Therefore, monadic types can be translated to newly defined types that are constrained to satisfy the monadic axioms. This constraining, carried out as an instantiation of type variables in the theory of monads, takes the form of a theory morphism, and can be carried out automatically by AWE, an implementation of parametrisation and theory morphism on top of Isabelle basic logic, that may also be used to extend HOL and HOLCF. The do notation can then be translated by unpacking it into monadic operators. Attempting to translate monadic notation without support AWE support will result in an error message.

Conclusion

The following is a list of constructs that are covered by all the translations.

- predefined types: boolean, integer.
- predifed type constructors: funtion, cartesian product, list.
- declaration of recursive datatype, including mutually recursive ones.
- predefined functions: equality, booelan constructors, connectives, list constructors, head and tail list functions, arithmetic operations.
- non-recursive functions, including conditional, let and case expressions.
- use of incomplete patterns (HOLCF) and of wildcards in case expressions.
- total primitive recursive functions (HOL) and partial recursive ones (HOLCF), including mutual ones (with restrictions in the HOL case).
- class and instance declarations.
- monad declarations and do notation.

The shallow embedding approach used by our translations should allow to take as much advantage as possible of the automation currently available on Isabelle, particularly in HOL.

Further work should include extending the translation to cover the whole of the Haskell Prelude. We hope that this will make it possible to support the verification of functional robotics programs.

A further extension, in line with the work on Programtica, would be the translation of P-logic. This translation, due to the character of the P-logic typing system, could be more easily carried out relying on some form of universal quantification on type variables, or else further relying on parametrisation.

References

[HMW05] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle-HOLCF. Research paper, OGI, 2005.

[Hud00] P. Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[JLB05] E. B. Johnsen, C. Lueth, and M. Bortin. Formal software development with Isabelle. Research paper, Universitaet Bremen, 2005.

[MMLW03] T. Mossakowski, C. Maeder, K. Luettich, and S. Woelfl. *The Heterogeneous Tool Set*, 2003.

[MNvOS99] O. Mueller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 1999.

[Pau94] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828. Springer, 1994.

[Tea00] The Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in Haskell. Technical report, OGI, 2000.

[Tho92] S. Thompson. Formulating Haskell. In *Functional Programming*. Springer, 1992.

[Tho94] S. Thompson. A logic for Miranda, revisited. *Formal Aspects of Computing*, 3, 1994.