Translating Haskell to Isabelle logics in Hets

Paolo Torrini, Till Mossakowski, Christian Maeder

No Institute Given

Abstract. **Automated, partial translations of Haskell into Isabelle higher-order logic (with or without computable functions) have been implemented as functions of Hets, a Haskell-based proof-management and program development system that allows for integration with other tools. The application, built on top of Programatica-style static analysis, can translate simple Haskell programs to HOLCF and, under stronger restrictions, to HOL. Both translations are based on shallow embedding and rely informally on denotational semantics.**

Work on the integration of compilers, analyzers and theorem-provers, as well as translations between programming and specification languages, may help on the way to making formal development and verification of programs more viable. In order to verify formally a program, we need to formulate the requirements in a logic that also allows for the program to be represented; to translate the program to the logic; finally, to carry out a proof of the correctness statement. Although specifications and proofs require normally the biggest amount of work, translations can also be a significant source of potential problems. For the verification to be reliable and efficient, the translation should rest on the definition of a formal semantics of the programming language in the specification logic; it should be carried out safely, at best automatically; it should also give program representations that do not make proofs more complex than necessary. It has long been argued that functional languages, based on notions closer to general, mathematical ones, can make the task of proving assertions about them easier, owing to the clarity and relative simplicity of their semantics [?].
In this report we are presenting automated translations based on denotational semantics of Haskell programs into Isabelle higher-order logic. Haskell is a strongly typed, purely functional language with lazy evaluation. It relies on a system of polymorphic types extended with type constructor classes, and has a syntax for side effects and pseudo-imperative code based on monadic operators [?]. The translations are implemented as functions of Hets [?], an Haskell-based application designed to support integration of specification formalisms for the development of Haskell programs. Hets supplies with parsing, static analysis, proof management and interface to various language-specific tools. From the point of view of generic theorem proving, Hets relies on an interface with Isabelle, a state-of-the-art interactive theorem-prover, written in SML, allowing for the formalization of a variety of logics [?]. Hets relies on Programatica [?] for the parsing and static analysis of Haskell programs. Programatica is a Haskell-specific formal development system built at OGI, envisioned to provide more than analysis. Its own proof management includes a specification logic and translations to different proof tools, notably to Isabelle [?]. In section it is going to be clarified how our work differs from theirs.

1 Isabelle

Isabelle-HOL (hereafter simply HOL) is the implementation in Isabelle of classical higher-order logic based on simply typed lambda calculus, extended with axiomatic type classes. It provides considerable support for reasoning about programming functions, both in terms of libraries and automation. Since the late nineties, it has essentially superseded FOL (classical first-order logic) as the logic of standard use in Isabelle. HOL has an implementation of recursive functions based on Knaster-Tarski fixed-point theorem. All functions are total; partiality may be dealt with by lifting types through the *option* type constructor.
HOLCF [?] is HOL conservatively extended with the logic of computable functions — a formalization of domain theory. In HOL, types are just sets; functions may not be computable, and a recursive function may require discharging proof obligations already at the definition stage — in fact, a specific measure has to be given for the function to be proved monotonic. In contrast, domain theory has each type interpreted as a *pcpo* (pointed complete partially ordered set) that is, as a set with a partial order which is closed w.r.t. ω-chains and has a bottom. All functions defined over domains, including partial ones, are continuous, therefore computable. Recursion can be expressed

in terms of least fixed-point operator, and so, in contrast with HOL, function definition is never dependant on proofs.
The Isabelle formalization of HOLCF is based on axiomatic type classes [?], following an approach that makes it possible to deal with complete partial orders while abstracting away from any specific relation. The class of types is *pcpo* — whereas in HOL it is *type*. Domain theory offers a good basis for the semantical analysis of programming languages; however, it may make proofs comparatively hard. After being spared the need to discharge proof obligations at the defining stage, one has to bear with assumptions over the continuity of functions while proving theorems. A standard strategy to get the best out of the two systems is to define as much as possible in HOL language, using HOLCF type constructors to lift types to domains only when this is needed.

2 Translations

A translation may be carried out relying on different semantical approaches and at different levels of depth, depending mainly on the expressiveness of the target logic. Different formalisms may make the embedding of certain features more or less hard. Translations into FOL such as those based on large-step operational semantics of Miranda [?,?,?] and Haskell [?] cannot deal straightforwardly at first order with higher-order features such as currying. The translation of Haskell to the Agda implementation of Martin-Loef type theory in [?] gets complicated dealing with Haskell polymorphism. The expressiveness of higher-order logic help overcome more plainly most of these obstacles, and makes it possible to adopt a higher-level approach based on denotational semantics, as proposed in [?] to translate Haskell to HOLCF, and used in [?] to translate ML to HOL. This approach allows for representations of programs that are closer to their specification as well as for proofs that are relatively more abstract and general.
Expressiveness plays an important role in the "depth" issue, as well. A shallow embedding is one that relies as much as possible on built-in features and packages provided with the implementation of the target language, especially with respect to general features such as types, classes, and recursion. The deeper the embedding, the less it relies on such features. This independence may be a plus from the point of view of semantic clarity and logical generality — object-logical, "deep" translation can be used to overcome *ad-hoc* limitations imposed by the built-in, meta-level features. Taking advantage of those features, on the other hand, may help make the theorem proving less tedious and make it easier to rely on common proof methods.
The translation of ML in [?] gives an example on the deep side — a class of types with bottom elements is defined in HOL for the sake of the embedding. On the other hand, the translation of Haskell to HOLCF proposed in [?] relies on a generic formalization of domain theory, particularly on the *fixrec* package for recursive functions (part of HOLCF in Isabelle 2006) developed in order to provide with a friendly syntax that covers mutually recursive definitions, too. Not even this translation is shallow as far as types are concerned, though. In order to capture an important feature of the Haskell type system, notably type constructor classes, Haskell types are not translated to HOLCF types, rather they are to terms, relying on a modelling of the Haskell type system at the object level. In this way it is possible to give a complete account of the type system. The practical drawback is that plenty of the automation built into the Isabelle type checking is lost, unless one is prepared to reimplement a lot.
The translations of Haskell to HOLCF and HOL that we are presenting here are based on denotational semantics keeping to a shallow approach. The translation to HOLCF relies on the *fixrec* package, along similar lines to those in [?]. In contrast with them, we translate Haskell types to HOLCF types, quite directly, and Haskell classes to Isabelle axiomatic classes, wishing to take effectively advantage of Isabelle type checking. We rely on the assumption that types in HOLCF are similar enough to those in Haskell to allow for embedding, save for possible implementation subtleties that we are not going to consider. We expect that operational equivalence between Haskell programs and their translation to HOLCF holds up to the level of typeable output. The translation covers a significant part of the syntax used in the Prelude, although it is still incomplete in one important respect — it does not include type constructor classes; we have plans, however, for an extention that should address this aspect as well.
Conceptually, type classes in Isabelle are quite different from those in Haskell. The formers are associated with sets of axioms, whereas the latters come with sets of function declarations. Isabelle classes may have at most a single type parameter. Most importantly, Isabelle does not allow for type constructor classes. The last limitation is serious, since it makes hard to cope with essential Haskell features such as monads and *do* notation. In alternative to the treatment of types proposed in [?],

we would like to get around the obstacle by relying on an extension of Isabelle based on theory morphism (see section). The AWE system [?] is in fact an implementation of such an extension.
Our translation to HOL, shallow as well, is much more limited. The most important restriction is related to recursion — only primitive recursive functions are covered. This limitation appears relatively hard to overcome, given the way syntax for full recursion works in HOL. Operational equivalence up to typeable output for the remaining fragment would require using option types, but we do not pursue it here, rather we rely on a restriction to terminating programs for semantical correctness. Under such restrictions, however drastic, this translation gives expressions that are particularly simple and therefore potentially useful to verify some properties.

3 Haskell2Isabelle

The Hets function *Haskell2Isabelle* supports the translation of simple Haskell programs to HOLCF and, with more restriction, to HOL. Not all the syntactical features used in the Prelude and main libraries are covered. Some of the most noticeable limitations are those related to built-in types, pattern-matching, local definitions, import and export. There are additional and more substantial restrictions in the case of HOL, related to termination and recursion. Each of the translations relies on a *base theory*. These are Isabelle theory files, respectively *HsHOLCF*, extending HOLCF, and *HsHOL*, extending HOL, which are included in the Hets distribution. Each of them provides definitions and axiomatizations of notions that are used in the corresponding translation — notably equality.
Information for the use of Hets may be found in [?] and a general outlook in [?]. The Haskell-to-Isabelle translation requires essentially GHC, Isabelle 2006 and Programatica — with respect to the latter, both analysis and translation functions in Hets make use of its modules. The command to run the application is
hets -v[1–5] -t Haskell2Isabelle[HOLCF — HOL] -o thy filename
where options set verbosity, target logic and name of the output file. The input file (last argument) must be a GHC source (*hs* extension). The Haskell program gets analyzed and translated. The result of a successful execution is an Isabelle theory file (*thy* extension) depending on the corresponding base theory.
The internal representation of Haskell in Hets (module *Logic_Haskell* and particularly *HatAna*) is essentially the same as in Programatica, whereas the internal representation of Isabelle (module *Logic_Isabelle* and particularly *IsaSign*) is a Haskell reworking of the ML definition of Isabelle own base logic, extended in order to allow for a straightforward representation of HOL and HOLCF.
Haskell programs as well as Isabelle theories are internally represented as Hets theories — each of them a data-structures formed by a signature and a set of sentences, fitting a theoretical framework described in [?]. Each translation, defined as composition of the signature translation with the translation of all sentences, has the structure of a morphism from theories in the internal representation of the source language to those in the representation of the target language. The distribution module *Haskell2IsabelleHOLCF* contains the main function, dependent on the target logic. The module *IsaPrint* provides the essential functions for the pretty-printing of Isabelle theories.
The following gives a list of reserved names, i.e. the names that are used in order to either rename or name automatically variables and constants in the translations. 1) Type variables, in the translation to HOL: *'vX*; any name terminating with *'XXn* where *n* is an integer. 2) Term variables, in both translations: *pXn*, *qXn*, with *n* integer. 3) Constants, in the translation to HOL: strings obtained by joining together names of defined functions, using *_X* as end sequence and separator.

3.1 HOLCF: Type signature

The translation to HOLCF keeps into account partiality, i.e. the fact that a function might be undefined for certain values, either because definition is missing, or because the program does not terminate. It also keeps into account laziness, i. e. the fact that by default function values in Haskell are passed by name and evaluated only when needed. Essentially, we are following the main lines of the "crude" denotational semantics for lazy evaluation in [?], pp. 216–217. Raising an exception is different from running forever, and both are different from stopping short of evaluation. However, from the point of view of the printable output, these behaviours are similar and can be treated semantically as such, i.e. using one and the same bottom element.
Each type in Isabelle has a sort, defined by the set of the classes of which it is member. Haskell type variables are translated to HOLCF ones, of class *pcpo*. Each built-in type is translated to

the lifting of its corresponding HOL type. Properly covered are Haskell booleans and unbounded integers, associated respectively to HOL booleans and integers. Bound integers and floating point numbers would need low-level modelling, and have not been covered. Bounded integers in particular are simply treated as unbounded in the translation. The HOLCF type constructor *lift* is used to lift HOL types to flat domains. In the case of booleans, we can use type *tr*, defined as equal to $bool\ lift$ in HOLCF. In the case of integers, we use *dInt*, defined in *HsHOLCF* to equal $int\ lift$. The types of Haskell functions and product are translated, respectively, to HOLCF function spaces and lazy product — i.e. such that $\bot = (\bot * \bot) \neq (\bot *' a) \neq ('a * \bot)$, consistently with lazy evaluation. Type constructors are translated to corresponding HOLCF ones (noticeably, parameters precede type constructors in Isabelle syntax). In particular, lists are translated to the domain *llist* defined in *HsHOLCF*. Function declarations are translated to HOLCF ones (keyword *consts*). Names (for types as well as for terms) are translated by a function (t here) that preserves them, up to avoidance of clashes with HOLCF keywords. Translation of types (minus mutual recursion) may be summed up as follows:

$$\begin{array}{ll}
\lceil a \rceil & = {}'a_t :: pcpo \\
\lceil Bool \rceil & = tr \\
\lceil Integer \rceil & = dInt \\
\lceil a \to b \rceil & = \lceil a \rceil \to \lceil b \rceil \\
\lceil (a, b) \rceil & = \lceil a \rceil * \lceil b \rceil \\
\lceil [a] \rceil & = \lceil a \rceil\ llist \\
\lceil TyCons\ a_1 \ldots a_n \rceil & = \lceil a_1 \rceil \ldots \lceil a_n \rceil\ TyCons_t
\end{array}$$

In HOL, datatype declarations define types of class *type* by keyword *datatype*; in contrast, in HOLCF, they define types of class *pcpo* (i.e. domains) by keyword *domain* (so we also may call them *domain declarations*). Each recursive datatype declaration in Haskell is translated to the corresponding one in HOLCF. The translation of mutually recursive datatypes relies on specific Isabelle syntax (keyword *and*), as in the next example.

$$\begin{array}{l}
data\ AType\ a\ b\ =\ ABase\ a \mid AStep\ (AType\ a\ b)\ (BType\ a\ b) \\
data\ BType\ a\ b\ =\ BBase\ b \mid BStep\ (BType\ a\ b)\ (AType\ a\ b)
\end{array}$$

This translates to HOLCF as the following.

$$\begin{array}{rl}
domain & ('a :: pcpo,'b :: pcpo)\ BType\ =\ BBase\ (BBase\_1 ::' b) \mid \\
 & BStep\ (BStep\_1 :: ('a,'b)\ BType)\ (BStep\_2 :: ('a,'b)\ AType) \\
and & ('a :: pcpo,'b :: pcpo)\ AType\ =\ ABase\ (ABase\_1 ::' a) \mid \\
 & AStep\ (AStep\_1 :: ('a,'b)\ AType)\ (AStep\_2 :: ('a,'b)\ BType)
\end{array}$$

Notably, domain declarations require an explicit introduction of destructors. Both translations (to HOL as well as to HOLCF) take care automatically of the order of datatype declarations — this is needed, insofar as, differently from Haskell, Isabelle requires them to be listed according to their order of dependency.

3.2 HOLCF: Sentences

Each Haskell function definition is translated to a corresponding one. Non-recursive definitions are translated to standard ones (keyword *defs*), whereas the translation of recursive definitions relies on the *fixrec* package. Lambda abstraction is translated as continuous abstraction (LAM), and function application as continuous application (the *dot* operator) — these notions coincide with the corresponding, HOL-defined ones, whenever their arguments are continuous.
Terms of built-in type (boolean and integers) are translated to lifted HOL values, using the HOLCF-defined lifting function *Def*. The bottom element $\bot$ is used for all undefined terms. The following operator, defined in *HsHOLCF*, is used to map binary arithmetic functions to lifted functions over lifted integers.

$$\begin{array}{l}
fliftbin :: ('a \Rightarrow\ 'b \Rightarrow\ 'c) \Rightarrow ('a\ lift \to\ 'b\ lift \to\ 'c\ lift) \\
fliftbin\ f\ ==\ LAM\ yx.\ (flift1\ (\%u.\ flift2\ (\%v.\ f\ v\ u))) \cdot x\ \cdot y
\end{array}$$

Booleans are translated to values of *tr* — *TT*, *FF* and $\bot$, and boolean connectives are translated to the corresponding HOLCF-defined lifted operators. HOLCF-defined *If then else fi* and *case* syntax

are used to translate, respectively, conditional and case expressions. There are some restrictions, however, on the latters, due to limitations in the translation of patterns (see section); in particular, the case term should always be a variable, and no nested patterns are allowed.
The translation of lists and list constructors relies on the following *HsHOLCF*-defined datatype.

$$domain\ 'a\ llist\ =\ lNil \mid lCons\ (lazy\ lHd ::'\ a)\ (lazy\ lTl ::'\ a\ llist)$$

Under the given interpretation, a stream as well as an undefined list function take value $\bot$. This may be regarded as a semantical weakness.
Haskell allows for local definitions by means of *let* and *where* expressions. The *let* expressions where the left-hand side is a variable are translated to similar Isabelle ones; other *let* expressions (i.e. those containing patterns on the left hand-side) and the *where* expressions are not covered. The translation of terms (minus mutual recursion) may be summed up, essentially, as follows:

$$\begin{array}{ll}
\lceil x :: a \rceil & = x_t :: \lceil a \rceil \\
\lceil c \rceil & = c_t \\
\lceil \backslash x\ \rightarrow\ f \rceil & = LAM\ x_t.\ \lceil f \rceil \\
\lceil (a, b) \rceil & = (\lceil a \rceil, \lceil b \rceil) \\
\lceil f\ a_1 \ldots a_n \rceil & = FIX\ f_t.\ f_t\ \cdot\ \lceil a \rceil \ldots \cdot\ \lceil a_n \rceil \\
 & \quad \text{where } f :: \tau,\ f_t :: \lceil \tau \rceil \\
\lceil let\ x_1\ \ldots\ x_n\ in\ exp \rceil & = let\ \lceil x_1 \rceil\ \ldots\ \lceil x_n \rceil\ in\ \lceil exp \rceil
\end{array}$$

In HOLCF all recursive functions can be defined by fixpoint operator — a function that, given as argument the defining term abstracted of the recursive call name, returns the corresponding recursive function. Coding this directly turns out to be rather cumbersome, particularly in the case of mutually recursive functions, where tuples of defining terms and tupled abstraction would be needed. In contrast, the *fixrec* package allows us to handle fixpoint definitions in a way much more similar to ordinary Isabelle recursive definitions, providing with friendly syntax for mutual recursion, as well. Continuing the example,

$$fun1\ ::\ (a \rightarrow c)\ \rightarrow\ (b \rightarrow d)\ \rightarrow\ AType\ a\ b\ \rightarrow\ AType\ c\ d$$

$$\begin{array}{lll}
fun1\ f\ g\ k & = & case\ k\ of \\
\quad ABase\ x & \rightarrow & ABase\ (f\ x) \\
\quad AStep\ x\ y & \rightarrow & AStep\ (fun1\ f\ g\ x)\ (fun2\ f\ g\ y)
\end{array}$$

$$fun2\ ::\ (a \rightarrow c)\ \rightarrow\ (b \rightarrow d)\ \rightarrow\ BType\ a\ b\ \rightarrow\ BType\ c\ d$$

$$\begin{array}{lll}
fun2\ f\ g\ k & = & case\ k\ of \\
\quad BBase\ x & \rightarrow & BBase\ (g\ x) \\
\quad BStep\ x\ y & \rightarrow & BStep\ (fun2\ f\ g\ x)\ (fun1\ f\ g\ y)
\end{array}$$

this code translates to HOLCF as follows:

consts

$$\begin{array}{ll}
fun1 & :: ('a :: pcpo \rightarrow' c :: pcpo)\ \rightarrow\ ('b :: pcpo \rightarrow' d :: pcpo)\ \rightarrow \\
 & \quad ('a :: pcpo,' b :: pcpo)\ AType\ \rightarrow\ ('c :: pcpo,' d :: pcpo)\ AType \\
fun2 & :: ('a :: pcpo \rightarrow' c :: pcpo)\ \rightarrow\ ('b :: pcpo \rightarrow' d :: pcpo)\ \rightarrow \\
 & \quad ('a :: pcpo,' b :: pcpo)\ BType\ \rightarrow\ ('c :: pcpo,' d :: pcpo)\ BType
\end{array}$$

$$\begin{array}{rl}
fixrec\ fun1 = & (LAMf.\ LAMg.\ LAMk.\ case\ k\ of \\
 & ABase \cdot pX1\ \ =>\ ABase \cdot (f \cdot pX1) \mid \\
 & AStep \cdot pX1 \cdot pX2\ \ => \\
 & \ AStep \cdot (fun1 \cdot f \cdot g \cdot pX1) \cdot (fun2 \cdot f \cdot g \cdot pX2)) \\
and\ fun2 = & (LAMf.\ LAMg.\ LAMk.\ case\ k\ of \\
 & BBase \cdot pX1\ \ =>\ BBase \cdot (g \cdot pX1) \mid \\
 & BStep \cdot pX1 \cdot pX2\ \ => \\
 & \ BStep \cdot (fun2 \cdot f \cdot g \cdot pX1) \cdot (fun1 \cdot f \cdot g \cdot pX2))
\end{array}$$

The translations take care automatically of the fact that, in contrast with Haskell, Isabelle requires patterns in case expressions to follow the order of datatype declarations.
In the Programatica representation of Haskell, class information about type parameters is given by adding dictionary parameters to normal ones. These are eliminated by the translation, as class

information in Isabelle may be given by annotating arguments. In particular, each definition includes the type of the defined function complete with class annotation, in order to allow for overloading (a detail we omitted to show for the sake of readability in some of the examples here).

3.3 HOL: Type signature

The translation to HOL is semantically rather crude, it takes into account neither partiality nor laziness, and so, for soundness, it requires all functions in the program to be total ones.
An account of partiality could be obtained using the *option* type constructor to lift types, along lines similar to those followed in HOLCF with *lift*. Here instead we are just mapping Haskell types to corresponding, unlifted HOL ones — so for booleans and integers. Type variables are of class *type*. HOL function type, product and list are used to translate the corresponding Haskell constructors. The translation of types (minus mutual recursion) may be summed up as follows.

$$\begin{array}{ll}
\lceil a \rceil & = {}'a_t :: type \\
\lceil Bool \rceil & = bool \\
\lceil Integer \rceil & = int \\
\lceil a \to b \rceil & = \lceil a \rceil \Rightarrow \lceil b \rceil \\
\lceil (a,b) \rceil & = \lceil a \rceil * \lceil b \rceil \\
\lceil [a] \rceil & = \lceil a \rceil \ list \\
\lceil TyCons\ a_1 \ldots a_n \rceil & = \lceil a_1 \rceil \ldots \lceil a_n \rceil \ TyCons_t
\end{array}$$

Recursive and mutually recursive data-types declarations are translated to HOL as datatype declaration.

$$\begin{array}{rl}
datatype\ \ ('a,'b)\ BType = & BBase\ 'b \mid \\
& BStep\ (('a,'b)\ BType)\ (('a,'b)\ AType) \\
and\ \ ('a,'b)\ AType = & ABase\ 'a \mid \\
& AStep\ (('a,'b)\ AType)\ (('a,'b)\ BType)
\end{array}$$

Metalevel features are essentially shared with the HOLCF translation.

3.4 HOL: Sentences

Non-recursive definitions are treated in an analogous way to the translation into HOLCF. Standard lambda-abstraction (%) and function application are used here instead of continuous ones. Partial functions, and particularly case expressions with incomplete patterns, are not allowed. The translation of terms (minus recursion and case expressions) may be summed up as follows:

$$\begin{array}{ll}
\lceil x :: a \rceil & = x_t :: \lceil a \rceil \\
\lceil c \rceil & = c_t \\
\lceil \backslash x\ \to\ f \rceil & = \%\ x_t.\ \lceil f \rceil \\
\lceil (a,b) \rceil & = (\lceil a \rceil, \lceil b \rceil) \\
\lceil f\ a_1 \ldots a_n \rceil & = f_t\ \lceil a \rceil \ldots\ \lceil a_n \rceil \\
& \quad \text{where } f :: \tau,\ f_t :: \lceil \tau \rceil \\
\lceil let\ x_1\ \ldots\ x_n\ in\ exp \rceil & = let\ \lceil x_1 \rceil\ \ldots\ \lceil x_n \rceil\ in\ \lceil exp \rceil
\end{array}$$

Recursive definitions set HOL and HOLCF apart. In HOL a distinction is drawn, and syntactically highlighted, between primitive recursive functions (introduced by keyword *primrec*) and generic recursive ones (by keyword *recdef*). Termination is guaranteed for each of the formers, by the fact that recursion is based on the datatype structure of one of the parameters. In contrast, termination is not a trivial matter for the latters. A strictly decreasing measure must be provided, associated to the parameters of the defined function. This requires a degree of ingenuity that cannot be easily dealt with automatically. For this reason, the translation to HOL is restricted to primitive recursive functions.
Mutual recursion is allowed for under additional restrictions — more precisely: 1) all the functions involved are recursive in the first argument; 2) recursive arguments are of the same type in each function. The translation of mutually recursive functions $a \to b, \ldots a \to d$ introduces a new function $a \to (b * \ldots * d)$ recursively defined, for each case pattern, as the product of the values correspondingly taken by the original, non-recursively defined ones.

```
fun3  ::  AType a b → (a → a) → AType a b

                              fun3 k f =  case k of
                                          ABase a → ABase (f a)
                                          AStep a b → AStep (fun4 a) b

fun4  ::  AType a b → AType a b

                          fun4 k =  case k of
                                    AStep x y → AStep (fun3 x (\z → z)) y
                                    ABase x → ABase x
```

These functions, satisfying the restrictions, will translate to the following.

```
consts
        fun3 ::           ('a :: type,' b :: type)AType ⇒ ('a :: type ⇒' a :: type) ⇒
                          ('a :: type,' b :: type)AType
        fun4 ::           ('a :: type,' b :: type)AType ⇒ ('a :: type ⇒' a :: type) ⇒
                          ('a :: type,' b :: type)AType
        fun3_Xfun4_X ::   ('a :: type,' b :: type)AType ⇒
                          (('aXX1 :: type ⇒' aXX1 :: type) ⇒
                          ('aXX1 :: type,' bXX1 :: type)AType)∗
                          (('aXX2 :: type ⇒' aXX2 :: type) ⇒
                          ('aXX2 :: type,' bXX2 :: type)AType)
defs

      fun3_def : fun3  ==  %k f. fst ((fun3_Xfun4_X ::
                  ('a :: type,' b :: type)AType ⇒ (('a :: type ⇒' a :: type)
                 ⇒ ('a :: type,' b :: type)AType) ∗ ((unit ⇒ unit) ⇒
                 (unit, unit)AType)) k) f
      fun4_def : fun4  ==  %k f. snd ((fun3_Xfun4_X ::
                 ('a :: type,' b :: type)AType ⇒ ((unit ⇒ unit) ⇒ (unit, unit)AType)∗
                 (('a :: type ⇒' a :: type) ⇒ ('a :: type,' b :: type)AType)) k) f

primrec
          fun3_Xfun4_X (ABase pX1)  =   (%f. ABase (f pX1),
                                         %f. ABase pX1)
          fun3_Xfun4_X (AStep pX1 pX2) =
                      (%f. AStep (snd  (fun3_Xfun4_X pX1) f) pX2,
                      %f. AStep (fst   (fun3_Xfun4_X pX1) f) pX2)
```

Calls of the recursive functions in the non-recursive definitions are annotated with type where exceeding type variables are instantiated with the unit type, as required by Isabelle in order to avoid definitions from which inconsistencies are derivable.

3.5 Patterns

Support of patterns in definitions and case expressions is more restricted in Isabelle than in Haskell. Nested patterns are overall disallowed. In case expressions, the case term is required to be a variable. Both of these restrictions apply to our translations. A further Isabelle limitation concerning case expressions — sensitiveness to pattern order — is dealt with automatically. Similarly, wildcards — something unknown to Isabelle — are dealt with, as well as, in HOLCF, incomplete patterns. The exclusion of nested patterns complicate the translation of some specific ones — in fact, guarded expressions and list comprehension are not covered; their use should be avoided here, using conditional expressions and *map* instead.
Multiple function definitions using top level pattern matching are translated as definitions based on a single case expression; this is due to HOL more than to HOLCF. In fact, multiple definitions in Isabelle are only allowed with the syntax of recursive ones. However, in HOL primitive recursive definitions, patterns are allowed for only in one parameter. In order to translate definitions with more patterns as arguments, without resorting to tuples and to more complex syntax (*recdef* instead of *primrec*) we translate multiple definitions by top level pattern matching as definitions by case

construct.

```
ctl  ::  Bool → Bool → Bool → Bool
ctl False a False  =  a
ctl True a False  =  False
ctl False a True =  True
ctl True a True  =  a
```

This translates to HOL as the following.

```
consts ctl ::  bool ⇒ bool ⇒ bool ⇒ bool

defs
                          ctl_def : ctl  == %qX1.%qX2.%qX3. case qX1 of
                                       False ⇒ case qX3   of
                                                          False ⇒ qX2 |
                                                          True ⇒ True |
                                       True ⇒ case qX3    of
                                                          False ⇒ False |
                                                          True ⇒ qX2
```

This example does not go through with the translation to HOLCF, as booleans there are translated to values of *tr*, which is not a recursive datatype. The following gives an alternative that can be handled (a new datatype is used instead of booleans).

```
data Two  = Fx | Tx
ctlx ::  Two → Two → Two → Two
ctlx Fx a Fx  =  a
ctlx Tx a Fx  =  Fx
ctlx Fx a Tx  =  Tx
ctlx Tx a Tx  =  a
```

This translates to HOLCF as follows.

```
domain Two  =  Fx | Tx
consts ctlx ::  Two → Two → Two → Two

defs
                          ctlx_def : map5  == LAM qX1 qX2 qX3. case qX1 of
                                           Fx ⇒ case qX3        of
                                                                Fx ⇒ qX2 |
                                                                Tx ⇒ Tx |
                                           Tx ⇒ case qX3        of
                                                                Fx ⇒ Fx |
                                                                Tx ⇒ qX2
```

In case expressions as well as in top level pattern matching definitions, wildcards may be used — though not in nested position. Incomplete patterns are translated to HOLCF using $\perp$ as default value.

3.6 Classes

Haskell defined classes are translated to Isabelle as classes with empty axiomatization. Isabelle allows classes with no more than one type parameter, therefore our translations can support only them — it might be possible to handle more than one parameter using tuples, but this would surely involve considerable complications dealing with conditional instances.
Instance declarations are translated to corresponding ones in Isabelle. Isabelle instances in general require proofs that class axioms are satisfied by the types, but as long as there are no axioms the proofs are trivial and can be handled automatically. Function declarations associated with Haskell

classes are translated as independent function declarations with appropriate class annotation. Function definitions associated with instance declarations are translated as overloaded function definitions, relying on class annotation of the typed variables.

```
classClassA a where
abase ::  a → Bool
astep ::  a → Bool

instance (ClassA a,  ClassA b) ⇒ ClassA (AType a b) where

                                        abase x  = case x of
                                                   ABase u  →  True
                                                   _  →  False
```

This code translates to HOLCF as follows.

```
axclass ClassA < pcpo
instance AType :: (pcpo, ClassA, pcpo, ClassA) ClassA
by intro_classes

consts
                                    abase ::          'a :: {ClassA, pcpo} → tr
                                    astep ::          'a :: {ClassA, pcpo} → tr
                                    default_abase :: 'a :: {ClassA, pcpo} → tr
                                    default_astep :: 'a :: {ClassA, pcpo} → tr

defs
        AType_abase_def :
                       abase :: ('a :: {ClassA, pcpo}, 'b :: {ClassA, pcpo}) AType → tr
                          == LAMx. case x of
                             ABase · pX1 ⇒ TT |
                             AStep · pX2 · pX1 ⇒ FF
        AType_astep_def :
                       astep :: ('a :: {ClassA, pcpo}, 'b :: {ClassA, pcpo}) AType → tr
                             == default_astep
```

Similarly, it translates to HOL.

```
axclass ClassA < type

instance AType :: ({type, ClassA}, {type, ClassA}) ClassA
by intro_classes

consts
                                   abase ::          'a :: {ClassA, type} ⇒ bool
                                   astep ::          'a :: {ClassA, type} ⇒ bool
                                   default_abase :: 'a :: {ClassA, type} ⇒ bool
                                   default_astep :: 'a :: {ClassA, type} ⇒ bool

defs
        AType_abase_def :
                      abase :: ('a :: {ClassA, type}, 'b :: {ClassA, type}) AType ⇒ bool
                         == %x. case x of
                            ABase pX1 ⇒ True |
                            AStep pX2 pX1 ⇒ False
        AType_astep_def :
                      astep :: ('a :: {ClassA, type}, 'b :: {ClassA, type}) AType ⇒ bool
                         == default_astep
```

3.7 Equality

At the moment equality is the only covered built-in class. The axiomatizations provided, respectively, in *HsHOLCF* and *HsHOL* are based on the abstract definition of the equality and inequality functions [?]. In both theories, *Eq* is declared as a subclass of *type* — in *HsHOLCF* this is done in order to instantiate it with lifted types.

consts

$$\begin{array}{l} hEq :: \quad ('a :: Eq)lift \;\rightarrow\; 'a\ lift \;\rightarrow\; tr \\ hNEq :: ('a :: Eq)lift \;\rightarrow\; 'a\ lift \;\rightarrow\; tr \end{array}$$

axioms

$$\begin{array}{c} axEq : ALLx.(hEq \cdot p \cdot q \;=\; Def x) \;= \\ (hNEq \cdot p \cdot q \;=\; Def(\sim x)) \end{array}$$

The instantiation of equality (consequently, of inequality) for boolean (and similarly for integer) is obtained by lifting HOL equality so that $\perp$ is returned whenever one of the argument is undefined.

$tr\_hEq\_def :\; hEq \;==\; fliftbin\ (\%(a :: bool)\ b.\ a \;=\; b)$

In *HsHOL* the axiomatization reflects the restriction to terminating programs.

consts

$$\begin{array}{l} hEq :: \quad ('a :: Eq) \;\Rightarrow\; 'a \;\Rightarrow\; bool \\ hNEq :: ('a :: Eq) \;\Rightarrow\; 'a \;\Rightarrow\; bool \end{array}$$

axioms

$$axEq : hEq\ p\ q \;==\sim hNEq\ p\ q$$

The instantiation of *hEq* for boolean and integer is simply taken to be HOL equality.

3.8 Monads

Isabelle does not allow for classes of type constructors, hence a problem in representing monads. We could deal with this problem relying on an axiomatization of monads that allows for the representation of monadic types as an axiomatic class, as presented in [?]. Monadic types should be translated to newly defined types that satisfy monadic axioms. This would involve defining a theory morphism, as an instantiation of type variables in the theory of monads. We are planning to rely on AWE [?], an implementation of theory morphism on top of Isabelle base logic that may be used to extend HOL as well.

4 Conclusion

The following is a list of features that are covered by our translations.

- predefined types: boolean, integer.
- predefined type constructors: function, 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 operators.
- non-recursive functions, including conditional, *case* and *let* and expressions (with restriction related to use of patterns).
- use of incomplete patterns (in HOLCF) and of wildcards in case expressions.
- total primitive recursive functions (in HOL) and partial recursive ones (in HOLCF), including mutual ones (with restrictions in the HOL case).
- single-parameter class and instance declarations.

The shallow embedding approach makes it possible to take the most out of the automation currently available on Isabelle, especially in HOL. Further work should include extending the translation to cover the whole of the Haskell Prelude. We would also be interested in carrying out an extension to cover P-logic [?], a specification formalism for Haskell programs included in the Programatica toolset.