Translating Haskell into Isabelle

Paolo Torrini, Till Mossakowski, Christian Maeder

Informatik, Universitaet Bremen

Abstract. **We have defined and implemented as functions of Hets two partial translations of Haskell into Isabelle higher-order logics. Hets is a Haskell-based proof-management and program development system that allows for integration with other verification tools. Our application can translate simple Haskell programs to HOLCF and, under stronger restrictions, to HOL. Both translations use the static analysis of Programatica, are based on shallow embedding and rely on denotational semantics.**

Translations between programming and specification languages, as well as the corresponding integration of compilers, analyzers and theorem-provers, can provide useful support to the formal development and verification of programs. Verifying a program involves a formulation of the requirements in a logic, a translattion of the program to the logic, and a proof of the correctness statement. The translation must rest on a formal semantics of the programming language given in the logic. It should allow for representations of programs that make proofs as easy as possible. And, not the least on account of safety, it should be carried out automatically. 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 [Tho92].

In the following we present automated translations of Haskell programs into Isabelle higher-order logic based on a denotational semantics. Haskell is a strongly typed, purely functional language with lazy evaluation, polymorphic types extended with type constructor classes, and a syntax for side effects and pseudo-imperative code based on monadic operators [PJ03]. The translations are implemented as functions of Hets [MMLW03], an Haskell-based application designed to support heterogeneous specification and formal development of programs. Hets supplies with parsing, static analysis, proof management and interface to various language-specific tools. Hets relies for interactive proofs on an interface with Isabelle, a generic theorem-prover written in SML that includes the formalization of many logics [Pau94]. Hets uses Programatica [HHJK04] for the parsing and the static analysis of Haskell programs. Programatica, a Haskell-specific formal development system built at OGI, has a proof management on its own that includes a specification logic and translations to different proof tools, notably to Isabelle [HMW05], albeit following a different approach from ours (see section 2).

1 Isabelle

Isabelle-HOL (hereafter simply HOL) is the implementation in Isabelle of classical higher-order logic based on simply typed lambda calculus and extended with axiomatic type classes. It provides considerable support for reasoning about programming functions, both in terms of rich libraries and efficient automation. Since the late nineties, it has essentially superseded FOL (classical first-order logic) as an Isabelle standard. 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 [MNvOS99] is HOL conservatively extended with the logic of computable functions — a formalization of domain theory. In HOL, types — elements of class *type* — 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 an element of class *pcpo* (pointed complete partially ordered sets) i.e. 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 never depends on proofs.
The Isabelle formalization of HOLCF is based on axiomatic type classes [Wen05] so to deal with complete partial orders in a totally abstract way. Domain theory offers a good basis for the semantical analysis of programming languages; nevertheless, 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, using HOLCF type constructors to lift types only when this is needed.

2 Translations

Translations depend first of all on the expressiveness of the target logic. Those into FOL based on large-step operational semantics of Miranda [Tho94,Tho89,HT95] and Haskell [Tho92] struggle with higher-order features such as currying. The Haskell translation to the Agda implementation of Martin-Loef type theory in [ABB$^+$05] struggles with Haskell polymorphism. Higher-order logic helps overcome such obstacles. It also allows for higher-level approaches based on denotational semantics, such as proposed in [HMW05] to translate Haskell to HOLCF and in [LP04] to translate ML to HOL. Denotational semantics makes it possible to give representations of programs that are closer to their specification as well as to give proofs that are relatively more abstract and general.
A shallow embedding into a logical language is one that relies quite radically on its extra-logical features, possibly taking advantage of built-in

packages provided with the implementation of that language, particularly with respect to features such as types, classes, and recursion. On the contrary, a deep embedding relies on the logical definition of all the relevant notions. This may give a plus in semantic clarity and, if the logic is expressive enough, in generality as well. Taking advantage of built-in features, on the other hand, may help rely on standard proof methods to make theorem proving less tedious.

The translation of ML in [LP04] based on the definition of a class of types with bottom elements in HOL gives an example of the deep sort. The translation of Haskell to HOLCF proposed in [HMW05] relies on a generic formalization of domain theory and particularly on the *fixrec* package for recursive functions, created to provide a friendly syntax covering mutually recursive definitions as well. In order to capture type constructor classes — an important feature of the Haskell type system — a deep embedding approach is used there as well. Haskell types are translated to terms, relying on a domain-theoretic modelling of the type system at the object level. The practical drawback of this approach is that plenty of the automation built into Isabelle type checking either needs to be reimplemented or is lost.

The translations of Haskell to HOLCF and HOL that we are presenting here are based on denotational semantics and on a shallow embedding approach. The translation to HOLCF relies on the *fixrec* package. In contrast with [HMW05], we rely on the similarity between type systems, translating Haskell types to HOLCF types, in a comparatively direct way, and Haskell classes to Isabelle axiomatic classes. The modelling we are relying on is a simplified one, and we expect operational equivalence between Haskell programs and their translation to HOLCF up to the level of typeable output. The translation covers a significant part of the Prelude syntax, although it is still incomplete in one important respect — it does not include type constructor classes; we have plans for an extention that should address this aspect.

Conceptually, type classes in Isabelle are quite different from those in Haskell. Where the former ones are associated with sets of axioms, the latter ones come with sets of function declarations. Moreover, 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 the *do* notation. In alternative to the method proposed in [HMW05], we would like to get around the obstacle by relying on an extension of Isabelle based on theory morphism (see section 3.8). The AWE system [BJL06] is in fact an implementation of such an extension.

The translation to HOL, similar in respect of shallow approach, nevertheless is much more limited. An important restriction has to do with recursion — only primitive recursive functions are covered. This limitation appears relatively hard to overcome, given the way syntax for full recursion is defined for HOL. Moreover, we have to restrict to total functions. Operational equivalence for a larger fragment could be obtained using option types, but it is not pursued here.

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 [Mos06] and a general outlook in [Mos06]. 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 [Mos05]. 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 [Win93], 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:

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

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.

$$data\ AType\ a\ b\ =\ ABase\ a \mid AStep\ (AType\ a\ b)\ (BType\ a\ b)$$
$$data\ BType\ a\ b\ =\ BBase\ b \mid BStep\ (BType\ a\ b)\ (AType\ a\ b)$$

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 $\perp$ 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.

$$fliftbin :: ('a \Rightarrow\ 'b \Rightarrow\ 'c) \Rightarrow ('a\ lift \rightarrow\ 'b\ lift \rightarrow\ 'c\ lift)$$
$$fliftbin\ f\ ==\ LAM\ yx.\ (flift1\ (\%u.\ flift2\ (\%v.\ f\ v\ u))) \cdot x\ \cdot y$$

Booleans are translated to values of *tr* — *TT*, *FF* and $\perp$, 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 3.5); 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 $\perp$. 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

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

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

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.

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

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

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

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:

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

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 $a \rightarrow b, \ldots a \rightarrow d$ introduces a new function $a \rightarrow (b * \ldots * d)$ recursively defined, for each case pattern, as the product of the values correspondingly taken by the original, non-recursively defined ones.

$$
fun3 \ :: \ AType \ a \ b \rightarrow (a \rightarrow a) \rightarrow AType \ a \ b
$$

$$
\begin{array}{l}
fun3 \ k \ f = \ case \ k \ of \\
\qquad ABase \ a \rightarrow ABase \ (f \ a) \\
\qquad AStep \ a \ b \rightarrow AStep \ (fun4 \ a) \ b
\end{array}
$$

$$
fun4 \ :: \ AType \ a \ b \rightarrow AType \ a \ b
$$

$$
\begin{array}{l}
fun4 \ k = \ case \ k \ of \\
\qquad AStep \ x \ y \rightarrow AStep \ (fun3 \ x \ (\backslash z \rightarrow z)) \ y \\
\qquad ABase \ x \rightarrow ABase \ x
\end{array}
$$

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

consts

$$
\begin{array}{ll}
fun3 \ :: & ('a :: type, 'b :: type) AType \Rightarrow ('a :: type \Rightarrow' a :: type) \Rightarrow \\
 & ('a :: type, 'b :: type) AType \\
fun4 \ :: & ('a :: type, 'b :: type) AType \Rightarrow ('a :: type \Rightarrow' a :: type) \Rightarrow \\
 & ('a :: type, 'b :: type) AType \\
fun3_Xfun4_X :: & ('a :: type, 'b :: type) AType \Rightarrow \\
 & (('aXX1 :: type \Rightarrow' aXX1 :: type) \Rightarrow \\
 & ('aXX1 :: type, 'bXX1 :: type) AType) * \\
 & (('aXX2 :: type \Rightarrow' aXX2 :: type) \Rightarrow \\
 & ('aXX2 :: type, 'bXX2 :: type) AType)
\end{array}
$$

$defs$

$$
\begin{aligned}
fun3_def : fun3 \ == \ \%k\ f.\ fst\ ((fun3_Xfun4_X :: \\
('a :: type, 'b :: type)AType \Rightarrow (('a :: type \Rightarrow' a :: type) \\
\Rightarrow ('a :: type, 'b :: type)AType) * ((unit \Rightarrow unit) \Rightarrow \\
(unit, unit)AType))\ k)\ f \\
fun4_def : fun4 \ == \ \%k\ f.\ snd\ ((fun3_Xfun4_X :: \\
('a :: type, 'b :: type)AType \Rightarrow ((unit \Rightarrow unit) \Rightarrow (unit, unit)AType)* \\
(('a :: type \Rightarrow' a :: type) \Rightarrow ('a :: type, 'b :: type)AType))\ k)\ f
\end{aligned}
$$

$primrec$

$$
\begin{aligned}
fun3_Xfun4_X\ (ABase\ pX1) \ = \ & (\%f.\ ABase\ (f\ pX1), \\
& \%f.\ ABase\ pX1) \\
fun3_Xfun4_X\ (AStep\ pX1\ pX2) = & \\
& (\%f.\ AStep\ (snd\ \ (fun3_Xfun4_X\ pX1)\ f)\ pX2, \\
& \%f.\ AStep\ (fst\ \ \ (fun3_Xfun4_X\ pX1)\ f)\ pX2)
\end{aligned}
$$

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.

$$
\begin{aligned}
& ctl\ :: \ Bool \rightarrow Bool \rightarrow Bool \rightarrow Bool \\
& ctl\ False\ a\ False \ = \ a \\
& ctl\ True\ a\ False \ = \ False \\
& ctl\ False\ a\ True = \ True
\end{aligned}
$$

$ctl\ True\ a\ True\ =\ a$

This translates to HOL as the following.

$consts\ ctl :: \ bool \Rightarrow bool \Rightarrow bool \Rightarrow bool$

$defs$

$$\begin{array}{lll} ctl_def : ctl\ == & \%qX1.\%qX2.\%qX3. & case\ qX1\ of \\ & False \Rightarrow case\ qX3 & of \\ & & False \Rightarrow qX2\ | \\ & & True \Rightarrow True\ | \\ & True \Rightarrow case\ qX3 & of \\ & & False \Rightarrow False\ | \\ & & True \Rightarrow qX2 \end{array}$$

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).

$data\ Two\ = Fx\ |\ Tx$
$ctlx :: \ Two \rightarrow Two \rightarrow Two \rightarrow Two$
$ctlx\ Fx\ a\ Fx\ =\ a$
$ctlx\ Tx\ a\ Fx\ =\ Fx$
$ctlx\ Fx\ a\ Tx\ =\ Tx$
$ctlx\ Tx\ a\ Tx\ =\ a$

This translates to HOLCF as follows.

$domain\ Two\ =\ Fx\ |\ Tx$
$consts\ ctlx :: \ Two \rightarrow Two \rightarrow Two \rightarrow Two$

$defs$

$$\begin{array}{lll} ctlx_def : map5\ == & LAM\ qX1\ qX2\ qX3. & case\ qX1\ of \\ & Fx \Rightarrow case\ qX3 & of \\ & & Fx \Rightarrow qX2\ | \\ & & Tx \Rightarrow Tx\ | \\ & Tx \Rightarrow case\ qX3 & of \\ & & Fx \Rightarrow Fx\ | \\ & & Tx \Rightarrow qX2 \end{array}$$

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.

$$
\begin{array}{l}
class ClassA\ a\ where \\
abase :: \ a \rightarrow Bool \\
astep :: \ a \rightarrow Bool
\end{array}
$$

$$
\begin{array}{l}
instance\ (ClassA\ a,\ ClassA\ b) \Rightarrow ClassA\ (AType\ a\ b)\ where \\
\qquad\qquad abase\ x\ = case\ x\ of \\
\qquad\qquad\qquad\qquad ABase\ u\ \rightarrow\ True \\
\qquad\qquad\qquad\qquad _\ \rightarrow\ False
\end{array}
$$

This code translates to HOLCF as follows.

$$
\begin{array}{l}
axclass\ ClassA < pcpo \\
instance\ AType :: (pcpo, ClassA, pcpo, ClassA)\ ClassA \\
by\ intro_classes
\end{array}
$$

$$
\begin{array}{l}
consts \\
\qquad\qquad abase :: \qquad\quad 'a :: \{ClassA, pcpo\} \rightarrow tr \\
\qquad\qquad astep :: \qquad\quad 'a :: \{ClassA, pcpo\} \rightarrow tr \\
\qquad\qquad default_abase :: 'a :: \{ClassA, pcpo\} \rightarrow tr \\
\qquad\qquad default_astep :: 'a :: \{ClassA, pcpo\} \rightarrow tr
\end{array}
$$

$$
\begin{array}{l}
defs \\
\\
AType_abase_def : \\
\qquad\qquad abase :: ('a :: \{ClassA, pcpo\}, 'b :: \{ClassA, pcpo\})\ AType \rightarrow tr \\
\qquad\qquad\quad == LAMx.\ case\ x\ of \\
\qquad\qquad\qquad\quad ABase \cdot pX1 \Rightarrow TT\ | \\
\qquad\qquad\qquad\quad AStep \cdot pX2 \cdot pX1 \Rightarrow FF \\
AType_astep_def : \\
\qquad\qquad astep :: ('a :: \{ClassA, pcpo\}, 'b :: \{ClassA, pcpo\})\ AType \rightarrow tr \\
\qquad\qquad\qquad == default_astep
\end{array}
$$

Similarly, it translates to HOL.

$$
\begin{array}{l}
axclass\ ClassA < type \\
\\
instance\ AType :: (\{type, ClassA\}, \{type, ClassA\})\ ClassA \\
by\ intro_classes
\end{array}
$$

consts

$$\begin{array}{ll} abase :: & 'a :: \{ClassA, type\} \Rightarrow bool \\ astep :: & 'a :: \{ClassA, type\} \Rightarrow bool \\ default_abase :: & 'a :: \{ClassA, type\} \Rightarrow bool \\ default_astep :: & 'a :: \{ClassA, type\} \Rightarrow bool \end{array}$$

defs

$$\begin{array}{l} AType_abase_def : \\ \quad abase :: ('a :: \{ClassA, type\}, 'b :: \{ClassA, type\})\ AType \Rightarrow bool \\ \quad\quad == \%x.\ case\ x\ of \\ \quad\quad\quad ABase\ pX1 \Rightarrow True\ | \\ \quad\quad\quad AStep\ pX2\ pX1 \Rightarrow False \\ AType_astep_def : \\ \quad astep :: ('a :: \{ClassA, type\}, 'b :: \{ClassA, type\})\ AType \Rightarrow bool \\ \quad\quad == default_astep \end{array}$$

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 [PJ03]. 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

$$\begin{array}{ll} hEq :: & ('a :: Eq)lift \ \rightarrow\ 'a\ lift \ \rightarrow\ tr \\ hNEq :: & ('a :: Eq)lift \ \rightarrow\ 'a\ lift \ \rightarrow\ tr \end{array}$$

axioms

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

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

$$\begin{array}{ll} hEq :: & ('a :: Eq) \ \Rightarrow\ 'a \ \Rightarrow\ bool \\ hNEq :: & ('a :: Eq) \ \Rightarrow\ 'a \ \Rightarrow\ bool \end{array}$$

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 [Lue05]. 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 [BJL06], 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 [Kie02], a specification formalism for Haskell programs included in the Programatica toolset.

References

[ABB+05] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying Haskell programs using constructive type theory. In *ACM-SIGPLAN 05*, 2005.

[BJL06] M. Bortin, E. B. Johnsen, and C. Lueth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 2006.

[HHJK04] T. Hallgren, J. Hook, M. P. Jones, and D. Kieburtz. An overview of the Programatica toolset. In *HCSS04*, 2004.

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

[HT95] S. Hill and S. Thompson. Miranda in Isabelle. In *Proceedings of the first Isabelle users workshop*, number 397 in Technical Report, pages 122–135. University of Cambridge Computer Laboratory, 1995.

[Kie02] R. Kieburtz. P-logic: property verification for haskell programs. Technical report, OGI, 2002.

[LP04] J. Longley and R. Pollack. Reasoning about CBV programs in Isabelle-HOL. In *TPHOL 04*, number 3223 in LNCS, pages 201–216. Springer, 2004.

[Lue05] C. Lueth. Modular modelling with monads. In *Methods of Category Theory in Software Engineering*. Technische Universitaet Dresden, 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.

[Mos05] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set, Habilitation Thesis, 2005.

[Mos06] T. Mossakowski. Hets user guide. Tutorial, Universitaet Bremen, 2006.

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

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[Tho89] S. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1, 1989.

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

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

[Wen05] M. Wenzel. Using axiomatic type classes in Isabelle. Tutorial, TU Muenchen, 2005.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.