Translating Haskell into Isabelle

Paolo Torrini, Till Mossakowski, Christian Maeder

Informatik, Universitaet Bremen

Abstract. **Two partial translations of Haskell into Isabelle higher-order logics have been implemented as functions of Hets, a Haskell-based system for proof-management and program development 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 and are based on a shallow embedding of denotations.**

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. This task involves translating programs to a logic in which requirements can be expressed, in order to prove the corresponding correctness statements. Program translation should rest on a formal semantics of the programming language allowing for proofs that are as easy as possible. In fact, it has long been argued that functional languages, based on notions closer to general, mathematical ones, can make the task of proving assertions on them easier, owing to the relative clarity and simplicity of their semantics [Tho92].

In the following we are presenting automated translations of Haskell programs into Isabelle higher-order logics that can be justified in terms of 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 and proof management, as well as with interfaces to various language-specific tools. As far as interactive proofs are concerned, it relies on an interface with Isabelle, a generic theorem-prover written in SML that includes the formalization of several useful logics [Pau94]. Moreover, Hets relies on Programatica [HHJK04] for the parsing and the static analysis of Haskell programs. Programatica (built at OGI) is another Haskell-specific system for formal development and it has a proof management on its own, including 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 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 rich libraries and efficient automation. Since the late nineties, it has essentially superseded FOL (classical first-order logic) in standard use. 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.
On the other hand, 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 stage of definition — in fact, a specific measure has to be given for the function to be proved monotonic. In contrast, HOLCF 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. In particular, the Isabelle formalization of HOLCF is based on axiomatic type classes [Wen05], making it possible to deal with complete partial orders in quite an abstract way.
Domain theory offers a good basis for the semantical analysis of programming languages. 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 does not depend on proofs. Nevertheless, proving theorems may turn out to be comparatively hard. After being spared the need to discharge proof obligations at the stage of giving definitions, one has to bear with assumptions over the continuity of functions while actually carrying out the proofs. 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 necessary.

2 Translations

Translations can depend on the expressiveness of the target logic. As examples, the translations into FOL of Miranda [Tho94,Tho89,HT95] and Haskell [Tho92], both based on large-step operational semantics, appear to struggle with higher-order features such as currying. The translation of Haskell to the Agda implementation of Martin-Loef type theory in [ABB+05] seems to struggle with Haskell polymorphism. Higher-order logic may in fact quite help overcome such obstacles. It also allows for higher-level approaches based on denotational semantics, such as already proposed in [HMW05] for a translation of Haskell into HOLCF, and in [LP04] for a translation of ML into HOL. By using denotational semantics, one may hope to give representations of programs that are closer to

their specification, and to give proofs that are relatively more abstract and general.
A shallow embedding into a logical language is one that relies heavily on its extra-logical features, possibly taking advantage of built-in packages provided with the implementation of that language, particularly with respect to types and recursion. On the contrary, a deep embedding relies on the logical definition of all the relevant notions. This may give a plus in semantical clarity and possibly, provided the logic is expressive enough, in generality as well. Taking advantage of built-in features, on the other hand, may help make theorem proving less specific and 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 also mutually recursive definitions. However, 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 it leads to plenty of the automation built into Isabelle type checking needing to be reimplemented.
In contrast with [HMW05], both the translations of Haskell to HOLCF and HOL presented here are based on a shallow embedding approach. In the case of HOLCF we use as well the *fixrec* package. Moreover, we tend to rely as much as possible on similarities between type systems, translating Haskell types to HOLCF types in a comparatively direct way. Haskell classes are translated to Isabelle axiomatic classes. Since we are trying to keep things as simple as possible, the modelling we rely on is not very deep either: our claim is that the equivalence between Haskell programs and their translation to HOLCF can be justified up to the level of typeable output. This translation covers a significant part of the syntax used in the Prelude, however there are several limitations related to built-in types, pattern-matching, local definitions, import and export. In particular, it does not cover type constructor classes yet, although we have plans for an extention that should address this aspect.
The translation to HOL, similar in matter of shallow approach, is nevertheless much more limited. First, it only covers primitive recursive functions. This limitation appears relatively hard to overcome, given the way syntax for full recursion is defined in HOL. Moreover, we have to restrict to total functions. Operational equivalence for a larger fragment could be obtained using option types, but this is not pursued here.

3 Translations in Hets

Information about Hets may be found in [Mos06] and [Mos06]. The Haskell-to-Isabelle translation requires GHC, Isabelle 2006 and Programatica. The command to run the application is
hets -v[1–5] -t Haskell2Isabelle[HOLCF — HOL] -o thy out in.hs

where arguments set options for verbosity, the logic, extension and name of the output file, the last one being the input — a Haskell program given as a GHC file (*in.hs*). This gets analyzed and translated, the result of a successful run being an Isabelle theory file (*out.thy*) in the given logic.
The internal representation of Haskell in Hets (modules *Logic_Haskell* and *HatAna*) is the same as in Programatica, whereas the internal representation of Isabelle (modules *Logic_Isabelle* and *IsaSign*) is essentially a Haskell reworking of the ML definition of Isabelle's own base logic, extended in order to allow for a straightforward representation of HOL and HOLCF.
Haskell programs and Isabelle theories are internally represented as Hets theories — each of them formed by a signature and a set of sentences, according to the theoretical framework described in [Mos05]. Each translation, defined in module *Haskell2IsabelleHOLCF* as composition of a signature translation with a translation of all sentences, is essentially a morphism from theories in the internal representation of the source language to those in the representation of the target language. The module *IsaPrint* contains functions for the pretty-printing of Isabelle theories.
Each translation relies on an Isabelle theory, respectively *HsHOLCF*, extending HOLCF, and *HsHOL*, extending HOL, which contain some useful notions — notably definitions of lifting functions and an axiomatisation for Haskell equality.

4 Naming conventions

Names of types as well as of terms are translated by a renaming function that preserves them, up to avoidance of clashes with Isabelle keywords. We also need to reserve a few names, as follows.
1) Names for type variables, in the translation to HOL: *'vX*; any string terminating with *'XXn* where n is an integer.
2) Names for term constants, in the translation to HOL: strings obtained by joining together names of defined functions, using *_X* as end sequence and separator.
3) Names for term variables, in both translations: *pXn*, *qXn*, with n integer.
4) Names for type destructors, in the translation to HOLCF: *C_n*, where C is a data constructor name and n is an integer.

4.1 Types

Our type translation are shallow and based on relative similarity of type systems. Isabelle, as well as Haskell, is based on simple types with polymorphism (which means, essentially, type variables, function and product types). They both have built-in types for Boolean values and integers, and a type constructor for lists. Both allows for sums, recursive and mutually recursive types in the form of datatype declarations. Finally, they both have a class mechanism — although not quite the same.
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. However, the idea that underlies the translation is rather a simplifying one: although raising an exception is different from running forever, and both are different from stopping short of evaluation, still, from the point of view of the printable output of ground types, these behaviours are similar and can be treated semantically as such, i.e. using one and the same bottom element. So, essentially, we are following the main lines of the "crude" denotational semantics for lazy evaluation in [Win93], pp. 216–217.

Haskell type variables are translated to HOLCF ones, of class *pcpo*. Each type in Isabelle has a sort, defined by the set of the classes of which it is a member. Each built-in type is translated to the lifting of its corresponding HOL type. The translation covers properly only Haskell Booleans and unbounded integers, respectively associated to HOL Booleans and integers. Bound integers and floating point numbers would require low-level modelling, and have not been covered. Bounded integers are simply treated as unbounded.

The HOLCF type constructor *lift* is used to lift HOL types to flat domains. In the case of Booleans, HOLCF provides with type *tr*, defined as $bool\ lift$. In the case of integers, we define *dInt* as $int\ lift$ in *HsHOLCF*. The types of Haskell functions and product are translated, respectively, to HOLCF function spaces and lazy product — i.e. such that $\perp = (\perp * \perp) \neq (\perp *' a) \neq ('a * \perp)$, consistently with lazy evaluation. Type constructors are translated to corresponding HOLCF ones (notably, parameters precede type constructors in Isabelle syntax). In particular, lists are translated to the domain *llist* defined in *HsHOLCF*.

Type translation to HOLCF, apart from mutual dependencies, may be summed up as follows (where t is a renaming function):

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

The translation to HOL is more crude. It takes into account neither partiality nor laziness; therefore, we need to require that all functions in the program are total ones. An account of partiality could be obtained, but here it is not, using the *option* type constructor to lift types, along lines similar to those followed in HOLCF with *lift*.

Haskell types are mapped into corresponding, unlifted HOL ones — thus so for Booleans and integers. All variables are of class *type*. HOL function type, product and list are used to translate the corresponding Haskell constructors. Type translation to HOL, apart from mutual dependencies, may be summed up as follows.

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

Under each translation, function declarations are associated, by using the keyword *consts*, to the corresponding ones in HOLCF and HOL, respectively. Datatype declarations are associated to the corresponding ones, as well, but this involve some difference in the two cases. In HOLCF datatype declarations define types of class *pcpo* by the keyword *domain* (hence we may call them *domain declarations*). In HOL they define types of class *type* by the keyword *datatype*. Notably, in contrast with Haskell and HOL, HOLCF datatype declarations require an explicit introduction of destructors; these are provided automatically according to the naming pattern in Section 4, point 4. Apart from this aspect, the meta-level features of the the two type translations are essentially similar.
Translation of mutually recursive datatypes, as the one shown in the following example, relies on specific Isabelle syntax (using the keyword *and*).

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

Translation to HOLCF gives the following.

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

Translation to HOL gives the following.

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

In contrast with Haskell, order of declarations matters in Isabelle, where they should always be listed according to their dependency. Both translations take care of this aspect automatically.

4.2 HOLCF: Sentences

Essentially, each function definition is translated to a corresponding ones. Non-recursive definitions are translated to standard Isabelle definitions (introduced by the keyword *defs*), whereas the translation of recursive

definitions relies on the *fixrec* package. Lambda abstraction is translated as continuous abstraction (LAM), function application as continuous application (the *dot* operator). These notions coincide with the corresponding ones in HOL, i.e. with lambda abstraction ($\%$) and standard function application, whenever all arguments are continuous.
Terms of built-in type (Boolean and integer) are translated to lifted HOL values, using the HOLCF-defined lifting function *Def*. The bottom element $\perp$ is used for all the undefined terms. We use the following operator, defined in *HsHOLCF*, to map binary arithmetic functions to lifted functions over lifted integers.

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

Boolean values are translated to values of *tr* — i.e. TT, FF and $\perp$. Boolean connectives are translated to the corresponding HOLCF lifted operators. HOLCF-defined *If then else fi* and *case* syntax are used to translate conditional and case expressions, respectively. There are some restrictions, however, on the latter, due to limitations in the translation of patterns (see Section 4.4); in particular, the case term should always be a variable, and no nested patterns are allowed.
The translation of lists relies on the following domain, defined in *HsHOLCF*.

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

Under the given interpretation, an infinite list as well as an undefined one evaluate to $\perp$. However, the value of finite prefixes, and particularly the printable output associated to them, will be the expected one, since the model of evaluation is lazy.
Haskell allows for local definitions by means of *let* and *where* expressions. Those *let* expressions in which the left-hand side is a variable are translated to similar Isabelle ones; neither other *let* expressions (i.e. those containing patterns on the left hand-side) nor *where* expressions are 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 quite more similar to ordinary Isabelle recursive definitions, providing also with friendly syntax for mutual recursion.

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

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

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

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

As an example, the above code translates to HOLCF as follows.

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

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

The translation take care automatically of the fact that, in contrast with Haskell, Isabelle requires patterns in case expressions to follow the order of datatype declarations.

4.3 HOL: Sentences

Non-recursive definitions are treated in an analogous way as in the translation to 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 one has to pay attention to the distinction between primitive recursive functions (introduced by the keyword *primrec*) and generic recursive ones (keyword *recdef*). Termination is guaranteed for each of the former, 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 latter. A strictly decreasing measure needs to be provided, in association with 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 some 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.

As an example, the translation of mutually recursive functions of type $a \to b, \ldots a \to d$, respectively, introduces a new function of type $a \to (b * \ldots * d)$ which is recursively defined, for each case pattern, as the product of the values correspondingly taken by the original ones. The following is a concrete example.

$$fun3 \;::\; AType\ a\ b \to (a \to a) \to AType\ a\ b$$

$$\begin{aligned} fun3\ k\ f = \ & case\ k\ of \\ & ABase\ a \to ABase\ (f\ a) \\ & AStep\ a\ b \to AStep\ (fun4\ a)\ b \end{aligned}$$

$$fun4 \;::\; AType\ a\ b \to AType\ a\ b$$

$$\begin{aligned} fun4\ k = \ & case\ k\ of \\ & AStep\ x\ y \to AStep\ (fun3\ x\ (\backslash z \to z))\ y \\ & ABase\ x \to ABase\ x \end{aligned}$$

The translation to HOL of these two functions gives the following.

consts

$$\begin{array}{ll} fun3\ :: & ('a :: type,'b :: type)\ AType \Rightarrow ('a \Rightarrow' a) \Rightarrow \\ & ('a,'b)\ AType \\ fun4\ :: & ('a :: type,'b :: type)\ AType \Rightarrow ('a \Rightarrow' a) \Rightarrow \\ & ('a,'b)\ AType \\ fun3\_Xfun4\_X :: & ('a :: type,'b :: type)\ AType \Rightarrow \\ & (('aXX1 :: type \Rightarrow' aXX1 :: type) \Rightarrow \\ & ('aXX1,'bXX1)\ AType) * \\ & (('aXX2 :: type \Rightarrow' aXX2 :: type) \Rightarrow \\ & ('aXX2,'bXX2)\ AType) \end{array}$$

$defs$

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

$primrec$

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

One may note that the type of the recursive function, for each of its call in the body of non-recursive definitions, is given by instantiations where the Isabelle unit type is replaced for each type variable which is not occurring on the right hand-side, i.e. for each variable which is not constrained by the type of the defined function. This is required by Isabelle, in order to avoid definitions from which inconsistencies could be derivable. Other meta-level features are essentially common to both translation.

4.4 Patterns

Multiple function definitions using top level pattern matching are translated as definitions based on a single case expression. This syntactical choice has more to do with HOL than with HOLCF. In fact, multiple definitions in Isabelle are only allowed with the syntax of recursive ones. In primitive recursive definitions, HOL allows for patterns only in one parameter. Therefore, in order to translate definitions having patterns in more than one, before resorting to a more complex syntax (with tuples and *recdef* instead of *primrec*), it turns out comparatively easier to internally deal with multiple definitions as with case expressions. An example follows.

$ctl\ ::\ Bool \rightarrow Bool \rightarrow Bool \rightarrow Bool$
$ctl\ False\ a\ False\ =\ a$
$ctl\ True\ a\ False\ =\ False$
$ctl\ False\ a\ True =\ True$
$ctl\ True\ a\ True\ =\ a$

Translation to HOL gives the following.

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

$defs$

$$
\begin{array}{rlll}
ctl\_def : ctl & == \%qX1\ qX2\ qX3. & case\ qX1\ of & \\
& False \Rightarrow case\ qX3 & of & \\
& & False \Rightarrow qX2 & \\
& & \mid True \Rightarrow True & \\
& \mid True \Rightarrow case\ qX3 & of & \\
& & \mid False \Rightarrow False & \\
& & True \Rightarrow qX2 &
\end{array}
$$

This example cannot be handled by the translation to HOLCF, owing to the mapping of Boolean values to type tr, which is not a recursive datatype. The following, where a defined datatype is used instead of Boolean, gives the closest alternative that can be dealt with.

$$
\begin{array}{l}
data\ TV\ \ = F \mid T \\
ctlx :: \ TV \rightarrow TV \rightarrow TV \rightarrow TV \\
ctlx\ F\ a\ F\ = \ a \\
ctlx\ T\ a\ F\ = \ F \\
ctlx\ F\ a\ T\ = \ T \\
ctlx\ T\ a\ T\ = \ a
\end{array}
$$

This translates to HOLCF as follows.

$$
\begin{array}{l}
domain\ TV\ = \ F \mid T \\
consts\ ctlx :: \ TV \rightarrow TV \rightarrow TV \rightarrow TV
\end{array}
$$

$defs$

$$
\begin{array}{rlll}
ctlx\_def : ctlx & == LAM\ qX1\ qX2\ qX3.\ case\ qX1\ of & & \\
& F \Rightarrow case\ qX3 & of & \\
& & F \Rightarrow qX2 & \\
& & \mid T \Rightarrow T & \\
& \mid T \Rightarrow case\ qX3 & of & \\
& & F \Rightarrow F & \\
& & \mid T \Rightarrow qX2 &
\end{array}
$$

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 — sensitiveness to the order of patterns in case expressions — is dealt with automatically. Similarly, wildcards, not available in Isabelle, are dealt with and may be used, in case expressions as well as in function definition, though not in nested position. The translation to HOLCF can also handle incomplete patters, also not allowed by Isabelle, in function definitions as well as in case expressions, by using $\perp$ as default value. As nested patterns are not covered, guarded expressions and list comprehension are neither; anyway these can be avoided easily enough, using conditional expressions and map instead.

4.5 Classes

Conceptually, type classes in Isabelle are quite different from those in Haskell. The former are associated with sets of axioms, whereas the latter come with sets of function declarations. Moreover, Isabelle allows only for classes with a single type parameter. Most importantly, Isabelle does not allow for type constructor classes. The last limitation is rather 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 4.7). The AWE system [BJL06] is in fact an implementation of such an extension.

Defined classes are translated to Isabelle as classes with empty axiomatization. Every class is declared as a subclass of *type* — also in the case of HOLCF, in order to allow for instantiation with lifted built-in types, as well. The translations stay simple and cover only classes with no more than one type parameter — one could probably do something more using tuples, but this would surely involve considerable complication dealing with conditional instantiations.

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 carried out automatically. Method declarations are translated to independent function declarations with appropriate class annotation on type variables. Method definitions associated with instance declarations are translated to overloaded function definitions, using type annotation. An example follows.

$class\ ClassA\ a\ where$
$abase :: \ a \rightarrow Bool$
$astep :: \ a \rightarrow Bool$

$instance\ (ClassA\ a,\ ClassA\ b) \Rightarrow ClassA\ (AType\ a\ b)\ where$

$$\begin{array}{ll} abase\ x & = case\ x\ of \\ & \quad ABase\ u\ \rightarrow\ True \\ & \quad \_\ \rightarrow\ False \end{array}$$

The translation of this code to HOLCF gives the following.

$axclass\ ClassA < type$
$instance\ AType :: (\{pcpo, ClassA\}, \{pcpo, ClassA\})\ ClassA$
$by\ intro\_classes$

$consts$

$$\begin{array}{ll} abase :: & 'a :: \{ClassA, pcpo\} \rightarrow tr \\ astep :: & 'a :: \{ClassA, pcpo\} \rightarrow tr \\ default\_abase :: & 'a :: \{ClassA, pcpo\} \rightarrow tr \\ default\_astep :: & 'a :: \{ClassA, pcpo\} \rightarrow tr \end{array}$$

$defs$

$$
\begin{aligned}
&ATType\_abase\_def: \ abase :: \\
&\quad ('a :: \{ClassA, pcpo\}, 'b :: \{ClassA, pcpo\}) \ AType \to tr \\
&\qquad == LAMx. \ case \ x \ of \\
&\qquad\quad ABase \cdot pX1 \Rightarrow TT \mid \\
&\qquad\quad AStep \cdot pX2 \cdot pX1 \Rightarrow FF \\
&AType\_astep\_def: \ astep :: \\
&\quad ('a :: \{ClassA, pcpo\}, 'b :: \{ClassA, pcpo\}) \ AType \to tr \\
&\qquad == default\_astep
\end{aligned}
$$

Translation to HOL, on the other hand, gives the following.

$axclass \ ClassA < type$

$instance \ AType :: (\{type, ClassA\}, \{type, ClassA\}) \ ClassA$
$by \ intro\_classes$

$consts$

$$
\begin{aligned}
&abase :: &&'a :: \{ClassA, type\} \Rightarrow bool \\
&astep :: &&'a :: \{ClassA, type\} \Rightarrow bool \\
&default\_abase :: &&'a :: \{ClassA, type\} \Rightarrow bool \\
&default\_astep :: &&'a :: \{ClassA, type\} \Rightarrow bool
\end{aligned}
$$

$defs$

$$
\begin{aligned}
&AType\_abase\_def: \ abase :: \\
&\quad ('a :: \{ClassA, type\}, 'b :: \{ClassA, type\}) \ AType \Rightarrow bool \\
&\qquad == \%x. \ case \ x \ of \\
&\qquad\quad ABase \ pX1 \Rightarrow True \\
&\qquad\quad \mid AStep \ pX2 \ pX1 \Rightarrow False \\
&AType\_astep\_def: \ astep :: \\
&\quad ('a :: \{ClassA, type\}, 'b :: \{ClassA, type\}) \ AType \Rightarrow bool \\
&\qquad == default\_astep
\end{aligned}
$$

The additional functions declared for default use in method definition are close mirrors of an internal feature of the Programatica representation. In the Programatica internal representation of Haskell, for each function, information about the class of type parameters is encoded by including in the internal representation of the function some extra arguments (*dictionary parameters*) on top of the original ones. This is particularly useful in the case of overloaded definitions. On the other hand, class information in Isabelle can be represented by direct annotation on the arguments. Therefore, the translation eliminates dictionary parameters and gives function definitions based on their external representation instead. However, for each definition, our translations gives a function explicitly annotated with its type, inclusive of class annotation, in order to allow for overloading (just for the sake of formatting, this type annotation has been delated in some of the examples shown).

4.6 Equality

Equality is the only built-in class which is covered by the two translations. Axiomatizations for the associated methods are provided in the base theories — *HsHOLCF* and *HsHOL*, respectively. Both axiomatizations are based on the abstract definition of equality and inequality given in [PJ03].

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

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

The definition of Boolean equality in *HsHOLCF* is obtained by lifting HOL equality, so that $\perp$ is returned whenever one of the argument is undefined.

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

All built-in methods for built-in types are defined in a similar way.
In *HsHOL* equality is axiomatized under the implicit assumption of restricting 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

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

Under that assumption, equality for built-in types can be identified with HOL equality.

4.7 Monads

Isabelle does not allow for classes of type constructors — hence the 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.

5 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, Boolean operators, 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 mutually recursive ones (with restrictions in the HOL case).
– single-parameter class and instance declarations.

The shallow embedding approach makes it possible to get most of the benefit out of the automation currently available on Isabelle. Further work might carry an extension to cover P-logic [Kie02], a specification formalism for Haskell programs that is 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.