Translating Haskell to Isabelle

Paolo Torrini, Verimag
Paolo.Torrini@imag.fr
Christoph Lueth, DFKI Lab Bremen
Christoph.Lueth@dfki.de
Christian Maeder, DFKI Lab Bremen
Christian.Maeder@dfki.de
Till Mossakowski, DFKI Lab Bremen
Till.Mossakowski@dfki.de

No Institute Given

Abstract. We present partial translations of Haskell programs to Isabelle that have been implemented as part of the Hets-Programatica system. The logics HOLCF and Isabelle-HOL are targets — under stronger restrictions in the latter case — to translations that are essentially based on a shallow embedding approach. The AWE package has been used to support a translation of monadic operators based on theory morphisms.

1 Introduction

Automating the translation from programming languages to the languagee of a generic prover may provide useful support for the formal development and the verification of programs. It has been argued that functional languages can make the task of proving assertions about programs written in them easier, owing to the relative simplicity of their semantics [Tho92,Tho94]. The idea of translating Haskell programs, came to us, more specifically, from an interest in the use of functional languages for the specification of reactive systems. 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]. Several languages based on Haskell have been proposed for application to robotics [PHH99,HCNP03]. In such languages, monadic constructors are extensively used to deal with side-effects. Isabelle is a generic theorem-prover implemented in SML supporting several logics — in particular, Isabelle-HOL and its extension to a theory of computable functions (HOLCF) [Pau94,MNvOS99].

We have implemented as functions of Hets translations of Haskell to Isabelle-HOL and HOLCF following an approach based on shallow embedding, mapping Haskell types to Isabelle ones, therefore taking full advantage of Isabelle built-in type-checking. Hets [Mos05b,Mos06,MML07] is an Haskell-based application designed to support heterogeneous specification and the formal development of programs. It has an interface with Isabelle, and relies on Programatica [HHJK04] for parsing and static analysis of Haskell programs. Programatica already includes a translation to HOLCF which, in contrast to ours, is based on an object-level modelling of the type system [HMW05].

Our translation to HOLCF covers at present Booleans, integers, basic constructors (function, product, list, *maybe*), equality, single-parameter type classes (with some limitations), *case* and *if* expressions, *let* expressions without patterns, mutually recursive data-types and functions. It relies on existing formalisations of lazy lists and *maybe*. It keeps into account partiality and laziness by following, for the main lines, the denotational semantics of lazy evaluation given in [Win93]. There are several limitations: *Predulde* syntax is covered only partially; list comprension, *where* expressions and *let* with patterns are not covered; further built-in types and type classes are not covered; imports are not allowed; constructor type classes are not covered at all — and so for monadic types beyond list and *maybe*. Of all these limitations, the only logically deep ones are those related to classes — all the other ones are just a matter of implementation.

The base translation to Isabelle-HOL is more limited, insofar as it covers only total primitive recursive functions. A better semantics with respect to partiality could be obtained by lifting the type of function values with *option*, but this has not been pursued here. Still, *option* has been used to translate *maybe*. On the other hand, laziness appears very hard to be captured with Isabelle-HOL. It also seems hard to overcome the limitation to primitive recursion. Other limitations are similar to those mentioned for the translation to HOLCF — with the notable exception of monads.

Isabelle does not allow for type constructor classes, therefore there is hardly a way shallow embedding of Haskell types may extend to cover them. This limitation is particularly acute with respect to monads and *do* notation. The problem is brilliantly avoided in [HMW05] by resorting to a deeper modelling of types. On the other hand, the main novelty in our work is to rely on theory morphisms and on their implementation for Isabelle in the package AWE [BJL06], in order to deal with special cases of monadic operator. This solution gives in general less expressiveness than the deeper approach — however, when we can get it to deal with cases of interest, it might make proofs easier. Currently Hets provides with an extension of the base translation to Isabelle-HOL which uses AWE and covers state monad types inclusive of *do* notation. Due to present limitations of AWE, this solution is available only for Isabelle-HOL at the moment, although in principle it could work for HOLCF as well.

Section 2 gives some background, section 3 introduces the system, section 4 gives the sublanguages of Haskell that can be translated, in section 5 we define the two translations, in section 6 we sketch a denotational semantics associated with the translation to HOLCF, in section 7 we show how translation of monads is carried out with AWE.

2 Logics and translations

Isabelle-HOL is the implementation in Isabelle of classical higher-order logic based on simply typed lambda calculus extended with axiomatic type classes. It provides support for reasoning about programming functions, both in terms of rich libraries and efficient automation. In this respect, it has essentially superseded Isabelle-FOL (classical first-order logic) as a standard. Isabelle-HOL has an implementation of recursive total functions based on Knaster-Tarski fixed-point theorem. HOLCF [MNvOS99] is Isabelle-HOL conservatively extended with the Logic of Computable Functions — a formalisation of domain theory.

In Isabelle-HOL types can be interpreted as sets (class *type*); functions are total and may not be computable. A non-primitive recursive function may require discharging proof obligations already at the stage of definition — in fact, a specific relation has to be given for a function to be proved total. In HOLCF each type can be interpreted as a pointed complete partially ordered set (class *pcpo*) i.e. a set with a partial order which is closed w.r.t. ω-chains and has a bottom. Isabelle formalisation, based on axiomatic type classes [Wen05], makes it possible to deal with complete partial orders in quite an abstract way. Functions are generally partial and computability can be expressed in terms of continuity. Recursion can be expressed in terms of least fixed-point operator, and so, in contrast with Isabelle-HOL, function definition does not depend on proofs. Nevertheless, proving theorems in HOLCF may turn out to be comparatively hard. After being spared the need to discharge proof obligations at the definition stage, one has to bear with assumptions on function continuity throughout the proofs. A standard strategy is then to define as much as possible in Isabelle-HOL, using HOLCF type constructors to lift types only when this is necessary.

Although there have been translations of functional languages to first-order systems — those to FOL of Miranda [Tho94,Tho89,HT95] and Haskell [Tho92], both based on large-step operational semantics; that of Haskell to Agda implementation of Martin-Loef type theory in [ABB$^+$05] — still, higher-order logic may be quite helpful in order to deal with features such as currying and polymorphism. Moreover, higher-order approaches may rely on denotational semantics — as for examples, [HMW05] translating Haskell to HOLCF, and [LP04] translating ML to HOL — allowing for program representation closer to specification as well as for proofs comparatively more abstract and general.

The translation of Haskell to HOLCF proposed in [HMW05] uses deep embedding to deal with types. Haskell types are translated to terms, relying on a domain-theoretic modelling of the type system at the object level, allowing explicitly for a clear semantics, and providing for an implementation that can capture most features, including type constructor classes. In contrast, we provide in the case of HOLCF with a translation that follows the lines of a denotational semantics under the assumption that type constructors and type application in Haskell can be mapped to corresponding constructors and built-in application in Isabelle without loss from the point of view of behavioural equivalence between programs — in particular, translating Haskell datatypes to Isabelle ones.

3 Translations in Hets

The Haskell-to-Isabelle translation in Hets requires GHC, Programatica, Isabelle and AWE. The application is run by a command that takes as arguments a target logic and an Haskell program, given as a GHC source file. The latter gets analysed and translated, the result of a successful run being an Isabelle theory file in the target logic.

The Hets internal representation of Haskell is similar to that of Programatica, whereas the internal representation of Isabelle is based on the ML definition of the Isabelle base logic, extended in order to allow for a simpler representation of Isabelle-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 [Mos05a]. Each translation, defined 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 theories in the representation of the target language. Each translation relies on a specific Isabelle theory included in the Hets distrubution — *HsHOLCF* for HOLCF and *HsHOL* for HOL, respectively, the latter of which uses AWE.

4 Sublanguage definitions

Programs can be regarded as theories in a language given by definitions of type, class, type schemas (types with a context where variables are sorted by class), patterns, terms, declarations (forming the signature — including function declaration, datatype declaration, type definition, as well as class and method declaration) and definitions (forming the theory body — including function definition, as well as instance and method definition).

4.1 HOLCF

In the following, we give a definition of the sub-language of Haskell H_c that is covered by the translation to HOLCF.

Types

$$\begin{array}{ll} \tau = () & \\ \quad Bool & \\ \quad Integer & \\ \quad v & \text{type variable} \\ \quad \tau_1 \rightarrow \tau_2 & \\ \quad (\tau_1, \tau_2) & \\ \quad [\tau] & \\ \quad Maybe\ \tau & \\ \quad T\ \tau_1\ \ldots \tau_n & \text{either datatype or defined type} \end{array}$$

Type classes

$$\begin{array}{l} K = Eq \text{ with default methods } ==,\ /= \\ \quad K \text{ defined type class} \end{array}$$

Contexts

$$\begin{array}{ll} ctx = \{K\ v\} \cup ctx & \\ \quad \{\} & \text{empty context} \end{array}$$

Type schemas

$$\begin{array}{l} \phi = ctx \Rightarrow \tau \\ \quad \text{where all variables in } ctx \text{ are in } \tau \end{array}$$

Simple patterns

$$\begin{array}{ll} sp = \_ & \text{wildcard} \\ \quad v & \text{variable of datatype} \\ \quad C\ \overline{v} & \text{case of datatype} \end{array}$$

Terms

$$
\begin{array}{lll}
t & = t \in \{True, False, \&\&, ||, not\} & \text{on Boolean} \\
& c \in \aleph & \text{Integer constant} \\
& t \in \{+, *, -, div, mod, negate, <, >\} & \text{on Integer} \\
& t \in \{:, head, tail\} & \text{on list} \\
& t \in \{==, \ /=\} & \text{on equality types} \\
& t \in \{Just, Nothing\} & \text{on } \textit{maybe} \text{ types} \\
& t \in \{fst, snd\} & \text{on pairs} \\
& (,) & \\
& x & \text{variable} \\
& f & \text{function symbol} \\
& C & \text{data constructor} \\
& if\ t\ then\ t_1\ else\ t_2 & \\
& case\ x\ of\ (sp_1 \to t_1;\ \ldots;\ sp_n \to t_n) & \\
& let\ (x_1 = t_1;\ \ldots;\ x_n = t_n)\ in\ t &
\end{array}
$$

Declarations

$$
\begin{array}{ll}
Decl & = type\ T\ \overline{v}\ = \ \tau \\
& data\ ctx\ \Rightarrow\ T\ \overline{v}\ =\ C_1\ \overline{x}_1\ |\ \ldots\ |\ C_k\ \overline{x}_k \\
& ctx\ \Rightarrow\ f\ ::\ \tau \\
& class\ K\ where\ (f_1 :: \tau_1; \ldots; f_n :: \tau_n) \\
& \quad \text{where } \tau_1, \tau_n \text{ have only one type variable}
\end{array}
$$

Definitions

$$
\begin{array}{ll}
Def & = f\ \overline{v}\ =\ t \\
& f\ \overline{v}_1\ sp_1\ \overline{w}_1\ =\ t_1;\ \ldots;\ f\ \overline{v}_n\ sp_n\ \overline{w}_n\ =\ t_n \\
& instance\ ctx \Rightarrow K\ \tau\ where\ (f_1 = t_1; \ldots;\ f_n = t_n) \\
& \quad \text{where } f_1, \ldots, f_n \text{ are methods of } K
\end{array}
$$

4.2 HOL

In contrast, the following gives a definition of the Haskell sub-language H_s that is covered by the translation to HOL.

Types, Type classes, Contexts, Type schemas, Simple patterns, Declarations
as before

Type constructor classes
$Monad$

Terms

$$
\begin{array}{lll}
t & = () & \\
& t \in \{True, False, \&\&, ||, not\} & \text{on Boolean} \\
& c \in \aleph & \text{Integer constant} \\
& t \in \{+, *, -, div, mod, negate, <, >\} & \text{on Integer} \\
& t \in \{:, head, tail\} & \text{on list} \\
& t \in \{==, \ /=\} & \text{on equality types} \\
& t \in \{Just, Nothing\} & \text{on Maybe} \\
& t \in \{return, bind\} & \text{on monadic types} \\
& t \in \{fst, snd\} & \text{on pairs} \\
& (,) & \\
& x & \text{variable} \\
& f & \text{function symbol} \\
& C & \text{data constructor} \\
& if\ t\ then\ t_1\ else\ t_2 & \\
& case\ x\ of\ (sp_1 \to t_1;\ \ldots;\ sp_n \to t_n) & \\
& \quad \text{with } sp_1, \ldots, sp_n \text{ complete match} & \\
& let\ (x_1 = t_1;\ \ldots;\ x_n = t_n)\ in\ t &
\end{array}
$$

Definitions

$$
\begin{array}{ll}
Def & = f\ \overline{v}\ =\ t \\
& \quad \text{where } f \text{ is totally defined and primitive recursive} \\
& f\ \overline{v}_1\ sp_1\ \overline{w}_1\ =\ t_1;\ \ldots;\ f\ \overline{v}_n\ sp_n\ \overline{w}_n\ =\ t_n \\
& \quad \text{where } f \text{ is totally defined and primitive recursive} \\
& f_1\ v_1 :: \tau\ \overline{w_1}\ =\ t_1;\ ldots;\ f_n\ v_n :: \tau\ \overline{w_n}\ =\ t_n \\
& \quad \text{where } f_1 :: \phi_1, \ldots, f_n :: \phi_n \text{ are totally defined, mutually} \\
& \quad \text{primitive recursive in the first argument, and forall} \\
& \quad 0 < i \leq n \text{ there exists type variable renaming } \sigma_i \text{ such} \\
& \quad \text{that } \tau_1 = \sigma_i(\tau_i) \text{ and all the variables in } \phi_i \text{ appear in } \tau_i \\
& instance\ ctx \Rightarrow K\ \tau\ where\ (f_1 = t_1; \ldots;\ f_n = t_n) \\
& \quad \text{where } f_1, \ldots, f_n \text{ are totally defined, primitive recursive} \\
& \quad \text{methods of } K
\end{array}
$$

5 Translation definitions

We can define recursively a translation from an Haskell program to an Isabelle theory along the line of theory morphisms [Mos05a]. Here we can simply take a translation $\omega\ ::\ L_1\ \rightarrow\ L_2$ to be partially defined as a set Ω of rules $r_{\in\Omega} :: L_1\ \rightarrow\ L_2$ and a renaming function t that preserves names, up to avoidance of name clashes (in our case, with Isabelle keywords), in the following sense: if α is a name (either a variable or a constant, for either a term or a type), then $\omega = t$; else, if there exists $r \in \Omega$ that matches α most closely, $\omega(\alpha)\ =\ r(\alpha)$; else undefined. In the following we assume ω to be total, we ignore t, we write α' for $\omega(\alpha)$ and we write rules in relational form, such as $\alpha\ \Longrightarrow\ r(\alpha)$.

5.1 HOLCF

The translation $\omega_c\ ::\ H_c\ \rightarrow\ HOLCF$ from programs in H_c to theories in HOLCF can be defined, semi-formally, with the following set of rules.

Types

$$
\begin{array}{ll}
a & \Longrightarrow\ 'a :: \{pcpo\} \\
() & \Longrightarrow\ unit\ lift \\
Bool & \Longrightarrow\ bool\ lift \\
Integer & \Longrightarrow\ int\ lift \\
\tau_1 \rightarrow \tau_2 & \Longrightarrow\ \tau_1'\ \rightarrow\ \tau_2' \\
(\tau_1, \tau_2) & \Longrightarrow\ (\tau_1' * \tau_2') \\
[\tau] & \Longrightarrow\ \tau'\ seq \\
Maybe\ \tau & \Longrightarrow\ \tau'\ maybe \\
T\ \tau_1\ \ldots\ \tau_n & \Longrightarrow\ \tau_1'\ \ldots\ \tau_n'\ T' \\
& \quad \text{with } T \text{ either datatype or defined type}
\end{array}
$$

Classes

$$
\begin{array}{ll}
Eq & \Longrightarrow\ Eq \\
K & \Longrightarrow\ K'
\end{array}
$$

Type schemas

$$
\begin{array}{ll}
(\{K\ v\}\ \cup\ ctx)\ \Rightarrow\ \tau & \Longrightarrow\ (ctx \Rightarrow \tau)'\ [(v' :: s)/(v' :: (K' \cup s))] \\
\{\}\ \Rightarrow\ \tau & \Longrightarrow\ \tau'
\end{array}
$$

Haskell type variables are translated to variables of class *pcpo*. Each type is associated to a sort in Isabelle, defined by the set of the classes of which it is member. Built-in types are translated to the lifting of its corresponding HOL type. The HOLCF type constructor *lift* is used to lift types to flat domains. The types of Haskell functions and product are translated, respectively, to HOLCF function spaces and lazy product — i.e. such that $\bot = (\bot * \bot) \neq (\bot *\ 'a) \neq ('a * \bot)$. Type constructors are translated to corresponding HOLCF ones (notably, parameters precede type constructors in Isabelle syntax). *Maybe* is translated to HOLCF-defined *maybe* (the disjoint union of the lifted unit type and the lifted domain parameter).

Terms

$$
\begin{array}{ll}
x :: \tau & \Longrightarrow\ x' :: \tau' \\
() & \Longrightarrow\ Def\ () \\
True & \Longrightarrow\ TT \\
False & \Longrightarrow\ FF \\
\&\& & \Longrightarrow\ trand \\
|| & \Longrightarrow\ tror \\
not & \Longrightarrow\ neg \\
c & \Longrightarrow\ Def\ c \\
t \in \{+, -, *, div, mod, <, >\} & \Longrightarrow\ fliftbin\ t \\
negate & \Longrightarrow\ flift2\ - \\
[] & \Longrightarrow\ nil \\
t : ts & \Longrightarrow\ t'\#\#ts' \\
head & \Longrightarrow\ HD \\
tail & \Longrightarrow\ TL \\
== & \Longrightarrow\ hEq \\
/= & \Longrightarrow\ hNEq \\
Just & \Longrightarrow\ return \\
Nothing & \Longrightarrow\ fail \\
C & \Longrightarrow\ C' \\
f & \Longrightarrow\ f' \\
\backslash\overline{x}\ \rightarrow\ t & \Longrightarrow\ LAM\ \overline{x}'.\ t' \\
t_1\ t_2 & \Longrightarrow\ t_1'\ \cdot\ t_2' \\
(t_1, t_2) & \Longrightarrow\ (t_1',\ t_2') \\
fst & \Longrightarrow\ cfst \\
snd & \Longrightarrow\ csnd \\
let\ (x_1 = t_1; & \\
\quad \ldots; & \\
\quad x_n = t_n) \quad in\ t & \Longrightarrow\ let\ (x_1' = t_1';\ \ldots;\ x_n' = t_n')\ in\ t' \\
if\ t\ then\ t_1\ else\ t_2 & \Longrightarrow\ If\ t'\ then\ t_1'\ else\ t_2'\ fi \\
case\ x\ of\ (p_1 \rightarrow t_1; & \\
\qquad \ldots; & \\
\qquad p_n \rightarrow t_n) & \Longrightarrow\ case\ x'\ p_1' \Rightarrow t_1' \mid \ldots \mid p_n' \Rightarrow t_n' \\
& \quad \text{if } p_1', \ldots, p_n' \neq \_ \text{ is a complete match} \\
& case\ x'\ p_1' \Rightarrow t_1' \mid \ldots \mid p_{n-1}' \Rightarrow t_{n-1}' \\
& \qquad\qquad \mid q_1 \Rightarrow t_n' \mid \ldots \mid q_k \Rightarrow t_n' \\
& \quad \text{if } p_n = \_, \text{ with } p_1', \ldots, p_{n-1}', q_1, \ldots, q_k \\
& \quad \text{complete match} \\
& case\ x'\ p_1' \Rightarrow t_1' \mid \ldots \mid p_n' \Rightarrow t_n' \\
& \qquad\qquad \mid q_1 \Rightarrow \bot \mid \ldots q_k \Rightarrow \bot \\
& \quad \text{with } p_1', \ldots, p_n', q_1, \ldots, q_k \text{ complete match}
\end{array}
$$

Terms of built-in type are translated using HOLCF-defined lifting function *Def*. The bottom element $\bot$ is used for undefined terms. HOLCF-defined $flift1 ::\ ('a \Rightarrow' b :: pcpo) \Rightarrow ('a\ lift \rightarrow' b)$ and $flift2 ::\ ('a \Rightarrow' b) \Rightarrow ('a\ lift \rightarrow' b\ lift)$ are used to lift operators, as well as the following, defined in *HsHOLCF*.

$$
\begin{array}{l}
fliftbin :: ('a \Rightarrow\ 'b \Rightarrow\ 'c) \Rightarrow ('a\ lift \rightarrow\ 'b\ lift \rightarrow\ 'c\ lift) \\
fliftbin\ f\ \ ==\ \ flift1\ (\lambda x.\ flift2\ (f\ x))
\end{array}
$$

Boolean values are translated to values of *bool lift* (*tr* in HOLCF) i.e. *TT*, *FF* and $\bot$, and Boolean connectives to the corresponding HOLCF operators. HOLCF-defined *If then else fi* and *case* syntax are used to translate conditional and case expressions, respectively. There are restrictions, however, on case epressions, due to limitations in the translation of patterns; in particular, the case term has to be a variable, and only simple patterns are allowed (no nested ones). On the other hand, Isabelle sensitiveness to the order of patterns in case expressions is dealt with. Multiple function definitions are translated as definitions based on case expressions. In function definitions as well as in case expressions, both wildcards — not available in Isabelle — and incomplete patterns — not allowed — are dealt with by elimination, $\perp$ being used as default value in the latters. Only let expressions without patterns on the left are dealt with; where expressions, guarded expressions and list comprehension are not covered.

Lists are translated to the domain *seq* defined in library IOA.

$$domain\ 'a\ seq\ =\ nil \mid \#\#\ (HD :: \ 'a)\ (lazy\ TL :: \ 'a\ seq)$$

Keyword *lazy* ensures that $x\ \#\#\ \perp\ \neq\ \perp$, allowing for partial sequences as well as for infinite ones [MNvOS99].

Declarations

$$\begin{array}{l}
class\ K\ where\ (Dec_1; \ldots ; Dec_n]) \implies class\ K'\ \subseteq\ pcpo;\ Dec'_1;\ \ldots;\ Dec'_n \\
f :: \phi \qquad\qquad\qquad\qquad \implies consts\ f'\ ::\ \phi' \\
type\ \tau\ =\ \tau_1 \qquad\qquad\quad \implies type\ \tau\ =\ \tau'_1 \\
(data\ \phi_1\ =\ C_{11}\ x_1 \ldots x_i \mid \ldots \mid C_{1p}\ y_1 \ldots y_j; \\
\ldots; \\
data\ \phi_n\ =\ C_{n1}\ w_1 \ldots w_h \mid \ldots \mid C_{1q}\ z_1 \ldots z_k) \implies \\
\quad domain\ \phi'_1\ =\ C'_{11}\ d_{111}\ x'_1\ \ldots\ d_{11i}\ x'_i \mid \ldots \mid C'_{1p}\ d_{1p1}\ y'_1\ \ldots\ d_{1pj}\ y'_j \\
\quad and\ \ldots \\
\quad and\ \phi'_n\ =\ C'_{n1}\ d_{n11}\ w'_1\ \ldots\ d_{n1h}\ w'_h \mid \ldots \mid C'_{nq}\ d_{nq1}\ z'_1\ \ldots\ d_{nqk}\ z'_k \\
\quad \text{where } \phi_1,\ \phi_n \text{ are mutually recursive datatype}
\end{array}$$

Definitions

$$\begin{array}{l}
f\ \overline{x}\ p_1\ \overline{x}_1\ =\ t_1;\ \ldots;\ f\ \overline{x}\ p_n\ \overline{x}_n\ =\ t_n \implies \\
\quad (f\ \overline{x}\ =\ case\ y\ of\ (p_1 \rightarrow (\backslash \overline{x}_1 \rightarrow\ t_1);\ \ldots;\ p_n(\rightarrow \backslash \overline{x}_n \rightarrow\ t_n)))' \\
f\ \overline{x}\ =\ t \qquad\qquad \implies defs\ f' :: \phi'\ ==\ LAM\ \overline{x}'.\ t' \\
\quad \text{with } f :: \phi \text{ not occurring in } t \\
(f_1\ \overline{v_1}\ =\ t_1;\ \ldots;\ f_n\ \overline{v_n}\ =\ t_n) \implies \\
\quad fixrec\ f'_1 :: \phi'_1\ =\ (LAM\ \overline{v_1}'.\ t'_1)\ and \\
\qquad \ldots \\
\qquad and\ f'_n :: \phi'_n\ =\ (LAM\ \overline{v_n}'.\ t'_n) \\
\quad \text{with } f_1 :: \phi_1, \ldots, f_n :: \phi_n \text{ mutually recursive} \\
instance\ ctx\ \Rightarrow\ K_T\ (T\ v_1\ \ldots\ v_n)\ where \\
\qquad\qquad (f_1 :: \tau_1 = t_1; \ldots;\ f_n :: \tau_n = t_n) \implies \\
\quad instance \\
\qquad \tau' ::\ K'_T\ (\{pcpo\}\ \cup\ \{K' : (K\ v_1) \in ctx\},\ \ldots, \\
\qquad\qquad\qquad \{pcpo\}\ \cup\ \{K' : (K\ v_n) \in ctx\}) \\
\qquad \text{with proof obligation;} \\
\quad defs\ f'_1 :: (ctx \Rightarrow \tau_1)' == t'_1;\ \ldots;\ f'_n :: (ctx \Rightarrow \tau_n)' == t'_n
\end{array}$$

Function declarations use Isabelle keyword *consts*. Datatype declarations in HOLCF are domain declarations and require explicitly destructors. Mutually recursive datatypes relies on specific Isabelle syntax (keyword *and*). Order of declarations is taken care of.

Non-recursive definitions are translated to standard definitions using Isabelle keyword *defs*. Recursive definitions rely on HOLCF package *fixrec* which provides nice syntax for fixed point definitions, including mutual recursion. Lambda abstraction is translated as continuous abstraction (LAM), function application as continuous application (the *dot* operator), equivalent to lambda abstraction (λ) and standard function application, respectively, when all arguments are continuous.

Classes in Isabelle and Haskell are built quite differently. In Haskell, a type class is associated to a set of function declarations, and it can be interpreted as the set of types where those functions are defined. In Isabelle, a type class has a single type parameter, it is associated to a set of axioms in a single type variable, and can be interpreted as the set of types that satisfy those axioms.

Not all the problems have been solved with respect to arities that may conflict in Isabelle, although they correspond to compatible Haskell instantiations. Moreover, Isabelle does neither allow for multi-parameter classes, nor for type constructor ones, therefore the same translation method cannot be applied to them.

Defined single-parameter classes are translated to HOLCF as subclasses of *pcpo* with empty axiomatization. Methods declarations associated with Haskell classes are translated to independent function declarations with appropriate class annotation on type variables. In Isabelle, each instance requires proofs that class axioms are satisfied by the instantiating type — anyway, as long as there are no axioms proofs are trivial and proof obligation may be automatically discharged. Method definitions associated with instance declarations are translated to independent function definitions, using type annotation and relying on Isabelle overloading.

In the internal representation of Haskell given by Programatica, function overloading is handled by means of dictionary parameters [Jon93]. This means that each function has additional parameters for the classes associated to its type variables. In fact, dictionary parameters are used to decide, for each instantiation of the function type variables, how to instantiate the methods called in the function body. On the other hand, overloading in Isabelle is obtained by adding explicitly type annotation to function definitions — dictionary parameters may thus be eliminated.

The translation of built-in classes may involve axioms — this is the case for equality. A HOLCF formalisation, based on the methods specification in [PJ03], has been given as follows in *HsHOLCF* (*neg* is lifted negation).

$consts$

$$heq :: \ 'a \ \rightarrow \ 'a \ \rightarrow \ tr$$
$$hneq :: 'a \ \rightarrow \ 'a \ \rightarrow \ tr$$

$axclass \ Eq \ < \ pcpo$
$eqAx : heq \cdot p \cdot q \ = \ neg \cdot (hneq \cdot p \cdot q)$

Functions *heq* and *hneq* can be defined, for each instantiating type, with the translation of equality and inequality, respectively. For each instance, a proof that the definitions satisfy *eqAx* needs to be given — the translation will simply print out *sorry* (a form of ellipsis in Isabelle). The definition of default methods for built-in types and the associated proofs can be found in *HsHOLCF*.

5.2 HOL

The translation $\omega_s \ :: \ H_s \ \rightarrow \ HOL$ from programs in H_s to theories in Isabelle-HOL extended with AWE can be defined with the following set of rules.

Types

$$\begin{array}{ll}
() & \Longrightarrow unit \\
a & \Longrightarrow 'a :: \{type\} \\
Bool & \Longrightarrow boolean \\
Integer & \Longrightarrow int \\
\tau_1 \rightarrow \tau_2 & \Longrightarrow \tau_1' \ \Rightarrow \ \tau_2' \\
(\tau_1, \tau_2) & \Longrightarrow (\tau_1' * \tau_2') \\
[\tau] & \Longrightarrow \tau' \ list \\
Maybe \ \tau & \Longrightarrow \tau' \ option \\
T \ \tau_1 \ \ldots \ \tau_n & \Longrightarrow \tau_1' \ \ldots \ \tau_n' \ T' \\
 & \quad \text{with } T \text{ either datatype or defined type}
\end{array}$$

Classes

$$\begin{array}{ll}
Eq & \Longrightarrow \ Eq \\
K & \Longrightarrow \ K'
\end{array}$$

Type schemas

$$\begin{array}{ll}
(\{K \ v\} \ \cup \ ctx) \ \Rightarrow \ \tau & \Longrightarrow \ (ctx \Rightarrow \tau)' \ [(v' :: s)/(v' :: (K' \cup s))] \\
\{\} \ \Rightarrow \ \tau & \Longrightarrow \ \tau'
\end{array}$$

Here we highlight the main differences the translation to HOLCF and this, semantically rather more approximative one to Isabelle-HOL (thereafter simply HOL). Function type, product and list are used to translate the corresponding Haskell constructors. Option types are used to translate *Maybe*. Haskell datatypes are translated to HOL datatypes. Type variables are of class *type*.

Standard lambda-abstraction (λ) and function application are used here, instead of continuous ones. Non-recursive definitions are treated in an analogous way as in the translation to HOLCF. However, partial functions and particularly case expressions with incomplete patterns are not allowed.

In HOL one has to pay attention to the distinction between *primitive recursive* functions (introduced by the keyword *primrec*) and generally recursive ones. Termination is guaranteed for each primitive recursive function by the fact that recursion is based on the datatype structure of one of the parameters. In contrast, termination is no trivial matter for recursion in general. A strictly decreasing measure needs to be association with the parameters. This cannot be dealt with automatically in general. Threferore here we restrict translation to primitive recursive functions.

Mutual primitive recursion is allowed for under additional restrictions — more precisely, given a set F of functions: 1) all the functions in F are recursive in the first argument; 2) all recursive arguments in F are of the same type modulo variable renaming; 3) each type variable occurring in the type of a function in F already occurs in the type of the first argument. The third conditions is a way to ensure that we do not end up with type variables which occurs on the right hand-side but not on the left hand-side of a definition. In fact, given mutually recursive functions $f_1, \ldots, f_n$ of type $A \to B_1, \ldots, A \to B_n$ after variable renaming, they are translated to projections of a new function of type $A \to (B_1 * \ldots * B_n)$ which is defined for cases of A with products of the corresponding values of $f_1, \ldots, f_n$. The expression $nth_n\ t$ used in the translation rule is simply an informal abbreviation for the HOL function, defined in terms of fst and snd, which extracts the n-th projection from a tuple no shorter than n.

Terms

$$
\begin{array}{lll}
x :: \tau & \Longrightarrow & x' :: \tau' \\
() & \Longrightarrow & () \\
True & \Longrightarrow & True \\
False & \Longrightarrow & False \\
\&\& & \Longrightarrow & \& \\
|| & \Longrightarrow & | \\
not & \Longrightarrow & Not \\
c & \Longrightarrow & c \\
t \in \{+,-,*,div,mod,<,>\} & \Longrightarrow & t \\
negate\ x & \Longrightarrow & -x \\
[] & \Longrightarrow & [] \\
t : ts & \Longrightarrow & t' \# ts' \\
head & \Longrightarrow & hd \\
tail & \Longrightarrow & tl \\
== & \Longrightarrow & hEq \\
/= & \Longrightarrow & hNEq \\
Just & \Longrightarrow & Some \\
Nothing & \Longrightarrow & None \\
return & \Longrightarrow & return \\
bind & \Longrightarrow & mbind \\
C & \Longrightarrow & C' \\
f & \Longrightarrow & f' \\
\backslash \overline{x}\ \rightarrow\ t & \Longrightarrow & \lambda\ \overline{x}'.\ t' \\
t_1\ t_2 & \Longrightarrow & t'_1\ t'_2 \\
(t_1, t_2) & \Longrightarrow & (t'_1,\ t'_2) \\
fst & \Longrightarrow & fst \\
snd & \Longrightarrow & snd \\
let\ (x_1 = t_1; & & \\
\quad \ldots; & & \\
\quad x_n = t_n) \quad in\ t & \Longrightarrow & let\ (x'_1 = t'_1;\ \ldots;\ x'_n = t'_n)\ in\ t' \\
if\ t\ then\ t_1\ else\ t_2 & \Longrightarrow & if\ t'\ then\ t'_1\ else\ t'_2 \\
case\ x\ of\ (p_1 \rightarrow t_1; & & \\
\qquad \ldots; & & \\
\qquad p_n \rightarrow t_n) & \Longrightarrow & case\ x'\ p'_1 \Rightarrow t'_1\ |\ \ldots\ |\ p'_n \Rightarrow t'_n \\
 & & \quad \text{if } p'_1, \ldots, p'_n \neq \_ \text{ is a complete match} \\
 & & case\ x'\ p'_1 \Rightarrow t'_1\ |\ \ldots\ |\ p'_{n-1} \Rightarrow t'_{n-1} \\
 & & \qquad |\ q_1 \Rightarrow t'_n\ |\ \ldots\ |\ q_k \Rightarrow t'_n \\
 & & \quad \text{if } p_n = \_, \text{ with } p'_1, \ldots, p'_{n-1}, q_1, \ldots, q_k \\
 & & \quad \text{complete match}
\end{array}
$$

Declarations

$$
\begin{array}{l}
class\ K\ where\ (Dec_1; \ldots; Dec_n]) \Longrightarrow class\ K' \subseteq type;\ Dec'_1;\ \ldots;\ Dec'_n \\
f :: \phi \qquad\qquad\qquad \Longrightarrow consts\ f'\ ::\ \phi' \\
type\ \tau\ =\ \tau_1 \qquad\qquad \Longrightarrow type\ \tau\ =\ \tau'_1 \\
(data\ \phi_1\ =\ C_{11}\ x_1 \ldots x_i\ |\ \ldots\ |\ C_{1p}\ y_1 \ldots y_j; \\
\ldots; \\
data\ \phi_n\ =\ C_{n1}\ w_1 \ldots w_h\ |\ \ldots\ |\ C_{1q}\ z_1 \ldots z_k) \Longrightarrow \\
\quad datatype\ \phi'_1\ =\ C'_{11}\ x'_1\ \ldots\ x'_i\ |\ \ldots\ |\ C'_{1p}\ y'_1\ \ldots\ y'_j \\
\quad and\ \ldots \\
\quad and\ \phi'_n\ =\ C'_{n1}\ w'_1\ \ldots\ w'_h\ |\ \ldots\ |\ C'_{nq}\ z'_1\ \ldots\ z'_k \\
\quad \text{where } \phi_1,\ \phi_n \text{ are mutually recursive datatype}
\end{array}
$$

Definitions

$$
\begin{array}{l}
f\ \overline{x}\ p_1\ \overline{x}_1\ =\ t_1;\ \ldots;\ f\ \overline{x}\ p_n\ \overline{x}_n\ =\ t_n\ \Longrightarrow\\
\quad (f\ \overline{x}\ =\ case\ y\ of\ (p_1 \to (\backslash \overline{x}_1 \to\ t_1);\ \ldots;\ p_n(\to \backslash \overline{x}_n \to\ t_n)))'\\
f\ \overline{x}\ =\ t \qquad\qquad \Longrightarrow\ defs\ f' :: \phi'\ ==\ \lambda\ \overline{x}'.\ t'\\
\quad \text{with } f :: \phi \text{ not occurring in } t\\
f_1\ \ y_1\ \overline{x_1}\ =\ t_1;\ \ldots;\ f_n\ y_n\ \overline{x_n}\ =\ t_n\ \Longrightarrow\\
\quad decl\ f_{new} :: (\sigma_1(ctx_1)\ \cup\ \ldots\ \cup\ \sigma_n(ctx_n)\ \Rightarrow\\
\qquad\qquad \sigma_1(\tau_{1a})\ \to\ (\sigma_1(\tau_1), \ldots, \sigma_n(\tau_n)))'\\
\quad primrec\ f_{new}\ sp_1\ =\ (\lambda\ \overline{x_1}'.\ t_1'[y_1'/sp_1], \ldots,\ \lambda\ \overline{x_n}'.t_n'[y_n'/sp_1]);\\
\qquad\qquad \ldots;\\
\qquad\qquad f_{new}\ sp_k\ =\ (\lambda\ \overline{x_1}'.\ t_1'[y_1'/sp_k], \ldots,\ \lambda\ \overline{x_n}'.\ t_n'[y_n'/sp_k]);\\
\quad defs\ f_1\ x\ ==\ nth_1\ (f_{new}\ x);\ \ldots;\ f_n\ x\ ==\ nth_n\ (f_{new}\ x)\\
\quad \text{with } f_1 :: (ctx_1 \Rightarrow \tau_{1a} \to \tau_1),\ \ldots,\ f_n :: (ctx_n \Rightarrow \tau_{na} \to \tau_n)\\
\quad \text{mutually recursive}\\
instance\ ctx\ \Rightarrow\ K_T\ (T\ v_1\ \ldots\ v_n)\ where\\
\qquad\qquad (f_1 :: \tau_1 = t_1; \ldots;\ f_n :: \tau_n = t_n)\ \Longrightarrow\\
\quad instance\\
\qquad \tau' ::\ K_T'\ (\{pcpo\}\ \cup\ \{K' : (K\ v_1) \in ctx\},\\
\qquad\qquad \ldots,\ \{pcpo\}\ \cup\ \{K' : (K\ v_n) \in ctx\})\\
\qquad \text{with proof obligation;}\\
\quad defs\quad f_1' :: (ctx \Rightarrow \tau_1)' == t_1';\ \ldots;\ f_n' :: (ctx \Rightarrow \tau_n)' == t_n'\\
instance\ Monad\ \tau\ where\ (def_{eta}; def_{bind})\ \Longrightarrow\\
\quad defs\quad def_{eta}';\ def_{bind}';\\
\quad t\_instantiate\ Monad\ mapping\ m\_\tau'\\
\qquad \text{with construction and proof obligations}\\
\quad \text{where } m_\tau' \text{ is defined as theory morphism associating}\\
\quad MonadType.M,\ MonadOpEta.eta,\ MonadOpBind.bind\\
\quad \text{to } tau',\ def_{eta}',\ def_{bind}' \text{ respectively;}
\end{array}
$$

Type classes are translated to subclasses of *type*. An axiomatisation of Haskell equality for total functions can be found in *HsHOL*.

consts

$$
\begin{array}{l}
heq ::\ \ 'a\ \to\ 'a\ \to\ bool\\
hneq :: 'a\ \to\ 'a\ \to\ bool
\end{array}
$$

$axclass\ Eq\ <\ type$

$eqAx : heq\ p\ q\ =\ Not\ (hneq\ p\ q)$

Given the restriction to total functions, equality on built-in types can be defined as HOL equality.

6 Semantics (for HOLCF)

Denotational semantics con be given as basis for the translation to HOLCF. Essentially, the claim here is that the expressions on the left hand-side of the following tables represent the denotational meaning of the Haskell expressions on the right hand-side, as well as of the HOLCF expressions to which they are translated. The language on the left hand-side is still based on HOLCF where type have been extended with abstraction (Λ) and fixed point (μ) in order to represent the denotational meaning of domain declarations.

$$
\begin{array}{ll}
\lceil a \rceil & = \ 'a :: pcpo \\
\lceil () \rceil & = \ unit\ lift \\
\lceil Bool \rceil & = \ bool\ lift \\
\lceil Integer \rceil & = \ int\ lift \\
\lceil \rightarrow \rceil & = \rightarrow \\
\lceil (,) \rceil & = \ * \\
\lceil [] \rceil & = \ seq \\
\lceil Maybe \rceil & = \ maybe \\
\lceil T_1\ T_2 \rceil & = \ \lceil T_1 \rceil\ \lceil T_2 \rceil \\
\lceil TC_i \rceil & = \ let\ F\ =\ \mu\ (X_1 * \ldots * X_k). \\
& \quad ((\Lambda\ v_{11}, \ldots, v_{1m}.\ \lceil \tau_{11} \rceil + \ldots + \lceil \tau_{1p} \rceil),\ \ldots, \\
& \quad (\Lambda\ v_{k1}, \ldots, v_{kn}.\ \ldots,\ \lceil \tau_{k1} \rceil + \ldots + \lceil \tau_{kq} \rceil))[X_1/TC_1, \ldots, X_k/TC_k] \\
& in\ nth_i(F) \\
\quad \text{with } 0 < i \leq k, & \text{when } data\ TC_1\ v_{11}\ \ldots\ v_{1m}\ =\ C_{11} :: \tau_{11} | \ldots | C_{1p} :: \tau_{1p}; \\
& \quad \ldots;\ data\ TC_k\ v_{k1}\ \ldots\ v_{kn}\ =\ C_{k1} :: \tau_{k1} | \ldots | C_{kq} :: \tau_{kq} \\
& \text{are mutually recursive declarations}
\end{array}
$$

The representation of term denotation is similar to what we get from the translation, except that for functions we give the representation of the meaning of *fixrec* definitions (FIX is the HOLCF fixed point operator).

$$
\begin{array}{ll}
\lceil x :: a \rceil & = \ x' :: \lceil a \rceil \\
\lceil c \rceil & = \ c' \\
\lceil \backslash x\ \rightarrow\ f \rceil & = \ LAM\ x_t.\ \lceil f \rceil \\
\lceil (a, b) \rceil & = \ (\lceil a \rceil, \lceil b \rceil) \\
\lceil t_1\ t_2 \rceil & = \ \lceil t_1 \rceil\ \cdot\ \lceil t_2 \rceil \\
\lceil let\ x_1\ \ldots\ x_n\ in\ exp \rceil & = \ let\ \lceil x_1 \rceil\ \ldots\ \lceil x_n \rceil\ in\ \lceil exp \rceil \\
\lceil f_i \rceil & = \ let\ g\ = FIX\ (x_1, \ldots, x_n).\ (\lceil t_1 \rceil, \ldots, \lceil t_n \rceil[f_1/x_1, \ldots, f_n/x_n] \\
& \quad in\ nth_i(g) \\
\quad \text{with } 0 < i \leq n, & \quad \text{where } f_1 = t_1,\ f_n = t_n \text{ are mutually recursive definitions}
\end{array}
$$

7 Monads with AWE

A monad is a type constructor with two operations that can be specified axiomatically — *eta* (injective) and *bind* (associative, with *eta* as left and right unit) [Mog89]. Isabelle does not have type constructor classes, therefore monads cannot be translated directly. The indirect solution that we are pursuing, is to translate monadic types as types that satisfy the monadic axioms. This solution can be expressed in terms of theory morphisms — maps between theories, associating signatures to signatures and axioms to theorems in ways that preserve operations and arities, entailing the definition of maps between theorems. Theory morphisms allow for theorems to be moved between theories by translating their proof terms, making it possible to implement parametrisation at the theory level (see [BJL06] for details). A *parameterised theory* Th has a sub-theory Th_P which is the parameter — this may contain axioms, constants and type declarations. Building a theory morphism from Th_P to a theory I provides the instantiation of the parameter with I, and makes it possible to translate the proofs made in the abstract setting of Th to the concrete setting of I — the result being an extension of I. AWE is an extension of Isabelle that can assist in the construction of theory morphisms [BJL06].

A notion of monad [BJL07] can be built in AWE by defining, on an abstract level, a hierarchy of theories culminating in *Monad*, based on the declaration of a unary type constructor M (in *MonadType*) with the two monad operations (contained in *MonadOpEta* and *MonadOpBind*, respectively) and the relevant axioms (in *MonadAxms*). To show that a specific type constructor forms a monad, we have to construct a theory morphism from *MonadAxms* to the specific theory; this involves giving specific definitions of the operators, as well as discharging proof obligations associated with specific instances of the axioms. The following gives an example.

$$
\begin{array}{l}
data\ LS\ a\ =\ N \mid C\ a\ (LS\ a) \\
instance\ Monad\ LS\ where
\end{array}
$$

$$
\begin{array}{ll}
return\ x & = C\ x\ N \\
x\ >>=\ f = & case\ x\ of \\
 & N\ \rightarrow\ N \\
 & C\ a\ b\ \rightarrow\ cnc\ (f\ a)\ (b\ >>=\ f)
\end{array}
$$

$$cnc :: \ LS\ a\ \rightarrow\ LS\ a\ \rightarrow\ LS\ a$$

$$
\begin{array}{l}
cnc\ x\ y = \ case\ x\ of \\
\quad N\ \rightarrow\ y \\
\quad C\ w\ z\ \rightarrow\ cnc\ z\ (C\ w\ y)
\end{array}
$$

These definitions are plainly translated to HOL, as follows. Note that these are not overloaded definitions.

$$
\begin{array}{l}
datatype\ 'a\ LS\ =\ N \mid C\ 'a\ ('a\ LS) \\
consts
\end{array}
$$

$$
\begin{array}{lll}
return\_LS & :: & 'a\ \Rightarrow'\ a\ LS \\
mbind\_LS & :: & 'a\ LS\ \Rightarrow\ ('a\ \Rightarrow\ 'b\ LS)\ \Rightarrow\ 'b\ LS \\
cnc & :: & 'a\ LS\ \Rightarrow\ 'a\ LS\ \Rightarrow\ 'a\ LS
\end{array}
$$

$defs$

$$return\_LS\_def : \ return\_LS :: \ ('a\ LS\ \Rightarrow'\ a)\ ==\ \lambda x.\ C\ x\ N$$

$primrec$

$$
\begin{array}{ll}
mbind\_LS\ N & = \lambda f.\ N \\
mbind\_LS\ (C\ pX1\ pX2) = & \lambda f.\ cnc\ (f\ pX1)\ (mbind\_LS\ pX2\ f)
\end{array}
$$

$primrec$

$$
\begin{array}{ll}
cnc\ N & = \lambda b.\ b \\
cnc\ (C\ pX1\ pX2) = & \lambda b.\ cnc\ pX2\ (C\ pX1\ b)
\end{array}
$$

In order to build up the instantiation of LS as a monad, here it is defined the morphism m_LS from $MonadType$ to the instantiating theory Tx, by associating M to LS.

$$thymorph\ m\_LS : \ MonadType\ \longrightarrow\ Tx$$

$$
\begin{array}{l}
maps\ [('a\ MonadType.M\ \mapsto\ 'a\ Tx.LS)] \\
renames : \ [(MonadOpEta.eta \mapsto return\_LS), (MonadOpBind.bind \mapsto mbind\_LS)]
\end{array}
$$

Renaming is used in order to avoid name clashes in case of more than one monads — here again, note the difference with overloading. Morphism m_LS is then used to instantiate the parameterised theory $MonadOps$.

$t\_instantiate\ MonadOps\ mapping\ m\_LS$

This instantiation gives the declaration of the instantiated methods, which may now be defined.

$defs$

$$
\begin{array}{l}
LS\_eta\_def : \ LS\_eta\ ==\ return\_LS \\
LS\_bind\_def : \ LS\_bind\ ==\ mbind\_LS
\end{array}
$$

In order to construct a mapping from $MonadAxms$ to Tx, the user needs to prove the monad axioms as HOL lemmas (in this case, by straightforward simplification). The translation will print out *sorry* instead.

$$
\begin{array}{ll}
lemma\ LS\_lunit : & LS\_bind\ (LS\_eta\ x)\ t\ =\ t\ x \\
lemma\ LS\_runit : & LS\_bind\ (t ::\ 'a\ LS)\ LS\_eta\ =\ t \\
lemma\ LS\_assoc : & LS\_bind\ (LS\_bind\ (s ::\ 'a\ LS)\ t)\ u\ = \\
 & LS\_bind\ s\ (\lambda x.\ LS\_bind\ (t\ x)\ u) \\
lemma\ LS\_eta\_inj : & LS\_eta\ x\ =\ LS\_eta\ y\ \Longrightarrow\ x\ =\ y
\end{array}
$$

Now, the morphism from *MonadAxms* to *Tx* can be built, and then used to instantiate *Monad*. This gives automatically access to the theorems proven in *Monad* and, modulo renaming, the monadic syntax which is defined there.

$$thymorph\ mon\_LS : \ MonadAxms \ \longrightarrow \ Tx$$

$$\begin{aligned} maps\ & [('a\ MonadType.M \ \mapsto \ 'a\ Tx.LS)] \\ & [(MonadOpEta.eta \ \mapsto \ Tx.LS\_eta), \\ & (MonadOpBind.bind \ \mapsto \ Tx.LS\_bind)] \end{aligned}$$

$t\_instantiate\ Monad\ mapping\ mon\_LS$
$renames : \ [\ldots]$

The *Monad* theory allows for the characterisation of single parameter operators. In order to cover other monadic operators, a possibility is to build similar theories for type constructors of fixed arity. An approach altogether similar to the one shown for HOL could be used, in principle, for HOLCF as well.

8 Conclusion and future work

The main advantage of shallow embedding is to get as much as possible out of the automation currently available in Isabelle, especially with respect to type checking. HOLCF in particular provides with an expressive semantics covering lazy evaluation, as well as with a smart syntax — also thanks to the *fixrec* package. The main disadvantage lies with lack of type constructor classes. Anyway, it is possible to get around the obstacle, at least partially, by relying on an axiomatic characterisation of monads and on a proof-reuse strategy that actually minimises the need for interactive proofs.

Future work should use this framework for proving properties of Haskell programs. For monadic programs, we are also planning to use the monad-based dynamic Hoare and dynamic logic that already have been formalised in Isabelle [Wal05]. Our translation tool from Haskell to Isabelle is part of the Heterogeneous Tool Set Hets and can be downloaded from `http://www.dfki.de/sks/hets`. More details about the translations can be found in [TLMM07].

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.

[BJL07] M. Bortin, E. B. Johnsen, and C. Lueth. The AWE extension package. Technical report, Universitaet Bremen, 2007.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[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.

[Jon93] M. P. Jones. Partial evaluation for dictionary-free overloading. Technical report, Yale University, 1993.

[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.

[MML07] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.

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

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, 1989.

[Mos05a] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set, Habilitation Thesis, 2005.

[Mos05b] T. Mossakowski. Heterogeneous theories and the heterogeneous tool set. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*. IBFI, Dagstuhl, 2005.

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

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

[PHH99] John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[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.

[TLMM07] P. Torrini, C. Lueth, C. Maeder, and T. Mossakowski. Translating Haskell to Isabelle. Technical report, Universitaet Bremen, 2007.

[Wal05] Dennis Walter. Monadic dynamic logic: Application and implementation, 2005.

[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.