Translating Haskell to Isabelle

Paolo Torrini[1] and Christoph Lueth[2] Christian Maeder[2] Till Mossakowski[2]

[1] Verimag, Paolo.Torrini@imag.fr
[2] DFKI Lab Bremen,
{Christoph.Lueth,Christian.Maeder,Till.Mossakowski}@dfki.de

Abstract. We present partial translations of Haskell programs to Isabelle that have been implemented as part of the Heterogeneous Tool Set. The the target logic is Isabelle/HOLCF, and the translation is based on a shallow embedding approach.

1 Introduction

Automating translation from programming languages to the language 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,Tho95]. 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 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. Isabelle/HOLCF [MNvOS99] [Pau94,MNvOS99] is Isabelle/HOL conservatively extended with the Logic of Computable Functions — a formalisation of domain theory.

We have implemented as functions of Hets translations of Haskell to Isabelle/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 [Mos05a,Mos06,MML07a,MML07b] 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 Isabelle/HOLCF which, in contrast to ours, is based on an object-level modelling of the type system [HMW05].

The paper is organised as follows: Section 2 discusses the semantic background of the translation, while the subsequent sections are devoted to the translation of types, datatypes, classes and function definitions, respectively. Sect. 7 shows some example proof, and Sect. 8 concludes the paper.

2 Semantic Background of the Translation

We firstly describe the subset of Haskell that we cover. Our translation to Isabelle/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 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: *Prelude* syntax is covered only partially; list comprehension, *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.

For the translation, we have followed the informal description of the operational semantics given in the Haskell report [PJ03], and also consulted the complete static semantics for Haskell 98 [Fax02], as well as the (fragmentary) dynamic semantics [HH92]. However, it should be noted that there is no denotational semantics of Haskell! This also has become clear after a query that one of the authors has sent to the Haskell mailing list [Mos05b]. Hence, our translation to Isabelle/HOLCF can be seen as the first denotational semantics given to a large subset of Haskell 98. This also means that there is no notion of correctness of this translation, because it just *defines* the denotational semantics. Of course, an interesting question is the coincidence of denotational and operational semantics. However, this is beyond the scope of the paper.

3 Translation of Types

In Isabelle/HOL types are 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 Isabelle/HOLCF each type is interpreted as a pointed complete partially ordered set (class *pcpo*) i.e. a set with a partial order which has suprema of ω-chains and has a bottom. Isabelle's 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 Isabelle/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 Isabelle/HOLCF type constructors to lift types only when this is necessary.

The provision of pcpos, domains and continuous functions by Isabelle/HOLCF eases the translation of Haskell types and functions a lot. However, special care is needed when trying to understand the Haskell semantics. If one reads the section of the Haskell report dealing with types and classes, one could come to the conclusion that a function space in Haskell can be mapped to the space of the continuous functions in Isabelle/HOLCF — this would correspond to a purely lazy language. However, Haskell is a mixed eager and lazy language, as it provides a function *seq* that enforces eager evaluation. This function is introduced in part 6 of the Haskell report, "Predefined Types and Classes", in section 6.2. We quote from there:

> However, the provision of *seq* has important semantic consequences, because it is available at every type. As a consequence, $\perp$ is not the same as $\lambda x \rightarrow \perp$, since *seq* can be used to distinguish them.

In order to enforce this distinction, each function space needs to be lifted. The same holds for products. We define these liftings in a specific Isabelle/HOLCF theory *HsHOLCF* (included in the Hets distribution) as follows:

```
defaultsort pcpo

domain ('a,'b) lprod = lpair (lazy 'a) (lazy 'b)

domain ('a,'b) LiftedCFun  = Lift (lazy "'a -> 'b")
syntax
  "LiftedCFun"  :: "[type, type] => type"      ("(_ --> _)" [1,0]0)

constdefs
  liftedApp :: "('a --> 'b) => ('a => 'b)" ("_$$_" [999,1000] 999)
                                                (* application *)
  "liftedApp f x == case f of
                    Lift $ g => g $ x"
constdefs
  liftedLam :: "('a => 'b) => ('a --> 'b)" (binder "Lam " 10)
                                                (* abstraction *)
  "liftedLam f == Lift $ (LAM x . f x)"
```

Our translation of Haskell types to Isabelle types is defined recursively. It is based on a translation of names for avoidance of name clashes that is not specified here. We write α' for both the recursive translation of item α and the

renaming according to the name translation. The translation of types is given by the following rules:
Types

$$
\begin{array}{lll}
a & \Longrightarrow & 'a :: \{pcpo\} \\
() & \Longrightarrow & unit \text{ lift} \\
Bool & \Longrightarrow & bool \text{ lift} \\
Integer & \Longrightarrow & int \text{ lift} \\
\tau_1 \to \tau_2 & \Longrightarrow & \tau_1' \texttt{ --> } \tau_2') \\
(\tau_1, \tau_2) & \Longrightarrow & (\tau_1' \ lprod \ \tau_2') \\
[\tau] & \Longrightarrow & \tau' \ llist \\
T \ \tau_1 \ \ldots \ \tau_n & \Longrightarrow & \tau_1' \ \ldots \ \tau_n' \ T' \\
 & & \text{with } T \text{ either datatype or defined type}
\end{array}
$$

Built-in types are translated to the lifting of the corresponding HOL type. The Isabelle/HOLCF type constructor *lift* is used to lift types to flat domains. The type constructor *llist* is discussed in the next section.

4 Translation of Datatypes

As explained in the Haskell report [PJ03], section 4.2.3, the following four Haskell declarations

```
data D1 = D1 Int
data D2 = D2 !Int
type S = Int
newtype N = N Int
```

have four different semantics. Indeed, the correct translation to Isabelle/HOLCF is as follows:

```
domain D1 = D1 (lazy D1_1::"Int lift")
domain D2 = D2 (D2_1::"Int lift")
types S = "Int lift"
pcpodef N = "{x:: Int lift . True}"
by auto
```

In Isabelle/HOLCF, the keyword *domain* defines a (possibly recursive) domain as solution of the corresponding domain equation. The keyword *lazy* ensures that the constructor $D1$ above is non-strict, i.e. $D1 \perp \neq \perp$. The keyword *pcpodef* can be used to define sub-pcpos of existing pcpos; here, we use it just to introduce an isomorphic copy of an existing pcpo — this is the semantics of Haskell *newtype* definitions. Although a domain with one strict unary constructor, such as $D2$, also introduces an isomorphic copy, the difference is that the "constructor" $Abs\_N :: Int \ lift \Rightarrow N$ introduced implicitly by the above *pcpodef* declaration is merely a representation function; it cannot be used for pattern matching. This means that in function definitions, any pattern for values of type N must be variable (which always matches), and the selector $Rep\_N :: N \Rightarrow Int \ lift$ needs

to be applied to this variable whereever the corresponding value of type *Int lift* is needed. This "always match" behaviour is exactly the behaviour specified for newtypes in the Haskell report.

Lists are translated to the domain *llist*, defined as follows in our prelude theory *HsHOLCF*:

```
domain 'a llist = lNil | ### (lazy 'a) (lazy 'a llist)
```

allowing for partial list as well as for infinite ones [MNvOS99].

For each datatype, we have to lift the constructors from the HOLCF continuous function spaces to our lifted function spaces:

```
constdefs
  cont2lifted2 :: "('a -> 'b -> 'c) => ('a --> 'b --> 'c)"
  "cont2lifted2 f == Lam x . Lam y. f $ x $ y"

  llCons :: "'a --> 'a llist --> 'a llist"
  "llCons == cont2lifted2 (op ###)"
```

The general scheme for translation of mutually recursive lazy Haskell datatypes to Isabelle/HOLCF domains is as follows:

$$
\begin{array}{l}
\mathit{data}\ \phi_1 \ =\ C_{11}\ x_1 \ldots x_i \mid \ldots \mid C_{1p}\ y_1 \ldots y_j \\
\ldots \\
\mathit{data}\ \phi_n \ =\ C_{n1}\ w_1 \ldots w_h \mid \ldots \mid C_{1q}\ z_1 \ldots z_k \qquad \Longrightarrow \\
\quad \mathit{domain}\ \phi'_1 \ = C'_{11}\ (\mathit{lazy}\ ::\ x'_1)\ \ldots\ (\mathit{lazy}\ ::\ x'_i) \mid \\
\qquad\qquad\qquad \ldots \mid \\
\qquad\qquad\qquad C'_{1p}\ (\mathit{lazy}\ ::\ y'_1)\ \ldots\ (\mathit{lazy}\ ::\ y'_j) \\
\quad \mathit{and}\ \ldots \\
\quad \mathit{and} \quad \phi'_n \ =\ C'_{n1}\ (\mathit{lazy}\ ::\ w'_1)\ \ldots\ (\mathit{lazy}\ ::\ w'_h) \mid \\
\qquad\qquad\qquad \ldots \mid \\
\qquad\qquad\qquad C'_{nq}\ (\mathit{lazy}\ ::\ z'_1)\ \ldots\ (\mathit{lazy}\ ::\ z'_k)
\end{array}
$$

Mutually recursive datatypes relies on specific Isabelle syntax (keyword *and*).[3] Order of declarations is taken care of. In case of strict arguments (indicated with ! in Haskell), the keyword *lazy* is omitted.

The translation scheme for definitions of type synonyms is simply as follows:

$$\mathit{type}\ \tau \ =\ \tau_1 \qquad \Longrightarrow \qquad \mathit{types}\ \tau' \ =\ \tau'_1$$

[3] Due to a bug in Isabelle/HOLCF 2005, declaration of mutually recursive lazy domains does not work. However, this will be fixed in Isabelle 2007; the fix currently is available in the development snapshot.

5 Translation of Kinds and Type Classes

Type classes in Isabelle and Haskell associate a set of functions to a type class identifier (these functions are also called the *methods* of the class). In Isabelle, type classes are typically further specified using a set of axioms; for example, the class *linorder* of total orders is specified using the usual total order axioms. Of course, such axiomatizations are not possible in Haskell. Indeed, in Haskell, there is no check that the class *Ord* actually consists of total orders only, and hence it would be inappropriate to translate it to *linorder* in Isabelle. Instead, we translate to a newly declared Isabelle class *Ord*. The only thing that we assume is that it is a subclass of *pcpo*, because all Haskell types are translated to pcpos. Hence, type classes are translated to Isabelle/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.

Class instance declarations declare that a particular type belongs to a class. In Isabelle, instance declarations generate proof obligations, namely that the methods for the type at hand indeed satisfy the axioms of the class. Since our translation only generates classes without axioms (beyond those of *pcpo*), proofs are trivial and proof obligation may be automatically discharged.

A Haskell class instance declaration that declares type T to belong to class C may define the behaviour of C's class methods for T. The same is possible with normal Isabelle constant definitions, if the type of the constant (function) is specialised to T in the definition. With this, we avoid an explicit handling of dictionaries, as described in the static semantics of Haskell [Fax02].

A restriction of Isabelle is that it does not allow for constructor classes. Therefore, the same restrictions apply to our translation.[4]

Classes

$$K \quad \Longrightarrow \quad K'$$

Type schemas

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

Haskell type variables are translated to variables of class *pcpo*. Each type is associated to a sort in Isabelle (in Haskell, the same concept is called "kind"), defined by the set of the classes of which it is member.

Class declarations

$$class\ K\ where\ (Dec_1; \ldots; Dec_n]) \Longrightarrow class\ K' \subseteq pcpo;\ Dec'_1;\ \ldots;\ Dec'_n$$

[4] Isabelle does not support multi-parameter classes (used in some Haskell 98 extensions) either.

Class instance definitions

$$\begin{array}{l} \mathit{instance}\ \mathit{ctx}\ \Rightarrow\ K_T\ (T\ v_1\ \ldots\ v_n)\ \mathit{where} \\ \qquad (f_1 :: \tau_1 = t_1; \ldots;\ f_n :: \tau_n = t_n) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow \\ \quad \mathit{instance} \\ \qquad \tau' ::\ K'_T\ (\{\mathit{pcpo}\}\ \cup\ \{K' : (K\ v_1) \in \mathit{ctx}\}, \\ \qquad\qquad \ldots, \\ \qquad\qquad \{\mathit{pcpo}\}\ \cup\ \{K' : (K\ v_n) \in \mathit{ctx}\}) \\ \quad \text{proof obligation;} \\ \quad \mathit{defs}\ f'_{\tau 1} :: (\mathit{ctx} \Rightarrow \tau_1)' = t'_1; \\ \qquad \ldots; \\ \qquad f'_{\tau n} :: (\mathit{ctx} \Rightarrow \tau_n)' = t'_n \end{array}$$

6 Translation of Function Definitions and Terms

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

```
lliftbin :: "('a::type => 'b::type => 'c::type) =>
               ('a lift --> 'b lift --> 'c lift)"
"lliftbin f == cont2lifted2 (flift1 (%x. flift2 (f x)))"
```

Boolean values are translated to values of *bool lift* (*tr* in Isabelle/HOLCF) i.e. *TT*, *FF* and $\bot$, and Boolean connectives to the corresponding Isabelle/HOLCF operators. Isabelle/HOLCF-defined *If then else fi* and *case* syntax are used to translate conditional and case expressions, respectively. There are restrictions, however, on case expressions, due to limitations in the translation of patterns; in particular, 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, $\bot$ being used as default value in the latter. Only let expressions without patterns on the left are dealt with; guarded expressions are translated as conditional ones; where expressions and list comprehension are not covered.

$$f :: \phi \quad \Longrightarrow \quad consts\ f' :: \phi'$$

$$f\ \overline{x}\ p_1\ \overline{x}_1 = t_1;\ \ldots;\ f\ \overline{x}\ p_n\ \overline{x}_n = t_n \quad \Longrightarrow$$
$$(f\ \overline{x} = case\ y\ of\ (p_1 \rightarrow (\backslash \overline{x}_1 \rightarrow t_1);\ \ldots;\ p_n(\rightarrow \backslash \overline{x}_n \rightarrow t_n)))'$$

$$f\ \overline{x} = t \quad \Longrightarrow \quad defs\ f' :: \phi' = Lam\ \overline{x}'.\ t'$$
with $f :: \phi$ not occurring in t

$$(f_1\ \overline{v_1} = t_1;\ \ldots;\ f_n\ \overline{v_n} = t_n) \quad \Longrightarrow$$
$$fixrec\ f_1' :: \phi_1' = (Lam\ \overline{v_1}'.\ t_1')$$
$$and\ \ldots$$
$$and\ f_n' :: \phi_n' = (Lam\ \overline{v_n}'.\ t_n')$$
with $f_1 :: \phi_1, \ldots, f_n :: \phi_n$ mutually recursive

Function declarations use Isabelle keyword *consts*. Non-recursive definitions are translated to standard definitions using Isabelle keyword *defs*. Recursive definitions rely on Isabelle/HOLCF package *fixrec* which provides nice syntax for fixed point definitions, including mutual recursion. Lambda abstraction is translated as continuous abstraction for lifted function spaces (*Lam*), function application as continuous application (the *$$* operator), see Sect. 3 above.

7 Example Proof

We illustrate our translation with a sample proof about the compatibility of map and composition. The Haskell declarations

```
comp ::  (b -> c) -> (a -> b) -> a -> c
comp f g x = f (g x)

map1 :: (a -> b) -> [a] -> [b]
map1 f [] = []
map1 f (x:xs) = f x : map1 f xs
```

are translated to

```
consts
X_comp :: "('b --> 'c) --> ('a --> 'b) --> 'a --> 'c"
map1 :: "('a --> 'b) --> 'a llist --> 'b llist"

defs
X_comp_def :
"(X_comp :: ('b --> 'c) --> ('a --> 'b) --> 'a --> 'c) ==
 Lam f. Lam g. Lam x. f $$ (g $$ x)"
```

```
fixrec "(map1 :: ('a --> 'b) --> 'a llist --> 'b llist) =
         (Lam qX1.  (Lam qX2. case qX2 of
             lNil => lNil |
             op ### $ pX1 $ pX2 =>
                 llCons $$ (qX1 $$ pX1) $$ (map1 $$ qX1 $$ pX2)))"
```

The fixrec definition leads to a bunch of theorems, one for simplification. However, the latter makes the Isabelle simplifier loop, and hence needs to be replaced with a more suitable simplification theorem:

```
lemmas map1_simps [simp del]
lemma map1_simp : "map1 $$ (f::('a --> 'b)) $$ (x::'a llist) =
    (case x of lNil => lNil |
               op ### $ pX1 $ pX2 =>
                    llCons $$ (f$$pX1)$$ (map1$$f$$pX2))"
  apply (subst map1_simps)
  by auto
```

Due to the recursion, this theorem cannot be fed into the simplifier either. However, using substitution, it helps in proving the following expected lemmas about the behaviour of *map*1:

```
lemma map1_UU [simp] : "map1 $$ f $$ UU = UU"
  apply (subst map1_simp)
  by simp

lemma map1_lNil[simp] : " map1 $$ f $$ lNil = lNil"
  apply (subst map1_simp)
  by simp

lemma map1_cons1[simp] :
               "map1 $$ f $$ (op ### $ x $ (xs::'a llist)) =
                op ### $ (f $$ x) $ (map1 $$ f $$ xs)"
  apply (subst map1_simp)
  apply (simp add: cont2lifted2_def)
done
```

With these, it is now easy to prove, via induction, that map distributes over composition:

```
theorem compMap :
   "X_comp $$ (map1 $$ f) $$ (map1 $$ g) $$ x =
    map1 $$ (X_comp $$ f $$ g) $$ x"
  apply (rule llist.ind)
  back
  apply (auto simp add: X_comp_def)
done
```

Isabelle/HOLCF also provides a coinduction method for recursive domains.

8 Conclusion and Related Work

We provide a shallow embedding of Haskell to Isabelle/HOLCF, which can be used for proving properties of Haskell programs. This translation also is the first denotational semantics that has been given to Haskell.

The main advantage of our shallow approach is to get as much as possible out of the automation currently available in Isabelle, especially with respect to type checking. Isabelle/HOLCF in particular provides with an expressive semantics covering lazy evaluation, as well as with a smart syntax — also thanks to the *fixrec* package. It is interesting to note that Haskell functions and product types have to be lifted due to the mixture of eager and lazy evaluation that Haskell exhibits due to the presence of the *seq* function. Type classes in Haskell are similar enough to Isabelle's type classes such that explicit handling of dictionaries can be avoided.

The main disadvantage of our approach is the 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.

Concerning related work, although there have been translations of functional languages to first-order systems — those to FOL of Miranda [Tho95,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 Isabelle/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 Isabelle/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. Our 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.

Future work should use this framework for proving properties of Haskell programs. Also, the lacking support of constructor classes should be overcome. Currently, Hets already provides some experimental translation of constructor classes and monads, also covering *do* notation, using theory morphisms as provided by the package AWE [BJL06]. However, there are (mainly syntactic) problems (with

name clashes) that currently prevent a proper integration with Isabelle/HOLCF. These problems should be solved in the near future.

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].

Acknowledgment

This work has been supported by the *Deutsche Forschungsgemeinschaft* under grants KR 1191/5-2 and KR 1191/7-2 . We thank Brian Huffmann for help with the Isabelle/HOLCF package and Erwin R. Catesbeiana for pointing out an inconsistency.

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.

[Fax02] Karl-Filip Faxén. A static semantics for Haskell. *J. Funct. Program*, 12(4&5):295–357, 2002.

[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.

[HH92] Kevin Hammond and Cordelia Hall. A dynamic semantics for Haskell, October 20 1992.

[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.

[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.

[MML07a] 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.

[MML07b] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Bernhard Beckert, editor, *VERIFY 2007*, volume 259 of *CEUR Workshop Proceedings*. 2007.

[MNvOS99] O. Mueller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

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

[Mos05b] Till Mossakowski, 2005. Mail to Haskell mailing list, see http://www.haskell.org/pipermail/haskell/2005-January/015238.html.

[Mos06] T. Mossakowski. Hets user guide. Technical report, 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(4):339–365, 1989.

[Tho92] Simon Thompson. Formulating haskell. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 258–268. Springer, 1992.

[Tho95] S. Thompson. A logic for Miranda, revisited. *Formal Aspects of Computing*, 7(4):412–429, 1995.

[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. Diploma thesis, University of Bremen.

[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.