Algebraic types and pattern matching in the logical language of the Why verification platform

Andrei Paskevich

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Algebraic types and pattern matching in the logical language of the WHY *verification platform*

Andrei Paskevich

N° 7128

Novembre 2009

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7128--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY – ÎLE-DE-FRANCE

Algebraic types and pattern matching in the logical language of the Why verification platform

Andrei Paskevich[*]

Thème : Programmation, vérification et preuves
Équipe-Projet Proval

Rapport de recherche n° 7128 — Novembre 2009 — 12 pages

Abstract: We introduce an extension of the logical language of a software verification tool Why with algebraic types and pattern matching expressions. We describe the corresponding additions to the syntax of Why and give the semantics of the new constructions in terms of first-order logic with polymorphic types as it is adopted in Why and the Alt-Ergo prover.

Key-words: formal language, first-order logic, algebraic types, pattern matching

[*] This work is supported by the ANR CAT project.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Les types algébriques et le filtrage par motif dans le langage logique de la plateforme de vérification Why

Résumé : On introduit une extension du langage logique de l'outil de vérification des logiciels Why avec des types algébriques et des expressions de filtrage par motif. On décrit les modifications correspondantes de la syntaxe de Why et on donne la sémantique des nouvelles constructions dans la logique du premier ordre avec des type polymorphes telle qu'elle est adoptée dans Why et dans le démonstrateur automatique Alt-Ergo.

Mots-clés : langage formel, logique du premier ordre, types algébriques, filtrage par motif

1 Motivation

This work was inspired by the recent experiments [1] with verification of floating-point computations in WHY [2]. According to the IEEE Standard 754, which specifies the representation and operation for the floating-point numbers, at any point a programmer can choose: one of five different encodings (binary numbers of single, double, and quadruple precision and decimal numbers of double and quadruple precision); one of five rounding algorithms; a computation mode with or without overflows. Correspondingly, the logical annotations in a floating-point program must take into account the encoding of a particular variable or constant, as well as the current rounding algorithm and computation mode. This can be done, of course, with a number of appropriately chosen predicates and series of «if-then-else» expressions. However, a more elegant solution would be to use three enumerated types, namely:

$$\texttt{FPencoding} = \{\texttt{BinSingle}, \texttt{BinDouble}, \texttt{BinQuad}, \texttt{DecDouble}, \texttt{DecQuad}\}$$
$$\texttt{FProunding} = \{\texttt{NearestTieEven}, \texttt{NearestTieInf}, \texttt{ToZero}, \texttt{ToPInf}, \texttt{ToNInf}\}$$
$$\texttt{FPoverflow} = \{\texttt{OFallowed}, \texttt{OFexception}\}$$

and to use branching constructions in logical formulas and terms. And since the logic of WHY supports polymorphism, it is just natural to treat enumerations as a special case of polymorphic algebraic types *à la* HASKELL or ML.

The principal objectives of this work are as follows:

1. Propose a syntax for algebraic types declaration and for pattern matching expressions which is consistent with the overall syntax of WHY.
2. Devise an appropriate semantics for algebraic types and pattern matching expressions in terms of first-order logic with polymorphism, which is handled by the ALT-ERGO prover [3] and (via encoding) by other automated SMT provers supported by WHY.
3. Based on this semantics, implement a translation procedure optimized for an efficient proof search in an SMT prover.

2 Syntax

We use BNF notation to present grammar rules. Non-terminals are written in italic (e.g. *typedefn*) and terminals in typewriter font (e.g. `match`). Grammar productions have the form:

$$\textit{nonterm} \;\rightarrow\; \textit{alt}_1 \mid \textit{alt}_2 \mid \ldots \mid \textit{alt}_n$$

and the following conventions are adopted:

$\textit{pat}_1 \mid \textit{pat}_2$	choice	$(\,\textit{pattern}\,)$	grouping
$[\,\textit{pattern}\,]$	optional	$\{\,\textit{pattern}\,\}$	zero or more repetitions

2.1 Type declaration

In the original language of Why, the declaration of an abstract logical type has the following syntax:

$$\begin{array}{rcl}
\mathit{typeDecl} & \to & \texttt{type}\ \mathit{typeHead} \\
\mathit{typeHead} & \to & \mathit{ident} \\
 & | & \mathit{typeVar}\ \mathit{ident} \\
 & | & \texttt{(}\ \mathit{typeVar}\ \texttt{,}\ \mathit{typeVar}\ \{\ \texttt{,}\ \mathit{typeVar}\}\ \texttt{)}\ \mathit{ident} \\
\mathit{typeVar} & \to & \texttt{'}\ \mathit{ident} \\
\mathit{ident} & \to & (\texttt{\_}|\texttt{a}|\ldots|\texttt{z}|\texttt{A}|\ldots|\texttt{Z})\ \{\texttt{\_}|\texttt{a}|\ldots|\texttt{z}|\texttt{A}|\ldots|\texttt{Z}|\texttt{0}|\ldots|\texttt{9}|\texttt{'}\}
\end{array}$$

The three variants of the *typeHead* non-terminal describe, respectively, the introduction of nullary, unary, and n-ary type constructors (for $n > 1$). In the third variant, all the type variables must be distinct. Built-in types and abstract types are the *pure types* of Why:

$$\begin{array}{rcl}
\mathit{pureType} & \to & \mathit{ident} \\
 & | & \mathit{pureType}\ \mathit{ident} \\
 & | & \texttt{(}\ \mathit{pureType}\ \texttt{,}\ \mathit{pureType}\ \{\ \texttt{,}\ \mathit{pureType}\}\ \texttt{)}\ \mathit{ident} \\
 & | & \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{real}\ |\ \texttt{unit} \\
 & | & \mathit{typeVar}
\end{array}$$

Every *ident* occurring in a pure type must be a previously declared type constructor of the corresponding arity.

To introduce algebraic types, we augment the above syntax as follows:

$$\begin{array}{rcl}
\mathit{typeDecl} & \to & \texttt{type}\ \mathit{typeHead}\ [\ \texttt{=}\ \mathit{typeDefn}\ \{\ \mathit{typeDeclCont}\ \}\ ] \\
\mathit{typeDeclCont} & \to & \texttt{and}\ \mathit{typeHead}\ \texttt{=}\ \mathit{typeDefn} \\
\mathit{typeDefn} & \to & [\ \texttt{|}\ ]\ \mathit{constructor}\ \{\ \texttt{|}\ \mathit{constructor}\ \} \\
\mathit{constructor} & \to & \mathit{ident}\ [\ \texttt{(}\ \mathit{pureType}\ \{\ \texttt{,}\ \mathit{pureType}\ \}\ \texttt{)}\ ]
\end{array}$$

An example of algebraic type declaration is given in Figure 1. It represents partially interpreted first-order formulas and terms with Hilbert's epsilon operator. The constructors `Forall` and `Epsilon` bind a variable in the underlying formula; bound variables are encoded as de Bruijn's indexes. The type of signature symbols and the carrier type are passed as the arguments (`'sym` and `'val`, respectively) to the type constructors `formula` and `term`.

An algebraic type declaration must have at least one constructor, since every type is inhabited in the logic of Why. Just as with abstract types, all type variables in an occurrence of *typeHead* must be distinct. In mutually recursive type declarations, the type constructors do not need to be given the same list of arguments (though they are in this example). Every type variable occurring in a constructor declaration must appear among the arguments of the corresponding type constructor in *typeHead*; for instance, heterogeneous lists are not supported.

```
type 'a list = Nil | Cons ('a, 'a list)

type ('sym,'val) formula =
  | False
  | Implies (('sym,'val) formula, ('sym,'val) formula)
  | Forall (('sym,'val) formula)
  | Atom ('sym, ('sym,'val) term list)

and ('sym,'val) term =
  | Value ('val)
  | Variable (int)
  | Epsilon (('sym,'val) formula)
  | Term ('sym, ('sym,'val) term list)
```

Figure 1: Algebraic type declaration

2.2 Match expressions

We extend the *logicExpr* non-terminal (which describes both terms and formulas in the logical language) with the following productions:

$$\begin{array}{rcl} \mathit{logicExpr} & \rightarrow & \ldots \\ & | & \texttt{match}\ \mathit{logicExpr}\ \texttt{with}\ \mathit{matchCases}\ \texttt{end} \\ \mathit{matchCases} & \rightarrow & [\ \texttt{|}\ ]\ \mathit{matchCase}\ \{\ \texttt{|}\ \mathit{matchCase}\ \} \\ \mathit{matchCase} & \rightarrow & \mathit{pattern}\ \texttt{->}\ \mathit{logicExpr} \\ \mathit{pattern} & \rightarrow & \mathit{ident}\ [\ \texttt{(}\ \mathit{ident}\ \{\ \texttt{,}\ \mathit{ident}\ \}\ \texttt{)}\ ] \end{array}$$

Match expressions can occur both as formulas and as terms. Matching must be exhaustive and all patterns must be linear (i.e. all the variables in a pattern are distinct). Currently, we do not support nested patterns in match expressions; any match expression must have exactly one branch per constructor. Note that constructors and pattern variables are represented by the same lexeme, *ident*. This does not pose a problem as long as all patterns are flat: the upper *ident* is necessarily a constructor and the rest are variables. For nested patterns, some discrimination rules will have to be established.

In the current syntax of Why, the underscore character `_` is a valid variable name. Thus, we cannot use it as an anonymous wildcard in patterns; in particular, several underscores in a pattern would violate linearity. While making a single underscore a reserved lexeme is a reasonable decision, this would create a non-conservative change of syntax and has to be taken with greater care.

Examples of match expressions are given in Figure 2. We do not consider recursive functions here, as Why does not currently support such definitions.

```
predicate isEmpty (l : 'a list) =
  match l with
  | Nil -> true
  | Cons (a,b) -> false
  end

type float

type FPencoding = BinSingle | BinDouble | BinQuad
                            | DecDouble | DecQuad

logic floatEnc : float -> FPencoding

function expMax (r : float) : int =
  match (floatEnc (r)) with
  | BinSingle -> 127
  | BinDouble -> 1023
  | BinQuad -> 16383
  | DecDouble -> 384
  | DecQuad -> 6144
  end
```

Figure 2: Match expressions

3 Translation

3.1 Type declaration

Let us consider a generic algebraic type declaration:

$$\begin{array}{ll} \texttt{type (}\alpha_1\texttt{, }\ldots\texttt{, }\alpha_m\texttt{)}\ D\ \texttt{=} & C_1\ \texttt{(}T_{1,1}\texttt{,}\ldots\texttt{,}T_{1,e_1}\texttt{)} \\ & \vdots \\ & \texttt{|}\ C_n\ \texttt{(}T_{n,1}\texttt{,}\ldots\texttt{,}T_{n,e_n}\texttt{)} \end{array}$$

We denote the type constructor by D, type variables by α_i, constructors by C_k, and pure types by $T_{k,l}$. This declaration corresponds to the following suite of declarations in the base logic of Why.

First of all, the type constructor is declared as an abstract type:

$$\texttt{type (}\alpha_1\texttt{, }\ldots\texttt{, }\alpha_m\texttt{)}\ D$$

Then each constructor is declared as an abstract function to this type:

$$\begin{array}{l} \texttt{logic}\ C_1\ \texttt{:}\ T_{1,1}\texttt{,}\ldots\texttt{,}T_{1,e_1}\ \texttt{->}\ \texttt{(}\alpha_1\texttt{,}\ldots\texttt{,}\alpha_m\texttt{)}\ D \\ \quad\vdots \\ \texttt{logic}\ C_n\ \texttt{:}\ T_{n,1}\texttt{,}\ldots\texttt{,}T_{n,e_n}\ \texttt{->}\ \texttt{(}\alpha_1\texttt{,}\ldots\texttt{,}\alpha_m\texttt{)}\ D \end{array}$$

In order to ensure uniqueness of matching, we define a discrimination function D_match of arity $n+1$ (below, β is a fresh type variable):

```
logic $D$_match : ($\alpha_1$,...,$\alpha_m$) $D$, $\beta$, ..., $\beta$ -> $\beta$
axiom $D$_match_$C_1$ :
   forall $y_1$,...,$y_n : \beta$. forall $x_1 : T_{1,1}$. ... forall $x_{e_1} : T_{1,e_1}$.
      $D$_match( $C_1$($x_1$,...,$x_{e_1}$), $y_1$, ..., $y_n$ ) = $y_1$
  ⋮
axiom $D$_match_$C_n$ :
   forall $y_1$,...,$y_n : \beta$. forall $x_1 : T_{n,1}$. ... forall $x_{e_n} : T_{n,e_n}$.
      $D$_match( $C_n$($x_1$,...,$x_{e_n}$), $y_1$, ..., $y_n$ ) = $y_n$
```

To ensure that the constructors are injective, each argument position in each constructor is provided with an access function:

```
logic $C_1$_proj_1 : ($\alpha_1$,...,$\alpha_m$) $D$ -> $T_{1,1}$
axiom $C_1$_proj_1_def : forall $x_1 : T_{1,1}$. ... forall $x_{e_1} : T_{1,e_1}$.
      $C_1$_proj_1( $C_1$($x_1$,...,$x_{e_1}$) ) = $x_1$
  ⋮
logic $C_1$_proj_$e_1$ : ($\alpha_1$,...,$\alpha_m$) $D$ -> $T_{1,e_1}$
axiom $C_1$_proj_$e_1$_def : forall $x_1 : T_{1,1}$. ... forall $x_{e_1} : T_{1,e_1}$.
      $C_1$_proj_$e_1$( $C_1$($x_1$,...,$x_{e_1}$) ) = $x_{e_1}$
  ⋮
logic $C_n$_proj_1 : ($\alpha_1$,...,$\alpha_m$) $D$ -> $T_{n,1}$
axiom $C_n$_proj_1_def : forall $x_1 : T_{n,1}$. ... forall $x_{e_n} : T_{n,e_n}$.
      $C_n$_proj_1( $C_n$($x_1$,...,$x_{e_n}$) ) = $x_1$
  ⋮
logic $C_n$_proj_$e_n$ : ($\alpha_1$,...,$\alpha_m$) $D$ -> $T_{n,e_n}$
axiom $C_n$_proj_$e_n$_def : forall $x_1 : T_{n,1}$. ... forall $x_{e_n} : T_{n,e_n}$.
      $C_n$_proj_$e_n$( $C_n$($x_1$,...,$x_{e_n}$) ) = $x_{e_n}$
```

Finally, we provide the inversion axiom postulating that every element of the type is the result of a constructor application:

```
axiom $D$_inversion : forall $x : $($\alpha_1$,...,$\alpha_m$) $D$.
      $x$ = $C_1$( proj_$C_1$_1($x$), ..., proj_$C_1$_$e_1$($x$) ) or ...
          ... or $x$ = $C_n$( proj_$C_n$_1($x$), ..., proj_$C_n$_$e_n$($x$) )
```

The minimality of the type carrier (i.e. that every element of the type is expressible as a finite superposition of constructors) is not directly expressible in a first-order language, and so we omit it.

The case of mutually recursive type declarations is handled in the same way, except that the abstract type declarations are put out first, before all the `logic` and `axiom` declarations.

The axioms for D`_match` given above are sufficient to prove, for instance, that two different constructors never produce the same value. However, such an inference is unlikely to be reproduced in an SMT prover such as ALT-ERGO. Therefore, it may be advantageous to introduce in translation a special instance of the D`_match` function:

```
logic $D$_to_int : ($\alpha_1$,...,$\alpha_m$) $D$ -> int
axiom $D$_to_int_$C_1$ : forall $x_1 : T_{1,1}$. ... forall $x_{e_1} : T_{1,e_1}$.
      $D$_to_int( $C_1$($x_1$,...,$x_{e_1}$) ) = 1
  ⋮
axiom $D$_to_int_$C_n$ : forall $x_1 : T_{n,1}$. ... forall $x_{e_n} : T_{n,e_n}$.
      $D$_to_int( $C_n$($x_1$,...,$x_{e_n}$) ) = $n$
```

In Figure 3, we show the automatically generated translation of the list type declaration (see Figure 1) in the syntax of ALT-ERGO. The terms in square brackets are so-called «triggers» which are used to guide instantiation in SMT provers: basically, a universally quantified formula is instantiated whenever its trigger matches a ground subterm in some other formula, initial or inferred. Note that we hint the subordinate SMT prover to instantiate definitions of projection and discrimination functions (C_i`_proj_`j and D`_to_int`) for every ground occurrence of a constructor.

3.2 Match expression

Translation of match expressions depends on whether they occur as terms or as formulas. Let us consider a generic match expression:

```
match $t$ with
| $C_1$ ($x_{1,1}$,...,$x_{1,e_1}$) -> $E_1$
⋮
| $C_n$ ($x_{n,1}$,...,$x_{n,e_n}$) -> $E_n$
end
```

Here, t is a term of some algebraic type (T_1,...,T_m) D; $C_1, \ldots, C_n$ are the constructors of this type; $x_{i,j}$ are the pattern variables bound in E_i; and E_i are either formulas or terms of the same type. Recall that $x_{i,1}, \ldots, x_{i,e_i}$ must be distinct variables.

If this match expression occurs as a term, it is translated to an application of the D`_match` function as follows:

```
$D$_match( $t$ , $E_1$[$x_{1,1}$/$C_1$_proj_1($t$), ..., $x_{1,e_1}$/$C_1$_proj_$e_1$($t$)] ,
                 ⋮
                 $E_n$[$x_{n,1}$/$C_n$_proj_1($t$), ..., $x_{n,e_n}$/$C_n$_proj_$e_n$($t$)] )
```

The pattern variables are simultaneously replaced with the corresponding projections in the arguments of D`_match`.

When the match expression is a formula, the expressions $E_1, \ldots, E_n$ are formulas, too, and so we cannot pass them as the arguments to D`_match`. Instead,

```
type ’a3 list

logic Nil : ’a1 list

logic Cons : ’a1, ’a1 list -> ’a1 list

logic list_match : ’a1 list, ’a2, ’a2 -> ’a2

axiom list_match_Nil :
  (forall x_1 : ’a2. (forall x_2 : ’a2
    [list_match(Nil, x_1, x_2)].
      (list_match(Nil, x_1, x_2) = x_1)))

axiom list_match_Cons :
  (forall x_1 : ’a2. (forall x_2 : ’a2.
  (forall x_3 : ’a1. (forall x_4 : ’a1 list
    [list_match(Cons(x_3, x_4), x_1, x_2)].
      (list_match(Cons(x_3, x_4), x_1, x_2) = x_2)))))

logic Cons_proj_1 : ’a1 list -> ’a1

axiom Cons_proj_1_def :
  (forall x_6 : ’a1. (forall x_7 : ’a1 list
    [Cons(x_6, x_7)].
      (Cons_proj_1(Cons(x_6, x_7)) = x_6)))

logic Cons_proj_2 : ’a1 list -> ’a1 list

axiom Cons_proj_2_def :
  (forall x_6 : ’a1. (forall x_7 : ’a1 list
    [Cons(x_6, x_7)].
      (Cons_proj_2(Cons(x_6, x_7)) = x_7)))

axiom list_inversion :
  (forall x_5 : ’a1 list.
    ((x_5 = Nil) or
     (x_5 = Cons(Cons_proj_1(x_5), Cons_proj_2(x_5)))))

logic list_to_int : ’a1 list -> int

axiom list_to_int_Nil :
  (list_to_int(Nil) = 0)

axiom list_to_int_Cons :
  (forall x_1 : ’a1. (forall x_2 : ’a1 list
    [Cons(x_1, x_2)].
      (list_to_int(Cons(x_1, x_2)) = 1)))
```

Figure 3: Algebraic type declaration (translation)

we choose one of the two possible translations:

$$\begin{aligned}
&\texttt{(} t \texttt{ = } C_1\texttt{(proj\_}C_1\texttt{\_1(}t\texttt{), ..., proj\_}C_1\texttt{\_}e_1\texttt{(}t\texttt{)) ->}\\
&\quad E_1[\, x_{1,1}/C_1\texttt{\_proj\_1(}t\texttt{)}, \ldots, x_{1,e_1}/C_1\texttt{\_proj\_}e_1\texttt{(}t\texttt{)}\,] \texttt{) and ...}\\
&\qquad\qquad \texttt{... and (} t \texttt{ = } C_n\texttt{(proj\_}C_n\texttt{\_1(}t\texttt{), ..., proj\_}C_n\texttt{\_}e_n\texttt{(}t\texttt{)) ->}\\
&\qquad\qquad\qquad\qquad E_n[\, x_{n,1}/C_n\texttt{\_proj\_1(}t\texttt{)}, \ldots, x_{n,e_n}/C_n\texttt{\_proj\_}e_n\texttt{(}t\texttt{)}\,] \texttt{)}
\end{aligned}$$

or

$$\begin{aligned}
&\texttt{(} t \texttt{ = } C_1\texttt{(proj\_}C_1\texttt{\_1(}t\texttt{), ..., proj\_}C_1\texttt{\_}e_1\texttt{(}t\texttt{)) and}\\
&\quad E_1[\, x_{1,1}/C_1\texttt{\_proj\_1(}t\texttt{)}, \ldots, x_{1,e_1}/C_1\texttt{\_proj\_}e_1\texttt{(}t\texttt{)}\,] \texttt{) or ...}\\
&\qquad\qquad \texttt{... or (} t \texttt{ = } C_n\texttt{(proj\_}C_n\texttt{\_1(}t\texttt{), ..., proj\_}C_n\texttt{\_}e_n\texttt{(}t\texttt{)) and}\\
&\qquad\qquad\qquad\qquad E_n[\, x_{n,1}/C_n\texttt{\_proj\_1(}t\texttt{)}, \ldots, x_{n,e_n}/C_n\texttt{\_proj\_}e_n\texttt{(}t\texttt{)}\,] \texttt{)}
\end{aligned}$$

The two formulas are equivalent in presence of the inversion axiom and the axioms for D`_match`. The current implementation always chooses the second one. In future, the choice will be driven by the polarity of the match expression's occurrence.

Thus, the definition of the `isEmpty` predicate from Figure 2 is translated into the syntax of ALT-ERGO as follows:

```
predicate isEmpty(l : 'a1 list) =
  (((l = Nil) and true) or
   ((l = Cons(Cons_proj_1(l), Cons_proj_2(l))) and false))
```

And here is the translation of the definition of the `expMax` function from Figure 2:

```
function expMax(r : float) : int =
  FPencoding_match(floatEnc(r), 127, 1023, 16383, 384, 6144)
```

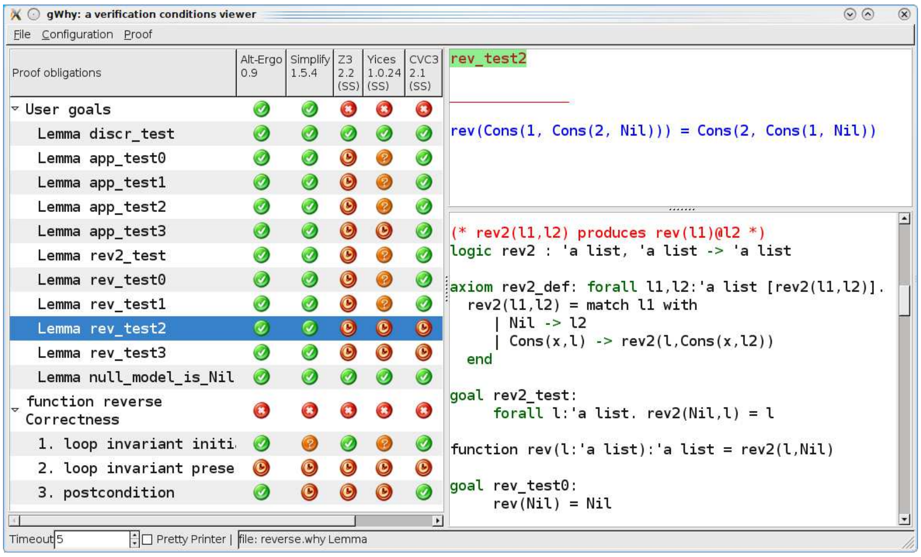
Finally, the `rev2_def` axiom shown at the screen-shot in Figure 4 is rendered by the following declaration:

```
axiom rev2_def :
  (forall l1: 'a1 list. (forall l2: 'a1 list [rev2(l1, l2)].
    (rev2(l1, l2) = list_match(l1, l2,
      rev2(Cons_proj_2(l1), Cons(Cons_proj_1(l1), l2))))))
```

4 Experiments

We made tests on several examples; see `examples/linked_lists/reverse.why` and `lib/why/floats_common.why` in the standard distribution of WHY [4].

In the first example, we introduce the algebraic type of lists (as in Figure 1) and define recursive functions `app` (concatenation of lists) and `rev2` (concatenation with reversal). Since recursive function definitions are not allowed in WHY, we declare `app` and `rev2` as abstract logical symbols and provide suitable axioms. See Figure 4 for the definition of `rev2`. Then we formulate several simple lemmas which do not require induction («User goals» section in Figure 4). With the help of provided triggers, the SMT provers ALT-ERGO and SIMPLIFY



Proof obligations	Alt-Ergo 0.9	Simplify 1.5.4	Z3 2.2 (SS)	Yices 1.0.24 (SS)	CVC3 2.1 (SS)
User goals	✓	✓	✗	✗	✗
Lemma discr_test	✓	✓	✓	✓	✓
Lemma app_test0	✓	✓	⏲	?	✓
Lemma app_test1	✓	✓	⏲	?	✓
Lemma app_test2	✓	✓	⏲	?	✓
Lemma app_test3	✓	✓	⏲	⏲	✓
Lemma rev2_test	✓	✓	⏲	?	✓
Lemma rev_test0	✓	✓	⏲	?	✓
Lemma rev_test1	✓	✓	⏲	?	✓
Lemma rev_test2	✓	✓	⏲	⏲	⏲
Lemma rev_test3	✓	✓	⏲	⏲	⏲
Lemma null_model_is_Nil	✓	✓	✓	✓	✓
function reverse Correctness	✗	✗	✗	✗	✗
1. loop invariant initi	✓	?	✓	?	✓
2. loop invariant prese	⏲	⏲	⏲	⏲	⏲
3. postcondition	✓	⏲	⏲	⏲	✓

Timeout 5 □ Pretty Printer | file: reverse.why Lemma

```
rev_test2

rev(Cons(1, Cons(2, Nil))) = Cons(2, Cons(1, Nil))
```

```
(* rev2(l1,l2) produces rev(l1)@l2 *)
logic rev2 : 'a list, 'a list -> 'a list

axiom rev2_def: forall l1,l2:'a list [rev2(l1,l2)].
  rev2(l1,l2) = match l1 with
      | Nil -> l2
      | Cons(x,l) -> rev2(l,Cons(x,l2))
  end

goal rev2_test:
     forall l:'a list. rev2(Nil,l) = l

function rev(l:'a list):'a list = rev2(l,Nil)

goal rev_test0:
     rev(Nil) = Nil
```

Figure 4: Simple lemmas about lists (GWHY interface)

[5] prove all these lemmas. Also, the CVC3 [6] prover, which does not rely on user-defined triggers, proves 9 of 11 lemmas.

The second example introduces enumerated types representing floating number formats and rounding modes as described in the introduction. Then several constants (e.g. the maximal and minimal representable value) are defined as functions of the number format using a match expression. A number of simple statements invoking these functions are then automatically proved.

5 Future work

We conclude by briefly enumerating the planned enhancements to this work in the upcoming versions of WHY.

Extended matching. The syntax of match expressions is to be generalized to support tuple matching and nested patterns with wildcards:

$$\begin{array}{rcl} \textit{logicExpr} & \rightarrow & \ldots \\ & | & \texttt{match}\ \textit{logicExpr}\ \{\ \texttt{,}\ \textit{logicExpr}\}\ \texttt{with}\ \textit{matchCases}\ \texttt{end} \\ \textit{matchCases} & \rightarrow & [\ \texttt{|}\ ]\ \textit{matchCase}\ \{\ \texttt{|}\ \textit{matchCase}\ \} \\ \textit{matchCase} & \rightarrow & \textit{pattern}\ \{\ \texttt{,}\ \textit{pattern}\}\ \texttt{->}\ \textit{logicExpr} \\ \textit{pattern} & \rightarrow & \textit{ident}\ [\ \texttt{(}\ \textit{pattern}\ \{\ \texttt{,}\ \textit{pattern}\ \}\ \texttt{)}\ ] \\ & | & \texttt{\_} \end{array}$$

Internally, these expressions will be compiled into superpositions of simple match expressions as it is done, e.g. in ML-like languages [7]. The exhaustiveness of matching would still be required.

Recursive functions and predicates. The potential of algebraic data types is quite limited without recursive functions and predicates. Currently, recursive definitions are not supported in Why. Instead, they are simulated with abstract `logic` declarations and appropriate axioms. We plan to extend the definition syntax of Why so as to allow occurrence of a defined symbol in the right-hand part of a definition. Simple forms of structural recursion over algebraic types would be recognized and translated, e.g. for the Coq proof assistant, into `Fixpoint` declarations. Otherwise, if the well-foundness of a definition cannot be established automatically, the definition will be translated into an axiom.

Proofs by induction. When dealing with first-order automated provers that do not support reasoning by induction, Why can be instructed to apply some induction rule before sending the problem to a prover. Specifically, a particular goal in the input file for Why can be annotated with an *induction term* having an algebraic type. In the simplest case, this induction term would be some universally quantified variable in the goal. Then Why can automatically generate the appropriate sub-goals for the base and step cases. When nested induction is needed, the inner statements must be separated into standalone lemmas.

References

[1] A. Ayad and C. Marché. Behavioral properties of floating-point programs. Hisseo publications, 2009. `http://hisseo.saclay.inria.fr/ayad09.pdf`

[2] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris-Sud 11, March 2003.

[3] F. Bobot, S. Conchon, É. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In Barrett and de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008.

[4] Why: a software verification platform. `http://why.lri.fr`

[5] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. Journal of the ACM, 2005

[6] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *CAV '07: 19th International Conference on Computer Aided Verification*, volume 4590 of Lecture Notes in Computer Science, 2007.

[7] Ph. Wadler. Efficient Compilation of Pattern Matching. In Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 6. Prentice-Hall, 1987.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399