

# Implementing Polymorphism in SMT solvers\*

François Bobot      Sylvain Conchon      Evelyne Contejean  
Stéphane Lescuyer

LRI, Univ. Paris-Sud, CNRS, Orsay F-91405  
& INRIA Futurs, ProVal, Orsay, F-91893  
FRANCE

## Abstract

Based on our experience with the development of Alt-Ergo, we show a small number of modifications needed to bring parametric polymorphism to our SMT solver. The first one occurs in the typing module where unification is now necessary for solving polymorphic constraints over types. The second one consists in extending triggers' definition in order to deal with both term and type variables. Last, the matching module must be modified to account for the instantiation of type variables. We hope that this experience is convincing enough to raise interest for polymorphism in the SMT community.

## 1 Introduction

The SMT-LIB [12] provides **ArraysEx**, a generic theory of arrays with extensionality, which introduces three sorts for *indices*, *elements* and *arrays*. Unfortunately, the sorts *indices* and *elements* are not parameters but constant types. Therefore, typing constraints prevent one from using a decision procedure of **ArraysEx** for arrays of integers, even though such a decision procedure would not depend on the sorts of indices and elements. Three other variants are thus provided by SMT-LIB: arrays containing integers (**Int\_ArraysEx**), bit vectors (**BitVector\_ArraysEx**) and arrays of reals (**Int\_Int\_Real\_Array\_ArraysEx**).

This replication issue also occurs with user-defined specifications. For instance, in the context of program verification, the formalization of memory models consists in a set of definitions and axioms which are actually independent of the type of memory cell contents [4, 9].

Polymorphic types [11] are an elegant solution to this problem: they allow a single set of definitions and axioms to be used with different types of data. Aside from the obvious gain of conciseness offered by such factoring of specifications, polymorphism also offers additional expressiveness, for instance the use of type variables as *phantom types* to ensure separation in memory models [6, 13].

In this paper, we show that only a small number of modifications are needed to bring polymorphic types to an SMT solver. In the first part, we detail the example

---

\*Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

of array theory so as to describe our polymorphic logic and its subtleties. In the second part, we describe in turn how the typing and matching mechanisms ought to be modified in order to deal with polymorphic theories. Finally, we argue why adding polymorphism to a solver is a better solution than other existing workarounds.

## 2 Polymorphic First-Order Logic

**A Case Study: Theory of Arrays.** In order to show how polymorphic types can circumvent the duplication of types and axioms, we transform the **ArrayEx** theory of the SMT-LIB given in Alt-Ergo-syntax in Figure 1 into a theory of arrays using polymorphic types (Figure 2). This theory will illustrate how a single set of axioms and definitions can be used to prove goals over several kinds of arrays (containing integers, bitvectors, other arrays, *et caetera*).

---

```

type array
type index
type element

logic select : array, index → element
logic store : array, index, element → array

axiom a1:
  forall a:array. forall i:index. forall e:element.
    select((store(a,i,e),i) = e

axiom a2:
  forall a:array. forall i,j:index. forall e:element.
    i<>j → select(store(a,i,e),j) = select(a,j)

axiom a3:
  forall a,b:array. (forall i:index. select(a,i)=select(b,i)) → a=b

```

---

Figure 1: The SMT-LIB theory of arrays in Alt-Ergo syntax

The first three lines in Figure 1 define the abstract sorts `array`, `index` and `element`. The next two lines define the signature of the usual functions `select` and `store` over arrays: given an array  $a$  and an index  $i$ , `select(a,i)` returns the  $i$ -th element of  $a$ , whereas `store(a,i,e)` returns the array  $a$  updated with  $e$  at index  $i$ .

The abstract nature of `index` and `element` prevents one to use this theory over actual arrays, *e.g.* of integers, since integers are of sort `int` and not `element`. It is desirable that `element` could be replaced by `int` (or any other sort), and similarly for indices. A well-known answer to this requirement is parametric polymorphism [11].

Indeed, it can be noticed that the axioms `a1`, `a2` and `a3` do not depend on the particular nature of indices and elements. As long as the latter can be compared

with respect to equality, they can be seen as black boxes. Informally, this means that the only sort which is specified by the theory is `array`, thus one would rather define a family of arrays parameterized by sorts for indices and elements. In Alt-Ergo syntax, this family is denoted by:

---

```
type ('i,'e) array
```

---

where `'i` and `'e` are type variables representing respectively indices and elements.

The signatures of functions and axioms have to be modified accordingly, by replacing every occurrence of `array` (resp. `index`, resp. `element`) by `('i,'e) array` (resp. `'i`, resp. `'e`). The resulting polymorphic theory is shown in Figure 2.

---

```
type ('i,'e) array
```

```
logic select : ('i,'e) array, 'i → 'e
logic store  : ('i,'e) array, 'i, 'e → ('i,'e) array
```

```
axiom a1 :
  forall a:(('i,'e) array). forall i:'i. forall e:'e.
  select((store(a,i,e),i) = e
```

```
axiom a2 :
  forall a:(('i,'e) array). forall i,j:'i. forall e:'e.
  i<>j → select(store(a,i,e),j) = select(a,j)
```

```
axiom a3 :
  forall a,b:(('i,'e) array). (forall i:'i. select(a,i)=select(b,i)) → a=b
```

---

Figure 2: The polymorphic theory of arrays

This parametric theory of arrays can be instantiated, for example to retrieve SMT-LIB's arrays:

```

ArraysEx      (index, element) array
Int_ArraysEx (int, int) array
BitVector_ArraysEx (bitvector, bitvector) array
Int_Int_Real_Array_ArraysEx (int, (int, real) array) array
```

Moreover, several distinct theory instances can be used conjointly, as illustrated in the following formula:

---

```
goal g1:
  forall i,j:int.
  forall m:(int, (int,int) Array) Array. forall r:(int,int) Array.
  r = select(m,i) → store(m,i,store(r,j,select(r,j))) = m
```

---

This goal involves two different instances of the theory of arrays, and mixes function symbols from both theories. Indeed, in the conclusion of this goal, the outer occurrence of the `store` function belongs to the theory of arrays of integer

arrays (integer matrices), whereas the inner occurrence of `store` belongs to the theory of integer arrays. It is worth noting that there is no syntactic distinction between these two instances of the same parametric function symbol. We explain in Section 3 how the typing system finds the correct instances for each function symbol.

**Polymorphism and its Subtleties.** Up to this point, we remained vague about the meaning of the free type variables in the definition of parametric symbols and polymorphic axioms. In order to highlight the subtleties introduced by polymorphism, we will now *explicitly* denote the *implicit* quantification of these type variables by using the symbol  $\forall$  and type variables instantiation (in terms) by brackets. With these new conventions, the beginning of the arrays theory becomes:

---

```

type :  $\forall 'i, 'e. ('i, 'e) \text{ array}$ 

logic select :  $\forall 'i, 'e. ('i, 'e) \text{ array}, 'i \rightarrow 'e$ 
logic store :  $\forall 'i, 'e. ('i, 'e) \text{ array}, 'i, 'e \rightarrow ('i, 'e) \text{ array}$ 

axiom a1 :
   $\forall 'i, 'e. \text{forall } a:('i, 'e) \text{ array}. \text{forall } i:'i. \text{forall } e:'e.$ 
   $\text{select}_{['i, 'e]}(\text{store}_{['i, 'e]}(a, i, e), i) = e$ 

```

---

The type `array` can be understood as a type family,  $i. e.$  a function yielding one array type for each pair of types  $('i, 'e)$ . Similarly, `select` introduces a function family: for each possible  $'i$  and  $'e$ , it provides a function of type  $('i, 'e) \text{ array}, 'i \rightarrow 'e$ .

The case of axiom `a1` is more informative: all the type variables in this axiom are universally quantified at the outer level of the definition. This outermost type quantification is general in Alt-Ergo and reflects our choice of prenex-polymorphism *a la* ML [10]. This choice will be discussed in Section 3. A very important consequence of this fact, and a major difference with the monomorphic multi-sorted first-order logic of the SMT-LIB, is that an axiom is different from an hypothesis in a goal. This can be seen in the following example:

---

```

type :  $\forall 'c. 'c \text{ t}$ 
logic P :  $\forall 'c. 'c \text{ t} \rightarrow \text{prop.}$ 
axiom a :  $\forall 'c. \text{forall } x: 'c \text{ t}. P_{['c]}(x).$ 
goal g2 :  $\forall 'a, 'b. (\text{forall } x: 'a \text{ t}. P_{['a]}(x)) \wedge (\text{forall } x: 'b \text{ t}. P_{['b]}(x))$ 

```

---

This goal can be easily proved by instantiating axiom `a` once for each type variable  $'a$  and  $'b$ . Now consider putting this axiom as an hypothesis in the goal:

---

```

goal g3 :  $\forall 'a, 'b, 'c.$ 
   $(\text{forall } x: 'c \text{ t}. P_{['c]}(x)) \rightarrow$ 
   $(\text{forall } x: 'a \text{ t}. P_{['a]}(x)) \wedge (\text{forall } x: 'b \text{ t}. P_{['b]}(x))$ 

```

---

The goal is not valid anymore since it is only provable when  $'a, 'b$  and  $'c$  are equal. This is a manifestation of the fact that in general the formulas  $(\forall x, P \Rightarrow Q)$

and  $(\forall x, P) \Rightarrow Q$  are not equivalent. Note, by the way, that even if the goal  $g_3$  is polymorphic, a goal can always be considered monomorphic<sup>1</sup>: for every type variable  $'a$  universally quantified in the goal, it suffices to introduce a fresh ground type  $t_a$  and to substitute  $'a$  by  $t_a$ . Proving this particular instantiation of the goal is equivalent to proving it for any instantiation, since no assumptions are made on the fresh types  $t_a$ .

The theory of polymorphic lists illustrates a new phenomenon which is not observable with arrays:

---

**type**  $\forall 'a. 'a \text{ list}$   
**logic**  $\text{nil} : \forall 'a. 'a \text{ list}$

---

In the above declaration, `nil` is a so-called *polymorphic constant*, that is a family of constants, one for each type. This implies that for each type  $'a$ , the type  $'a \text{ list}$  is inhabited by `nil[ $'a$ ]`, even if  $'a$  is not! A more pernicious declaration is

---

**logic**  $\text{any} : \forall 'a. 'a$

---

which makes *every* type inhabited. Polymorphic constants will have to be dealt with carefully during triggers' definition and matching (*cf.* Section 4).

We have now gained enough understanding about what parametric polymorphism means to be able to give a good intuition of the semantics of polymorphic first-order logic. Suppose we have a polymorphic theory  $\mathbf{T}_{\text{PFOL}}$  and a goal  $\mathbf{G}$ ; we wish to explain what it means for  $\mathbf{T}_{\text{PFOL}}$  to *entail*  $\mathbf{G}$ , in symbols  $\mathbf{T}_{\text{PFOL}} \models_{\text{PFOL}} \mathbf{G}$ . As argued above, we can consider that  $\mathbf{G}$  is monomorphic without restriction. Let  $\mathcal{T}$  be the set of all ground types that can be built from the signature in  $\mathbf{T}_{\text{PFOL}}$  along with an infinitely countable set of arbitrary fresh constant types<sup>2</sup>. Now, let us write  $\mathbf{T}_{\text{FOL}}$  the monomorphic multi-sorted theory such that:

- the set of its types is  $\mathcal{T}$  ;
- its function symbols are all monomorphic instances (with types in  $\mathcal{T}$ ) of the function symbols defined in  $\mathbf{T}_{\text{PFOL}}$ ;
- similarly, its axioms are all the possible monomorphic instances of the axioms in  $\mathbf{T}_{\text{PFOL}}$ .

Given such a theory, we have that  $\mathbf{T}_{\text{PFOL}}$  entails  $\mathbf{G}$  if and only if  $\mathbf{T}_{\text{FOL}}$  entails  $\mathbf{G}$  in monomorphic first-order logic, ie.  $\mathbf{T}_{\text{PFOL}} \models_{\text{PFOL}} \mathbf{G} \Leftrightarrow \mathbf{T}_{\text{FOL}} \models_{\text{FOL}} \mathbf{G}$ . Furthermore, since the proof of  $\mathbf{G}$  in  $\mathbf{T}_{\text{FOL}}$  is finite, only a finite number of monomorphic instances of the definitions in  $\mathbf{T}_{\text{PFOL}}$  are necessary to establish a proof of  $\mathbf{G}$ . Altogether, this means that the task of solving a polymorphic problem amounts to finding the right monomorphic instances of the definitions in the problem and then solving the monomorphic problem we obtain in this manner. The task of generating and finding monomorphic instances is discussed in Sections 4.

---

<sup>1</sup>This means that pushing a polymorphic axiom into a goal makes it monomorphic.

<sup>2</sup>This technical requirement ensures that  $\mathcal{T}$  be non-empty and that polymorphic goals can be monomorphized.

### 3 Type-checking

This section is devoted to the research of proper instances of polymorphic symbols occurring in a monomorphic formula (the case of polymorphic axioms will be treated in the next section). We chose the prenex-version of polymorphism since it is quite expressive and light to handle for a user, compared with polymorphism *a la system F*, which is more powerful, but requires types' annotations for *every* occurrence of polymorphic symbols to ensure a decidable type inference. A possible compromise would be polymorphism *a la ML<sup>F</sup>* [8] where annotations are needed only when *defining* polymorphic symbols, but the whole machinery is much more complex than in the case of prenex-polymorphism.

We illustrate the well-known inference mechanism for prenex-polymorphism [11] on the goal `g1`, in particular on the subterm `select (m, i)`.

1. Since `select` is a polymorphic function, the first step is to build an instance with *fresh* type variables `x1` and `x2`, yielding `select[x1,x2] (m, i)`.
2. The constraints given by the signature of `select` are the following:

$$\text{select}_{[x1, x2]}(m, i) : x2 \quad m : (x1, x2) \text{ array} \quad i : x1$$

Combined with the type annotations of `g1`:

$$m : (\text{int}, (\text{int}, \text{int}) \text{ array}) \text{ array} \quad i : \text{int}$$

we get that `x1 = int` and `x2 = (int, int) array` by unification [1].

Unification of the typing constraints enables finding (most general) instances such that all terms and formulas are well-typed. On the contrary, unification will fail if there is no way to find instances such that the formulas are well-typed. We can remark that the equality has type  $\forall 'a. 'a, 'a \rightarrow \mathbf{prop}$ , so well-typedness ensures that every occurrence of an equality is homogeneous. In our example, the constraint that both terms `r` and `select (m, i)` have the same type is verified.

Introducing fresh type variables in the first step guarantees that different occurrences of the same polymorphic symbol can be instantiated independently, such as the two occurrences of `store` in goal `g1` or, more evidently, the occurrences of `x` in the following example:

---

**type** `'a t`

**logic** `x :  $\forall 'a. 'a t$`

**logic** `f :  $\forall 'a. 'a t \rightarrow 'a$`

**goal** `g4 :  $\forall 'a. f(x_{[\text{int}]}) = 0 \text{ or } f(x_{[\text{real}]})) = 4.5 \text{ or } x_{['a]} = x_{[a]} \text{ or } f(x_{[\text{int}]})) = 3$`

---

The first and last `x` are the same instance `x[int]`, and thus represent the same constant. In `x=x`, the sole constraint on the type of `x` is that it be the same on both sides of the equality symbol. Therefore, both `x` are instantiated on the same fresh

(universally quantified) type variable  $'a$ , and after the monomorphization of the goal, this variable will become a new constant type, distinct from all other types.

After finding the right instances for all symbols, we must keep the types inferred in the abstract syntax tree (AST) of formulas, in order to be able to use this type information in the *matching* process, as explained in the next section.

## 4 Polymorphic Triggers and Matching

This section is illustrated by the theory of polymorphic lists, defined by

---

```

type  $\forall 'a. 'a \text{ list}$ 
logic  $\text{nil} : \forall 'a. 'a \text{ list}$ 
logic  $\text{cons} : \forall 'a. 'a, 'a \text{ list} \rightarrow 'a \text{ list}$ 
logic  $\text{length} : \forall 'a. 'a \text{ list} \rightarrow \text{int}$ 

axiom  $11 : \forall 'a. \text{length}_{[a]}(\text{nil}_{[a]}) = 0$ 
axiom  $12 : \forall 'a. \text{forall } x:'a. \text{forall } l:'a \text{ list.}$ 
            $\text{length}_{[a]}(\text{cons}_{[a]}(x, l)) = \text{length}_{[a]}(l) + 1$ 

```

---

In order to prove the goal  $g : \text{length}_{[\text{int}]}(\text{cons}_{[\text{int}]}(3, \text{nil}_{[\text{int}]})) = 1$ , a human being will first instantiate 12 by  $\mu_1 = \{ 'a \mapsto \text{int} \}$  and  $\sigma_1 = \{ x \mapsto 3, l \mapsto \text{nil}_{[\text{int}]} \}$ . Then (s)he is left with  $\text{length}_{[\text{int}]}(\text{nil}_{[\text{int}]}) + 1 = 1$ , which can be proved by using 11 instantiated by  $\mu_2 = \{ 'a \mapsto \text{int} \}$ . An SMT solver needs to have a similar mechanism in order to "infer" the useful instances. This mechanism is based on so-called *triggers*, as presented in [3]. In this particular example, the solver "knows" a set of ground terms  $\Gamma$ , and terms are equipped with their type:

$$\Gamma = \text{length}_{[\text{int}]}(\text{cons}_{[\text{int}]}(3, \text{nil}_{[\text{int}]})) : \text{int}, \text{cons}_{[\text{int}]}(3, \text{nil}_{[\text{int}]}) : \text{int list}, \\ 1, 3 : \text{int}, \text{nil}_{[\text{int}]} : \text{int list}$$

Given a lemma and its trigger, the matching module of Alt-Ergo searches  $\Gamma$  for an instance of the trigger (possibly modulo the equalities discovered so far), and applies the resulting substitutions for type and term variables to the lemma, yielding a ground *fact* which can feed the SAT-solver.

Assume that we have chosen as trigger for 12 its subterm  $\text{cons}_{[a]}(x, l)$ , a possible instance is  $\text{cons}_{[\text{int}]}(3, \text{nil}_{[\text{int}]})$ , and the substitutions are  $\mu_1$  and  $\sigma_1$ . The matching module thus has to instantiate both type and term variables. When there are polymorphic constants, such as *any* or *nil*, although they are in a sense ground terms since they do not contain any *term* variables, they are not fully ground since their type may contain *type* variables. This explains why triggers can be terms without (term) variables. In particular, our axiom 11 looks like it is a fact already, but it actually has to be instantiated before being fed to the SAT-solver. In our example, if the trigger of 11 is  $\text{length}_{[a]}(\text{nil}_{[a]})$ , the solver can compute the instance  $\mu_2$ .

We now turn to a more formalized definition of matching. Let us assume a set of *ground terms*  $\Gamma$  along with their types, for which we write  $t : \tau \in \Gamma$  when

a ground term  $t$  of ground type  $\tau$  belongs to  $\Gamma$ . We also consider an *equivalence relation*  $\Delta$  on terms in  $\Gamma$ , representing the equalities discovered so far in the context of the solver. Given these sets, we first define a few notions, and go on to describing the matching algorithm itself:

- a *trigger* is a term where variables and constants are decorated with (possibly polymorphic) types: namely, we write  $x^{\tau[\bar{\alpha}]}$  (resp.  $c^{\tau[\bar{\alpha}]}$ ) when the variable  $x$  (resp. the constant  $c$ ) is annotated with a type  $\tau$  and  $\bar{\alpha}$  are the free type variables in  $\tau$ ; given a trigger  $p$ , we write  $\mathcal{V}(p)$  to denote the variables of  $p$ , and  $\mathcal{TV}(p)$  the set of all type variables appearing in the variables of  $p$ ;
- a *substitution*  $\sigma$  mapping variables  $x_1, \dots, x_n$  to terms  $t_1, \dots, t_n$  in  $\Gamma$  is denoted  $\{x_1 \mapsto t_1; \dots; x_n \mapsto t_n\}$ ; its *support*  $Supp(\sigma)$  denotes the set of variables  $x_1, \dots, x_n$ . Substitutions  $\sigma_1, \dots, \sigma_n$  are said to be  $\Delta$ -*compatible* if the following holds:

$$\forall x_i, x, x \in Supp(\sigma_i) \cap Supp(\sigma_j) \Rightarrow \sigma_i(x) =_{\Delta} \sigma_j(x);$$

- a *type substitution*  $\mu$  mapping some type variables  $\bar{\alpha}$  to ground types  $\bar{\tau}$  is denoted  $\{\bar{\alpha} \mapsto \bar{\tau}\}$ , and its support  $Supp(\mu)$  is defined as above; type substitutions  $\mu_1, \dots, \mu_n$  are said to be *compatible* if the following holds:

$$\forall \alpha_i, \alpha, \alpha \in Supp(\mu_i) \cap Supp(\mu_j) \Rightarrow \mu_i(\alpha) = \mu_j(\alpha).$$

Given  $n$  such compatible type substitutions, we write  $\biguplus \mu_i$  the merging of these substitutions; similarly, we write  $\biguplus \sigma_i$  the merging of  $\Delta$ -compatible term substitutions.

The *matching function*  $\mathcal{M}$  depends on  $\Gamma$  and  $\Delta$ , ie. on the current ground environment of the prover. Given a trigger  $p$ ,  $\mathcal{M}(p)$  is a set of tuples  $(t, \sigma, \mu)$  where  $t \in \Gamma$ ,  $\sigma$  is a substitution from  $\mathcal{V}(p)$  to  $\Gamma$ , and  $\mu$  a type substitution from  $\mathcal{TV}(p)$  to ground types, such that  $p[\mu]\sigma =_{\Delta} t$ . The computation of  $\mathcal{M}(p)$  can be recursively defined on the structure of the trigger  $p$ , in the following way:

$$\begin{aligned} \mathcal{M}(x^{\tau[\bar{\alpha}]}) &= \{(t, \{x \mapsto t\}, \{\bar{\alpha} \mapsto \bar{\tau}\}) \mid t : \tau[\bar{\tau}] \in \Gamma\} \\ \mathcal{M}(c^{\tau[\bar{\alpha}]}) &= \{(c_{[\bar{\tau}]}, \{\}, \{\bar{\alpha} \mapsto \bar{\tau}\}) \mid c_{[\bar{\tau}]} : \tau[\bar{\tau}] \in \Gamma\} \\ \mathcal{M}(f(p_1, \dots, p_n)) &= \left\{ (t, \biguplus \sigma_i, \biguplus \mu_i) \left| \begin{array}{l} t = f(t_1, \dots, t_n) \in \Gamma \\ \forall i, (t_i, \sigma_i, \mu_i) \in \mathcal{M}(p_i) \\ \sigma_1, \dots, \sigma_n \Delta\text{-compatible} \\ \mu_1, \dots, \mu_n \text{ compatible} \end{array} \right. \right\} \end{aligned}$$

It can be shown that if  $(t, \sigma, \mu) \in \mathcal{M}(p)$ , all variables in  $\mathcal{V}(p)$  are instantiated in  $\sigma$  and similarly, all type variables in  $\mathcal{TV}(p)$  are instantiated in  $\mu$ . In other words,  $t$  is indeed a ground and monomorphic instance of the original pattern  $p$ . This definition is not computationally efficient and in an actual implementation, better strategies can be used:

- when matching a pattern  $f(p_1, \dots, p_n)$ , it is possible to retrieve all the terms of the form  $f(t_1, \dots, t_n)$  in  $\Gamma$  and to match the sub-patterns  $p_i$  only against the relevant  $t_i$ ;
- the substitutions can be built incrementally through the matching process in order to only check compatibility at the variables' level and to avoid the cost of merging substitutions.

Finally, it is straightforward to extend our definition of matching to the case of *multi-triggers*, by considering each trigger in turn and discarding incompatible substitutions.

## 5 Discussion and Conclusion

So far, we have seen how polymorphism had been added to the logic of our SMT solver Alt-Ergo. It is worth noting that, as argued in Section 2, provided that the instantiation mechanism can generate the monomorphic instances of polymorphic axioms, the task of solving a particular problem comes down to solving a monomorphic problem. The important consequence of this remark is that no other modification to our solver was required: in particular, the congruence closure mechanism, the SAT solver or the built-in decision procedures for theories such as linear arithmetic needed not be changed since they would always be used on ground monomorphic terms and formulas. Another consequence is that our approach does not require the combination of parametric theories as presented in [7].

Although it seems reasonably easy to add parametric polymorphism to an SMT solver, one may wonder if it is up to a solver to manipulate logics with complicated features. Indeed, SMT solvers are typically used to certify problems that have been automatically generated by systems such as *verification condition generators* (VCG). It could be argued that even if more refined logics are desirable in such systems (so as to achieve more expressiveness and more compact specifications of programs), the additional burden should be borne by VCG themselves. We have actually studied and used such alternatives in order to employ multi sorted or unsorted provers with the Why toolkit [5]. In [2], the authors show that the problem of encoding polymorphic first-order logic in a weaker logic is not an easy one: intuitive solutions happen to be either impractical, or incorrect. They also show that there is no evidence that it is always possible to statically generate all the monomorphic instances necessary to solve a problem. The encodings they propose provide a better solution, but still have some shortcomings: less easy to implement, they generate formulas that are bigger than the original ones, and require additional lemmas to smoothly deal with built-in decision procedures. All in all, this results in a loss of efficiency on the solver's side.

We have presented our experience of implementing polymorphism in our SMT solver Alt-Ergo. We showed that adding parametric types to the logic required changing the typing system and the matching mechanism in order to account for

type variables. Because the extent of these modifications is relatively moderate, we are convinced that they are worth the extra cost in design and engineering. While not suffering from the same disadvantages as other solutions such as encodings, they bring the expressiveness and conciseness of polymorphism over to the SMT solver. For that reason, we hope that in the future, polymorphism could be added to the SMT-LIB standard.

## References

- [1] F. Baader and W. Snyder. Unification Theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 447–533. Elsevier Science, 2001.
- [2] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.
- [3] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
- [4] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM*, pages 15–29, 2004.
- [5] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590, pages 173–177, Berlin, Germany, July 2007.
- [6] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, Mar. 2007. <http://www.lri.fr/~marche/hubert07hav.pdf>.
- [7] S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.
- [8] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, Aug. 2003.
- [9] K. R. M. Leino and A. Wallenburg. Class-local object invariants. In *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, pages 57–66, New York, NY, USA, 2008. ACM.
- [10] R. Milner. A theory of type polymorphism programming. *J. Comput. Syst. Sci.*, 17, 1978.
- [11] B. C. Pierce. *Types and Programming Languages*, chapter V. MIT Press, 2002.
- [12] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). <http://www.smtcomp.org>, 2006.
- [13] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 97–108, Nice, France, Jan. 2007.