

Implementing Polymorphism in SMT solvers

François Bobot, Sylvain Conchon
Evelyne Contejean, Stéphane Lescuyer

SMT Workshop 2008



how to **define** and **use** a generic theory within SMT-Lib?

- theories are made generic by introducing abstract datatypes;
→ these types cannot be instantiated
- one copy for each instance type
→ **only one** copy of built-in theories per file

example: the theory ArraysEx (1/2)

```
(theory ArraysEx
:sorts (Index Element Array)
:funs ((select Array Index Element)
      (store Array Index Element Array))
:axioms (
  (forall (?a Array) (?i Index) (?e Element)
    (= (select (store ?a ?i ?e) ?i) ?e))
  (forall (?a Array) (?i Index) (?j Index) (?e Element)
    (or (= ?i ?j)
      (= (select (store ?a ?i ?e) ?j) (select ?a ?j))))
  (forall (?a Array) (?b Array)
    (implies
      (forall (?i Index) (= (select ?a ?i) (select ?b ?i)))
      (= ?a ?b)))
))
```

example: the theory `ArraysEx` (2/2)

particular instances:

- `Int_ArraysEx` arrays of integers;
- `BitVector_ArraysEx` arrays of bitvectors;
- `Int_Int_Real_Array_ArraysEx` real matrices;

example: the theory `ArraysEx` (2/2)

particular instances:

- `Int_ArraysEx` arrays of integers;
- `BitVector_ArraysEx` arrays of bitvectors;
- `Int_Int_Real_Array_ArraysEx` real matrices;

what about real arrays, string arrays, integer matrices etc. ?

example: the theory `ArraysEx` (2/2)

particular instances:

- `Int_ArraysEx` arrays of integers;
- `BitVector_ArraysEx` arrays of bitvectors;
- `Int_Int_Real_Array_ArraysEx` real matrices;

what about real arrays, string arrays, integer matrices etc. ?

what about arrays of user-defined datatypes ?

example: the theory `ArraysEx` (2/2)

particular instances:

- `Int_ArraysEx` arrays of integers;
- `BitVector_ArraysEx` arrays of bitvectors;
- `Int_Int_Real_Array_ArraysEx` real matrices;

what about real arrays, string arrays, integer matrices etc. ?

what about arrays of user-defined datatypes ?

how to mix different instances in a single file?

achieving genericity through polymorphism

- overloading (ad hoc polymorphism)
 - instances can be mixed but have to be manually defined and replication remains an issue

achieving genericity through polymorphism

- overloading (ad hoc polymorphism)
 - instances can be mixed but have to be manually defined and replication remains an issue
- parameterized modules (templates, functors, generics)
 - user-defined theories are written once but explicit module application is required
 - different instances have different names

achieving genericity through polymorphism

- overloading (ad hoc polymorphism)
 - instances can be mixed but have to be manually defined and replication remains an issue
- parameterized modules (templates, functors, generics)
 - user-defined theories are written once but explicit module application is required
 - different instances have different names
- system F
 - built-in and user-defined theories are written once but each application function may have to be manually annotated (undecidability of type inference)

achieving genericity through polymorphism

- overloading (ad hoc polymorphism)
 - instances can be mixed but have to be manually defined and replication remains an issue
- parameterized modules (templates, functors, generics)
 - user-defined theories are written once but explicit module application is required
 - different instances have different names
- system F
 - built-in and user-defined theories are written once but each application function may have to be manually annotated (undecidability of type inference)
- ML parametric polymorphism (Why/Alt-Ergo)
 - restricted form of system F where type inference is decidable: theories are defined once and no annotations from the user is required

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

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

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

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

ArraysEx (index, element) array

Int_ArraysEx (int, int) array

BitVector_ArraysEx (bitvector, bitvector) array

Int_Int_Real_Array_ArraysEx (int, (int, real) array) array

ArraysEx (index, element) array

Int_ArraysEx (int, int) array

BitVector_ArraysEx (bitvector, bitvector) array

Int_Int_Real_Array_ArraysEx (int, (int, real) array) array

- these different instances can be used in the same formula:

goal g:

$\forall i, j: \text{int.}$

$\forall m: (\text{int}, (\text{int}, \text{int}) \text{ Array}) \text{ Array.}$

$\forall r: (\text{int}, \text{int}) \text{ Array.}$

$r = \text{select}(m, i) \rightarrow \text{store}(m, i, \text{store}(r, j, \text{select}(r, j))) = m$

ArraysEx (index, element) array

Int_ArraysEx (int, int) array

BitVector_ArraysEx (bitvector, bitvector) array

Int_Int_Real_Array_ArraysEx (int, (int, real) array) array

- these different instances can be used in the same formula:

goal g:

$\forall i, j: \text{int.}$

$\forall m: (\text{int}, (\text{int}, \text{int}) \text{ Array}) \text{ Array.}$

$\forall r: (\text{int}, \text{int}) \text{ Array.}$

$r = \text{select}(m, i) \rightarrow \text{store}(m, i, \text{store}(r, j, \text{select}(r, j))) = m$

- retrieving the right instances is done during typechecking

how to handle polymorphism in an SMT solver?

we experimented the three following methods

- monomorphization
- type encoding
- built-in polymorphism

monomorphization (1/2)

the goal to be proved can always be considered monomorphic

we statically

- find the instances of polymorphic datatypes needed to prove the goal
- replicate symbols and lemmas' definitions for each instance

monomorphization (1/2)

the goal to be proved can always be considered monomorphic

we statically

- find the instances of polymorphic datatypes needed to prove the goal
- replicate symbols and lemmas' definitions for each instance

for example, in order to prove

`goal g:`

`∀i,j:int.`

`∀m:(int,(int,int) Array) Array.∀r:(int,int) Array.`

`r = select(m,i) → store(m,i,store(r,j,select(r,j))) = m`

one has to generate instances of lemmas a1, a2 and a3 for the types

- `(int,(int,int) Array) Array`
- `(int,int) Array`

this method raises several problems

this method raises several problems

- termination/completeness
 - how to find the right instances?
 - how to avoid generating useless instances?
 - is it always possible?

this method raises several problems

- termination/completeness
 - how to find the right instances?
 - how to avoid generating useless instances?
 - is it always possible?
- size blow-up
 - extra lemmas and definitions can slow down proof search
 - does not scale at all in presence of phantom types
 - more than linear when axioms involve several type variables

encoding polymorphic first-order formulas in multi-sorted (or unsorted) logics

encoding polymorphic first-order formulas in multi-sorted (or unsorted) logics

- erasing types
 - incorrect with finite types

encoding polymorphic first-order formulas in multi-sorted (or unsorted) logics

- erasing types
 - incorrect with finite types
- types as predicates
 - strongly disturb SMT solvers (lots of disjunctions are introduced)

encoding polymorphic first-order formulas in multi-sorted (or unsorted) logics

- erasing types
 - incorrect with finite types
- types as predicates
 - strongly disturb SMT solvers (lots of disjunctions are introduced)
- types as terms
 - bigger terms (not human-readable anymore)
 - disturb trigger selection
 - example: a variable trigger x now becomes a non-variable trigger $\text{sort}(x, t)$
 - requires administrative lemmas

adding polymorphism *a la* ML in an SMT solver amounts to

adding polymorphism *a la* ML in an SMT solver amounts to

- 1 modifying its type system

adding polymorphism *a la* ML in an SMT solver amounts to

- 1 modifying its type system
- 2 extending its matching module and its trigger selection

adding polymorphism *a la* ML in an SMT solver amounts to

- 1 modifying its type system
- 2 extending its matching module and its trigger selection

the quantifier free kernel is left **unchanged**

typing issues

a couple of subtleties

a couple of subtleties

- prenex polymorphism \Rightarrow axioms are different from hypotheses
for instance, the following goal g can be proved by instantiating axiom a on type variables $'a$ and $'b$

```
axiom a :  $\forall x: 'c \text{ t.P}(x)$ .
```

```
goal g :  $(\forall x: 'a \text{ t.P}(x))$  and  $(\forall x: 'b \text{ t.P}(x))$ 
```

but putting the same axiom as an hypothesis prevent from proving g

```
goal g :
```

```
 $(\forall x: 'c \text{ t.P}(x)) \rightarrow (\forall x: 'a \text{ t.P}(x))$  and  $(\forall x: 'b \text{ t.P}(x))$ 
```

a couple of subtleties

- prenex polymorphism \Rightarrow axioms are different from hypotheses
for instance, the following goal g can be proved by instantiating axiom a on type variables $'a$ and $'b$

```
axiom a :  $\forall x: 'c$  t.P(x).
```

```
goal g : ( $\forall x: 'a$  t.P(x)) and ( $\forall x: 'b$  t.P(x))
```

but putting the same axiom as an hypothesis prevent from proving g

```
goal g :
```

```
( $\forall x: 'c$  t.P(x))  $\rightarrow$  ( $\forall x: 'a$  t.P(x)) and ( $\forall x: 'b$  t.P(x))
```

- a polymorphic constant (e.g. `logic nil : 'a list`) represents a family of constants, one for each type
 - \rightarrow thus `'a list` is inhabited even if `'a` is not
 - \rightarrow the declaration `logic any: 'a` makes every type inhabited

matching issues

- terms are equipped with their types

- terms are equipped with their types
- triggers covers both term and type variables

- terms are equipped with their types
- triggers covers both term and type variables
- the matching algorithm has to instantiate both type and term variables

- terms are equipped with their types
- triggers covers both term and type variables
- the matching algorithm has to instantiate both type and term variables

a subtlety: triggers can now be terms without term variables

```
type 'a list
logic nil : 'a list
logic length : 'a list → int
```

```
axiom l1 : length(nil) = 0
```

axiom l1 is not a *monomorphic* fact, but a *polymorphic* lemma whose triggers can be either `nil` or `length(nil)`

polymorphism offers conciseness and additional expressiveness

- different kind of genericity
- several implementation issues
- some typing or matching subtleties

polymorphism offers conciseness and additional expressiveness

- different kind of genericity
- several implementation issues
- some typing or matching subtleties

adding parametric polymorphism in Alt-Ergo has required

- changing the typing system
- modifying the trigger selection mechanism
- extending the matching algorithm

polymorphism offers conciseness and additional expressiveness

- different kind of genericity
- several implementation issues
- some typing or matching subtleties

adding parametric polymorphism in Alt-Ergo has required

- changing the typing system
- modifying the trigger selection mechanism
- extending the matching algorithm

but no modification of the quantifier free part!