SMTtoTPTP -
A Converter for Theorem Proving Formats

Peter Baumgartner

NICTA

Australian National University

Introduction

TPTP (Thousands of Problems for Theorem Proving)

Languages: clause logic, [typed]FOL[+arithmetics], HOL

Problem library: $>$ 20k problems

Infrastructure: utilities, solutions to problems

SMT-LIB

Language: sorted FOL $+$ background theories (e.g., arithmetics, arrays)

Problem library: $>$ 100k problems

Infrastructure: utilities

SMTtoTPTP

Translation SMT-LIB problems $\Rightarrow$ TPTP problems

Who benefits?

(Remark: "sort" $=$ "type" in this talk)

Who Benefits?

Maintainers of TPTP problem collections

SMTtoTPTP makes it easy to add existing SMT-LIB benchmarks to TPTP

Developers of TPTP theorem provers

SMTtoTPTP provides a front-end for problems written in SMT-LIB

Users of SMT solvers

SMTtoTPTP provides the link to (also) use TPTP theorem provers

Rest of this talk

Example SMT-LIB ⇒ TPTP transformation

SMTtoTPTP algorithm

SMT-LIB Scripts

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S) (Pair Int S))

(declare-fun get-int ((Int-Pair Color))
  Int)

(declare-fun int-color-pair (Int Color)
  (Pair Int Color))

(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))

(check-sat)
```

SMT-LIB Scripts

Uninterpreted function symbols + LIA

```
(set-logic UFLIA)
(declare-sort Color 0)
(declare-fun red () Color)
(declare-sort Pair 2)
(define-sort Int-Pair (S) (Pair Int S))
(declare-fun get-int ((Int-Pair Color))
  Int)
(declare-fun int-color-pair (Int Color)
  (Pair Int Color))
(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))
(check-sat)
```

SMT-LIB Scripts

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S) (Pair Int S))

(declare-fun get-int ((Int-Pair Color))
  Int)

(declare-fun int-color-pair (Int Color)
  (Pair Int Color))

(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))

(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

SMT-LIB Scripts

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S) (Pair Int S))

(declare-fun get-int ((Int-Pair Color))
  Int)

(declare-fun int-color-pair (Int Color)
  (Pair Int Color))

(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))

(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

Color-constant red

SMT-LIB Scripts

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S) (Pair Int S))

(declare-fun get-int ((Int-Pair Color))
  Int)

(declare-fun int-color-pair (Int Color)
  (Pair Int Color))

(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))

(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

Color-constant red

2-ary sort Pair

SMT-LIB Scripts

```
(set-logic UFLIA)
(declare-sort Color 0)
(declare-fun red () Color)
(declare-sort Pair 2)
(define-sort Int-Pair (S) (Pair Int S))
(declare-fun get-int ((Int-Pair Color))
  Int)
(declare-fun int-color-pair (Int Color)
  (Pair Int Color))
(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))
(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

Color-constant red

2-ary sort Pair

Macro Sort $\mapsto$ Sort

SMT-LIB Scripts

```
(set-logic UFLIA)
(declare-sort Color 0)
(declare-fun red () Color)
(declare-sort Pair 2)
(define-sort Int-Pair (S) (Pair Int S))
(declare-fun get-int ((Int-Pair Color))
  Int)
(declare-fun int-color-pair (Int Color)
  (Pair Int Color))
(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))
(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

Color-constant red

2-ary sort Pair

Macro Sort $\mapsto$ Sort

get-int:

(Pair Int Color) $\mapsto$ Int

SMT-LIB Scripts

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S) (Pair Int S))

(declare-fun get-int ((Int-Pair Color))
  Int)

(declare-fun int-color-pair (Int Color)
  (Pair Int Color))

(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))

(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

Color-constant red

2-ary sort Pair

Macro Sort $\mapsto$ Sort

`get-int`:

`(Pair Int Color)` $\mapsto$ `Int`

(Well-sorted) input formula

SMT-LIB Scripts

```
(set-logic UFLIA)
(declare-sort Color 0)
(declare-fun red () Color)
(declare-sort Pair 2)
(define-sort Int-Pair (S) (Pair Int S))
(declare-fun get-int ((Int-Pair Color))
  Int)
(declare-fun int-color-pair (Int Color)
  (Pair Int Color))
(assert (forall ((i Int) (c Color))
    (= (get-int (int-color-pair i c)) i)))
(check-sat)
```

Uninterpreted function symbols + LIA

0-ary sort Color

Color-constant red

2-ary sort Pair

Macro Sort $\mapsto$ Sort

get-int:

(Pair Int Color) $\mapsto$ Int

(Well-sorted) input formula

Translation into TPTP? Compatibility with TPTP format?

SMT-LIB ⇒ TPTP: (In)Compatibilities

(✓ = compatible ✗ = incompatible)

Sorts

✓ SMT-LIB arithmetic sorts ≈ TPTP arithmetic sorts

✗ SMT-LIB: n-ary user sorts ≠ TPTP: 0-ary user sorts

Overloaded operators

✓ SMT-LIB equality = TPTP equality

$=$: S × S ↦ `Bool` for any sort S

✓ SMT-LIB arithmetic operators ≈ TPTP arithmetic operators

✗ SMT-LIB overloaded array operators (predefined)

`(declare-sort Array 2)`

`select`: (`Array` S T) × S ↦ T for any sorts S and T

`store`: (`Array` S T) × S × T ↦ (`Array` S T)

⇒ It is the types that require the most attention in the transformation

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
        (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Color', type, 'Color': $tType).


tff('Pair', type,
    'Pair[Int,Color]': $tType).


tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).


tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').


tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
          (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Color', type, 'Color': $tType).

tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Constant red: unused hence forget

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
        (int-color-pair i c))
     i)))

(check-sat)
```

```
ype).

tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

Color → 'Color'

Instance (Pair Int Color) → 'Pair[Int,Color]'

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

Color → 'Color'

Instance (Pair Int Color) → 'Pair[Int,Color]'

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
        (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
          (int-color-pair i c))
      i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

Color → 'Color'

Instance (Pair Int Color) → 'Pair[Int,Color]'

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
      i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

Color → 'Color'

Instance (Pair Int Color) → 'Pair[Int,Color]'

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
       (get_int(int_color_pair(I, C))
         = I))).
```

Example SMT-LIB ⇒ TPTP

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
   ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

Example SMT-LIB ⇒ TPTP

```
(set-logic UFLIA)

(declare-sort Color 0)

(declare-fun red () Color)

(declare-sort Pair 2)

(define-sort Int-Pair (S)
  (Pair Int S))

(declare-fun get-int
  ((Int-Pair Color)) Int)

(declare-fun int-color-pair
  (Int Color) (Pair Int Color))

(assert
  (forall ((i Int) (c Color))
    (= (get-int
         (int-color-pair i c))
     i)))

(check-sat)
```

Ignore

Color → ‘Color’

Instance (Pair Int Color) → ‘Pair[Int,Color]’

```
tff('Pair', type,
    'Pair[Int,Color]': $tType).

tff(get_int, type, get_int:
    'Pair[Int,Color]' > $int).

tff(int_color_pair, type, int_color_pair:
    ($int * 'Color') > 'Pair[Int,Color]').

tff(formula, axiom,
  ( ! [I:$int, C:'Color'] :
      (get_int(int_color_pair(I, C))
        = I))).
```

SMTtoTPTP Algorithm

(1) Abstract syntax tree (AST)

Input SMT-LIB commands are parsed into AST

- Scala parser combinators library
- ASTs over Scala classes for Declarations, definitions, assertions etc

If arrays are needed (e.g. via (`set-logic` AUFLIA)) add declarations

```
(declare-sort Array 2)
(declare-parametric-fun (I E) select ((Array I E) I) E)
(declare-parametric-fun (I E) store ((Array I E) I E) (Array I E))
```

`declare-parametric-fun` ?

- Not an SMT-LIB command, but OK, as hidden from user
- Useful also for datatypes, see below

SMTtoTPTP Algorithm

(2) Semantic analysis

Decompose commands into their constituents

Result: various Scala tables related to input signature

- Declared/defined sorts, arities of declared/defined fns

These tables make it easy to compute the sort of any subterm in any assertion

SMTtoTPTP Algorithm

(3) Transformations

(1) Defined functions by introducing equations

```
(define-fun inc ((i Int)) Int (+ i 1)) →
tff(inc, axiom, ! [i:$int] : (inc(i) = $sum(i, 1)))
```

(Alternatively could expand terms with defined functions)

(2) Let-terms

Let σ(t) be the sort of term t

Replace let-term by ∃-quantification in smallest Bool-sorted context

```
(assert ( … ( … (let ((x t)) s) … ) … )) →
(assert ( … (exists ((x σ(t))) (and (= x t) (… s …))) … ))
```

Not shown above: renaming of x for avoiding unintended binding

If σ(t) = Bool instead replace let-term by expansion

SMTtoTPTP Algorithm

(3) Transformations

(3) If-then-else terms (ITE)

User option:

Translation into TPTP ITE

OR

Expansion

```
(< (+ (ite (< 1 2) 3 4) 5) 6) ⇝

(and
  (=>
    (< 1 2)
    (< (+ 3 5) 6))
  (=>
    (not (< 1 2))
    (< (+ 4 5) 6)))
```

SMTtoTPTP Algorithm

(3) Transformations

(4) Arrays (not predefined in TPTP)

```
(declare-fun a1 () (Array Color Int))
(declare-fun a2 () (Array Int Int))
...
```

Add standard axioms, incl equality, for all used sort instances

```
(forall ((a (Array Color Int)) (i Color) (e Int))
  (= (select (store a i e) i) e))

(forall ((a (Array Int Int)) (i Int) (e Int))
  (= (select (store a i e) i) e))

...
```

SMTtoTPTP Algorithm

(4) TPTP Generation

Main Problem: overloaded operators

Multiple sort-instances of f-terms, e.g., `(select a1 red)` `(select a2 1)`

Cannot simply use `select` as a (monomorphic) TPTP identifier

Solution: monomorphization

Suppose SMT-LIB term $t = (f\ t_1 \cdots t_n)$

Translation $f \Rightarrow f^{TFF}$ where $\sigma(t)$ is the sort of t

Append argument/result sorts: $f^{TFF} =$ ‘$f{:}\sigma(t_1){*} \cdots {*}\sigma(t_n){>}\sigma(t)$’

Add declaration `tff(`f`, type, `f^{TFF}`: (`$\sigma(t_1)$` * ··· * `$\sigma(t_n)$`) > `$\sigma(t)$`).`

Now t can be recursively transformed into TPTP, e.g.,

```
'select:Array[Color,Int]*Color>Int'(a1, red)
'select:Array[Int,Int]*Int>Int'(a2, 1)
```

SMTtoTPTP Algorithm

(4) TPTP Generation miscellaneous

- No type inference, sometimes explicit coercion is needed

  Instead of empty list `nil` use coerced version `(as nil (List Int))`

- SMT-LIB and TPTP identifiers are rather different (unpleasant)

- SMT-LIB operator annotations `chainable`, `associative` and `pairwise` are respected. E.g., $=$ is `chainable`

  $$(= t_1 \cdots t_n) \;\rightsquigarrow\; (\mathrm{and}\ (= t_1\ t_2) \cdots (= t_{n-1}\ t_n))$$

- SMT-LIB equations between `Bool`-sorted terms are turned into bi-implications

Limitations and Extensions

Unsupported

Logic: bit vector

Tokens: hexadecimal, binary, string, indexed identifier `(_ a 5)`

Commands: ignored: get-proof, check-sat, ... error: push, pop

Extension: Z3-style datatypes

```
(declare-datatypes () ((Color red green blue)))
(declare-datatypes (S T) ((Pair (mk-pair (first S) (second T)))))
(declare-datatypes (T) ((List nil (insert (head T)
                                          (tail (List T))))))
```

Parametric function declarations and axioms for constructors, destructors etc are added automatically

Availability

GPL'ed source/jar at https://bitbucket.org/peba123/smttotptp