Satisfiability Modulo Theories
Lecture 9 - Extending OpenSMT for fun and profit*

(slides revision: Saturday 14$^{\text{th}}$ March, 2015, 11:47)

Roberto Bruttomesso

Seminario di Logica Matematica
(Corso Prof. Silvio Ghilardi)

22 Dicembre 2011

Outline

1. A simple logic
2. Phase 0 - Compiling OpenSMT
3. Phase 1 - Setting up files and directories
4. Phase 2 - Connecting the $\mathcal{T}$-solver
5. Phase 3 - Implementing the $\mathcal{T}$-solver

A simple logic

For this tutorial we use a theory that we call "simple order" ($\mathcal{SO}$)

Atoms of this theory are of the form

$$x \leq y$$

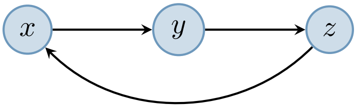
but $\leq$ it is just a symbol to intend "x is before y"

A set of constraints is unsatisfiable iff there is a cycle

$$x \leq y,\ y \leq z,\ \ldots,\ w \leq x$$

A model is therefore is any "acyclic" set of constraints

$x \leq y$
$y \leq z$
$z \leq x$



unsat

A simple logic

For this tutorial we use a theory that we call “simple order” ($\mathcal{SO}$)

Atoms of this theory are of the form

$$x \leq y$$

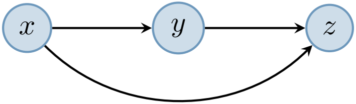
but $\leq$ it is just a symbol to intend “x is before y”

A set of constraints is unsatisfiable iff there is a cycle

$$x \leq y,\ y \leq z,\ \ldots,\ w \leq x$$

A model is therefore is any “acyclic” set of constraints

$x \leq y$
$y \leq z$
$x \leq z$



sat

Phase 0 - Compiling OpenSMT for the first time

```
$ autoreconf --install [--force]

$ mkdir debug

$ cd debug

debug $ ../configure --disable-optimization
                     [--with-gmp=/opt/local]

debug $ make [-j4]
debug $ cd ..
```

Phase 1 - Setting up files and directories

1. Create a new directory for $\mathcal{SO}$-solver
   a. `$ cd src/tsolvers`
   b. `src/tsolvers $ cp -r emptysolver sosolver`

Phase 1 - Setting up files and directories

1. Create a new directory for $\mathcal{SO}$-solver
   a. `$ cd src/tsolvers`
   b. `src/tsolvers $ cp -r emptysolver sosolver`

2. Rename files
   a. `src/tsolvers $ cd sosolver`
   b. `src/tsolvers/sosolver $ mv EmptySolver.h SOSolver.h`
   c. `src/tsolvers/sosolver $ mv EmptySolver.C SOSolver.C`

Phase 1 - Setting up files and directories

1. Create a new directory for $\mathcal{SO}$-solver
   a. `$ cd src/tsolvers`
   b. `src/tsolvers $ cp -r emptysolver sosolver`
2. Rename files
   a. `src/tsolvers $ cd sosolver`
   b. `src/tsolvers/sosolver $ mv EmptySolver.h SOSolver.h`
   c. `src/tsolvers/sosolver $ mv EmptySolver.C SOSolver.C`
3. Adjust Makefile.am
   a. `src/tsolvers/sosolver $ rm Makefile.in`
   b. `src/tsolvers/sosolver $ vim Makefile.am`
   c. Find-replace “`empty`” with “`so`” and “`Empty`” with “`SO`” in the file (`:%s/empty/so/g` and `:%s/Empty/SO/g` with vim)

Phase 1 - Setting up files and directories

4. Adjust source files
   a. `src/tsolvers/sosolver $ vim SOSolver.h`
   b. Find-replace "`EMPTY`" with "`SO`" (`:%s/EMPTY/SO/g` with vim)
   c. Find-replace "`Empty`" with "`SO`" (`:%s/Empty/SO/g` with vim)
   d. `src/tsolvers/sosolver $ vim SOSolver.C`
   e. Find-replace "`Empty`" with "`SO`" (`:%s/Empty/SO/g` with vim)

Phase 1 - Setting up files and directories

4. Adjust source files
   a. `src/tsolvers/sosolver $ vim SOSolver.h`
   b. Find-replace "`EMPTY`" with "`SO`" (`:%s/EMPTY/SO/g` with vim)
   c. Find-replace "`Empty`" with "`SO`" (`:%s/Empty/SO/g` with vim)
   d. `src/tsolvers/sosolver $ vim SOSolver.C`
   e. Find-replace "`Empty`" with "`SO`" (`:%s/Empty/SO/g` with vim)

5. Adjust `../Makefile.am`
   a. `src/tsolvers/sosolver $ cd ..`
   b. `src/tsolvers $ vim Makefile.am`
   c. Add `sosolver` in `SUBDIR` list
   d. Add `sosolver/libsosolver.la`

Phase 1 - Setting up files and directories

4. Adjust source files
   a. `src/tsolvers/sosolver $ vim SOSolver.h`
   b. Find-replace "`EMPTY`" with "`SO`" (`:%s/EMPTY/SO/g` with vim)
   c. Find-replace "`Empty`" with "`SO`" (`:%s/Empty/SO/g` with vim)
   d. `src/tsolvers/sosolver $ vim SOSolver.C`
   e. Find-replace "`Empty`" with "`SO`" (`:%s/Empty/SO/g` with vim)

5. Adjust `../Makefile.am`
   a. `src/tsolvers/sosolver $ cd ..`
   b. `src/tsolvers $ vim Makefile.am`
   c. Add `sosolver` in `SUBDIR` list
   d. Add `sosolver/libsosolver.la`

6. Adjust `../../configure.ac`
   a. `src/tsolvers $ cd ../..`
   b. `$ vim configure.ac`
   c. Add `-I\${top_srcdir}/src/tsolvers/sosolver \\\`
   d. Add `src/tsolvers/sosolver/Makefile \`

Phase 1 - Setting up files and directories

7. Compile again
   a. `$ cd debug`
   b. `debug $ make -j4`
   c. `debug $ cd ..`

Phase 2 - Connecting the $\mathcal{T}$-solver

1. Create a new logic
   a. `$ vim src/common/Global.h`
   b. Add `, QF_SO` around line 196
   c. Add `else if ( l == QF_SO ) return "QF_SO";` around line 312
   d. Add `using opensmt::QF_SO;` around line 346
   e. `$ vim src/api/OpenSMTContext.C`
   f. Add `else if ( strcmp( str, "QF_SO" ) == 0 ) l = QF_SO;` around line 88
   g. Compile again `$ cd debug; make; cd ..` (to see if any typo was introduced)

Phase 2 - Connecting the $\mathcal{T}$-solver

2. Allocate the solver

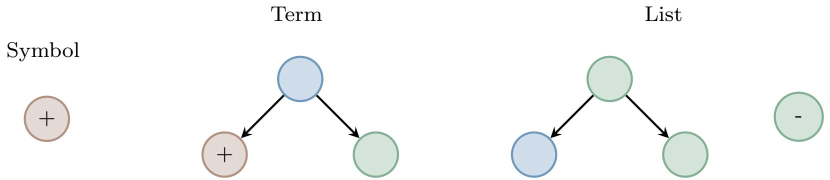
   a. `$ vim src/egraph/EgraphSolver.C`
   b. Add `#include "SOSolver.h"` around line 29
   c. Add around line 853

```
else if ( config.logic == QF_SO )
{
  tsolvers.push_back( new SOSolver( tsolvers.size( )
                                   , "Simple Order Solver"
                                   , config
                                   , *this
                                   , sort_store
                                   , explanation
                                   , deductions
                                   , suggestions ) );
  #ifdef STATISTICS
    tsolvers_stats.push_back( new TSolverStats( ) );
  #endif
}
```

2. Compile again `$ cd debug; make; cd ..` (to see if any typo was introduced)

The **`Enode`** is the data structure that stores any term and formula

There are three kinds of Enode

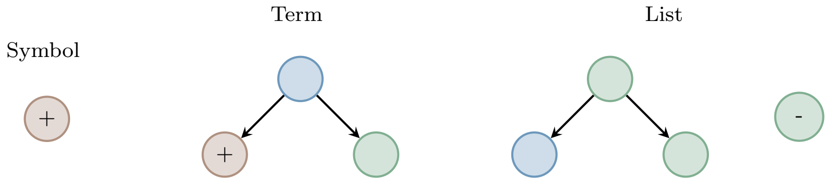


Phase 3 - Implementing the solver

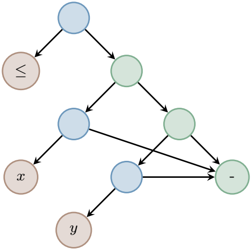
Playing with the Enodes

The **`Enode`** is the data structure that stores any term and formula

There are three kinds of Enode



E.g. the term $x \leq y$ is represented as



If **`e`** is the Enode for $x \leq y$, we retrieve the Enode for **`x`** with

`Enode * lhs = e->get1st( )`

and the Enode for **`y`** with

`Enode * rhs = e->get2nd( )`

The polarity of an Enode can be retrieved with

```
lbool sign = e->getPolarity( );
```

and it could be `l_True` or `l_False`

An Enode `e` can be simply printed with

```
cerr << "printing enode: " << e << endl;
```

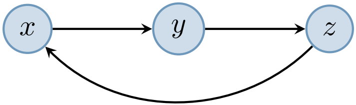
Phase 3 - Implementing the solver

A set of constraints is unsatisfiable iff there is a cycle

$$x \leq y,\ y \leq z,\ \ldots,\ w \leq x$$

A model is therefore is any "acyclic" set of constraints

$x \leq y$
$y \leq z$
$z \leq x$



unsat

Therefore we need to

1. Represent the graph from the received constraints
2. Check if the graph contains a cycle

Phase 3 - Implementing the solver

Representing the graph

We represent the graph with **adjacency lists**, i.e., every node (variable) is assigned to a list of outgoing edges

We use STL to make the following data structure

```
map< Enode *, vector< Enode * > > adj_list;
```

that we declare as private attribute in `SOSolver.h`, so that it will be accessible everywhere in the class

When we receive a new constraint with `assertLit`, we update `adj_list` with

```
adj_list[ from ].push_back( e );
```

Phase 3 - Implementing the solver

Checking for a cycle

When `check` is called we have to see if there is a cycle in the current graph

We use a simple depth-first search using the following high-level recursive function:

Input: a node "from"
Output: *true* iff a cycle containing "from" is found

```
function findCycle( Enode * from )

    if ( "from was seen before" ) return true

    for each "from ≤ y" in adj_list of from
        res = findCycle( y )
        if ( res == true ) return true

    return false
```

Phase 3 - Implementing the solver

Checking for a cycle

In SOSolver.C

```
...
bool SOSolver::findCycle( Enode * from )
{
  if ( seen.find( from ) != seen.end( ) )
    return true;

  seen.insert( from );
  vector< Enode * > & adj_list_from = adj_list[ from ];

  for ( size_t i = 0 ; i < adj_list_from.size( ) ; i ++ )
  {
    const bool cycle_found = findCycle( adj_list_from[ i ]->get2nd( ) );
    if ( cycle_found )
      return true;
  }

  seen.erase( from );
  return false;
}
...
```

Phase 3 - Implementing the solver

Checking for a cycle

In SOSolver.C

```
bool SOSolver::check( bool complete )
{
  for ( map< Enode *, vector< Enode * > >::iterator it = adj_list.begin( )
      ; it != adj_list.end( )
      ; it ++ )
  {
    seen.clear( );

    Enode * from = it->first;
    const bool cycle_found = findCycle( from );

    if ( cycle_found )
      return false;
  }

  return true;
}
```

In SOSolver.h

```
private:

  bool findCycle ( Enode * );

  map< Enode *, vector< Enode * > > adj_list;
  set< Enode * > seen;
```

We can try to compile, and test it with files so_example_1.smt2 and so_example_2.smt2

Phase 3 - Implementing the solver

Computing conflicts

Conflicts can be computed by keeping track of the edges that form the last cycle detected

We add a `map< Enode *, Enode * > parent_edge` map which we use to store the edge used to reach a node in the depth-serch traversal

When during findCycle we discover an already visited node, we backward visit the parent_edge relation to retrieve the cycle

```
void SOSolver::computeExplanation( Enode * from )
{
  assert( explanation.empty( ) );
  Enode * x = from;
  do
  {
    x = parent_edge[ x ]->get1st( );
    explanation.push_back( parent_edge[ x ] );
  }
  while( x != from );
}
```

Again in SOSolver.C

```
void SOSolver::findCycle( Enode * from )
{
  if ( seen.find( from ) != seen.end( ) )
  {
    computeExplanation( from );
    return true;
  }
  ...
}
```

Again in SOSolver.h

```
  map< Enode *, Enode * > parent_edge;
```

Last thing we need is to handle incrementality/backtrackability

For simplicity, solving will not be incremental nor backtrackable in this lecture

However we still have to keep `adj_list` updated with constraints received/dropped

For this aim we introduce two more vectors in SOSolver.h

```
vector< Enode * > used_constr;
vector< size_t >  backtrack_points;
```

`used_constr` keeps track of the order of constraints received, while `backtrack_points` keeps track of the **size** of `used_constr` when a new backtrack point is requested

```
void SOSolver::pushBacktrackPoint ( )
{
  backtrack_points.push_back( used_constr.size( ) );
}


void SOSolver::popBacktrackPoint ( )
{
  assert( !backtrack_points.empty( ) );
  const size_t new_size = backtrack_points.back( );
  backtrack_points.pop_back( );

  while ( new_size < used_constr.size( ) )
  {
    Enode * e = used_constr.back( );
    Enode * from = e->get1st( );
    Enode * to   = e->get2nd( );
    assert( adj_list[ from ].back( ) == e );
    adj_list[ from ].pop_back( );
    used_constr.pop_back( );
  }
}
```

Exercizes

The current code for `check` has worst-case quadratic complexity (a lot of redundant work is done). Modify it to make it linear (hint: use another `set< Enode * >` to keep track of . . .)