Formalization of Incremental Simplex Algorithm by Stepwise Refinement

Mirko Spasić, Filip Marić

Faculty of Mathematics,
University of Belgrade

FM2012, 30. August 2012.

Overview

Overview

1. Introduction
   - Formal Verification of SMT solvers?
2. Approach and Techniques
3. Linear Arithmetic, Incremental Simplex
4. Evaluation
5. Conclusions

SMT solvers

- SMT solvers are very important tools in formal hardware and software verification.
- Quis custodiet ipsos custodes? — who will guard the guards?
- How to trust SMT solvers results, having in mind their complexity?
- Several approaches:
  - formal verification of solvers (and their underlying algorithms),
  - generating and checking certificates.
- Certificate checking shows very good results in practice and therefore it has been the dominant approach in industry (e.g., Böhme and Weber 2010., Armand et al. 2011.).

Why formal verification?

Still, we advocate that formal verification of SMT solving algorithms within a proof assistant may have its own merits.

- Mathematical proofs have two main components: justification (certification) and explanation (message).
- Approach to formalization may be more important then the final result itself.
- Apart from giving assurance that a procedure is correct, formalization effort should carry important messages for the reader.
- Formalization offers clear explanations for subtle details.
- The formalization is a contribution to the growing body of verified theorem proving algorithms.

Overview

Approach to verification

- Shallow embedding in the proof assistant Isabelle/HOL.
    - HOL treated as a functional programming language.
    - Functional model of the procedure implemented in HOL and verified.
    - Executable code can be extracted (in SML, Haskell, Scala, OCaml, . . .).
    - By means of reflection, the procedure can be used within the proof assistant.

Approach to verification — refinement

- Procedure is developed trough a long series of small refinement steps.
- Refinement is a verifiable transformation of abstract formal (high-level) specification into a concrete executable (low-level) program.
- Stepwise refinement assumes that the refinement process is performed through a series of simple steps.

Refinement

- Top-down approach.
- Correct-by-construction.
- Each step reduces the amount of non-determinism in a program.
- Rich history (systematically explored by E. W. Dijkstra and N. Wirth in 1960s, formal treatment given by R. J. Back in 1970s).

Data vs Algorithm refinement

- Data refinement assumes replacing abstract data structures by concrete ones.
- Algorithm (program) refinement assumes replacing abstract algorithms (operations) by concrete ones.

Benefits of using refinement in our formalization

- The procedure can be analyzed and understood on different levels of abstraction.
- Abstract layers in the formalization allow easy porting of the formalization to other systems.
- Makes the formalization suitable for teaching formal methods.
- Makes the correctness proofs significantly simpler.

Code generation as a refinement framework of Isabelle/HOL

- Haftmann and Nipkow, 2010.
- No axiomatic specification is used.
- Specification is done in terms of a reference implementation (usually simple and abstract).
- Correctness proofs for the system rely only on the reference implementation, while concrete representations are used only during code generation.

Code generation as a refinement framework of Isabelle/HOL

- Algorithm refinement:
  - Give a new (better) implementation of a function.
  - Prove the equivalence with the reference implementation.
  - Instruct the code generator to use the new implementation.
- Data refinement:
  - Define an abstract data type representation and functions operating on this representation.
  - Define a concrete data type representation, functions operating on this representation and the conversion from the concrete to the abstract representation.
  - Prove the equivalence.
  - Instruct the code generator to use the concrete representation.

Program refinement in Isabelle/HOL by using locales

- Locales — Isabelle's version of parametrized theories.
- A locale is a named context of functions $f_1, \ldots, f_n$ and assumptions $P_1, \ldots, P_m$:
  locale loc = **fixes** $f_1, \ldots, f_n$ **assumes** $P_1, \ldots, P_m$
- Locales can be hierarchical as in:
  locale $loc = loc_1 + loc_2 +$ **fixes** ...
- Locales are ideal for giving axiomatic function specifications:

Example

locale $sorting =$
  fixes $sort :: ''\,'a\ list \Rightarrow\ 'a\ list''$
  assumes
    $sorted : let\,l' = sort\ l\ in\ \forall i < length\ l' - 1.\ l'_{[i]} \leq l'_{[i+1]}$
    $elems : multiset\_of\ \ (sort\ \ l) = multiset\_of\ \ l$

Program refinement by using locales

- In the context of a locale, definitions can be made and theorems can be proved.
- Locales can be interpreted by concrete instances of $f_1, \ldots, f_n$, and then it must be shown that these satisfy assumptions $P_1, \ldots, P_m$.
- Locales are naturally combined with the code generation.

Program refinement by using locales

- A locale l is a sublocale of a locale l' if all functions of loc' can be defined using the functions of l and all assumptions of l' can be proved using the assumptions of l.
- Then every interpretation for loc can be automatically converted to an interpretation of loc'.

Program refinement by using locales

Example

locale *min_selection* =
  fixes *min* :: "'a list ⇒ 'a × 'a list"
  assumes
  "let $(m, l') = min\ l$ in $multiset\_of\ (m\#l') = multiset\_of\ l$"
  "let $(m, l') = min\ l$ in $\forall x \in set\ l'.\ m \le x$"
begin
  function *ssort* **where**
  "$ssort\ l = (if\ l = []\ then\ []\ else\ let\ (m, l') = min\ l\ in\ m\#ssort\ l')$"
end

sublocale *min_selection* < *sort ssort*
proof
...
qed

Overview

Linear arithmetic

- A first order theory (usually semantically specified).
- Atomic formulae of the form $c_1x_1 + \ldots c_nx_n \bowtie c$, where $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$, and $c_1, \ldots, c_n, \ldots c$ are integer (or rational) constants.
- Usually, only universally quantified fragment is assumed (i.e., satisfiability of ground formulae is checked).
- Several variants:
  - LRA — satisfiability over $\mathbb{Q}$
  - LIA — satisfiability over $\mathbb{Z}$

Example

Are there rational constants x and y such that

$$x \leq -4 \ \wedge \ x > -8 \ \wedge \ y - x < 1 \ \wedge \ x + y \geq 2?$$

SMT solvers

- Formulae encountered in verification practice are not only conjunctions of literals and have rich propositional structure. E.g., $(3x + 4y > 0 \ \vee x + y < 3) \Rightarrow (2x - 3y \geq 5 \wedge x < 0)$.
- SMT solvers combine powerful SAT solvers for propositional reasoning with decision procedures for conjunctions of literals in concrete theories.
- Maximal efficiency requires modification of both SAT solvers and decision procedures.

Decision procedures for linear arithmetic

- Decidable theory.
- Different decision procedures. Most popular are based on:
    - Fourier-Motzkin elimination (in some aspects similar to Gaussian elimination for equality systems),
    - Simplex algorithm (Dantzig, 1947, linear programming and elimination algorithm).

Incremental Simplex for SMT

- Duterte and de Moura, 2006.
- Yices solver.
- Adopted by many state-of-the-art SMT solvers.
- Dual-simplex with Bland's rule for ensuring termination.
- Basic solver for LRA with extensions for LIA (branch-and-bound, Gomory's cuts).
- Only proof sketch of termination (partial correctness not proved).

Polynomials

- Polynomials are of the form $a_1 \cdot x_1 + \ldots + a_n \cdot x_n$.
- Abstract representation:
  - Functions mapping variables x_i into coefficients a_i, such that only finitely many variables have a non-zero coefficient.
  - The sum of p_1 and p_2 is the polynomial $\lambda\ x.\ p_1\ x + p_2\ x$.
  - The value of the polynomial p for the valuation v, denoted by $p\{\!|v|\!\}$ is $\sum x \in \{x.\ p\ x \neq 0\}.\ p\ x \cdot v\ x$
- Concrete representations:
  - Lists of coefficients.
  - Red-black tree implemented mappings.

Linear constraints

Linear constraints are of the form $p \bowtie c$ or $p_1 \bowtie p_2$:

- p, p_1 i p_2 su linearni polinomi,
- c is a rational constant,
- $\bowtie \in \{<, >, \leq, \geq, =\}$.

datatype constraint =
LT linear_poly rat |
GT linear_poly rat | ...

Semantics of linear constraints

- $\mathsf{v} \models_c \ \mathsf{c}$ — valuation v satisfies the constraint c
  - $\mathsf{v} \models_c \mathsf{LT}\ \mathsf{l}\ \mathsf{r} \longleftrightarrow \mathsf{l}\{\!|\mathsf{v}|\!\} < \mathsf{r}$
  - $\mathsf{v} \models_c \mathsf{GT}\ \mathsf{l}\ \mathsf{r} \longleftrightarrow \mathsf{l}\{\!|\mathsf{v}|\!\} > \mathsf{r}$
- $\mathsf{v} \models_{cs} \mathsf{cs}$ — valuation v satisfies the list of constraints cs
  - $\mathsf{v} \models_{cs} \mathsf{cs} \equiv \forall\ \mathsf{c} \in \mathsf{set}\ \mathsf{cs}.\ \mathsf{v} \models_c \mathsf{c}$

Procedure specification

locale Solve =

— Decide if the given list of constraints is satisfiable. Return the satisfiability status and, in the satisfiable case, one satisfying valuation.

fixes solve :: "constraint list ⇒ bool × rat valuation option"

— If the status *True* is returned, then returned valuation satisfies all constraints.

assumes "let (sat, v) = solve cs in sat $\longrightarrow$ v $\models_{cs}$ cs"

— If the status *False* is returned, then constraints are unsatisfiable.

assumes "let (sat, _) = solve cs in $\neg$ sat $\longrightarrow$ $\neg$ ($\exists$ v. v $\models_{cs}$ cs)"

Eliminating non-strict inequalities

- $p < c$ can be replaced by $p \leq c - \delta$,
- $p > c$ can be replaced by $p \geq c + \delta$
- All further computations are done in the structure $\mathbb{Q}_\delta$ (ordered vector space over $\mathbb{Q}$)
  - elements are of the form $a + b \cdot \delta$, $a, b \in \mathbb{Q}$,
  - $(a_1 + b_1 \cdot \delta) + (a_2 + b_2 \cdot \delta) = (a_1 + a_2) + (b_1 + b_2) \cdot \delta$,
  - $c \cdot (a + b \cdot \delta) = c \cdot a + c \cdot b \cdot \delta$,
  - $(a_1 + b_1 \cdot \delta) < (a_2 + b_2 \cdot \delta) \longleftrightarrow a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 < b_2)$.

Specification of eliminating strict constraints

locale To_ns =

— Convert a constraint list to an equisatisfiable non-strict constraint list.

fixes to_ns :: "constraint list ⇒ 'a::lrv ns_constraint list"

assumes "v $\models_{cs}$ cs $\Longrightarrow$ ∃ v'. v' $\models_{nss}$ to_ns cs"

— Convert the valuation that satisfies all non-strict constraints to the valuation that satisfies all initial constraints.

fixes from_ns :: "(var ⇒ 'a) ⇒ 'a ns_constraint list ⇒ (var ⇒ rat)"

assumes "⟨v'⟩ $\models_{nss}$ to_ns cs $\Longrightarrow$ ⟨from_ns v' (to_ns cs)⟩ $\models_{cs}$ cs"

Implementation of the solve function

Assuming that there is a function solve_ns solving the non-strict constraints (with a specification analogous to the one for the function solve), the solve function can be implemented simply:

```
solve cs ≡ let cs' = to_ns cs; (sat, v) = solve_ns cs' in
  if sat then (True, Some (from_ns v cs')) else (False, None)
```

Preprocessing

In the next step, the list of non-strict constraints is transformed into:

- a tableau — list of linear equalities
- list of atoms atom of the form $x_i \bowtie b_i$, such that x_i is a variable, and b_i is a constant from $\mathbb{Q}_\delta$

For example, $[x_1 + x_2 \leq b_1, x_1 + x_2 \geq b_2, x_2 \geq b_3]$ is transformed into $[x_3 = x_1 + x_2]$ and atoms $[x_3 \leq b_1, x_3 \geq b_2, x_2 \geq b_3]$

Formalization of tableau and atoms

type eq = var × linear_poly

v $\models_e$ (x, p) ≡ v x = p ⦃ v ⦄

type tableau = eq list

Tableau is normalized (denoted by △ t) if variables on the left sides are all different and do not occur on the right side.

datatype 'a atom = Leq var 'a | Geq var 'a

"v $\models_a$ Leq x c ⟷ v x ≤ c" | "v $\models_a$ Geq x c ⟷ v x ≥ c"

"v $\models_{as}$ as ≡ ∀ a ∈ as. v $\models_a$ a"

Preprocessing specification

locale Preprocess = **fixes** preprocess::"'a::lrv ns_constraint list $\Rightarrow$ tableau$\times$'a atom list"
assumes
— The returned tableau is always normalized.
"let (t, as) = preprocess cs in $\triangle$ t"
— Tableau and atoms are equisatisfiable with starting non-strict constraints.
"let (t, as) = preprocess cs in v $\models_{as}$ set as $\wedge$ v $\models_t$ t $\longrightarrow$ v $\models_{nss}$ cs"
"let (t, as) = preprocess cs in v $\models_{nss}$ cs $\longrightarrow$ ($\exists$ v'. v' $\models_{as}$ set as $\wedge$ v' $\models_t$ t)"

Implementation of solve_ns

Assuming that the assert_all function, which has the precondition that the tableau is normalized, and the effect similar to the function solve, the function solve_ns can be easily implemented:

solve_ns s $\equiv$ let (t, as) = preprocess s in assert_all t as

Overview

Proof metrics

- Around 8K lines of proof (3K devoted to termination).
- A previous ,,monolithic" attempt was abandoned when it went over 25K lines of proofs.
- Crucial aspects for proof simplification: refinement approach and treatment of symmetric cases.

Run-time comparison with other implementations

- Verified (Chaieb and Nipkow — Isabelle/HOL)
- Semi-verified (Spasić and Marić — C++)
- Unverified (SMT solvers Z3 and OpenSMT)

Choice of benchmarks

- We are handling only conjunctions of constraints.
- Benchmarks contain many conjuncts with many variables (up to 100×50).
- Randomly generated.
- Dense — non realistic for the SMT applications.

What did the experiments show?

- Simplex was several orders of magnitude faster then previously verified algorithms in Isabelle/HOL.
- Much slower then its counterpart C++ implementation and Z3.
- C++ not always much slower then Z3 (different variable orderings).

Surprising profiling results

- The cause of inefficiency: functional (non-destructive) data structures vs imperative data-structures?
- No!
- Reference C++ used exact rationals of the GMP library, while the extracted code reduces everything to ML native integers (also backed up by GMP).
- Manually changed the generated code to use native rationals (this time in Haskell since ML does not support rationals natively).
- The tweaked Haskell code slightly outperformed the C++ implementation!
- More then 80% of the times is spend doing rational arithmetic, so in this scenario it does not matter whether imperative or functional data structures are used.

Overview

Conclusions

- Verified incremental Simplex algorithm in Isabelle/HOL.
- In some scenarios, generated code is competitive with state-of-the art solvers.
- Much more important then the result itself is the approach to the formalization.
- Refinement — many layers of abstraction give strength to a formalization attempt.