Combining Superposition and Induction: a Practical Realization

Abdelkader Kersani and Nicolas Peltier

Laboratoire d'Informatique de Grenoble/CNRS
CAPP team - ASAP project (ANR-09-BLAN-0407-01)

FROCOS 2013 - September 2013 - Nancy

Introductory example

$$\text{length\_at\_least}(l;n) \quad , \quad n = 0 \_ \; \exists x; l'; n' \, (l = cons(x;l') \wedge n = s(n') \wedge \text{length\_at\_least}(l';n'))$$

$$\text{nth}(x;l;n) \quad , \quad \exists l' \, l = cons(y;l') \wedge (n = s(0) \wedge x = y) \_ \; \exists n' \, (n = s(n') \wedge \text{nth}(x;l';n'))$$

Check that the following holds:

$\forall n \in \mathbb{N}; \forall l \, (\text{length\_at\_least}(l;n) \wedge n \neq 0 \rightarrow \exists x \, \text{nth}(x;l;n))$

Introductory example (2)

- This problem cannot be stated in first-order logic ($n\ 2\ \mathbb{N}$)
- An inductive property of the form $8n; l\ 9x$
- Must combine:
  - Standard equational reasoning with unification to:
    1. Find the value of x (w.r.t. n, l)
    2. Check that it indeed fulfills the desired property
  - Inductive reasoning on n

Introductory example (3)

- Straightforward approach: use standard proof procedures for first-order logic together with explicit induction schemes

$$(\ \ (0) \wedge \forall n\ \ (n)\ \ ) \quad (s(n)))\ \ ) \ \forall n\ \ (n)$$

  for some "well-chosen" formula
- Our approach: try to discover automatically such inductive lemmata, by detecting cycles in the search space

Plan of the talk

- The language
- A proof procedure: superposition + loop detection
- A cycle detection algorithm
- Experimentations

The language

Clausal (first-order) logic + a (unique) arithmetic parameter n

- Two sorts (standard terms) and $!$ (natural numbers), with $0 : !; s : ! \; ! \; !$
- A special constant symbol n denoting a natural number
- Terms, (equational) literals and clauses are defined as usual
  do *not* contain the special symbol n
- n-clauses: constrained clauses of the form

$$[C \; j \; X]$$

where:

- C is a clause
- X is of the form $\bigwedge_{i=1}^{k} n = t_i$, where $t_1; : : : ; t_k$ (k 0) are terms of sort $!$

Semantics

- The special symbol n is interpreted as a term of the form $s^m(0)$ $(m\ 2\ \mathbb{N})$
- 0 and s are interpreted as 0 and successor function
- The other symbols are interpreted as usual
- $[C\ j\ \bigwedge_{i=1}^{k} n = t_i]$ holds in I iff for every substitution such that $I(n) = t_i$, C holds in I

The language (2)

Remarks:

- A strict extension of first-order logic
- The constant n does not occur in the clauses
  A formula of the form $f(n) = a$ must be written:

$$[f(x) = a \, j \, n = x]$$

- Extension to formulæ with several parameters

A theoretical limitation

Theorem

The set of satisfiable sets of n-clauses is neither recursively enumerable (of course !) nor co-recursively enumerable

Depart from:

- First-order logic (unsatisfiability is semi-decidable)
- Rewrite-based inductive theorem proving (non-provability is semi-decidable)

The language (3)

Proposition

Every (non-tautological) n-clause is equivalent to an n-clause of the form $[C\ j\ >]$ or $[C\ j\ n = t]$

Proof: $\bigwedge_{i=1}^{k} n = t_i \ \ , \ \ n = t_1 \wedge \bigwedge_{i=2}^{k} t_1 = t_i$, thus

$$[C\ j \bigwedge_{i=1}^{k} n = t_i] \ \ , \ \ [C \quad j\ n = t_1 \ \ ]$$

where $\quad = \mathrm{mgu}(t_1; : : : ; t_k)$ and

$$[C\ j \bigwedge_{i=1}^{k} n = t_i] \ \ , \ \ >$$

if $t_1; : : : ; t_k$ are not unifiable

The language (4)

3 kinds of n-clauses:

1. Standard first-order clauses: express universal properties, not depending on the value of n
2. $[C \; j \; n = s^k(0)]$: expresses a property that holds only if n has some specific value $(n = k)$
3. $[C[x] \; j \; n = s^k(x)]$: expresses a property C that holds for $x = n \quad k$

The language (4)

3 kinds of n-clauses:

1. Standard first-order clauses: express universal properties, not depending on the value of n rank ?
2. $[C\ j\ n = s^k(0)]$: expresses a property that holds only if n has some specific value $(n = k)$ no rank
3. $[C[x]\ j\ n = s^k(x)]$: expresses a property C that holds for $x = n$ k rank k

$S[i]$ denotes the set of n-clauses of rank i in S

Proof Procedure: Constrained superposition calculus

Superposition:

$$\frac{[C \_ t \ ./ \ s \ j \ X]; [D \_ u = v \ j \ Y]}{[C \_ D \_ t[v]_p \ ./ \ s \ j \ X \ \hat{} \ Y]}$$

If $./2 \ f=; 6=g$, $= \text{mgu}(u; tj_p)$, $u \ \ 6 \ v \ ; t \ \ 6 \ s \ , tj_p$ is not a variable, $(t \ ./ \ s) \ \ 6\in C \ , (u = v) \ \ 6\in D \ .$

Proof Procedure: Constrained superposition calculus (2)

Reflection:

$$\frac{[C \_ t \not\approx s \mid X]}{[C \mid X]}$$

If $\sigma = \mathrm{mgu}(t; s)$, $(t \not\approx s)\sigma \in C$

Factorisation:

$$\frac{[C \_ t = s \_ u = v \mid X]}{[C \_ s \not\approx v \_ t = s \mid X]}$$

If $\sigma = \mathrm{mgu}(t; u)$, $t\sigma \not\approx s\sigma$, $u\sigma \not\approx v\sigma$, $(t = s)\sigma \in C$.

Proof Procedure

Remarks:

- The parameter n is abstracted away from the clauses: $f(n) = a \quad ! \quad [f(x) = a \mid n = x]$
- Allows for a lazy instantiation of this parameter: $[f(x) = a \mid n = x]; f(0) \not\simeq a \vdash [\square \mid n = 0]$
- "Weakly" complete: if $S \not\models n \not\simeq k$ (for some $k \in \mathbb{N}$) then $S \vdash [\square \mid n = k]$ (modulo subsumption)
- Not complete: no contradiction is derived in finite time (almost never terminates)

A trivial example

Prove the following:

$$p(0) \wedge \forall x\, p(x) \;\Rightarrow\; p(s(x)) \models \forall n \in \mathbb{N}\, p(n)$$

A trivial example (2)

Use the superposition calculus:

1	$p(0) = \texttt{true}$	
2	$p(x) \not= \texttt{true} \_ p(s(x)) = \texttt{true}$	
3	$[p(x) \not= \texttt{true} \mid n = x]$	
4	$[\Box \mid n = 0]$	(superposition, 1, 3)
5	$[p(x) \not= \texttt{true} \mid n = s(x)]$	(superposition, 2, 3)
6	$[\Box \mid n = s(0)]$	(superposition, 1, 5)
...	...	
	$[\Box \mid n = s^k(0)]$	

Uncomplete calculus

If S is unsatisfiable, we have:

$$\forall k \in \mathbb{N}\ S \vdash n \neq k$$

but not:

$$S \vdash \forall k \in \mathbb{N}\ n \neq k$$

$$(\quad ?)$$

A trivial example (3)

A "cycle" in the search space:
Clause 5 : $[p(x) \not\simeq \texttt{true} \mid n = s(x)]$ is almost identical to Clause 3 : $[p(x) \not\simeq \texttt{true} \mid n = x]$, up to a translation on n.
Clause 3 $p(n)$
Clause 5 $p(n \quad 1)$
Idea: detect those cycles and use them to prune the search space

Cycle Detection

First step: Formalize the notion of translation

- $S\#_i \quad S\,f\,n \quad n \quad i\,g$
- $[C\ j\ n = t] \quad ! \quad [C\ j\ n \quad i = t]$

Cycle Detection

First step: Formalize the notion of translation

- $S\#_i \quad S\,f\,n \quad n \quad i\,g$
- $[C\ j\ n = t] \quad ! \quad [C\ j\ n \quad i = t]$

Cycle Detection

First step: Formalize the notion of translation

- $S\#_i \quad S\,f\,n \quad n \quad i\,g$
- $[C\ j\ n = t] \quad ! \quad [C\ j\ n = t + i]$

Cycle Detection

First step: Formalize the notion of translation

- $S\#_i \quad S\,f\,n \quad n \quad i\,g$
- $[C\ j\ n = t] \quad ! \quad [C\ j\ n = s^i(t)]$

Cycle Detection

Second step:

Cycle Detection Rule

If there exists S_{ind} S such that:

1. $S_{ind} \not\models n \not\approx l$, for every l 2 $[i; i + j[$
2. and $S_{ind} \not\models S_{ind}\#_j$,

then $S \not\models n < i$ (i.e. $S \not\models [\Box\, j\, n = s^i(x)]$)

Proof: by "descente infinie"

Cycle Detection

In practice:

- S is the whole search space (set of generated n-clauses)
- S_{ind} S
- Decidable conditions are needed

Cycle Detection

In practice:

- S is the whole search space (set of generated n-clauses)
- S_{ind} S
- Decidable conditions are needed
  - Condition 1: Check that $[\Box\ j\ n = l]$ has been derived from S_{ind}
  - Condition 2: Check that some set of n-clauses S_{loop} has been derived from S_{ind}, with $S_{loop} = S_{ind}\#_j$

Cycle Detection

In practice:

- S is the whole search space (set of generated n-clauses)
- S_{ind} S
- Decidable conditions are needed
  - Condition 1: Check that $[\Box\ j\ n = l]$ has been derived from S_{ind}
  - Condition 2: Check that some set of n-clauses S_{loop} has been derived from S_{ind}, with S_{loop} subsumes $S_{ind}\#_j$

Cycle Detection

In practice:

- S is the whole search space (set of generated n-clauses)
- S_{ind} S
- Decidable conditions are needed
  - Condition 1: Check that $[\Box \; j \; n = l]$ has been derived from S_{ind}
  - Condition 2: Check that some set of n-clauses S_{loop} has been derived from S_{ind}, with S_{loop} subsumes $S_{ind}\#_j$ (or any decidable entailment relation)

Cycle Detection

In practice:

- S is the whole search space (set of generated n-clauses)
- S_{ind} S
- Decidable conditions are needed
  - Condition 1: Check that $[\Box\ j\ n = l]$ has been derived from S_{ind}
  - Condition 2: Check that some set of n-clauses S_{loop} has been derived from S_{ind}, with S_{loop} subsumes $S_{ind}\#_j$ (or any decidable entailment relation)
  - A further restriction: assume that all n-clauses in S_{ind} have the same rank i (or $?$)

Example (continued)

1	$p(0) = \texttt{true}$	
2	$p(x) \not\simeq \texttt{true} \_ p(s(x)) = \texttt{true}$	
3	$[p(x) \not\simeq \texttt{true} \mid n = x]$	
4	$[\square \mid n = 0]$	(superposition, 1, 3)
5	$[p(x) \not\simeq \texttt{true} \mid n = s(x)]$	(superposition, 2, 3)
6	$[\square \mid n = s(0)]$	(superposition, 1, 5)
...	...	
	$[\square \mid n = s^k(0)]$	

- $S_{ind} = \{1; 2; 3\}$, $S_{loop} = \{1; 2; 5\}$, $i = 0$, $j = 1$
- $[\square \mid n = x]$ can be derived
- Unsatisfiability is detected

Cycle Detection in Practice

How to generate effectively the numbers $i; j$ and the sets $S_{ind}; S_{loop}$?

An algorithm to compute $S_{ind}; S_{loop}$ (for fixed $i; j$)

Properties:

- Sound: the computed sets $S_{ind}; S_{loop}$ satisfy the desired property
- Complete: if some sets $S_{ind}; S_{loop}$ satisfy the desired property, then the algorithm succeeds (but not necessarily with output $S_{ind}; S_{loop}$)
- Efficient: polynomial w.r.t. the size of the set S
- Based on a greatest fixpoint computation

The Algorithm

S_0 $fn \not\doteq k; k \in [i; i+j[g$
S_{ind} $S[i]$
if $S_{ind} \not\in S_0$ **then**
  return `false`
end if
S_{loop} $fD \in S[i+j] \mid S_{ind} \setminus fDgg$
while $\exists C \in S_{ind} \mid S_{loop} \not= fC\#_j g$ **do**
  S_{ind} $S_{ind} \cap fCg$
  if $S_{ind} \not\in S_0$ **then**
    return `false`
  end if
  Remove from S_{loop} all the n-clauses D s.t. $S_{ind} \not\in fDg$
end while
return `true`

Experimentations

Implemented in Prover9
Use n-clauses to model *schemata of formulæ*

- Formulæ depending on some parameter n
- Constructed using special connectives $\bigvee_{i=a}^{b}$ and $\bigwedge_{i=a}^{b}$

Example: n-bit adder

$$Sum_i(p;q;c;r) \stackrel{\text{def}}{=} r_i \quad , \quad (p_i \quad q_i) \quad c_i$$

$$Carry_i(p;q;c) \stackrel{\text{def}}{=} c_{i+1} \quad , \quad (p_i \wedge q_i) \_ (c_i \wedge p_i) \_ (c_i \wedge q_i)$$

$$Adder(p;q;c;r) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} Sum_i(p;q;c;r) \wedge \bigwedge_{i=1}^{n} Carry_i(p;q;c) \wedge \ : c_1$$

Translation into clausal form (1)

$$\bigvee_{i=0}^{n} \quad ! \; p(n)$$

with:

$$p(0) \;\; , \quad fi \;\; ! \;\; 0g$$

$$8x \, p(x+1) \;\; , \quad fi \;\; ! \;\; x+1g \_ \, p(x)$$

$$\bigvee_{i=a}^{n+b} \quad ! \; \bigvee_{i=0}^{n} (\quad \wedge q_i) \_ \quad fi \;\; ! \;\; n+1g \_ : : : \_ \quad fi \;\; ! \;\; n+bg$$

with:

$$: q(0) \wedge : : : \wedge \; : q(a \quad 1) \wedge q(a)$$

$$8x \, q(x) \;\; ! \;\; q(s(x))$$

Translation into clausal form (E)

Eliminate terms of the form $s(t)$ where t is not a variable:

$$p_{s(t)} \quad ! \quad p'_t$$

with:

$$\forall x \, p_{s(x)} \quad , \quad p_x$$

Experimentations

Example	Time	# of calls to Cycle_2	# clauses
Ripple-carry adder ($A + 0 = A$)	0.48	336	33833
Ripple-carry adder (commutativity)	0.03	102	2003
Ripple-carry adder (associativity)	0.09	207	10154
Unicity of the result (ripple-carry)	0.7	150	50901
Carry-propagate adder (commutativity)	0.02	14	1980
Carry-propagate adder (associativity)	0.01	20	3972
Equivalence between the ripple-carry and the carry-propagate adders	0.03	14	1980
Totality of $<$ ($n_1 \quad n_2 \_ n_1 < n_2$)	0.01	47	185

Summary

- A technique to combine superposition calculus and inductive theorem proving
- Automated discovery of (some) inductive invariants
- Completeness can be ensured in some cases (CADE), e.g. if the formulæ contain no non-arithmetic variable (schemata of propositional formulæ)
- An implementation based on Prover9

Future Work

- Incremental loop detection
- Heuristics to "guess" the values of i and j or to trigger the application of the loop detection rule
- Improve the implementation, more experimentations