Mixing Finite Success and Finite Failure in an Automated Prover

Alwen Tiu[1], Gopalan Nadathur[2] and Dale Miller[3]

[1]INRIA Lorraine, France

[2]University of Minnesota, USA

[3]INRIA Futurs/École polytechnique, France

ESHOL 2005 – Montego Bay, Jamaica
December 2, 2005

The Context for this Work

A declarative treatment of models of computational systems

In particular:

- Logic based encodings for structural operational semantics descriptions
- Executability of such encodings
- Logic based support for reasoning about encodings

Such capabilities are discussed, for instance, by the POPLmark challenge.

An Approach to Meeting these Requirements

Based on exploiting logic programming and proof search:

- SOS rules translate naturally into program clauses extended with
  - higher-order features for encoding λ-tree abstract syntax
  - new primitives for manipulating such encodings
- Proof search over program clauses leads to animation
- Reasoning about specifications realized via definitions/fixed points [Schroeder-Heister, LICS'93, Girard 92].

Approach has been developed by [McDowell & Miller, 2000] and [Miller & Tiu, 2004]

Here, we combine these ideas in a limited way into an extended logic programming system.

Outline of the Rest of the Talk

- Abstract syntax based on λ-trees
- Definitions and rules for reasoning about definitions
- The logic $FO\lambda^{\Delta\nabla}$ [Miller and Tiu, 2003]
- The Level 0/1 prover
- Concluding remarks

λ-Tree Abstract Syntax

- A variant of higher-order abstract syntax, based on using the simply typed λ-calculus
- λ-abstraction is used to encode binding impact of object language operators such as
  - quantifiers in logical formulas
  - function arguments in programs
  - restriction and bound input/output actions in the π-calculus
- Meta-level treatment of λ-terms supports notions such as
  - α-equivalence,
  - capture-avoiding substitution, and
  - binding respecting destructuring

Encoding π-Calculus Terms (Example)

Consider the π-calculus process $(x)a(y).\bar{y}x.0$

This reads as

Input a name y through the channel a and output a fresh name x through the channel y

Its encoding as a λ-term might be

$$\nu\ (\lambda x.\text{in}\ a\ \lambda y.(\text{out}\ y\ x\ 0))$$

where ν, in and out are constructors representing π-calculus operators.

Abstraction is used to capture the binding effects of restriction and bounded input.

Specifying π-Calculus Transition Rules (Example)

Consider the restriction transition rule for the π-calculus:

$$\frac{\mathsf{P} \xrightarrow{\alpha} \mathsf{P}'}{(x)\mathsf{P} \xrightarrow{\alpha} (x)\mathsf{P}'} \; x \notin \mathrm{n}(\alpha)$$

This can be rendered into the (extended) logic programming clause

$$\frac{\forall x (Px \xrightarrow{A} P'x)}{\nu(\lambda x.Px) \xrightarrow{A} \nu(\lambda x.P'x)}$$

Proof search with such translations supports animation.

Reasoning about Computations

If p and q are defined predicates, then we want to read

$$\forall x.p\ x \supset q\ x$$

as follows:

> *For every term t for which there is a proof of $p\ t$, there is also a proof of $q\ t$.*

Thus, this goal should succeed given the clauses

$$\{(p\ a),(p\ b),(q\ a),(q\ b),(q\ c)\}.$$

Such an interpretation is important for describing properties of computations like bisimulation.

A logical treatment of this interpretation can be obtained as follows:

- Recast program clauses as *definition clauses* of the form $H \triangleq B$, where H is an atomic formula.
- Add the following *definition introduction* rules:

$$\frac{\{B\theta, \Gamma\theta \vdash C\theta \mid A\theta = H\theta, H \triangleq B\}}{A, \Gamma \vdash C}\ def\mathcal{L}$$

$$\frac{\Gamma \vdash B\theta}{\Gamma \vdash A}\ def\mathcal{R}, A = H\theta, H \triangleq B$$

In *def*$\mathcal{L}$, *all* definition clauses and *all* substitutions have to be considered in the premiss.

Using Definitions (Example)

Let the set of definition clauses be

$$p\ a \triangleq \top, \quad p\ b \triangleq \top, \quad q\ a \triangleq \top, \quad q\ b \triangleq \top, q\ c \triangleq \top$$

Then the following is a successful derivation:

$$\cfrac{\cfrac{\cfrac{;\ \vdash \top}{;\ \vdash q\ b}\ \mathit{def}\mathcal{R} \qquad \cfrac{;\ \vdash \top}{;\ \vdash q\ c}\ \mathit{def}\mathcal{R}}{Y\ ;\ p\ Y \vdash q\ Y}\ \mathit{def}\mathcal{L}}{\vdash \forall x.p\ x \supset q\ x}\ \forall\mathcal{R}; \supset\mathcal{R}$$

Notice that eigenvariables are instantiated by the *def*$\mathcal{L}$ rule.

The Treatment of Names

- New names are treated in proof search through universal quantifiers
- Unfortunately, universal quantifiers do not enforce distinctness of names that is important in some contexts.

  For example,

  $$\forall x \forall y (p\ x\ y) \supset \forall z (p\ z\ z)$$

  is valid in intuitionistic logic.
- An elegant solution to this problem is obtained introducing a new quantifier ∇. [Miller and Tiu, 2003]

The logic $FO\lambda^{\Delta\nabla}$

The full logic has the following characteristics:

- It is an extension of Gentzen's intuitionistic logic
- It incorporates definitions and definitional reflection
- It includes the ∇ quantifier and sequents as a result have the structure

$$\Sigma;\ \sigma_1 \triangleright B_1, \ldots, \sigma_n \triangleright B_n \longrightarrow \sigma_0 \triangleright B_0$$

where Σ is *global* signature and the σ_is are *local* signatures

Formulas for a Two-Level Prover

The formulas themselves reflect a kind of stratification:

Level 0: $G ::= \top \mid \bot \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid \nabla x.G$

Level 1: $D ::= \top \mid \bot \mid A \mid D \wedge D \mid D \vee D \mid \exists x.D \mid \nabla x.D \mid \forall x.D \mid G \supset D$

where atomic formulas have definition clauses such that

- Level 0 "atoms" are defined by level 0 formulas, and
- Level 1 "atoms" are defined by level 1 formulas

The prover attempts to prove *D* formulas.

A Two Phase Proof Strategy

An observation concerning sequents seen by the prover:

> *Only G formulas appear on the left and all the rules applicable to them are invertible*

Thus, proof search for $G \supset D$ can use the following strategy:

- **Step 1** Run a logic programming interpreter with G, treating eigenvariables as logic variables and using λ-abstractions to process ∇
- **Step 2** Collect *all* answer substitutions in Step 1 and attempt to prove D under each.

If there are *no* answers in Step 1, the prover succeeds immediately.

Implementation

- The prover has been implemented in SML of New Jersey.
- Two main ingredients in the implementation:
  - a new, suspension calculus based implementation of higher-order pattern unification [Nadathur and Linnell, ICLP'05]
  - a logic programming interpreter that produces *all* answers in a lazy stream based manner
- Has been used in some interesting applications:
  - bisimulation checking in the π-calculus
  - model checking in a modal logic for the π-calculus
- Available on the web: `http://www.lix.polytechnique/~tiu/lincproject`

An Incompleteness for the Prover

Eigenvariables and logic variables present together in a formula in the left can cause problems.
For example, consider the goal

$$\forall x.\exists y.(px \wedge py \wedge x = y \supset \bot),$$

where p is defined as

$$\{pa \triangleq \top, pb \triangleq \top, pc \triangleq \top\}$$

Solving this goal requires solving *disunification* problems: For each x, find an y such that $x \neq y$.
The current prover forbids occurrences of logic variables in lefthand side formulas.

Conclusions and Future work

- Described a prover that extends logic programming notions but
  - uses Prolog technology and
  - relies on finite success and finite failure
- Extensions of the prover capability may be possible. (E.g. using tabling ideas like in XSB (extended by Pientka) may lead to finiteness in more cases)
- Experimentation with more applications is needed: (E.g. encoding of spi calculus and perhaps the modal-μ calculus.)