Bedwyr

Logic Programming for Protocol Verification

David Baelde

GdT SECSI, Nov 2012

Introduction

Logic programming

- High-level, declarative programming
- Execute purely logical specifications
- Interoperate with other provers

Bedwyr

- Extend Prolog with “model-checking” abilities
- Manipulate syntactic expressions with binding
- Applications: proof & type systems, op. semantics, etc.

From Horn to Definitions

Origins in logic programming:

$$nat\ 0 \text{ :- } \top \qquad nat\ (s\ X) \text{ :- } nat\ X$$

Focused proofs correspond to natural numbers:

$$\dfrac{\dfrac{}{\mathcal{T} \vdash nat\ 0}\ axiom}{\vdots}\ \forall L, \supset L, \ldots$$
$$\mathcal{T} \vdash nat\ (s\ 0)$$

From Horn to Definitions

Origins in logic programming:

$$nat\ 0 \stackrel{\triangle}{=} \top \qquad nat\ (s\ X) \stackrel{\triangle}{=} nat\ X$$

Definitions provide more structure and expressiveness

$$\frac{\{\ (\Gamma, B \vdash P)\theta\ :\ A \stackrel{\triangle}{=} B \text{ and } A'\theta \doteq A\theta\ \}}{\Gamma, A' \vdash P} \qquad \frac{\Gamma \vdash B\theta \quad A \stackrel{\triangle}{=} B}{\Gamma \vdash A\theta}$$

Derivations correspond to natural numbers:

$$\cfrac{\cfrac{}{\vdash nat\ 0}}{\vdash nat\ (s\ 0)}$$

Examples

$$\dfrac{\dfrac{\dfrac{}{\mathit{nat}\ x \vdash \mathit{nat}\ x}\ \text{axiom}}{\mathit{nat}\ (s\ x) \vdash \mathit{nat}\ x}\ \text{def-L}}{x = y, \mathit{nat}\ (s\ x) \vdash \mathit{nat}\ y}\ \text{def-L}$$

Examples

$$\dfrac{\dfrac{\dfrac{}{\mathit{nat}\ x \vdash \mathit{nat}\ x}\ \text{axiom}}{\mathit{nat}\ (s\ x) \vdash \mathit{nat}\ x}\ \text{def-L}}{x = y, \mathit{nat}\ (s\ x) \vdash \mathit{nat}\ y}\ \text{def-L}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash f\ y = f\ y \vee f\ y = z}\ \vee R, = R \quad \dfrac{\dfrac{}{\vdash z = y \vee z = z}\ \vee R, = R}{\mathrm{mem}\ (f\ x)\ [f\ z] \vdash x = f\ y \vee x = z}\ \text{def-L}}{\mathrm{mem}\ (f\ x)\ [f\ (f\ y); f\ z] \vdash x = f\ y \vee x = z}\ \text{def-L}}{\mathrm{mem}\ (f\ x)\ [g\ a; f\ (f\ y); f\ z] \vdash x = f\ y \vee x = z}\ \text{def-L}$$

Bedwyr

Level 0/1

Fragment where eager case analysis is complete for "finite behavior" provability.

Bedwyr searches for proofs of Level 1 formulas; Level 0 hypotheses are fully invertible:

$$\begin{array}{rcl} L0 & ::= & L0 \wedge L0 \mid L0 \vee L0 \mid s = t \mid p\vec{t} \\ & \mid & \nabla x.L0x \mid \exists x.L0x \\ L1 & ::= & L1 \wedge L1 \mid L1 \vee L1 \mid s = t \mid p\vec{t} \\ & \mid & \nabla x.L1x \mid \exists x.L1x \\ & \mid & \forall x.L1x \mid L0 \supset L1 \end{array}$$

+ implicit conditions on definitions $p\vec{x} \stackrel{\triangle}{=} B$.

Examples

Horn specs fit naturally in Level 0, their negations in Level 1

Graph properties in Level 1:

```
Define connected, disconnected, tree by
connected G :=
  forall X Y, node G X -> node G Y -> edgepath G X Y ;
disconnected G :=
  exists X Y, node G X /\ node G Y /\ (edgepath G X Y -> false)
tree G :=
  connected G /\
  forall A Gp, remove_edge G A Gp -> disconnected Gp.
```

Simulation, unbounded number of implications:

```
Define sim : state -> state -> prop by
  sim P Q :=
    forall P1 A, next P A P1 ->
      exists Q1, next Q A Q1 /\ sim P1 Q1.
```

Proof search, operationally

How to find θ (over logic variables) such that $\vdash (A \supset B)\theta$:

1. Collect all σ_i (over universal variables) such that $\vdash A\sigma_i$;
2. Find θ such that, for all i, $\vdash B\sigma_i\theta$.

In particular, finite failure on A trivially yields success on $A \supset \bot$.

Proof search, operationally

How to find θ (over logic variables) such that $\vdash (A \supset B)\theta$:

1. Collect all σ_i (over universal variables) such that $\vdash A\sigma_i$;
2. Find θ such that, for all i, $\vdash B\sigma_i\theta$.

In particular, finite failure on A trivially yields success on $A \supset \bot$.

Standard proof search engine, ran in two modes:

- only accept $\forall$ and $\supset$ on the right;
- on the left, $\exists$ introduces universal variables;
- on the left, unify universal variables instead of logic variables;
- logic variables are not allowed on the left.

Completeness, in practice

Bedwyr is a programming language, not a decision procedure.

Various sources of incompleteness:

- depth first strategy,
- logic variable on the left, other unification errors.

Outcomes of a query

- Yes + substitution: there is a proof
- No: there is no (more) finite behavior proof
- Loop or error: depends

Tabling

Memoization

Do not search for the same goal twice

Cyclic proofs

Classify cycle as success or failure
depending on (co)inductive nature of the defined predicate:

$$\dfrac{\dfrac{?}{\vdash d\,\overline{x}}}{\dfrac{\vdots \quad \vdots \quad \vdots}{\vdash d\,\overline{x}}}$$

$\rightsquigarrow$ tabling enabled by declaring predicate as `(co)inductive`

Examples: cyclic graphs, (multi)simulation

Reasoning about binding

Higher-Order Abstract Syntax

- Representing objects: higher-order terms
- Representing properties using proof-level binding: ∀

```
% wt :  env -> term -> ty -> prop
wt E (var A) T :- member (E,A) T.
wt E (app X Y) T :- wt E X (arrow H T), wt E Y H.
wt E (abs F) (arr Tx T) :- ∀x, wt (x,Tx)::E (F (var x)) T.
```

Reasoning about binding

Higher-Order Abstract Syntax

- Representing objects: higher-order terms
- Representing properties using proof-level binding: ∀

```
% wt :  env -> term -> ty -> prop
wt E (var A) T :- member (E,A) T.
wt E (app X Y) T :- wt E X (arrow H T), wt E Y H.
wt E (abs F) (arr Tx T) :- ∀x, wt (x,Tx)::E (F (var x)) T.
```

Generic Quantification

- Case analysis: ∀ does not fit
- As for closed world assumption and fixed points:
  need to express genericity inside the logic

Proof theory of generic quantification

Local generic context in sequents:

$$\frac{\Gamma, (\sigma, x) \rhd Hx \vdash \sigma' \rhd G}{\Gamma, \sigma \rhd \nabla x.Hx \vdash \sigma' \rhd G} \quad \frac{\Gamma \vdash (\sigma, x) \rhd Gx}{\Gamma \vdash \sigma \rhd \nabla x.Gx} \quad \frac{}{\Gamma, \sigma \rhd G \vdash \sigma \rhd G}$$

A few theorems relating ∇ and other connectives:

$$\begin{aligned} \nabla x.(Bx \wedge Cx) &\equiv \nabla x.Bx \wedge \nabla x.Cx \\ \nabla x_\alpha \forall y_\beta.Bxy &\equiv \forall h_{\alpha\to\beta} \nabla x.Bx(hx) \\ \nabla x.(t = s) &\equiv (\lambda x.t) = (\lambda x.s) \end{aligned}$$

That's enough for our immediate needs, no need to answer hard questions such as $(\nabla x.P) \equiv P$ or $(\forall x.P\ x) \supset (\nabla x.\ P\ x)$.

Generic quantification in Bedwyr

```
Define wt :  env -> term -> ty -> prop by
 wt E (var A) T := member (E,A) T;
 wt E (app X Y) T := wt E X (arrow H T) ∧ wt E Y H;
 wt E (abs F) (arr Tx T) := ∇x, wt (x,Tx)::E (F (var x)) T.
```

Semantics

- Logical meaning: adequacy and cut elimination
- Operational meaning: “for a fresh constant”

Examples

- λ-calculus, π-calculus and open bisim, logics, etc.

SPEC

SPEC

Spi-calculus bisimulation

Shown to be decidable & implemented by A. Tiu
Implemented in Bedwyr, with a user-friendly front-end
Required new non-logical features (some harmless, some not)

Secrecy

A simple case study to get a feel for the tool

1. Build (complete) traces from one-step transitions
2. Reify logic variables
3. Translate traces to deducibility constraints, solve them
4. Formalize leakage rather than secrecy:

$$\texttt{leak P := trace P T} \wedge \texttt{traceleak T}$$

Conclusion & Future

Bedwyr

- Very high-level programming: concise, maintainable
- Proof theoretical support for
  $\forall\exists$ search, generic quantification, tabling

Possible future directions

- A nice playground for new experiments
  trace equivalence, attack reconstruction,
  analyze concurrency to avoid state explosion
- It should be only about logic:
  reasoning on same specs, possibly interactively