Reasoning about Computations Using Two-Levels of Logic

Dale Miller

INRIA-Saclay & LIX/École Polytechnique
Palaiseau, France

APLAS 2010, 1 December 2010, Shanghai

Overview of high-level goals

- *Design* a logic for reasoning about computation: e.g., capture
  - inductive and co-inductive reasoning,
  - may and must judgments, and
  - binding and substitution.
- *Reason* directly on logic specifications of computation.
- *Formalize* the reasoning logic as proof theory in the tradition of Gentzen and Girard.
- *Implement* the proof theory and apply to examples.

This research effort spans the years 1997 to 2010 and has involved about 6 researchers.

Outline

A logic for specifications

The open and closed world assumptions

Generic quantification

The Abella prover

Related work: nominal logic and POPLMark

Outline

A range of specification languages

For *dynamic semantics:*

- process calculus: CCS, CSP, π-calculus
- abstract machines: Krivine machine, SECD
- finite state machines
- Petri nets

For *static semantics:*

- typing judgments of many kinds

In recent years,

- *operational semantics* has become the standard for *defining* dynamic semantics, while
- *denotational semantics* can sometimes capture *deep results* about computation.

An example of operational semantics

$$\textsc{par}: \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$$

$$\textsc{com}: \quad \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad \textsc{close}: \quad \frac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

$$\textsc{res}: \quad \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'} \quad y \notin \mathrm{n}(\alpha) \qquad \textsc{open}: \quad \frac{P \xrightarrow{\overline{x}y} P'}{(y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin \mathrm{fn}((y)P') \end{array}$$

Some operational semantic rules cut from Milner, Parrow, & Walker, "A Calculus of Mobile Processes, Part II" (1989).

Logic programming specifications

Most operational semantics specifications can be encode within first-order Horn clauses.

Prolog can *animate* such specifications.

The *quality* of such encodings is, however, extremely important when attempting to reason about what is encoded.

A serious quality issue is the treatment of *bindings* in syntactic expressions and computation traces.

- programming languages, type systems
- λ-calculus
- π-calculus

Abstract syntax

Approaches to encoding syntax have slowly grown more abstract over the years.

Strings: Formulas-as-strings: "well-formed formulas (wff)". Church and Gödel did meta-logic with strings (!).

Parse trees: Removing white space, parenthesis, infix/prefix operators, and keywords yields recursive term structures for syntax.

Abstract syntax

Approaches to encoding syntax have slowly grown more abstract over the years.

Strings: Formulas-as-strings: "well-formed formulas (wff)". Church and Gödel did meta-logic with strings (!).

Parse trees: Removing white space, parenthesis, infix/prefix operators, and keywords yields recursive term structures for syntax.

However: bindings are treated too concretely. One of the oldest of the approaches to making bindings more abstract is:

λ-trees: Syntax is treated via α-conversion and weak forms of β-reduction (eg, typed β-conversion or β_0). Unification (modulo $\alpha\beta$) is used to decompose syntax.

Abstract syntax

Approaches to encoding syntax have slowly grown more abstract over the years.

Strings: Formulas-as-strings: "well-formed formulas (wff)". Church and Gödel did meta-logic with strings (!).

Parse trees: Removing white space, parenthesis, infix/prefix operators, and keywords yields recursive term structures for syntax.

However: bindings are treated too concretely. One of the oldest of the approaches to making bindings more abstract is:

λ-trees: Syntax is treated via α-conversion and weak forms of β-reduction (eg, typed β-conversion or β_0). Unification (modulo $\alpha\beta$) is used to decompose syntax.

(Sometimes also called *higher-order abstract syntax* but that term is also confused with another encoding technique.)

An example: call-by-name evaluation

$$\frac{}{\lambda x.R \Downarrow \lambda x.R} \qquad \frac{M \Downarrow \lambda x.R \qquad R[x/N] \Downarrow V}{(M\ N) \Downarrow V}$$

Application $app : tm \to (tm \to tm)$.
Abstraction $abs : (tm \to tm) \to tm$.
Evaluation *eval* binary predicate over type tm.

$$\forall R\ [eval\ (abs\ R)\ (abs\ R)]$$

$$\forall M, N, V, R\ [eval\ M\ (abs\ R) \wedge eval\ (R\ N)\ V \supset eval\ (app\ M\ N)\ V]$$

The variable R is of higher-type $tm \to tm$ and the application $(R\ U)$ is a "meta-level" β-redex.

An example: simple typing

$$\frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}\ \dagger \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash (M\ N) : \beta}$$

Proviso $\dagger$: x does not occur in Γ (x is "new").

An example: simple typing

$$\frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}\ \dagger \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash (M\ N) : \beta}$$

Proviso $\dagger$: x does not occur in Γ (x is "new").

Arrow type constructor $arr : ty \to ty \to ty$.
Typing judgment of is a binary predicate between tm and ty.

$$\forall R, A, B\ [\ \forall x[of\ x\ A \supset of\ (R\ x)\ B] \supset of\ (abs\ R)\ (arr\ A\ B)]$$

$$\forall M, N, A, B\ [of\ M\ (arr\ A\ B) \wedge of\ N\ A \supset of\ (app\ M\ N)\ B]$$

Where did the proviso $\dagger$ go?

An example: simple typing (continued)

Consider building a proof of a universally quantified implications (in Gentzen's natural deduction proof system):

$$\dfrac{\begin{array}{c}(of\ x\ A)\\ \vdots\\ of\ (R\ x)\ B\end{array}}{\dfrac{\forall x[of\ x\ A \supset of\ (R\ x)\ B]}{of\ (abs\ R)\ (arr\ A\ B)}}\ \dagger$$

The proviso † requires that the eigenvariable x is not free in any non-discharged assumption.

This proviso is pushed into the logic: specifications within the logic do not need to deal with it directly.

Outline

We need the open-world assumption

To prove $\forall x[of\ x\ A \supset of\ (R\ x)\ B]$

- generate a *new* "constant," say c, and
- assume a *new* assumption about c and then
- prove $of\ c\ A \vdash of\ (R\ c)\ B$

Our logic must be willing to accept new constants and scoped assumptions about them.

Thus, we need the *open-world assumption* in the specification logic to support the λ-tree abstraction.

We need the closed-world assumption

Consider proving the theorem:

$$\forall n[\ \mathit{fib}(n) = n^2 \ \supset \ n \leq 20\ ].$$

We do not want to assume the existence of a new natural number n such that the n^{th} Fibonacci number is n^2.

Instead, we solve for n and get 0, 1, and 12, then show that

$$0 \leq 20 \wedge 1 \leq 20 \wedge 12 \leq 20.$$

The set of natural numbers is a *closed* type.
Closedness is needed for induction.

How can we have both an open and closed world?

Our solution here:

How can we have both an open and closed world?

Our solution here: Use two logics.

How can we have both an open and closed world?

Our solution here: Use two logics.

The *specification logic* is a restricted second-order intuitionistic logic. Proofs are given by, say, Gentzen's LJ.

How can we have both an open and closed world?

Our solution here: Use two logics.

The *specification logic* is a restricted second-order intuitionistic logic. Proofs are given by, say, Gentzen's LJ.

The *reasoning* logic:

- Church's Simple Theory of Types (intuitionistic or classical)
- (this includes induction and co-inductive proof rules)
- Provability of the specification logic is a predicate:
  The binary predicate $\{\Gamma \vdash G\}$ holds exactly when the sequent $\Gamma \longrightarrow G$ is provable in the specification logic.
- plus one more thing...

Examples of reasoning logic theorems

The following should be theorems of the reasoning logic.

- $\forall M, V, A\ [\{\vdash \mathit{eval}\ M\ V\} \wedge \{\vdash \mathit{of}\ M\ A\} \supset \{\vdash \mathit{of}\ V\ A\}]$
- $\forall A\ \neg\{\vdash \mathit{of}\ (\mathit{abs}\ \lambda x.(\mathit{app}\ x\ x)\ A)\}$
- If Ω is the term

$$(\mathit{app}\ (\mathit{abs}\ \lambda x.(\mathit{app}\ x\ x))\ (\mathit{abs}\ \lambda x.(\mathit{app}\ x\ x)))$$

  then $\forall V.\ \neg\{\vdash \mathit{eval}\ \Omega\ V\}$.

The reasoning logic can quantify over the terms, formulas, and contexts in the specification logic.

Outline

Quiz

Let $\langle x, y \rangle$ be a pairing constructor. If the formula $\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle]$ follows from the assumptions

$$\Delta = \{\forall x \forall y [q\ x\ x\ y],\ \forall x \forall y [q\ x\ y\ x],\ \forall x \forall y [q\ y\ x\ x]\}$$

what can we say about the terms t_1, t_2, and t_3?

Answer:

Quiz

Let $\langle x, y \rangle$ be a pairing constructor. If the formula $\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle]$ follows from the assumptions

$$\Delta = \{\forall x \forall y [q\ x\ x\ y],\ \forall x \forall y [q\ x\ y\ x],\ \forall x \forall y [q\ y\ x\ x]\}$$

what can we say about the terms t_1, t_2, and t_3?

Answer: the terms t_2 and t_3 are equal.

The answer concerns proofs and not models: *i.e.*, the domain of the quantifiers $\forall u \forall v$ does not matter.

Quiz

Let $\langle x, y \rangle$ be a pairing constructor. If the formula $\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle]$ follows from the assumptions

$$\Delta = \{\forall x \forall y [q\ x\ x\ y],\ \forall x \forall y [q\ x\ y\ x],\ \forall x \forall y [q\ y\ x\ x]\}$$

what can we say about the terms t_1, t_2, and t_3?

Answer: the terms t_2 and t_3 are equal.

The answer concerns proofs and not models: *i.e.*, the domain of the quantifiers $\forall u \forall v$ does not matter.

The following should be a theorem in the reasoning logic:

$$\forall t_1, t_2, t_3\ [\{\Delta \vdash \forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle]\} \supset t_2 = t_3]$$

Another example

Let c be a constant. It is not possible to prove

$$\forall w.\ w = c$$

in the open-world setting.
Thus, the following should be a theorem of the reasoning logic.

$$\forall w.\neg\{\vdash \forall x.\ x = w\}$$

Another example

Let c be a constant. It is not possible to prove

$$\forall w.\ w = c$$

in the open-world setting.
Thus, the following should be a theorem of the reasoning logic.

$$\forall w.\neg\{\vdash \forall x.\ x = w\}$$

How do we capture the "intensional" aspects of the specification logic universal quantifier in the reasoning logic?

Still other examples

There are other examples in computer science (apart from logic) where new names and scoping for them is needed.

- names in the π-calculus,
- nonces and session keys in security protocols, and
- reference locations in imperative programming.

Proof in the specification logic as an inductive definition

Object-logic provability can be defined inductively using the following Prolog-like clauses.

$$\{\Delta \vdash \top\} \text{ :- } \top.$$

$$\{\Delta \vdash A\} \text{ :- } \mathit{memb}\ D\ \Delta, \mathit{instan}\ D\ (G \supset A), \{\Delta \vdash G\}.$$

$$\{\Delta \vdash G_1 \wedge G_2\} \text{ :- } \{\Delta \vdash G_1\}, \{\Delta \vdash G_2\}.$$

$$\{\Delta \vdash D \supset G\} \text{ :- } \{D, \Delta \vdash G\}.$$

$$\{\Delta \vdash \exists x.\ G\ x\} \text{ :- } \exists x.\{\Delta \vdash G\ x\}.$$

$$\{\Delta \vdash \forall x.\ G\ x\} \text{ :- } \nabla x.\ \{\Delta \vdash G\ x\}.$$

If ∇ (pronounced "nabla") is replaced by $\forall$, then the previous examples are not provable. But what is ∇?

∇-quantification

This is third quantifier along with $\forall$ and $\exists$.

It is used in the reasoning logic and not in the specification logic.

Seems to be of little interest if the specification logic does not involve binding.

Logical aspects of the ∇-quantifier

Some theorems

$$\nabla x \neg Bx \equiv \neg \nabla x Bx \qquad \nabla x(Bx \wedge Cx) \equiv \nabla x Bx \wedge \nabla x Cx$$
$$\nabla x(Bx \vee Cx) \equiv \nabla x Bx \vee \nabla x Cx \qquad \nabla x(Bx \supset Cx) \equiv \nabla x Bx \supset \nabla x Cx$$
$$\nabla x \forall y\ Bxy \equiv \forall h \nabla x\ Bx(hx) \qquad \nabla x \exists y\ Bxy \equiv \exists h \nabla x\ Bx(hx)$$
$$\nabla x\ \top \equiv \top \qquad \nabla x\ \bot \equiv \bot$$

Thus ∇ can always be given atomic scope within formulas.

Some non-theorems

$$\nabla x \nabla y Bxy \supset \nabla z Bzz \qquad \nabla x Bx \supset \exists x Bx$$
$$\nabla z Bzz \supset \nabla x \nabla y Bxy \qquad \forall x Bx \supset \nabla x Bx$$
$$\forall y \nabla x Bxy \supset \nabla x \forall y Bxy \qquad \exists x Bx \supset \nabla x Bx$$

Two structural rules for ∇-quantification

The following two equivalences are not forced.
The exchange principle

$$\nabla x \nabla y . B\, x\, y \equiv \nabla y \nabla x . B\, x\, y$$

seems natural.

The following principle

$$(\nabla x_\tau . B) \equiv B \qquad\qquad (x \text{ is not free in } B)$$

implies that there are an infinite number of members of the type τ.
This assumption is awkward in some settings but natural in others.

The Abella prover accepts both of these principles.

∇-quantification and equality

Notice $(\forall x.t = s) \equiv (\lambda x.t = \lambda x.s)$ fails in general. For example,

$$\forall w.\neg.(\lambda x.x = \lambda x.w)$$

is a desired theorem but $\forall w.\neg.\forall x.(x = w)$ is not a theorem since it is false in the singleton model.

The following is the equivalence we want:

$$(\nabla x.t = s) \equiv (\lambda x.t = \lambda x.s)$$

This equivalence also suggests how to implement ∇: if your system has equality / unification on λ-terms, you are already close....

Outline

An interactive theorem prover for the reasoning logic. Implemented in OCaml. Written by Andrew Gacek as part of his PhD (University of Minnesota) and postdoc (INRIA-Saclay).

π-calculus in Abella: syntax of processes and actions

```
kind name, proc   type.

type null         proc.
type taup         proc -> proc.
type plus, par    proc -> proc -> proc.
type match, out   name -> name -> proc -> proc.
type in           name -> (name -> proc) -> proc.
type nu           (name -> proc) -> proc.

kind action       type.
type tau          action.
type up, dn       name -> name -> action.

type one          proc ->          action  ->          proc  -> o.
type oneb         proc -> (name -> action) -> (name -> proc) -> o.
```

This is a λProlog signature file. The first lines are roughly equivalent to:

$$\mathrm{P} ::= 0 \mid \bar{x}y.\mathrm{P} \mid x(y).\mathrm{P} \mid \tau.\mathrm{P} \mid (x)\mathrm{P} \mid [x = y]\mathrm{P} \mid \mathrm{P}|\mathrm{P} \mid \mathrm{P} + \mathrm{P}.$$

π-calculus in Abella: one-step transitions

```
oneb (in X M) (dn X) M.
one  (out X Y P) (up X Y) P.

one  (taup P) tau P.

one  (match X X P) A Q :- one  P A Q.
oneb (match X X P) A M :- oneb P A M.

one  (plus P Q) A R :- one  P A R.
one  (plus P Q) A R :- one  Q A R.
oneb (plus P Q) A M :- oneb P A M.
oneb (plus P Q) A M :- oneb Q A M.

one  (par P Q) A (par P1 Q) :- one P A P1.
one  (par P Q) A (par P Q1) :- one Q A Q1.
oneb (par P Q) A (x\par (M x) Q) :- oneb P A M.
oneb (par P Q) A (x\par P (N x)) :- oneb Q A N.

one  (nu P) A (nu Q)          :- pi x\ one  (P x) A (Q x).
oneb (nu P) A (y\ nu x\Q x y) :- pi x\ oneb (P x) A (y\ Q x y).
oneb (nu M) (up X) N          :- pi y\ one  (M y) (up X y) (N y).

one (par P Q) tau (nu y\ par (M y) (N y)) :- oneb P (dn X) M , oneb Q (up X) N.
one (par P Q) tau (nu y\ par (M y) (N y)) :- oneb P (up X) M , oneb Q (dn X) N.
one (par P Q) tau (par (M Y) T) :-  oneb P (dn X) M, one Q (up X Y) T.
one (par P Q) tau (par R (M Y)) :-  oneb Q (dn X) M, one P (up X Y) R.
```

This is a λProlog module file. This is roughly equivalent to a page of SOS inference rules (but with no side conditions!)

The π-calculus in Abella: Theorems and proofs

```
CoDefine sim : proc -> proc -> prop by
  sim P Q :=
    (forall A P1, {one P A P1}      -> exists Q1,{one Q A Q1} /\ sim P1 Q1) /\
    (forall X M,  {oneb P (dn X) M} -> exists N, {oneb Q (dn X) N} /\
                                                 forall W, sim (M W) (N W)) /\
    (forall X M,  {oneb P (up X) M} -> exists N, {oneb Q (up X) N} /\
                                                 nabla w, sim (M w) (N w)).

Theorem sim_refl : forall P, sim P P.
coinduction. intros. unfold.
  intros. apply CH with P = P1. search.
  intros. exists M. split. search.
    intros. apply CH with P = M W. search.
  intros. exists M. split. search.
    intros. apply CH with P = M n1. search.

Theorem sim_trans : forall P Q R, sim P Q -> sim Q R -> sim P R.
coinduction. intros. case H1. case H2. unfold.
  intros. apply H3 to H9. apply H6 to H10. apply CH to H11 H13. search.
  intros. apply H4 to H9. apply H7 to H10.
  exists N1. split. search.
    intros. apply H11 with W = W. apply H13 with W = W.
    apply CH to H14 H15. search.
  intros. apply H5 to H9. apply H8 to H10.
  apply CH to H11 H13. search.
```

Thus, simulation is a pre-order. The bisimulation corresponds to *open bisimulation*.

A type preservation theorem

$$\forall M, V, A\ [\{\vdash \mathit{eval}\ M\ V\} \wedge \{\vdash \mathit{of}\ M\ A\} \supset \{\vdash \mathit{of}\ V\ A\}]$$

Proof. By induction on $\{\vdash \mathit{eval}\ M\ V\}$.

Base case: $M = V = (\mathit{abs}\ R)$. The conclusion is trivial.

Inductive case: Here, $M = (\mathit{app}\ M'\ N)$ and both $\{\vdash \mathit{eval}\ M'\ (\mathit{abs}\ R)\}$ and $\{\vdash \mathit{eval}\ (R\ N)\ V\}$ have shorter proofs (for some $R : \mathit{tm} \to \mathit{tm}$). From $\{\vdash \mathit{of}\ (\mathit{app}\ M'\ N)\ A\}$ we have $\{\vdash \mathit{of}\ M'\ (\mathit{arr}\ B\ A\}$ and $\{\vdash \mathit{of}\ N\ B\}$ (for some $B : \mathit{ty}$). By induction, we have $\{\vdash \mathit{of}\ (\mathit{abs}\ R)\ (\mathit{arr}\ B\ A)\}$ Thus, $\{\vdash \forall x.\ \mathit{of}\ x\ B \supset \mathit{of}\ (R\ x)\ A\}$. Since this is logic, we know

$$\{\mathit{of}\ N\ B \supset \mathit{of}\ (R\ N)\ A\} \quad \text{and} \quad \{\vdash \mathit{of}\ (R\ N)\ A\}.$$

By induction again, we know that $\{\vdash \mathit{of}\ V\ A\}$. QED.
A substitution theorem for free!

Other examples done in Abella

- Process calculi
  - π-calculus
    - Various examples of bisimulation (model checking)
    - Meta-theorems: eg, bisim is a congruence
  - Calculus of communicating systems
- λ-calculus
  - Strong and weak normalization for simply-typed terms
  - Church-Rosser
  - Standardization
  - Evaluation and typing
  - Type uniqueness for simply-typed terms
- Programming languages
  - POPLmark Challenge problems 1a and 2a
  - Evaluation by explicit substitution
  - PCF: Programming language for Computable Functions

Outline

Related work: Nominal Logic

Pitts & Gabbay developed a semantic approach to a first order theory of names and binding called *nominal logic*.

It contains a self-dual quantifier similar to ∇.

Relating nominal logic to the logic here has been difficult since their semantic assumptions are hard to capture in proof theory.

If one weakens the logic sufficiently, one can get results:

- Gacek [PPDP 10] shows an exact match between αProlog (using nominal techniques) and a logic programming language using λ-terms and ∇-quantification.

Related work: POPLMark Challenge

Can *existing* theorem provers can be augmented (with, say, packages) so that they can treat binding structures well.

Solutions: Nominal Isabelle, Coq libraries for locally nameless encodings, etc.

In this talk: bindings are fundamentally part of logic.

- The very nature of logic and proof requires bindings in the form of quantification and substitutions.
- Binding is not an "add-on" package.
- Start with a logic (Church's Simple Theory of Types) where binders and λ-conversions are primitive.
- Add just ∇-quantification.

References

Theory, Design, Applications

- λProlog: M, Nadathur (ICLP 86, LICS 89, etc)
- Two-level logic (pre-∇): McDowell, M (LICS 97, ToCL 02)
- ∇-quantification: M, Tiu (LICS 03, ToCL 05)
- π-calculus and ∇: Tiu (CONCUR 05, ToCL 10)

Systems

- Teyjus: Nadathur, *et. al.* (CADE 99, ICLP 05, LPAR 05) (a compiler for λProlog)
- Bedwyr: Baelde, Gacek, M, Nadathur, Tiu (ESHOL 05, CADE 07) (model checker for a fragment of Abella)
- Abella: Gacek, M, Nadathur (IJCAR 08, LICS 08, I&C 11)

All papers and systems are available for download.

Thank you