A logic for reasoning about logic specifications

Dale Miller
INRIA/Futurs and École polytechnique

Outline

1. A new architecture for a theorem prover.
2. Proof search, logic programming, proof theory
3. A proof theoretic approach to definitions
4. A new quantifier ∇ (nabla)
5. Example: object-level provability
6. Example: π-calculus simulation
7. Conclusions

Traditional structure of theorem provers for reasoning about computation

(1) Implement mathematics

- Choose among constructive mathematics, classical logic, set theory, etc.
- Provide abstractions such as sets and/or functions.

(2) Reduce computation to mathematics

- via denotational semantics and/or
- via inductively defined data types for data and inference systems.

What could be wrong with this approach? Isn't mathematics the universal language?

"Intensional aspects" of specifications — bindings, names, resource accounting, etc — generally require heavy encodings.

Advances in the proof search paradigm

Logic programming has been extended in recent years in several directions, including the following two extensions.

- *Higher-order abstract syntax (HOAS)* is captured with higher-type quantification and logic support for λ-terms. λProlog was the first programming language that incorporated HOAS. Specification languages, such as Isabelle and Twelf, also provide it. Sometimes called *λ-tree syntax* to distinguish the functional programming approach to HOAS.
- *Linear logic (LL)* greatly increases the expressiveness of logic programming, allowing direct and modular OS specification of state, exceptions, continuations, and concurrency in programming languages. Lolli and Forum are examples of linear logic programming languages (admitting also HOAS).

HOAS and LL strain traditional theorem provers

Coding HOAS and LL into traditional mathematics is often complex and can obscure meaning.

- HOAS is really an approach to *syntax*.
  - Encoding it using functions in a rich, higher-order logic is problematic: too many functions (exotic terms), extensional equality identifies too many terms, induction is difficult, etc.
  - Encoding using first-order terms is problematic: one must re-implement many complex logical concepts (substitution, alpha-conversion, etc). The logic of binders is captured only indirectly.
- Encoding linear logic via semantics is difficult. Operational encodings, via multiset rewriting, ACI unification, etc, can generally capture only some aspects linear logic (additives/multiplicatives, quantifiers, modals, cut-elimination, etc).

A new architecture for a theorem prover

One meta-logic. This is a formalized replacement for mathematical reasoning, incorporating induction, co-induction, HOAS, and intuitionistic logic. Atomic judgments will include provability within an object-level logic.

A few object-logics. Here, specifications are given as proof search specifications (logic programs) using such things as Horn clauses, or higher-order linear logic (*e.g.*, Forum).

Consider, for example, the π-calculus.

1. The one step transitions $P \xrightarrow{A} P'$ for the π-calculus is given by a simple logic program in an object-level logic (Horn clauses).
2. Simulation of P and Q is a meta-level predicate defined such that forall A and P' if it is *provable* that $P \xrightarrow{A} P'$ then there exists a Q' such that $Q \xrightarrow{A} Q'$ is *provable* and P' is simulated by Q'.

Thus proving properties of simulation requires reasoning about the proof search specification in the object-level language.

Structure of the meta-logic

The meta-logic features the following.

- Intuitionistic logic (no linear logic at the "mathematics level"). Could be classical as well.
- Induction and co-induction are generally needed.
- A new quantifier ∇ will be used to provide a meta-level treatment of object-level eigenvariables.
- Object-level provability is specified via logic programs.
- Case analysis of how object-level judgments can be proved.

The proof theoretic notion of *definitions* is used to address the last two points.

A proof theoretic notion of definition

A *definition* is a finite set of *clauses*

$$\forall \bar{x}[p_1(\bar{t}_1) \stackrel{\triangle}{=} B_1] \quad \ldots \quad \forall \bar{x}[p_n(\bar{t}_n) \stackrel{\triangle}{=} B_n] \quad (n \geq 0)$$

For $i = 1, \ldots, n$,

- p_i is a predicate constant,
- free variables of B_i are also free in the list $\bar{t}_i$, and
- all variables free in $\bar{t}_i$ are contained in the list $\bar{x}_i$.

The formula B_i is the *body* and $p_i(\bar{t}_i)$ is the *head* of the i^{th} clause.

The predicate symbols $p_1, \ldots, p_n$ are not distinct predicates: definitions act to define predicates by mutual recursion.

The symbol $\stackrel{\triangle}{=}$ is not a logical connective: it is used just to denote definitional clauses.

Using $\stackrel{\triangle}{=}$ directly as logical equivalence can damage proof search. We need something more clever.

Right introduction for defined atoms

Left and right introduction rules for atomic formulas are given for a fixed definition and equality theory.

$$\frac{\Delta \longrightarrow B\theta}{\Delta \longrightarrow A}\ def\mathcal{R}, \text{ where } A = H\theta \text{ for some clause } \forall \bar{x}.[H \stackrel{\triangle}{=} B].$$

If we think of a definition as a logic program, then this rule is *backchaining*.

Notice that (reading from bottom up)

- *matching* is used to select a clause from a definition, and
- the atom is replace by *some* body of a matching clause.

Left introduction for defined atoms

$$\frac{\{B\theta, \Delta\theta \longrightarrow C\theta \mid \theta \in csu(A, H) \text{ for some clause } \forall\bar{x}.[H \stackrel{\triangle}{=} B]\}}{A, \Delta \longrightarrow C}\ def\mathcal{L}.$$

The variables $\bar{x}$ need to be chosen so that they are not free in any formula of the lower sequent. This rule is due to [Eriksson 91].

The set of premises can be empty, finite, or infinite since definitions and the set $csu(A, H)$ can be infinite. In some theories, minimal CSUs are not effectively computable.

While the formal theory of definitions handles this general case, we shall only use this left rule when CSUs can be replaced with MGUs (most general unifiers).

Notice that (reading from bottom up)

- *unification* is used to select a clause from a definition, and
- the atom is replace by *all* bodies of unifying clauses.

Example: computing max

$$
\begin{array}{rl}
a\ (s\ z) & \triangleq \top. \\
a\ (s\ (s\ (s\ z))) & \triangleq \top. \\
a\ z & \triangleq \top. \\
maxa\ N & \triangleq (a\ N) \wedge \forall x(a\ x \supset x \leq N). \\
z \leq N & \triangleq \top. \\
(s\ N) \leq (s\ M) & \triangleq N \leq M.
\end{array}
$$

$maxa\ N$ holds if and only if N is the maximum value for a.

$$
\dfrac{\dfrac{\longrightarrow \top}{\longrightarrow a\ 3}\ def\mathcal{R} \qquad \dfrac{\dfrac{\longrightarrow 1 \leq 3 \quad \longrightarrow 3 \leq 3 \quad \longrightarrow 0 \leq 3}{x : a\ x \longrightarrow x \leq 3}\ def\mathcal{L}}{\longrightarrow \forall x(a\ x \supset x \leq 3)}\ \forall\mathcal{R}, \supset\mathcal{R}}{\longrightarrow maxa\ 3}\ def\mathcal{R}
$$

Example: evaluation of a conditional

Consider defining some rules for a conditional (if) in a functional programming language.

$$\vdots$$

$$(if\ B\ M\ N) \Downarrow V \triangleq B \Downarrow true\ \wedge M \Downarrow V.$$
$$(if\ B\ M\ N) \Downarrow V \triangleq B \Downarrow false \wedge N \Downarrow V.$$

$$\vdots$$

Now consider the following fragment of a proof

$$\dfrac{\dfrac{B \Downarrow true\ , M \Downarrow V \longrightarrow M \Downarrow V}{B \Downarrow true\ \wedge M \Downarrow V \longrightarrow M \Downarrow V}\ \wedge\mathcal{L} \quad \dfrac{B \Downarrow false, M \Downarrow V \longrightarrow M \Downarrow V}{B \Downarrow false \wedge M \Downarrow V \longrightarrow M \Downarrow V}\ \wedge\mathcal{L}}{(if\ B\ M\ M) \Downarrow V \longrightarrow M \Downarrow V}\ def\,\mathcal{L}$$

Roles for $def\mathcal{R}$ and $def\mathcal{L}$

$$\frac{\vdots}{\longrightarrow A}\ def\mathcal{R}$$ corresponds to *backchaining*.

$$\frac{\vdots}{A \longrightarrow}\ def\mathcal{L}$$ corresponds to *finite failure*.

$$\frac{\vdots}{A \longrightarrow B}\ def\mathcal{L} + def\mathcal{R}$$ corresponds to *simulation*[†].

[†] McDowell, Miller, and Palamidessi. *Encoding transition systems in sequent calculus.* *TCS*, 2001.

Restrictions on definitions

In general, cut can not be eliminated which out restrictions on definitions. Consider:

$$p \stackrel{\triangle}{=} p \supset \bot \ .$$

The literature contains three ways to restriction definitions so that cut-elimination can hold.

1. Do not allow the body of definitions to contain implications. This is a rather strong restriction, but corresponds to Horn clauses. [Schroeder-Heister]
2. Remove contraction, which moves us away from intuitionistic logic to linear or relevant logics. [Girard, Schroeder-Heister]
3. Give predicates and formulas a level and require definitions to be stratified [McDowell & Miller].

The Level 0/1 prover

An attempt to do proof search using both $def\mathcal{R}$ and $def\mathcal{L}$. It combines two logic programming interpreters: **Level 1** works on hereditary Harrop formulas and **Level 0** works on Horn clauses.

Level 1

- works on sequents of the form $\Sigma: \longrightarrow R$, where Σ contains the eigenvariables of the proof search.
- uses $A \stackrel{\triangle}{=} B$ in the direction $A \supset B$ only.
- employs logic variables and unification (under a mixed prefix) in the usual way.

As soon as proof search encounters a sequent with a formula on the left (from an implication-right rule) $\Sigma: L \longrightarrow R$, it immediately calls Level 0 with goal L and classifies the variables in Σ as logic variables (!) for Level 0.

Level 0

- uses $A \stackrel{\triangle}{=} B$ in the direction $A \subset B$ only.
- Logic variables inside Level 0 are eigen-variables the above level.
- If Level 1 leaves some logic variables in L, Level 0 aborts.
- If Level 0 encounters an atomic goal defined using an implication, it aborts.

Despite its theoretical limitations, Level 0/1 has been used to implement symbolic bisimulation, some model checking, game playing (tic-tac-toe), and negation in logic programming.

It suggests new theory on which to work:

- "Unification under a mixed prefix" needs to be extended to include inequalities (∇ will probably be important too).
- There shouldn't be two provers but just one which calls itself in a completely symmetric fashion, making failure and success completely symmetric (for terminating computation, of course). Game semantics might provide a flexible foundations for exploring this approach.

The Level 0/1 prover (cont)

Alwen Tiu implemented this with SML code contributions from Nadathur & Linnell for unification and from Miller for stream-based proof search. Tiu added many innovations and features as well. It is available from the Tiu's web site.

A new implementation based on OCaml is planned. Major enhancements should include: improved performance and tabling to recognize cycles.

Many additional examples should be explored. So far each major example suggested new theory to work out.

E.g.: dealing with the difference between "open" and "late" bisimulation in the π-calculus suggests that some details of "unification failure" can be used in a positive way to guide proof search.

An example: call-by-name evaluation and simple typing

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

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

$$\forall M, N, A, B\ [typeof\ M\ (A \to B) \wedge typeof\ N\ A \supset typeof\ (app\ M\ N)\ B]$$

$$\forall R, A, B\ [\forall x[typeof\ x\ A \supset typeof\ (R\ x)\ B] \supset typeof\ (abs\ R)\ (A \to B)]$$

The first three clauses are Horn clauses; the fourth is not.

Proof of type preservation

Theorem: If P evaluates to V and P has type T then V has type T.

Proof: Prove by structural induction on a proof of $\mathit{eval}\ \ P\ V$: for all T, if $\vdash \mathit{typeof}\ P\ T$ then $\vdash \mathit{typeof}\ V\ T$.

The proof of $\mathit{eval}\ \ P\ V$ must end by backchaining on one of the formulas encoding evaluation.

Case 1: Backchaining on the *eval* of *abs*: thus P and V are equal to $\mathit{abs}\ R$, for some R, and the consequent is immediate.

Case2: Backchaining on the *eval* of *app*: thus P is of the form $(app\ M\ N)$ and for some R, there are shorter proofs of $eval\ \ M\ (abs\ R)$ and $eval\ \ (R\,N)\ V$.

Since $\vdash typeof\,(app\ M\ N)\ T$, this typing judgment must have been proved using backchaining and, hence, there is a U such that $\vdash typeof\,M\ (arr\,U\ T)$ and $\vdash typeof\,N\ U$.

Using the inductive hypothesis, we have $\vdash typeof\,(abs\ R)\ (arr\,U\ T)$. This formula must have been proved by backchaining on the *typeof* formula for *abs*, and, hence, $\vdash \forall x.[typeof\,x\ U \supset typeof\,(R\,x)\ T]$.

Since our logic of judgments is intuitionistic logic, we can instantiate this quantifier with N and use cut and cut-elimination to conclude that $\vdash typeof\,(R\,N)\ T$. Using the inductive hypothesis a second time yields $\vdash typeof\,V\ T$.

QED

The collapse of eigenvariables

A cut-free proof search of

$$\forall x \forall y. P\, x\, y$$

first introduces two new eigenvariables c and d and then attempts to prove $P\, c\, d$.

Eigenvariables have been used to encode names in π-calculus [Miller93], nonces in security protocols [Cervesato, et. al. 99], reference locations in imperative programming [Chirimar95], etc.

Since

$$\forall x \forall y. P\ x\ y \supset \forall z. P\ z\ z$$

is provable, it follows that the provability of $\forall x \forall y. P\, x\, y$ implies the provability of

$$\forall z. P\, z\, z.$$

That is, there is also a proof where the eigenvariables c and d are identified.

Thus, eigenvariables are unlikely to capture the proper logic behind things like nonces, references, names, etc.

A new quantifier ∇

The problem illustrated on the previous slide is that the eigenvariables c and d should be *object-level* eigenvariables and not *meta-level* eigenvariables.

To fix this problem of scope, we introduce a new meta-level quantifier, $\nabla x.B\,x$, and a new context to sequents. Sequents will have one *global* signature (the familiar Σ) and several *local* signatures, used to scope object-level eigenvariables.

$$\Sigma : B_1, \ldots, B_n \longrightarrow B_0$$

$$\Downarrow$$

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

Σ is a set of eigenvariables, scoped over the sequent and σ_i is a list of variables, locally scoped over the formula B_i.

The expression $\sigma_i \triangleright B_i$ is called a *generic judgment*. Equality between judgments is defined up to renaming of local variables.

See: Miller and Alwen Tiu, *Encoding generic judgments* in LICS03.

The ∇ and $\forall$-quantifier

The ∇-introduction rules modify the local contexts.

$$\frac{\Sigma : (\sigma, y_\gamma) \triangleright B[y/x], \Gamma \longrightarrow \mathcal{C}}{\Sigma : \sigma \triangleright \nabla x_\gamma.B, \Gamma \longrightarrow \mathcal{C}}\ \nabla\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow (\sigma, y_\gamma) \triangleright B[y/x]}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \nabla x_\gamma.B}\ \nabla\mathcal{R}$$

Since these rules are the same on the left and the right, this quantifier is *self-dual*.

Both the global and local signatures are abstractions over their respective scopes.

The universal quantifier rules are changed to account for the local context.
(Rules for $\exists$ are simple duals of these.)

$$\frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma : \sigma \triangleright B[t/x], \Gamma \longrightarrow \mathcal{C}}{\Sigma : \sigma \triangleright \forall_\gamma x.B, \Gamma \longrightarrow \mathcal{C}}\ \forall\mathcal{L} \qquad \frac{\Sigma, h : \Gamma \longrightarrow \sigma \triangleright B[(h\ \sigma)/x]}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \forall x.B}\ \forall\mathcal{R}$$

The familiar *raising* technique from higher-order unification is used to manage scoping of variables: if σ is $x_1, \ldots, x_n$ then $(h\ \sigma)$ is $(h\ x_1 \cdots\ x_n)$, where h is a higher-order variable of the proper type.

Unification and matching in definitions is extended to these context by identifying local signature with λ-binders.

Some results involving ∇

$$\nabla x \neg Bx \equiv \neg \nabla x Bx \qquad \nabla x(Bx \wedge Cx) \equiv \nabla x Bx \wedge \nabla x Cx$$

$$\nabla x(Bx \vee Cx) \equiv \nabla x Bx \vee \nabla x Cx \qquad \nabla x(Bx \Rightarrow Cx) \equiv \nabla x Bx \Rightarrow \nabla x Cx$$

$$\nabla x \forall y Bxy \equiv \forall h \nabla x Bx(hx) \qquad \nabla x \exists y Bxy \equiv \exists h \nabla x Bx(hx)$$

$$\nabla x \forall y Bxy \Rightarrow \forall y \nabla x Bxy \qquad \nabla x.\top \equiv \top, \quad \nabla x.\bot \equiv \bot$$

Theorem. Given a fixed stratified definition, a sequent has a proof if and only if it has a cut-free proof.

Theorem. Given a *noetherian* definition, the following formula is provable.

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

Theorem. If we restrict to Horn definitions (no implication and negation in the body of the definitions) then

1. $\forall$ and ∇ are interchangeable in definitions,
2. $\vdash \nabla x.B\,x \supset \forall x.B\,x$ for noetherian definitions.

Example: reasoning with an object-logic

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

$$\forall x \forall y[q\ x\ x\ y] \qquad \forall x \forall y[q\ x\ y\ x] \qquad \forall x \forall y[q\ y\ x\ x]$$

only if terms t_2 and t_3 are equal.

We would like to prove a meta-level formula like

$$\forall x, y, z[pv\ (\hat{\forall}\, u\, \hat{\forall}\, v[q\ \langle u, x\rangle\ \langle v, y\rangle\ \langle v, z\rangle]) \supset y = z]$$

Example: reasoning with an encoded object-logic (cont)

We can encode provability of a first-order logic using the following definitions.

$$pv\ (\hat{\forall}\, G) \triangleq \nabla x.pv\ (Gx) \qquad pv\ A \triangleq \exists D.prog\ D \wedge inst\ D\ A$$
$$pv\ (G\ \&\ G') \triangleq pv\ G \wedge pv\ G'$$

$$inst\ (q\ X\ Y\ Z)\ (q\ X\ Y\ Z) \triangleq \top \qquad prog\ (\hat{\forall}\, x\, \hat{\forall}\, y\ q\ x\ x\ y) \triangleq \top$$
$$inst\ (\hat{\forall}\, D)\ A \triangleq \exists t.\ inst\ (D\ t)\ A \qquad prog\ (\hat{\forall}\, x\, \hat{\forall}\, y\ q\ x\ y\ x) \triangleq \top$$
$$X = X \triangleq \top \qquad prog\ (\hat{\forall}\, x\, \hat{\forall}\, y\ q\ y\ x\ x) \triangleq \top$$

$$\frac{\Xi_1 \quad \Xi_2 \quad \Xi_3}{\dfrac{x, y, z : u, v \triangleright pv\ (q\ \langle u, x\rangle\ \langle v, y\rangle\ \langle v, z\rangle) \longrightarrow y = z}{x, y, z : pv\ (\hat{\forall}\, u\, \hat{\forall}\, v[q\ \langle u, x\rangle\ \langle v, y\rangle\ \langle v, z\rangle]) \longrightarrow y = z}}$$

Ξ_1 : $\lambda u \lambda v \langle u, x\rangle = \lambda u \lambda v \langle v, y\rangle$. Unification failure, so sequent is proved.
Ξ_2 : $\lambda u \lambda v \langle u, x\rangle = \lambda u \lambda v \langle v, z\rangle$. Unification failure, so sequent is proved.
Ξ_3 : $\lambda u \lambda v \langle v, y\rangle = \lambda u \lambda v \langle v, z\rangle$. Unifier $[y \mapsto z]$ yields new trivial sequent

$$x, z :\longrightarrow z = z.$$

Encoding π-calculus transitions

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

Processes can make transitions via various *actions*. There are three constructors for actions: $\tau : a$ for *silent* actions, $\downarrow : n \to n \to a$ for *input* actions, and $\uparrow : n \to n \to a$ for *output* actions.

Following usual conventions: $\downarrow xy$ represents the action of inputting name y on channel x, and $\uparrow xy$ represents the action of outputting name y on channel x.

The abstraction $\uparrow x : n \to a$ denotes outputting of an abstracted variable, and $\downarrow x : n \to a$ denotes inputing of an abstracted variable.

The one-step transition relation is encoded as two different predicates:

$P \xrightarrow{A} Q$	$A : a$
$P \xrightarrow{\downarrow x} M$	bound input action, $\downarrow x : n \to a$, $M : n \to proc$
$P \xrightarrow{\uparrow x} M$	bound output action, $\uparrow x : n \to a$, $M : n \to proc$

π-calculus: one step transitions

- Operational semantics:

$$\frac{}{\bar{x}y.\mathbf{P} \xrightarrow{\bar{x}y} \mathbf{P}}\ \text{OUTPUT}-\text{ACT} \qquad \frac{\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}{[x=x]\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}\ \text{MATCH} \qquad \frac{\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}{(y)\mathbf{P} \xrightarrow{\alpha} (y)\mathbf{P}'}\ \text{RES}, y \notin \mathrm{n}(\alpha)$$

- Encoding restriction using $\forall$ is problematic.

$$\begin{array}{lrcl} \text{RES}: & (x)Px \xrightarrow{\alpha} (x)P'x & \triangleq & \forall x.(Px \xrightarrow{\alpha} P'x) \\ \text{OUTPUT}-\text{ACT}: & \bar{x}y.P \xrightarrow{\bar{x}y} P & \triangleq & \top \\ \text{MATCH}: & [x=x]P \xrightarrow{\alpha} P' & \triangleq & P \xrightarrow{\alpha} P' \end{array}$$

- Consider the process $(y)[x=y]\bar{x}z.0$. It cannot make any transition, since y has to be "new"; that is, it cannot be x.

- The following statement should be provable

$$\forall x \forall Q \forall \alpha.[((y)[x=y](\bar{x}z.0) \xrightarrow{\alpha} Q) \supset \bot]$$

If restriction is translated to the meta-level $\forall$, then we need to prove

$$\{x, z, Q, \alpha\} : \forall y.([x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \longrightarrow \bot$$

Proceeding now yields a sequent of the form

$$\{z\} : ([w = w](\bar{w}z.0) \xrightarrow{\bar{w}z} 0) \longrightarrow \bot$$

which is not provable.

Scoping and newness are captured precisely by ∇:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x, z, Q, \alpha\} : w \triangleright ([x = w](\bar{x}z.0) \xrightarrow{\alpha} Q) \longrightarrow \bot}\, def\mathcal{L}}{\{x, z, Q, \alpha\} : \cdot \triangleright \nabla y.([x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \longrightarrow \bot}\, \nabla\mathcal{L}}{\{x, z, Q, \alpha\} : \cdot \triangleright ((y)[x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \longrightarrow \bot}\, def\mathcal{L}}{\{x, z, Q, \alpha\} :\longrightarrow \cdot \triangleright ((y)[x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \supset \bot}\, \supset\mathcal{R}$$

The success of $def\mathcal{L}$ follows the failure of unification problem $\lambda w.x = \lambda w.w$.

Encoding simulation in the (finite) π-calculus

If the premises for the one step transition systems use ∇ instead of $\forall$, then simulation for the (finite) π-calculus is simply the following:

$$\begin{aligned} sim\ P\ Q \stackrel{\triangle}{=}\ & \forall A \forall P'\ [(P \xrightarrow{A} P') \Rightarrow \exists Q'.(Q \xrightarrow{A} Q') \\ & \qquad \wedge\ sim\ P'\ Q'] \wedge \\ & \forall X \forall P'\ [(P \xrightarrow{\downarrow X} P') \Rightarrow \exists Q'.(Q \xrightarrow{\downarrow X} Q') \\ & \qquad \wedge \nabla w.sim\ (P'w)\ (Q'w)] \wedge \\ & \forall X \forall P'\ [(P \xrightarrow{\uparrow X} P') \Rightarrow \exists Q'.(Q \xrightarrow{\uparrow X} Q') \\ & \qquad \wedge \nabla w.sim\ (P'w)\ (Q'w)] \end{aligned}$$

Deduction with this formula will compute simulation. This is a direct translation of the "official definition" but where names are handled entirely using scoping mechanisms of the meta-logic.

There is a "cheap" λProlog program that will emulate this deduction and do "symbolic simulation".

Conclusion

- When computation is described via provability in the proof search paradigm, HOAS and linear logic can be used for expressive advantage.
- A few pieces of a meta-logic that should allow us to reason directly on provability of specifications was illustrated: in particular, definitions and ∇.
- When reasoning about HOAS specifications, something like ∇ seems required. We have no examples of ∇ that do not involve HOAS.
- The main area of application of these ideas seem to be in the operational semantic specifications of rich, symbolic systems (programming languages, specification languages, security protocols, type systems, etc).