Reasoning about proof search 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.

Roles of Logic in the Specification of Computation

In the specification of computational systems, logics are generally used in one of two approaches.

Computation-as-model: Computations are mathematical structures representing computations via nodes, transitions, and states (for example, Turing machines, etc). Logic is used in an external sense to make statements *about* those structures. E.g. Hoare triples, modal logics.

Computation-as-deduction: Pieces of logic are used to model elements of computation directly.

Functional programming. Programs are proofs and computation is proof normalization (λ-conversion, cut-elimination).

Logic programming. Programs are theories and computation is the search for (cut-free) sequent proofs. The dynamics of computation are encoded in the changes to sequents that occur during the search for a proof. A successful domain of logic programming is *operational semantics*.

Operational semantics of computation systems

OS is probably the dominate form of programming language semantic specification: used for both *dynamic* and *static* semantics.

Operational semantics is generally given using inference rules. Below is an example rule specifying call-by-value evaluation:

$$\frac{M \Downarrow (abs\ R) \qquad N \Downarrow U \qquad (R\ U) \Downarrow V}{(app\ M\ N) \Downarrow V}$$

Such specifications have a natural and immediate connection to proof search (logic programming).

For example, it is trivial to translate the above inference rule as a logic program clause (using λProlog syntax):

```
eval (app M N) V :- eval M (abs R), eval N U, eval (R U) V.
```

This connection means that the recent advances in proof search can often be translated into advances in the specification of operational semantics.

Advances in the proof search paradigm

Logic programming has been extended in recent years to encompass 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.
- *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.

Forum is a presentation of all of higher-order linear logic as a logic programming language. It modularly extends the logical foundations of Prolog, λProlog, and Lolli.

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. These will not be discussed in this talk.
- 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.

The sequent calculus in brief

A single-conclusion *sequent* is a triple written as $\Sigma\colon \Delta \longrightarrow G$ where

Σ: The *signature* of this sequent: a set of eigenvariables used in the formulas G and Δ.

Δ: The *antecedent* or *left-hand side* or *program*.

G: The *succedent* or *right-hand side* or *goal*.

One way to understand the context Σ in the sequent

$$\Sigma\colon \Delta \longrightarrow G$$

is that for every substitution for the variables in Σ, the sequent

$$\Delta\theta \longrightarrow G\theta$$

is provable.

A proof theoretic notion of definition

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

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

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 $\triangleq$ is not a logical connective: it is used just to denote definitional clauses.

Using $\triangleq$ 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.

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

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

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

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.

Examples: $1+2=3$ and $1+2\neq 1$

Define addition on a simple encoding of natural numbers.

$$\begin{array}{rcl} sum\ z\ N\ N & \triangleq & \top. \\ sum\ (s\ N)\ M\ (s\ P) & \triangleq & sum\ N\ M\ P. \end{array}$$

We can prove that $1+2=3$

$$\dfrac{\dfrac{\longrightarrow \top}{\longrightarrow sum\ z\ (s\ (s\ z))\ (s\ (s\ z))}\ def\mathcal{R}}{\longrightarrow sum\ (s\ z)\ (s\ (s\ z))\ (s\ (s\ (s\ z)))}\ def\mathcal{R}$$

and that $1+2\neq 1$.

$$\dfrac{\dfrac{}{sum\ z\ (s\ (s\ z))\ z \longrightarrow}\ def\mathcal{L}}{sum\ (s\ z)\ (s\ (s\ z))\ (s\ z) \longrightarrow}\ def\mathcal{L}$$

More generally, eigenvariables can be instantiated:

$$\dfrac{\Sigma, n : \Gamma[n/m], \top \longrightarrow G[n/m]}{\Sigma, n, m : \Gamma, sum\ z\ n\ m \longrightarrow G}\ def\mathcal{L}$$

Example: evaluation of a conditional

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

$$\vdots$$

$$\begin{array}{lcl} (if\ B\ M\ N) \Downarrow V & \triangleq & B \Downarrow true\ \wedge M \Downarrow V. \\ (if\ B\ M\ N) \Downarrow V & \triangleq & B \Downarrow false \wedge N \Downarrow V. \end{array}$$

$$\vdots$$

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 $\mathit{def}\mathcal{R}$ **and** $\mathit{def}\mathcal{L}$

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

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

$$\frac{\vdots}{A \longrightarrow B}\ \mathit{def}\mathcal{L} + \mathit{def}\mathcal{R}$$ 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:

$$p \triangleq p \supset \bot \ .$$

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 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.

Intuitionistic logic with ∇

$$\frac{}{\Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{B}}\ \mathit{init} \qquad \frac{\Sigma : \Delta \longrightarrow \mathcal{B} \qquad \Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{C}}{\Sigma : \Delta, \Gamma \longrightarrow \mathcal{C}}\ \mathit{cut}$$

$$\frac{\Sigma : \mathcal{B}, \mathcal{B}, \Gamma \longrightarrow \mathcal{C}}{\Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{C}}\ c\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow \mathcal{C}}{\Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{C}}\ w\mathcal{L}$$

$$\frac{}{\Sigma : \sigma \triangleright \bot, \Gamma \longrightarrow \mathcal{B}}\ \bot\mathcal{L} \qquad \frac{}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \top}\ \top\mathcal{R}$$

$$\frac{\Sigma : \sigma \triangleright B_i, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B_1 \wedge B_2, \Gamma \longrightarrow \mathcal{D}}\ \wedge\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright B_1 \qquad \Sigma : \Gamma \longrightarrow \sigma \triangleright B_2}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B_1 \wedge B_2}\ \wedge\mathcal{R}$$

$$\frac{\Sigma : \sigma \triangleright B_1, \Gamma \longrightarrow \mathcal{D} \qquad \Sigma : \sigma \triangleright B_2, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B_1 \vee B_2, \Gamma \longrightarrow \mathcal{D}}\ \vee\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright B_i}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B_1 \vee B_2}\ \vee\mathcal{R}$$

$$\frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \qquad \Sigma : \sigma \triangleright C, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B \supset C, \Gamma \longrightarrow \mathcal{D}}\ \supset\mathcal{L} \qquad \frac{\Sigma : \sigma \triangleright B, \Gamma \longrightarrow \sigma \triangleright C}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \supset C}\ \supset\mathcal{R}$$

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.

Clear? Or were you thinking about the domain of interpretation of u and v?

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.

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

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

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

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

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

Example: encoding π calculus

π-calculus is a formal model for concurrency. The main entities are *processes* and *names*. The syntax is the following:

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

We pick the π-calculus because it is an interesting case where the conventional approach to encoding require complicated uses of side conditions involving names.

Encoding the transition system for the π-calculus into HOAS has been know for a number of years and is pretty straightforward. For example:

restriction $(x)P$ is encoded using a constant of type $(\mathrm{nm} \rightarrow \mathit{proc}) \rightarrow \mathit{proc}$.

input $x(y).P$ is encoded using a constant of type $\mathrm{nm} \rightarrow (\mathrm{nm} \rightarrow \mathit{proc}) \rightarrow \mathit{proc}$.

Encoding π-calculus transitions

Processes can make transitions via various *actions*. There are three constructors for actions: $\tau : \mathrm{act}$ for *silent* actions, $\downarrow : \mathrm{nm} \rightarrow \mathrm{nm} \rightarrow \mathrm{act}$ for *input* actions, and $\uparrow : \mathrm{nm} \rightarrow \mathrm{nm} \rightarrow \mathrm{act}$ 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 $\uparrow x : \mathrm{nm} \rightarrow \mathrm{act}$ denotes outputting of an abstracted variable, and $\downarrow x : \mathrm{nm} \rightarrow \mathrm{act}$ denotes inputing of an abstracted variable.

Bound output is responsible for sending a locally bound variable outside its scope to other processes: *scope extrusion*.

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

$$
\begin{array}{ll}
P \xrightarrow{A} Q & A : act \\
P \xrightharpoonup{\downarrow x} M & \text{bound input action, } \downarrow x : nm \rightarrow act,\ M : nm \rightarrow proc \\
P \xrightharpoonup{\uparrow x} M & \text{bound output action, } \uparrow x : nm \rightarrow act,\ M : nm \rightarrow proc
\end{array}
$$

π-calculus: one step transitions

- Operational semantics:

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

- 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

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

There are at least two instantiations of variables that identify x and y:

1. $y \mapsto w$, $x \mapsto w$, $\alpha \mapsto \bar{w}z$, $Q \mapsto 0$: (wrong *scoping*)

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

2. $y \mapsto x$, $\alpha \mapsto \bar{x}z$, $Q \mapsto 0$: (*newness* assumption on y is violated)

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

Scoping and newness are captured precisely by ∇:

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

The success of $\mathit{def}\mathcal{L}$ 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:

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

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".

Future and related work

- Induction and co-induction fit naturally with definitions: simply specify which definitions should be considered least fixed points and which greatest fixed points. Doing this in the sequent calculus takes some care. See Momigliano and Tiu in TYPES 2003.
- Pitts and Gabbay have a "new name quantifier". It is similar in that it is also self-dual and they are targeting similar applications as HOAS. It differs in most other respects.
- Alwen Tiu is constructing a prototype theorem prover for doing experiments, based on earlier work in Jérémie Wajs's 2002 PhD.
- Carsten Schürmann is developing a meta-logical system for reasoning about LF specifications.
- Simon Ambler, Roy Crole, and Alberto Momigliano have used their Hybrid package in Isabelle/HOL to reason using definitions and HOAS.
- Do existing theorem provers allow for this style reasoning? Should a full scale implementation be undertaken?

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).

Names

Reprose from REasoning about PROof SEarch.

Prosea from PROof SEArch, but this is taken by a maritime lobbying group.

Sarah since Paulson named his prover after Huet's daughter.

LINC for lambda, induction, nabla, and co-induction (Tiu). Also stands for "LINC is not Coq".