Representing and reasoning with operational semantics

Dale Miller
INRIA-Futurs and LIX/École polytechnique
Parsifal Project

Setting the stage: mathematics vs computing

On Thursday, B. Buchberger said:

For automated reasoning, mathematics is the main goal.

This talk will miss this main goal for a much more modest and seemingly unrelated goal, namely:

To use automated reasoning to prove theorems about computational systems.

Typical theorems in the target of this goal:

1. processes P and Q are bisimilar;
2. the applet J does not access memory locations M;
3. miniML has determinate evaluation and the type preservation property; and
4. bisimulation for a process calculus is a congruence.

Setting the stage: theory vs tools

There exists a number of mature theorem proving systems. There is a great deal of interest in using and improving these tools: better user interaction, more libraries, more decision procedures, improved interoperability, etc.

In the area of operational semantics, there is a similar focus on tools. For example, the PoplMark Challenge focuses energies on solving various challenge problems with existing systems.

Our focus here is

logical foundations and theory,
aimed at providing a new architecture for tools
to reason about the meaning of computational systems.

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?

Various "intensional aspects" of computational specifications — bindings, names, resource accounting, etc — are challenges to this approach to reasoning about computation.

Challenge area 1: Higher-order abstract syntax

"Use the meta-level binder to encode object-level binder."

A natural counterpart to the usual practice of using meta-level application to encode object-level application.

- Huet & Lang, 1978: the Mentor system, second-order matching
- Miller & Nadathur, 1986: λProlog, full higher-order unification
- Pfenning & Elliot, 1988: coined the term HOAS

What flavor is your meta-logic? Is the following a theorem?

$$\forall w_i.\ \lambda x.x \neq \lambda x.w$$

In logic programming/proof search: Yes, since variable capture is not possible. This is a question about syntax.

In functional programming: Depends on whether or not the domain is a singleton or not. This seems to be more than about syntax.

Challenge area 2: Linear logic

Linear logic has increased greatly the expressive strengths of proof search from its classical (Horn clauses) and intuitionistic (hereditary Harrop formulas) setting.

All of linear logic can be seen as logic programming and focused proofs provide the operational semantics.

Linear logic has been used to encode concurrency, functional programming with side-effects, security protocols, Petri nets, etc.

Used as a logical framework for specifying sequent calculus (instead of natural deduction).

Linear logic can, of course, be encoded in theorem proving. But since "linear logic is the logic behind (computational) logic", there seems to be better avenues one should be taking.

I will mostly avoid speaking about linear logic in the rest of this talk.

Static and Dynamic Semantics

We shall focus on reasoning about the semantics of specification and programming languages.

Programming language semantics is often divided into two parts:

- **Dynamic semantics:** evaluation, communications, side-effects, etc.
  - **Small-step semantics**: e.g., labeled transition, SOS,
  - **Big-step semantics**: e.g., natural semantics
- **Static semantic:** Typing

Semantics as inference rules

CCS and π-calculus transition system:

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin fn((y)P') \end{array}$$

Functional programming evaluation:

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

Simple typing of terms: used in functional (SML) and logic (λProlog) programming.

$$\frac{\Gamma, x{:}\tau \vdash t{:}\sigma}{\Gamma \vdash \lambda x.t{:}\tau \to \sigma} \quad x \notin fn(\Gamma)$$

Operational semantics of computation systems

Can these be seen as expressions in logic? Does proof theory, an approach to inference, have a role to play here?

Can $\dfrac{A_1 \quad \cdots \quad A_n}{A_0}$ be encoded as $\forall \bar{x}[(A_1 \wedge \ldots \wedge A_n) \supset A_0]$

$A_0 \texttt{:-}\ A_1, \ldots, A_n.$

Particular problems:

- Ordered premises: particularly in functional programming with side-effects. But $\wedge$ is commutative.
- The status of bindings substitutions in terms must be explained.
- Side-conditions: many deal with occurrences of names and variables.

HOAS approach illustrated

Instead of the rule

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

where various things are still to be explained, consider, instead,

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

Here, R is a higher-order variable and the logic presumably has built-in $\alpha\beta$-conversion.

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.
```

Two slogans about bindings

(I) From Alan Perlis's *Epigrams on Programming*: As Will Rogers would have said, "There is no such thing as a free variable."

(II) The *names* of binders are the same kind of fiction as *white space*: they are artifacts of how we write expressions and have *zero semantic content*.

To specify or implement a logic for dealing with bindings, one must, of course, deal with the complexity of names.

Church provided a specification of such a logic in 1940 with his paper on "A Formulation of the Simple Theory of Types." We shall work in this *Paradise of (the) Church*.

Example: Binding a variable in a proof

When proving a universal quantifier, one uses a "new" or "fresh" variable.

$$\frac{B_1, \ldots, B_n \longrightarrow Bv}{B_1, \ldots, B_n \longrightarrow \forall x_\tau . Bx}\ \forall\mathcal{R},$$

provided that v is a "new" variable (not free in the lower sequent). Such new variables are called *eigenvariables*.

But this violates the "Perlis principle." Instead, we write

$$\frac{\Sigma, v{:}\tau : B_1, \ldots, B_n \longrightarrow Bv}{\Sigma : B_1, \ldots, B_n \longrightarrow \forall x_\tau . Bx}\ \forall\mathcal{R},$$

Here, we assume that the variables in the new context (signature) are bindings over the sequent.

Eigenvariables are bound variables within a proof.

Dynamics of binders during proof search

During computation, binders can be *instantiated:*

$$\frac{\Sigma : \Delta, \mathit{typeof}\ c\ (\mathit{int} \to \mathit{int}) \longrightarrow C}{\Sigma : \Delta, \forall\alpha(\mathit{typeof}\ c\ (\alpha \to \alpha)) \longrightarrow C}\ \forall\mathcal{L}$$

They also have *mobility* (they can move):

$$\frac{\Sigma, x : \Delta, \mathit{typeof}\ x\ \alpha \longrightarrow \mathit{typeof}\ \lceil B \rceil\ \beta}{\dfrac{\Sigma : \Delta \longrightarrow \forall x(\mathit{typeof}\ x\ \alpha \supset \mathit{typeof}\ \lceil B \rceil\ \beta)}{\Sigma : \Delta \longrightarrow \mathit{typeof}\ \lceil \lambda x.B \rceil\ (\alpha \to \beta)}}\ \forall\mathcal{R}$$

In this case, the binder named x moves from *term-level* (λx) to *formula-level* ($\forall x$) to *proof-level* (as an eigenvariable in Σ, x).

Note: To account for the mobility of binders, only a weak form of β-conversion is needed (together with α-conversion):

$$(\beta_0) \qquad (\lambda x.B)x = B$$

Higher-order pattern unification uses only this form of β-conversion.

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

We want to do more than "animate" or "execute" a specification. We want to prove properties about the specifications. We illustrate with a proof of type preservation (subject-reduction).

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

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

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

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

The first three clauses are Horn clauses; the fourth is not. Here, *app* is a constant of type $tm \to (tm \to tm)$ and *abs* is a constant of type $(tm \to tm) \to tm$).

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 $eval\ \ P\ V$: for all T, if $\vdash typeof\ P\ T$ then $\vdash typeof\ V\ T$.

The proof of $eval\ \ P\ V$ 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 $(abs\ \ R)$, 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 $\vdash typeof\,(R\,N)\ T$. (Substitution lemma for free!)

Using the inductive hypothesis a second time yields $\vdash typeof\ V\ T$. QED

Analyzing this informal proof

We wish to have a formal setting where this proof can be performed. This suggests that the following features would be valuable in the meta-logic.

1. *Two distinct logics.* The *object logic* captures judgments, e.g. about typability and evaluation. The *meta logic*, written here in English, has atomic formulas that are judgments about the object-logic.
2. *Induction*. Co-induction is needed for other theorems.
3. *Instantiation of meta-level eigenvariables.* The meta-level eigenvariable P was instantiated to $(abs\ R)$ and to $(app\ M\ N)$.
4. *Case analysis of the proof of an assumed judgment.* We needed to invert inference rules.

A proof theoretic approach to "definitions", "fixed points", and the closed-world-assumption can be used to address the last three points above. See work by Schroeder-Heister, Girard, and McDowell/Miller/Tiu.

One additional meta-level feature is still to be motivated.

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.

Bindings now have more places to move.

See papers by Miller and Alwen Tiu in LICS03 and ToCL 2005.

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)

The following definition encodes a part of object-level provability.

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

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

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

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 $(n \to p) \to p$.

input $x(y).P$ is encoded using a constant of type $n \to (n \to p) \to p$.

Encoding π-calculus transitions

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 inputting 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:

$$P \xrightarrow{A} Q \qquad A : a$$

$$P \xrightarrow{\downarrow x} M \qquad \text{bound input action, } \downarrow x : n \to a,\ M : n \to p$$

$$P \xrightarrow{\uparrow x} M \qquad \text{bound output action, } \uparrow x : n \to a,\ M : n \to p$$

π-calculus: one-step transitions

- Operational semantics: Rules for OUTPUT-ACT, MATCH, and RES.

$$\frac{}{\bar{x}y.\mathbf{P} \xrightarrow{\bar{x}y} \mathbf{P}} \qquad \frac{\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}{[x=x]\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'} \qquad \frac{\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}{(y)\mathbf{P} \xrightarrow{\alpha} (y)\mathbf{P}'}\ y \notin \mathrm{n}(\alpha)$$

- Encoding restriction using $\forall$ is problematic.

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

- 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. It is bisimilar to 0.

- The following statement should be provable

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

Given the encoding of restriction using $\forall$, this reduces to proving the sequent

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

No matter what is used to instantiate the $\forall y$, the eigenvariable x can instantiated to the same thing (say, w), and this case leads to the non-provable sequent

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

The universal quantifier was not the correct choice. 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:

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

Deduction with this formula will compute simulation. Bisimulation is easy to encode (just add additional cases above).

Bisimulation corresponds to *open* bisimulation. If the meta-logic is enhanced to be classical, then *late* bisimulation is captured. The difference can be reduced to the excluded middle $\forall x \forall y.\ x = y \vee x \neq y$.

LINC: the meta-logic

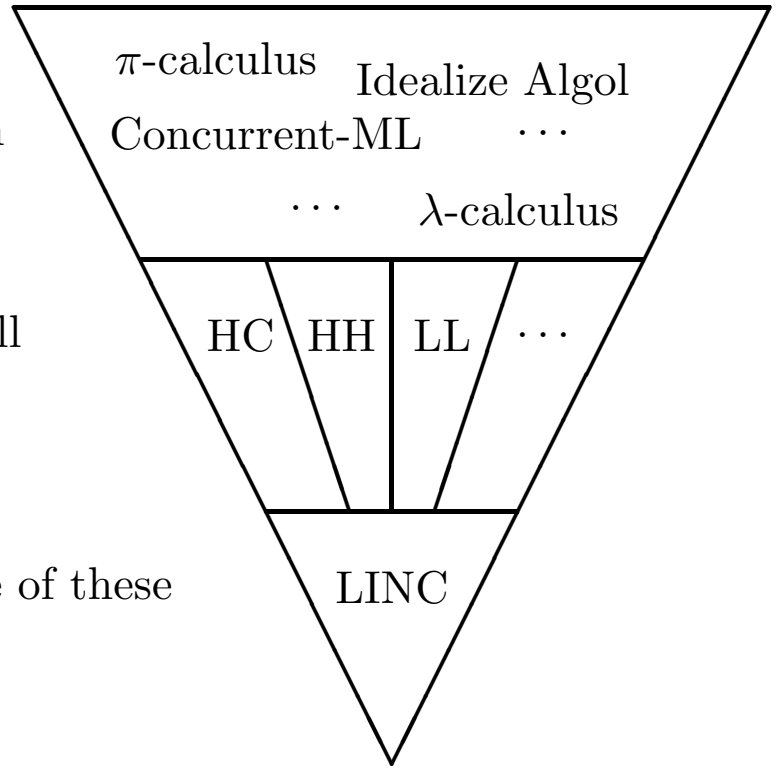
- LINC stands for a logic with Lambdas, Induction, Nabla, and Co-induction. (Also: LINC Is Not Coq.) Details in Tiu's PhD thesis (2004).
- Extends $FO\lambda^{\Delta\mathbb{N}}$ of McDowell/Miller (ToCL 2002).
- Intuitionistic logic (no linear logic at the "mathematics level"). Classical logic is of interest as well.
- It is a big logic, providing a framework for proving properties about logic specifications (current target: operational semantics).
- Allows induction and co-induction on the proof search approach to HOAS.

A possible architecture

I. Applications: focus on operational semantics

II. Object-logics: A small number of these

III. Meta-logic: Just one of these



Bedwyr: a proof search implementation of part of LINC

Bedwyr is an OCaml implementation of small part of LINC.

It is a generalization of logic programming in that it uses both finite success and finite failure for the search for proofs. Also includes tabling.

Bedwyr's application areas look similar to those for model checking, game playing, and bisimulation checking.

For example, it completely implements open bisimulation for finite π-calculus. The implementation is the natural specification.

Open source: http://slimmer.gforge.inria.fr/bedwyr/

Current implementation and design team:
INRIA & LIX: D. Baelde (PhD student) & D. Miller
University of Minnesota: A. Gacek (PhD student) & G. Nadathur
Australian National University: A. Tiu

Future and related work

- A interactive theorem prover for LINC is planned.
- Semantics for the ∇-quantifier should be developed. A candidate is Ulrich Schoepp's LFMTP 2006 paper.
- There are similarities and differences between ∇ and the Pitts-Gabbay "new name quantifier". Their connection needs to be be understood. See Tiu's LFMTP 2006 paper.
- Format rules for SOS: generalize tfyt/txyt format rules for process calculi with name-passing so that open bisimulation is a congruence (Ziegler/Miller/Palamidessi, SOS 2005).
- Format rule for specifying inference rules in object-level sequent calculus that guarantee elimination of initial and cut rules (Pimentel/Miller, LPAR 2005).
- Lots of other related work at cited in the paper in the proceedings.

Conclusion

- When dynamic and static semantic specifications of computation are presented as inference rules, these can often be converted to logical theories or definitions (fixed points).
- Proof theory and proof search ideas can provide useful tools for encoding and reasoning with such specifications.
- In particular, object-level bound variable always remain bound. This frees one from the many non-semantically interesting aspects of variable names.
- When reasoning about HOAS specifications, something like ∇ seems required (at least when negation is involved). 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).