An overview of a proof theoretical approach to reasoning about computation

Dale Miller
INRIA-Saclay & LIX/École Polytechnique
Parsifal Project

LFMTP'08, Pittsburgh, 23 June 2008

References can be found in the short article in the conference proceedings.

This talk provides an overview to some of the work by David Baelde, Andrew Gacek, Ray McDowell, Gopalan Nadathur, Alwen Tiu, and myself. Miller was funded by the IST-2005-015905 MOBIUS and the INRIA "Equipes Associées" Slimmer.

Outline

- Background and motivation
- Computing and reasoning with bindings
- Definitions and ∇-quantification
- A couple examples
- Designing a theorem prover

Proof theory as an alternative to model theory

Gödel's Completeness Theorem [1929]

A formula is a theorem if and only if it is valid.

The first deep connection between syntactically presented proofs (witness for theoremhood) and model theoretic validity.

In many circles, it is largely unquestioned that the proper way to attribute meaning to logic specifications is via model theory or categorical theory.

- In the past 80 years, enough *semantic muscle* has been developed to justify almost anything. If seems that both syntax *and* semantic can be *ad hoc*.
- Model theory is based on the "infinite" while proof theory is based on the "finite". Eg: is a binding encoded as a function or an expression?

Proof theory can be regarded as organizing the principles behind syntax.

Of course, the important things (like the set of theorems) always have more than one description.

How abstract is your syntax?

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

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

Parse trees: Removing white space, parenthesis, infix/prefix operators yields first-order term structures. The recursive structure of syntax is better represented.

Bindings are treated too concretely. Of the numerous approaches to making bindings more abstract, I'll focus on probably the oldest approach.

λ-trees: Syntax is forced to be treated via α-conversion and weak forms of β-reduction (eg, typed β-conversion or β_0). Unification is used to decompose syntax. Support from the meta-level for changing contexts.

Proof-search versus proof-normalization

We use proof-search specifications (logic programs) instead of proof-normalization specifications (functional programs).

- SOS rules, typing rules, etc, are relational specifications that are immediately and naturally realized as logic programming clauses.
- The proof search embraces λ-tree syntax directly.

	Functional Prog	Logic Prog
f.o. variables	datatypes	datatypes
abstraction	higher-order prog	Over relations: higher-order prog
	$(\alpha \to \beta) \to \alpha^* \to \beta^*$	$(\alpha \to \beta \to o) \to \alpha^* \to \beta^* \to o$
	—	Over datatypes: λ-tree syntax
		$(term \to term) \to term$

Higher-order abstract syntax (hoas) uses "meta-level binders" for "object-level binder." Since this term is ambiguous, we use *λ-tree syntax* for the proof search approach to hoas.

The traditional approach to reasoning about computation

Step 1: **Implement mathematics**

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

Step 2: **Reduce computation to mathematics**

- via denotational semantics and/or
- via inductive definitions for datatypes and inference rules.

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

Various "intensional aspects" of computational specifications — bindings, names, resource accounting, etc, seem poorly supported in mathematical systems.

Eg, contrast modeling memory via linear logic or via encodings of sets of pairs or as arrays.

Drop mathematics as the intermediate

Church [1940] provided a framework for mixing logic and λ-abstraction.

- Using only axioms 1-6 yields a logic too weak for mathematics but strong enough to capture λ-tree. Eg, λProlog is based on an intuitionistic subset of these axioms.
- Subsequent axioms provided for extensionality, infinity, and choice functions. Such an extension provides a framework for mathematics.

Example: Is the following a theorem?

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

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

In proof normalization: No, since the domain might a singleton. This seems to be about more than syntax.

Here, types denote syntactic types (categories such as π-calculus expressions, names, actions, etc) and not the more general notion of "semantic" types.

Three slogans about bindings

Two from Alan Perlis's *Epigrams on Programming* and one from me.

(I) As Will Rogers would have said, "There is no such thing as a free variable."

A variable is always declared somewhere and it's usually a good idea to be explicit about locating where that binding is declared.

(II) One man's constant is another man's variable.

It's a matter of scope. When executing a functional program, the `fact` program is a constant. When compiling and linking such a program, it is a variable: it can be different low-level code and occupy different memory locations.

(III) 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*.

Bindings have semantic importance but not their names. It is the implementations of logic and type systems that must deal with specific devices such as names or de Bruijn indices.

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 signature context are bindings over the sequent.

Eigenvariables are bound variables within a proof.

Two key ingredients for computing with bindings

- Unification modulo α, η, and enough β conversion.
- Mobility of binders. For this, (β_0) $(\lambda x.B)x = B$ is sufficient.

  In the expression $(\lambda x.B)x$, occurrences in B of x refer to a *local* binding, while in the second occurrence of B, the x refers to some more *remote* binding.

Unification modulo $\alpha\beta_0\eta$ is decidable given that β is so strongly restricted (β_0-reduction yields smaller terms).

Pattern unification is a restricted subset of $\alpha\beta_0\eta$-unification that is also unary.

Treatment of binders in sequent calculus

During computation, binders can be *instantiated:*

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

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

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

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).

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

Both of the theorem provers **Abella** (Gacek 2008) and **Taci** (Baelde, Snow, Viel 2008) can interactively build this proof.

The collapse of eigenvariables

Eigenvariables are not sufficient alone as a proof-level binding.

This is despite their various uses as abstractions in computational specifications: eigenvariables have been used to encode *names* in π-calculus [Miller 1993], *reference locations* in imperative programming [Chirimar 1995], *nonces* in security protocols [Cervesato, et. al. 1999], etc.

A cut-free proof of

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

uses two *different* eigenvariables c and d and contains a proof of $P\, c\, d$. Since

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

is provable, it follows that

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

is provable: in particular, the *same* proof works after identifying c and d.

Thus, eigenvariables are unlikely to capture the proper logic behind things like nonces, references, names, etc, particularly if *inequality* is important.

Logical specifications as definition

The *closed world assumption* assumption has been given a proof theory account using either *definitions* or *fixed points*.

We shall ignore most aspects of a proof theory of definitions except to say

- The definition clause $\forall x_1 \cdots \forall x_n[p(x_1, \ldots, x_n) \stackrel{\triangle}{=} Body]$ means that any occurrence of the atomic formula $p(t_1, \ldots, t_n)$ is a sequent can be replaced by $Body[x_1/t_1, \ldots, n_n/t_n]$.
- Equality has the following introduction rules:

$$\frac{}{\Delta \longrightarrow t = t} \qquad \frac{\delta\theta \longrightarrow C\theta}{\Delta, s = t \longrightarrow C} \text{ if } \theta = mgu(s, t)$$

A *failure* of s and t to unify is a *success* in the proof

$$\frac{}{\Delta, s = t \longrightarrow C} \text{ if } s \text{ and } t \text{ are not unifiable}$$

A new quantifier ∇

This problem illustrates a confusion about *where* eigenvariables are bound: at the object-level or meta-level (inside or outside the *provability* predicate).

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.

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

$$\Downarrow$$

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

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 a *generic judgment*.

Binder mobility is now more expressive given this new proof-level abstraction.

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.

Changes to the equality rules

Equality has the following introduction rules:

$$\frac{}{\Delta \longrightarrow \sigma \triangleright t = t} \qquad \frac{\Delta\theta \longrightarrow C\theta}{\Delta, \sigma \triangleright s = t \longrightarrow C} \text{ if } \theta = mgu(\lambda\sigma.s, \lambda\sigma.t)$$

$$\frac{}{\Delta, \sigma \triangleright s = t \longrightarrow C} \text{ if } \lambda\sigma.s \text{ and } \lambda\sigma.t \text{ are not unifiable}$$

If σ is the list $x_1, \ldots, x_n$ then $\lambda\sigma.t$ is an abbreviation for $\lambda x_1 \ldots \lambda x_n.t$.

Usually one aims for these unification problems to be within the pattern unification fragment. The more general setting can be accommodated in the theory.

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 $\forall u \forall v[q\ \langle u, t_1\rangle\ \langle v, t_2\rangle\ \langle v, t_3\rangle]$ is *provable* 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 should be able to prove the meta-level formula

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

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

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

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

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

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

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

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

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 by making various *actions*: constructor $\tau : a$ for *silent* actions and constructors $\downarrow, \uparrow : n \to n \to a$ for *input* and *output* actions.

$\downarrow xy$ 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 a *bound output action* and $\downarrow x : n \to a$ denotes a *bound input action*.

The one-step transition relation is encoded using two 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. Use ∇ instead:

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

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:

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

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

If the meta-logic is intuitionistic and free variables are interpreted as being $\forall$ quantified, then bisimulation corresponds to *open* bisimulation.

If the meta-logic is classic and free variables are interpreted as being ∇ quantified, then bisimulation corresponds to *closed* bisimulation.

What about infinite systems?

The proof theory for ∇ with noetherian definitions (e.g., finite π-calculus) appears unproblematic.

In such finite systems, one does not need either the cut rule or the initial rule. Thus, one does not need to ask the question:

when should $\sigma \triangleright B$ be equal to $\sigma' \triangleright B'$?

∇ commutes with all logical connectives. Another question remains:

how does ∇ commute with the definition mechanism?

Both questions need to be addressed when developing induction and coinduction principles.

These questions are addressed in different ways in the recent work by Baelde, Gacek, Nadathur, and Tiu. Some of this will be presented later today.

Putting this all together

How can we exploit ∇ and the proof theory of proof search more generally to reason about logic specifications?

Object-logic: write your specifications using Horn clauses, hereditary Harrop formulas, or linear logic. The proof theory of such logics is well developed (via results about uniform proofs and focusing proofs). The main judgment of interest here is

Does the sequent $x_1, \ldots, x_n : \Delta \longrightarrow G$ have a proof?

The eigenvariables $x_1, \ldots, x_n$ arise via uses of the $\forall$ quantifier: ∇ has no role.

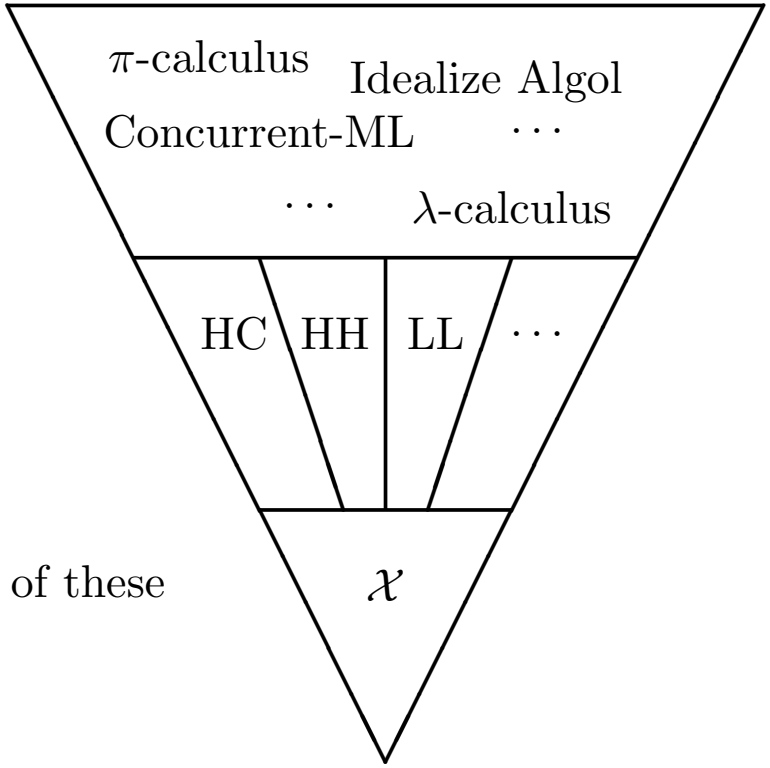
Meta-logic: Map object-level formulas as meta-level terms and now translate the above sequent judgment into the generic judgment $x_1, \ldots, x_n \triangleright \mathit{seq}\ \Delta\ G$. Here, *seq* translates the object-level $\forall$ to the meta-level ∇.

A two-level architecture

I. Applications: focus on operational semantics

II. Object-logics: Maybe Girard's LU instead

III. Meta-logic: Just one of these



Here, $\mathcal{X}$ is one of the logics being developed by Baelde, Gacek, and Tiu.

Implemented systems

Teyjus (Nadathur et. al.; Version 2, April 2008) is an implementation of λProlog. Useful for animating a wide range of logic specifications (as well as doing more general purpose programming).

Bedwyr (Baelde, Gacek & Tiu; November 2006) is an OCaml implementation of a model checker for logic involving ∇ and finite logic specifications. It can compute open bisimulation for finite π-calculus expressions (as well as much more).

Abella (Gacek; February 2008) is a proof editor containing induction and coinductive principles. A range of meta-level theorems have been constructed in this system.

Taci (Baelde, Snow & Viel; private release June 2008) is a proof editor for a logic with ∇ and explicit least and greatest fixed point operators. It features a tactic for doing significant automatic reasoning.

All these systems are written in OCaml.

Related work and conclusion

I summarize making a few brief comments.

- Although they come from different motivations and foundational considerations, there are lots of similarities to note with the nominal approach of Pitts, Gabbay, and others.
- The Twelf/M_2 approach to reasoning about specification has many overlapping considerations. Relating the type-theoretic and the proof-theoretic approaches is not, however, as easy as it should be.
- Format rules for SOS: Given that the connection between the logic and bisimulation was so strong, can the connection be exploited further? A generalize tfyt/txyt format rules for process calculi with name-passing has been defined: it guarantees that that open bisimulation is a congruence (Ziegler/Miller/Palamidessi, SOS 2005).
- The ∇-quantifier has passed a number of tests (cut-elimination, applications, implementations). It should have a model theoretic semantic characterization. A candidate might be Ulrich Schoepp's LFMTP 2006 paper.