Formalizing SOS specifications in logic

Dale Miller, *INRIA-Saclay & LIX, École Polytechnique*

Based on technical results in:

- M & Tiu: "Generic Judgments", lics03, ToCL 2005
- Tiu: *Model Checking for π-Calculus*, concur05
- Ziegler, M, Palamidessi: *A congruence format for name-passing*, sos05
- Gacek, M, Nadathur: *Combining generic judgments with recursive definitions*, lics08.

Collaboration between the INRIA team Parsifal, the University of Minnesota, and the Australian National University.

The overview of the next 10^{-6} century

Brief remarks about the uses of logic in computing

Making syntax more abstract and declarative

The π-calculus as an example and counterexample

The ∇-quantifier

More about the π-calculus

Roles of Logic in the Specification of Computation

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 captured by changes to sequents that occur during proof search.

Two "logic programming" approaches: Processes-as-formulas

- Combinators of process calculus are mapped to logical connectives: for example, $\mid$ is $\otimes$ and restriction is $\exists$.
- Substructural logics (e.g., linear logic, multiset rewriting) are often needed.

This approach to specification is exciting but limited.

If $P \vdash Q$ means (multi-step) transition, then $\vdash P \equiv Q$ is the finest equivalence possible: hence, this style approach does not capture bisimulation (a more fine equivalence).

Two "logic programming" approaches: Processes-as-terms

Processes are modelled using terms: e.g., $|$ and $+$ are binary terms constructors.

This more conventional approach (involves intuitionistic or classical logic). Sometimes called the *relational approach* to SOS.

We focus here on this style of semantic specification and the challenges to make logic expressive enough.

Two goals of this work: to use

- *proof theory* approaches to specify meaning for SOS, and
- *automated deduction* techniques to build tools for supporting SOS specification and reasoning.

Some tools we're building for SOS

Since we need to support binding in terms and formulas, these tools were built from scratch and implement aspects of *higher-order unification* and various *extensions* to Horn clauses.

Animation: *λProlog* (1989) work well to animate many SOS specifications: particularly, using the *Teyjus* (2008) implementation.

Model Checking: *Bedwyr* (2006) is a deduction system that can be used as a model checker. Successful examples: completely declarative bisimulation checker for the finite π-calculus.

Theorem Proving: To prove richer properties about possibly infinite systems, we are implementing some theorem provers: *Abella* and *Taci*. Example theorems: open bisimulation is a congruence, subject-reduction theorems.

The evolving nature of specifications

Denotational Semantics: computationally similar to functional programming (Scheme, ML, etc).

Structural Operational Semantics: computationally similar to logic programming, especially if the paradigm is generalized to

- treat λ-bindings in terms,
- explicit fixed point constructions (closed-world assumption), and
- various extension to Horn clauses.

Teaching of SOS: Logic programming can be used to animate and experiment with SOS specifications: especially a modern updating of Prolog including typing, modules, higher-order quantification (λProlog and Teyjus again).

Making syntax more abstract

Syntax as strings: White space, infix/prefix, parentheses. Much too concrete. Church and Gödel did meta-theory in logic viewing formulas as strings. Despite this choice, they achieved interesting results!

Syntax as parse trees: Parse string and remove white space, infix/prefix distinctions, etc. Organize as trees to encode recursive structures.

Syntax as λ-trees: Bound variable names are still treated too concretely. Treat these modulo $\alpha\beta\eta$-conversion. Requires more support from logic than is provided by Horn clauses.

Example: encoding finite π calculus

Concrete syntax of π-calculus processes:

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

Three syntactic types: n for names, a for actions, and p for processes. The type n may or may not be inhabited.

Three constructors for actions: $\tau : a$ and $\downarrow$ and $\uparrow$ (for input and output actions, resp), both of type $n \to n \to a$.

Abstract syntax for processes uses λ-bindings: $(y)Py$ is coded using a constant $nu : (n \to p) \to p$ as $nu(\lambda y.Py)$ or just $(nu\ P)$. Input prefix $x(y).Py$ is encoded using a constant $in : n \to (n \to p) \to p$ as $in\ x\ (\lambda y.Py)$ or just $(in\ x\ P)$. Other constructions are encoded similarly.

π-calculus: one step transitions

The "free action" arrow $\cdot \xrightarrow{\cdot} \cdot$ relates p and a and p.

The "bound action" arrow $\cdot \xrightharpoonup{\cdot} \cdot$ relates p and $n \to a$ and $n \to p$.

$P \xrightarrow{A} Q$ free actions, $A : a$ (τ, $\downarrow xy$, $\uparrow xy$)

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

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

Some small-step rules presented as formulas:

output–act:	$\forall x, y, P.$	$\top$	$\supset$	$\bar{x}y.P \xrightarrow{\uparrow xy} P$
input–act:	$\forall x, M.$	$\top$	$\supset$	$x(y).My \xrightharpoonup{\downarrow x} M$
match:	$\forall x, P, Q, \alpha.$	$P \xrightarrow{\alpha} Q$	$\supset$	$[x = x]P \xrightarrow{\alpha} Q$
res:	$\forall P, Q, \alpha.$	$\forall x(Px \xrightarrow{\alpha} Qx)$	$\supset$	$(x)Px \xrightarrow{\alpha} (x)Qx$

Proving positives but not negatives

The following can be proved.

Adequacy Theorem: The following are provable from the specification of the π-calculus

$$P \xrightarrow{A} P' \qquad P \xrightarrow{\uparrow X} M \qquad P \xrightarrow{\downarrow X} M$$

if and only if the "corresponding" transition holds in the π-calculus.

But:

You cannot prove interesting negations, even if you turn specification into "bi-conditionals" ($\triangleq$). *E.g.*, there is no proof of

$$\forall x \forall A \forall P.\ \neg[(y)[x = y].\bar{x}x.0 \xrightarrow{A} P]$$

Say good-bye to proving bisimulation.

The fault is in the use of eigenvariables at the meta-level.

Problem: eigenvariables collapse

An attempt to prove $\forall x \forall y. P\, x\, y$ first introduces two new and different 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.

Generic judgments and a new quantifier

Gentzen's introduction rule for $\forall$ on the left is *extensional:* $\forall x$ mean a (possibly infinite) conjunction indexed by terms.

The quantifier $\nabla x.B\,x$ provides a more *"intensional"*, *"internal"*, or *"generic"* reading. It uses a new local context in sequents.

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

$$\Downarrow$$

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

Σ is a list of distinct eigenvariables, scoped over the sequent and σ_i is a list of distinct 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.

The ∇-quantifier

The left and right introductions for ∇ (nabla) are the same.

$$\frac{\Sigma : (\sigma, x : \tau) \triangleright B, \Gamma \longrightarrow \mathcal{C}}{\Sigma : \sigma \triangleright \nabla_\tau x.B, \Gamma \longrightarrow \mathcal{C}} \qquad \frac{\Sigma : \Gamma \longrightarrow (\sigma, x : \tau) \triangleright B}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \nabla_\tau x.B}$$

Standard proof theory design: Enrich context and add connectives dealing with these context.

Quantification Logic: Add the eigenvariable context; add $\forall$ and $\exists$.

Linear Logic: Add multiset context; add multiplicative connectives.

Also: hyper-sequents, calculus of structures, etc.

Such a design, augmented with cut-elimination, provides modularity of the resulting logic.

Properties of ∇

This quantifier moves through all propositional connectives:

$$\begin{array}{cc} \nabla x \neg Bx \equiv \neg \nabla x Bx & \nabla x (Bx \supset Cx) \equiv \nabla x Bx \supset \nabla x Cx \\ \nabla x . \top \equiv \top & \nabla x (Bx \wedge Cx) \equiv \nabla x Bx \wedge \nabla x Cx \\ \nabla x . \bot \equiv \bot & \nabla x (Bx \vee Cx) \equiv \nabla x Bx \vee \nabla x Cx \end{array}$$

It moves through the quantifiers by *raising* them.

$$\begin{array}{rcl} \nabla x_\alpha \forall y_\beta . Bxy & \equiv & \forall h_{\alpha \to \beta} \nabla x_\alpha . Bx(hx) \\ \nabla x_\alpha \exists y_\beta . Bxy & \equiv & \exists h_{\alpha \to \beta} \nabla x_\alpha . Bx(hx) \end{array}$$

Consequence: ∇ can always be given atomic scope within formulas, at the "cost" of raising quantifiers. Finally,

$$(\nabla \bar{x} . t = s) \quad \text{iff} \quad (\lambda \bar{x} . t) = (\lambda \bar{x} . s).$$

Non-theorems

$$\nabla x \nabla y Bxy \supset \nabla z Bzz \qquad \nabla x Bx \supset \exists x Bx^{\dagger}$$

$$\nabla z Bzz \supset \nabla x \nabla y Bxy \qquad \forall x Bx \supset \nabla x Bx^{\dagger}$$

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

† These are theorems using the "new" quantifier of Pitts. (More comparisons later.)

Meta theorems

Theorem: *Cut-elimination.* Given a fixed stratified definition, a sequent has a proof if and only if it has a cut-free proof. (Tiu 2003: also when induction and co-induction are added.)

Theorem: For a fixed formula B,

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

Theorem: If we restrict to *Horn specification* (no implication or negations in the body of the clauses) then

1. $\forall$ and ∇ are interchangeable in specifications.
2. For a fixed B, $\vdash \nabla x . B\, x \supset \forall x . B\, x$.

Returning to the π-calculus

Replace $\forall$ in premises with ∇: e.g.,

$$\text{res:}\quad \forall P, Q.[\nabla x(Px \xrightarrow{\alpha} Qx) \supset (x)Px \xrightarrow{\alpha} (x)Qx]$$

We can now prove

$$\forall w \forall A \forall P.\ \neg.(x)[w = x].\bar{w}w.0 \xrightarrow{A} P$$

This proof requires observing that the equation

$$\lambda x.w = \lambda x.x.$$

has no solution for any instance of w (unification failure).

π-calculus: encoding (bi)simulation

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

This definition clause is not Horn and helps to illustrate the differences between $\forall$ and ∇.

Bisimulation (*bisim*) is easy to write: it has 6 cases.

The early version of bisimulation is a change in quantifier scope.

Learning something from our encoding

Theorem: For the finite π-calculus we have:

P is *open bisimilar* to Q if and only if $\vdash_I \forall \bar{x}.bisim\ P\ Q$.

P is *late bisimilar* to Q if and only if

$$\forall w \forall y (w = y \vee w \neq y) \vdash_I \nabla \bar{x}.bisim\ P\ Q.$$

Should one assume this instance of *excluded middle*?

The *Bedwyr* prover, which implements ∇ and fixed point extensions to logic, can prove bisimulation for (finite) π-calculus. Note that this is an implementation of a *logic* that can be used for a range of SOS-related tasks.

Modal logics

Tiu [concur05] specified modal logics for the π-calculus:

$$
\begin{array}{lcl}
P \models \langle\uparrow X\rangle A & \triangleq & \exists P'(P \xrightarrow{\uparrow X} P' \wedge \nabla y.P'y \models Ay). \\
P \models [\uparrow X]A & \triangleq & \forall P'(P \xrightarrow{\uparrow X} P' \supset \nabla y.P'y \models Ay).
\end{array}
$$

$$
\begin{array}{lcl}
P \models \langle\downarrow X\rangle A & \triangleq & \exists P'(P \xrightarrow{\downarrow X} P' \wedge \exists y.P'y \models Ay). \\
P \models \langle\downarrow X\rangle^l A & \triangleq & \exists P'(P \xrightarrow{\downarrow X} P' \wedge \forall y.P'y \models Ay). \\
P \models \langle\downarrow X\rangle^e A & \triangleq & \forall y \exists P'(P \xrightarrow{\downarrow X} P' \wedge P'y \models Ay).
\end{array}
$$

$$
\begin{array}{lcl}
P \models [\downarrow X]A & \triangleq & \forall P'(P \xrightarrow{\downarrow X} P' \supset \forall y.P'y \models Ay). \\
P \models [\downarrow X]^l A & \triangleq & \forall P'(P \xrightarrow{\downarrow X} P' \supset \exists y.P'y \models Ay). \\
P \models [\downarrow X]^e A & \triangleq & \exists y \forall P'(P \xrightarrow{\downarrow X} P' \supset P'y \models Ay).
\end{array}
$$

Generalizing format rules for mobility: tyft

In the first order case:

$$\frac{\cdots \quad [P_i \xrightarrow{A_i} Y_i] \quad \cdots}{(f\ X_1\ \ldots\ X_n) \xrightarrow{A} Q}$$

Here, $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ are distinct (first-order) variables.

This format generalizes naturally to the following:

$$\frac{\cdots \quad \nabla u_1 \ldots \nabla u_k [P_i \xrightarrow{A_i} (Y_i u_1 \ldots u_n)] \quad \cdots}{(f\ X_1\ \ldots\ X_n) \xrightarrow{A} Q}$$

The distinct variables $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ are bound universally around the inference rule.

This format guarantees that *open bisimulation* is a congruence.

Alternations between $\forall$ and ∇ leads to the notion of *distinctions* that are used to define open bisimulation.

Future Work

Develop more examples. Currently we deal with many aspects of the π-calculus, λ-calculus, functional and imperative programming.

Improve the automatic model checker (Bedwyr) and the interactive provers (Abella, Taci).

Modularity of reasoning depends of achieving suitable abstractions over SOS theories: more use of higher-order logic (and maybe linear logic) may help here.

How to implement *late bisimulation*? How to automate effectively the instances of the excluded middle for equality?

What is a good model-theoretic semantics for ∇?