Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 162 (2006) 243–247

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Proof Theoretic Approach to Operational Semantics

Dale Miller[1],[2]

INRIA & LIX, École Polytechnique
Palaiseau, France

Abstract

Proof theory can be applied to the problem of specifying and reasoning about the operational semantics of process calculi. We overview some recent research in which λ-tree syntax is used to encode expressions containing bindings and sequent calculus is used to reason about operational semantics. There are various benefits of this proof theoretic approach for the π-calculus: the treatment of bindings can be captured with no side conditions; bisimulation has a simple and natural specification in which the difference between bound input and bound output is characterized using difference quantifiers; various modal logics for mobility can be specified declaratively; and simple logic programming-like deduction involving subsets of second-order unification provides immediate implementations of symbolic bisimulation. These benefits should extend to other process calculi as well. As partial evidence of this, a simple λ-tree syntax extension to the tyft/tyxt rule format for name-binding and name-passing is possible that allows one to conclude that (open) bisimilarity is a congruence.

Keywords: operational semantics, proof theoretic specifications, λ-tree syntax, rule formats, π-calculus

A number of frameworks have been used to formalize the semantics of process calculi and, more generally, programming languages. For example, algebra, category theory, and I/O automata have been used to provide formal settings for not only specifying but also reasoning about the operational semantics of calculi and languages. In this note, we overview recent results in making use of *proof theory* to encode and reason about such operational semantics. By the term "proof theory" we refer the study of proofs for logics, particularly in the style initiated by Gentzen.

To illustrate an immediate and natural connection between operational semantics and a proof theoretic approach to logic, notice that operational semantics (either "big-step" or "small-step") is often presented as inference rules. Occasionally, it is possible to encode such inference rules directly as theories in a logic: typically, as Horn clauses in a first-order logic. To achieve such an encoding, process calculus

[1] Support for this work comes from INRIA through the "Equipes Associées" Slimmer and from the ACI grants GEOCAL and Rossignol.

[2] Email: dale.miller@inria.fr

doi:10.1016/j.entcs.2005.12.089

expressions, actions, labels, *etc*, are encoded as first-order (algebraic) terms, one step transitions as atomic formulas, and inferences rules, such as,

$$\frac{A_1 \quad \cdots \quad A_n}{A_0} \qquad \text{as the formula} \qquad \forall \bar{x}[A_1 \wedge \ldots \wedge A_n \supset A_0]\ (n \geq 0).$$

Here, $A_0, \ldots, A_n$ are atomic formulas and the explicitly quantified variables in the list $\bar{x}$ are the *schema variables* implicitly quantified in the inference rule on the left. When such an encoding works, one expects that an atomic formula A is provable from the inference rules if and only if the corresponding Horn clauses (viewed as a theory or logic program) prove A.

There are several possible benefits for encoding operational semantics into logic in this fashion. For example, logic programming technology, such as unification and backtracking search, can convert operational semantics into an executable specifications [1]. In addition, some properties of semantic specifications, such as reachability and bisimilarity, can be automated [9].

It is not always possible to encode inference rules in this simple fashion. For example, side conditions are frequently added to inference rules and these conditions must also be encoded into logic and used as additional premises. If a process calculi has a notion of binder, such as in the π-calculus, the join-calculus, or Concurrent ML, then a number of side conditions are usually employed to enforce that variable names respect scope. A large number of such side conditions can, in fact, be eliminated by encoding inference rule into a logic that directly encodes λ-abstraction and higher-type quantification over λ-terms. Church's Simple Theory of Types provides a good starting point for such a logic. The addition of λ-abstractions and higher-type quantification is now a well studied and frequently implemented enhancement to logic programming that provides an internalization of term-level binders as well as α-conversion and object-level substitution [8].

The encoding of syntax involving binders as simply typed λ-terms in a logic with an equality that includes α, β, and η conversion is called the *λ-tree syntax* approach [3] to *higher-order abstract syntax*.

To illustrate this approach to encoding syntax, let type p denote the syntactic category of processes and consider encoding process expressions of the π-calculus into that type. For example, encoding the expression $P + Q$ can be done by introducing a constructor *plus* of type $p \to p \to p$ and (using "logic-level" application) forming the term $(plus\ P'\ Q')$, where P' and Q' are the encodings of P and Q, respectively. Similarly, the expression $x(y).P$, where x is a name and y is a binding with scope P, can be encoded using a constructor *in* of type $n \to (n \to p) \to p$ as the expression $(in\ x\ (\lambda y.P'))$, where the type n denotes the syntactic type of names and the expression P' denotes the encoding of P. Notice that a "logic-level abstraction" has been used to form the expression $\lambda y.P'$ of type $n \to p$. Similarly, other process combinators that do not involve bindings can be encoded with constructors with "algebraic type" (first-order type) while those involving binders would use second-order types. In the π-calculus, the only other combinator that requires a binder is the restriction operator: for this, the constants ν of type $(n \to p) \to p$ can be used to encode restriction (we abbreviate $\nu(\lambda x.P)$ as simply $\nu x.P$).

An important lesson learned from using computational systems involving λ-tree syntax is that bindings within terms need to be matched with bindings in formula (via quantifiers) and bindings in proofs (such as eigenvariables). In particular, binders have *mobility* from terms to formulas to proofs: a bound variable never becomes a free variable (or vice versa) during proof search [2].

To illustrate how bindings can be treated declaratively in operational semantics, consider specifying the operational semantics of the π-calculus. First, we shall use the up arrow $\uparrow$ and down arrow $\downarrow$ to encode input and output actions, respectively: in particular, the expression $(\uparrow Xy)$ denotes an input action on channel X of value y. Notice that the two expressions, $\lambda y.\uparrow Xy$ and $\uparrow X$, denoting *abstracted actions*, are equal up to η-conversion and can be used interchangeably. Second, we use the horizontal arrow $\longrightarrow$ to relate a processes with an action and a continuation (a process), and the "harpoon" $\longrightharpoonup$ to relate a process with an *abstracted* action and an *abstracted* continuation.

The following three rules are part of the specification of one-step transitions for the π-calculus: the full specification using λ-tree syntax can be found in [3,10,11].

$$\frac{\nabla n(Nn \xrightarrow{A} Mn)}{\nu n.Nn \xrightarrow{A} \nu n.Mn}\,(\textsc{res}) \qquad \frac{\nabla y(Ny \xrightarrow{\uparrow Xy} My)}{\nu y.Ny \xrightharpoonup{\lambda y.\uparrow Xy} \lambda y.My}\,(\textsc{open})$$

$$\frac{P \xrightharpoonup{\downarrow X} M \qquad Q \xrightharpoonup{\uparrow X} N}{P \mid Q \xrightarrow{\tau} \nu y.(My \mid Ny)}\,(\textsc{close})$$

The (close) rule illustrates that a bounded input and bounded output action can yield a τ step involving a new restriction in the continuation. The (res) rule illustrates how λ-tree syntax and appropriate quantification can remove the need for side conditions: since substitution in *logic* does not allow for the capture of bound variables, all instances of the premise of this rule has a horizontal arrow in which the action label does not contain the variable n free. Thus, the usual side condition for this rule is treated declaratively. Both the (res) and (open) rules illustrate the ∇-quantifier that was introduced in [5,4] for encoding "generic judgments". For our purposes here, the expression $\nabla x_\gamma.Bx$ can be thought of as provable if, given a newly constructed object c of type γ, the formula Bc is provable. This rule should be seen as being hypothetical: no assumption about whether or not the domain of the type γ is non-empty is made.

With rules in this style, it is easy to provide definitions for simulation and bisimulation: for example, the following equivalence can be used to define simulation between two π-calculus expressions.

$$\begin{aligned} sim(P,Q) \equiv\ & \forall A \forall P' \left[P \xrightarrow{A} P' \supset \exists Q' \left(Q \xrightarrow{A} Q' \wedge sim(P',Q') \right) \right] \wedge \\ & \forall X \forall N \left[P \xrightarrow{\downarrow X} N \supset \exists M \left(Q \xrightarrow{\downarrow X} M \wedge \forall w.sim(Nw, Mw) \right) \right] \wedge \\ & \forall X \forall N \left[P \xrightarrow{\uparrow X} N \supset \exists M \left(Q \xrightarrow{\uparrow X} M \wedge \nabla w.sim(Nw, Mw) \right) \right] \end{aligned}$$

Notice that bound inputs require the $\forall$ quantifier to quantify the comparisons of their continuation while bound outputs require the ∇ quantifier to quantify the comparisons of their continuation. Formally speaking, in order for this equivalence to correctly encode the greatest fixed point of the equivalence (bisimilarity), one must deal with co-induction explicitly within inference rules, following, for example, [7].

As described in [10], it is also possible to specify the modal operators of [6] in a similar, declarative style. Again, the need for side conditions on names, their scopes, and their occurrences is taken care of declaratively by logic.

Implementing the logic containing the ∇-quantifier does not require significant new technical devices. For example, rather straightforward extensions of higher-order logic programming techniques [8] have been used to build the deductive system described in [12], which computes not only one-step transitions but also symbolic bisimulation for finite π-calculus expressions (those not involving replication).

Given the high degree of declarativeness of specifications written using λ-trees syntax, it has been possible to define a generalization [13] of the tyft/tyxt rule format that captures name-binding and name-passing calculi and for which (open) bisimilarity is a congruence.

References

[1] Despeyroux, J., *Proof of translation in natural semantics*, in: *Proceedings of Symposium on Logic in Computer Science*, Cambridge, Mass, 1986, pp. 193–205.

[2] Miller, D., *Bindings, mobility of bindings, and the ∇-quantifier*, in: J. Marcinkowski and A. Tarlecki, editors, *18th International Workshop CSL 2004*, LNCS **3210**, 2004, p. 24.

[3] Miller, D. and C. Palamidessi, *Foundational aspects of syntax*, ACM Computing Surveys **31** (1999).

[4] Miller, D. and A. Tiu, *A proof theory for generic judgments: An extended abstract*, in: *Proc. 18th IEEE Symposium on Logic in Computer Science (LICS 2003)* (2003), pp. 118–127.

[5] Miller, D. and A. Tiu, *A proof theory for generic judgments*, ACM Transactions on Computational Logic **6** (2005), pp. 749–783.

[6] Milner, R., J. Parrow and D. Walker, *Modal logics for mobile processes*, Theoretical Computer Science **114** (1993), pp. 149–171.

[7] Momigliano, A. and A. Tiu, *Induction and co-induction in sequent calculus*, in: M. C. Stefano Berardi and F. Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in LNCS, 2003, pp. 293 – 308.

[8] Nadathur, G. and D. Miller, *An Overview of λProlog*, in: *Fifth International Logic Programming Conference* (1988), pp. 810–827.

[9] Ramakrishna, Y. S., C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift and D. S. Warren, *Efficient model checking using tabled resolution*, in: *Proceedings of the 9th International Conference on Computer Aided Verification (CAV97)*, number 1254 in LNCS, 1997, pp. 143–154.

[10] Tiu, A., *Model checking for π-calculus using proof search*, in: M. Abadi and L. de Alfaro, editors, *CONCUR*, Lecture Notes in Computer Science **3653** (2005), pp. 36–50.

[11] Tiu, A. and D. Miller, *A proof search specification of the π-calculus*, in: *3rd Workshop on the Foundations of Global Ubiquitous Computing*, ENTCS **138**, 2004, pp. 79–101.

[12] Tiu, A., G. Nadathur and D. Miller, *Mixing finite success and finite failure in an automated prover*, in: *Proceedings of ESHOL'05: Empirically Successful Automated Reasoning in Higher-Order Logics*, 2005, pp. 79 – 98.

[13] Ziegler, A., D. Miller and C. Palamidessi, *A congruence format for name-passing calculi*, in: *Proceedings of SOS 2005: Structural Operational Semantics*, 2005.