Electronic Notes in Theoretical Computer Science 58 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume58.html 20 pages

Encoding Generic Judgments: Preliminary results

Dale Miller [1]

Computer Science and Engineering Department, 220 Pond Lab, Pennsylvania State University Park, PA 16802-6106 USA

Abstract

Operational semantics is often presented in a rather syntactic fashion using relations specified by inference rules or equivalently by clauses in a suitable logic programming language. As it is well known, various syntactic details of specifications involving bound variables can be greatly simplified if that logic programming language has term-level abstractions (λ-abstraction) and proof-level abstractions (eigenvariables) and the specification encodes object-level binders using λ-terms and universal quantification. We shall attempt to extend this specification setting to include the problem of specifying not only relations capturing operational semantics, such as one-step evaluation, but also properties and relations about the semantics, such as simulation. Central to our approach is the encoding of generic object-level judgments (universally quantified formulas) as suitable atomic meta-level judgments. We shall encode both the one-step transition semantics and simulation of (finite) π-calculus to illustrate our approach.

1 Introduction

The operational semantics of a programming or specification language is often given in a relational style using inference rules following a small-step approach (a.k.a. structured operational semantic [36]) or big-step approach (a.k.a. natural semantics [13]). In either case, algebraic (first-order) terms are often used to encode the language being specified and the first-order theory of Horn clauses is often used to formalize and largely mechanize such semantic specifications [9].

For example, consider specifying a functional programming language that has a conditional specified using the following inference rule (following the

[1] Email: dale@cse.psu.edu

natural semantics style specification).

$$\frac{B \Downarrow \mathit{true} \qquad M \Downarrow V}{(\mathit{if}\ B\ M\ N) \Downarrow V} \qquad \frac{B \Downarrow \mathit{false} \qquad N \Downarrow V}{(\mathit{if}\ B\ M\ N) \Downarrow V}$$

These two inference figures can be mapped into the two first-order Horn clauses

$$\forall B \forall M \forall N \forall V [B \Downarrow \mathit{true} \wedge M \Downarrow V \ \supset (\mathit{if}\ B\ M\ N) \Downarrow V]$$
$$\forall B \forall M \forall N \forall V [B \Downarrow \mathit{false} \wedge N \Downarrow V \ \supset (\mathit{if}\ B\ M\ N) \Downarrow V]$$

Here, the down arrow is a non-logical, predicate symbol and an expression such as $N \Downarrow V$ is an atomic formula. A simple adequacy result shows that atomic formulas provable from such Horn clauses are exactly those for which there is a proof using the corresponding inference rules.

If these two rules are the only rules describing the evaluation of the conditional, then it should follow that if $(\mathit{if}\ B\ M\ M) \Downarrow V$ is provable then so is $M \Downarrow V$. In what logic can this be formalized and proved? For example, how might we prove the sequent

$$(\mathit{if}\ B\ M\ M) \Downarrow V \longrightarrow M \Downarrow V,$$

where B, M, and V are eigenvariables (universally quantified)? Since such a sequent contains no logical connectives, the standard sequent inference rules that introduce logical connective will not directly help here. One natural extension of the sequent calculus is then to add left and right introduction rules for atoms. Lars Hallnäs and Peter Schroeder-Heister [7,8,40], Girard [6], and more recently, McDowell and Miller [16,14,17] have all considered just such introduction rules for non-logical constants. We outline this kind of introduction rule in the next section.

2 A proof theoretic form of definitions

A *definition* is a finite collection of definition *clauses* of the form $\forall \bar{x}[H \stackrel{\triangle}{=} B]$, where H is an atomic formula (the one being defined), every free variable of the formula B is also free in H, and all variables free in H are contained in the list $\bar{x}$ of variables. Since all free variables in H and B are universally quantified, we often leave these quantifiers implicit when displaying definitional clauses. The atomic formula H is called the *head* of the clause, and the formula B is called the *body*. The symbol $\stackrel{\triangle}{=}$ is used simply to indicate a definitional clause: it is not a logical connective. The same predicate may occur in the head of multiple clauses of a definition: it is best to think of a definition as a mutually recursive definition of the predicates in the heads of the clauses.

Given a definition, the following two inference rules are used to introduce defined predicates. The right introduction rule is

$$\frac{\Gamma \longrightarrow B\theta}{\Gamma \longrightarrow A}\ \mathit{def}\mathcal{R}\ ,$$

provided that there is a clause $\forall\bar{x}[H \stackrel{\triangle}{=} B]$ in the given definition such that A is equal to $H\theta$. The left-introduction rule is

$$\frac{\{B\theta, \Gamma\theta \longrightarrow C\theta \mid \theta \in CSU(A,H) \text{ for some clause } \forall\bar{x}[H \stackrel{\triangle}{=} B]\}}{A, \Gamma \longrightarrow C}\ \mathit{def}\mathcal{L}\ ,$$

where the variables $\bar{x}$ are chosen to be distinct from the (eigen)variables free in the lower sequent of the rule. The set $CSU(A,H)$ denotes a complete set of unifiers for A and H: when the CSUs and definition are finite, this rule will have a finite number of premises. (A set S of unifiers for t and u is *complete* if for every unifier ρ of t and u there is a unifier $\theta \in S$ such that ρ is $\theta \circ \sigma$ for some substitution σ [12].) There are many important situations where CSUs are not only finite but are also singleton (containing a most general unifier) whenever terms are unifiable. One such case is, of course, the first-order case. Another case is when the application of functional variables are restricted to distinct bound variables in the sense of *higher-order pattern* unification [20,32]. In this paper, many unification problems will fall into this latter case.

We must also restrict the use of implication in the bodies of definitional clauses, otherwise cut-elimination does not hold [39]. To that end we assume that each predicate symbol p in the language is associated with it a natural number $\mathrm{lvl}(p)$, the *level* of the predicate. We then extend the notion of level to formulas and derivations. Given a formula B, its *level* $\mathrm{lvl}(B)$ is defined as follows:

(i) $\mathrm{lvl}(p\,\bar{t}) = \mathrm{lvl}(p)$
(ii) $\mathrm{lvl}(\bot) = \mathrm{lvl}(\top) = 0$
(iii) $\mathrm{lvl}(B \wedge C) = \mathrm{lvl}(B \vee C) = \max(\mathrm{lvl}(B), \mathrm{lvl}(C))$
(iv) $\mathrm{lvl}(B \supset C) = \max(\mathrm{lvl}(B) + 1, \mathrm{lvl}(C))$
(v) $\mathrm{lvl}(\forall x.B) = \mathrm{lvl}(\exists x.B) = \mathrm{lvl}(B)$.

We now require that for every definitional clause $\forall\bar{x}[p\,\bar{t} \stackrel{\triangle}{=} B]$, $\mathrm{lvl}(B) \leq \mathrm{lvl}(p\,\bar{t})$. (If the definition is based on Horn clauses, then this restriction is trivial to satisfy since no implications would occur in the body of definitional clauses.) Cut-elimination for this use of definition within intuitionistic logic was proved in [14] and [17] and is modeled on proofs by Tait and Martin-Löf that use the technical notions of normalizability and reducibility. In fact, that proof also allowed the logic to contain a formulation of induction, a topic we return to later.

We can think of definitions as a technique to introduce logical equivalences in such a way that we do not introduce into proof search meaningless cycles: that is, if we simply considered $H \equiv B$, then when proving a sequent containing H, we could replace it with B, which could then be replaced with H, etc.

To illustrate the strengthening of logic that can result from adding definitions in this way, consider the motivating sequent above. We first convert the two Horn clauses representing the evaluation rules for the conditional into the following definitional clauses. (We employ the usual convention that free variables in displayed definitional clauses are implicitly universally quantified around that clause.)

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

This sequent then has the following simple and immediate proof.

$$\dfrac{\dfrac{\dfrac{}{B\Downarrow \mathit{true}, M\Downarrow V \longrightarrow M\Downarrow V}\ \mathit{initial}}{B\Downarrow \mathit{true}\wedge M\Downarrow V \longrightarrow M\Downarrow V}\ \wedge L \qquad \dfrac{\dfrac{}{B\Downarrow \mathit{false}, M\Downarrow V \longrightarrow M\Downarrow V}\ \mathit{initial}}{B\Downarrow \mathit{false}\wedge M\Downarrow V \longrightarrow M\Downarrow V}\ \wedge L}{(\mathit{if}\ B\ M\ M)\Downarrow V \longrightarrow M\Downarrow V}\ \mathit{def}\mathcal{L}$$

In the paper [18], the expressive strength of definitions was studied in greater depth. One example considered there involved attempting to capture the notion of simulation and bisimulation for labelled transition systems. In particular, assume that $P \xrightarrow{A} P'$ is defined via clauses to which are added the two clauses in Figure 1. These two clauses are a direct encoding of the closure conditions for simulation and bisimulation. In [18] it was proved that if the labeled transition system is finite (noetherian) then simulation and bisimulation coincided exactly with provability of $\mathit{sim}\ P\ Q$ and $\mathit{bisim}\quad P\ Q$. The restriction to noetherian transition systems is necessary since in such situations, these closure clauses have unique fixed points and since provability yields atoms that are true in all fixed points, provability correctly characterizes that fixed point. In transition systems with infinite paths, the least fixed point and greatest fixed point differ, so provability no longer captures only the greatest fixed point (simulation and bisimulation are greatest fixed points). If, however, the transition system is finitely branching, then induction can be used to characterize the greatest fixed point by repeatedly applying the closure operator to the trivially true relation.

In Section 5 we show how we can capture simulation and bisimulation for the (finite) π-calculus in a similar style.

3 Should we explicitly reference provability?

Although we have now succeeded in giving the sequent $(\mathit{if}\ B\ M\ M)\Downarrow V \longrightarrow M\Downarrow V$ a natural proof using this proof theoretic notion of definition, it appears that we need to revisit what we really have in mind for that sequent. It seems

$$
\begin{aligned}
\mathit{sim}\ P\ Q &\triangleq \forall A \forall P'.\ P \xrightarrow{A} P' \supset \exists Q'.\ Q \xrightarrow{A} Q' \wedge \mathit{sim}\ P'\ Q' \\
\mathit{bisim}\ P\ Q &\triangleq [\forall A \forall P'. P \xrightarrow{A} P' \supset \exists Q'. Q \xrightarrow{A} Q' \wedge \mathit{bisim}\ P'\ Q'] \wedge \\
&\quad\ [\forall A \forall Q'. Q \xrightarrow{A} Q' \supset \exists P'. P \xrightarrow{A} P' \wedge \mathit{bisim}\ Q'\ P']
\end{aligned}
$$

Fig. 1. Simulation and bisimulation as definitions.

more natural that what we intend to prove is rather: "if $(\mathit{if}\,B\ M\ M) \Downarrow V$ is *provable* then $M \Downarrow V$ is *provable*." To explore this possibility, consider introducing the predicate $\triangleright \cdot$ that serves as an operator for provability. In this case, we now need to distinguish between two logics, one meta-level logic and one object-level logic. To do so, we shall use the type o to denote meta-level logical expressions and obj to denote object-level logical expressions. The meta-logic uses the symbols $\forall_\sigma$ of type $(\sigma \to o) \to o$, $\exists_\sigma$ of type $(\sigma \to o) \to o$ and $\wedge$ and $\supset$, both of type $o \to o \to o$ for universal and existential quantification at type σ, for conjunction, and for implication, respectively. The object-logic uses the symbols $\bigwedge_\sigma$ of type $(\sigma \to \mathit{obj}) \to \mathit{obj}$ and & and $\Rightarrow$, both of type $\mathit{obj} \to \mathit{obj} \to \mathit{obj}$ for universal quantification at type σ, for conjunction, and for implication, respectively. (The type subscripts for $\forall_\sigma$ and $\bigwedge_\sigma$ will often be dropped if they can be easily inferred or are not important.) As is the usual convention, the expression $\bigwedge \lambda x$ will be abbreviated as simply $\bigwedge x$.

To encode the provability relation for the object-logic, we copy the structure of a logic programming interpreter following the completeness theorems for uniform proofs and backchaining found in, say, [19], or the notion of focused proof [1,22]. Our interpreter will use the following four predicates: provability is denoted by $\triangleright$ and has type $\mathit{obj} \to o$, backchaining is denoted by the infix symbol $\triangleleft$ and has type $\mathit{obj} \to \mathit{obj} \to o$, *atomic* of type $\mathit{obj} \to o$ decides if an object-level formula is atomic, and *prog*, also of type $\mathit{obj} \to o$, decides if a formula is an object-level assumption (object-level logic program). A Horn clause interpreter will be written as the definition in Figure 2. Notice that in both the $\triangleright \cdot$ and $\cdot \triangleleft \cdot$ expression, the triangle points to the formula for which (object-level) introductions rules are considered (right-rules for $\triangleright \cdot$ and left-rules for $\cdot \triangleleft \cdot$).

The full specification of provability at the object level would then require additional definitional clauses for specifying what are atomic object-level formulas and what formulas constitute the object-level Horn clause specification. Examples of such clauses are given in Figure 3. Here, $\Downarrow$ has type $\mathit{tm} \to \mathit{tm} \to \mathit{obj}$, where tm is the type of the intended programming language that we are attempting to encode the operational semantics. We can now prove the sequent

$$\triangleright(\mathit{if}\,B\ M\ M) \Downarrow V \longrightarrow \triangleright M \Downarrow V.$$

Proposition 3.1 *An atomic judgment has a proof using inference figures if and only if it has a proof using $\triangleright \cdot$ (Figure 2) in which the inference figures are encoded as* atomic $\cdot$ *and prog $\cdot$ clauses (as in Figure 3).*

$$\begin{aligned}
\rhd(G \mathbin{\&} G') &\triangleq \rhd G \wedge \rhd G'.\\
\rhd A &\triangleq \mathit{atomic}\ A \wedge \mathit{prog}\ D \wedge D \lhd A.\\
A \lhd A &\triangleq \mathit{atomic}\ A.\\
(G \Rightarrow D) \lhd A &\triangleq D \lhd A \wedge \rhd G.\\
(\textstyle\bigwedge_\sigma .Dx) \lhd A &\triangleq \exists_\sigma t\ (D\ t \lhd A).
\end{aligned}$$

Fig. 2. Interpreter for object-level specifications.

$$\begin{aligned}
\mathit{atomic}\ (M \Downarrow V) &\triangleq \top.\\
\mathit{prog}\ (\textstyle\bigwedge B \bigwedge M \bigwedge N \bigwedge V[B \Downarrow \mathit{true} \mathbin{\&} M \Downarrow V \supset (\mathit{if}\, B\ M\ \ N) \Downarrow V]) &\triangleq \top.\\
\mathit{prog}\ (\textstyle\bigwedge B \bigwedge M \bigwedge N \bigwedge V[B \Downarrow \mathit{false} \mathbin{\&} N \Downarrow V \supset (\mathit{if}\, B\ M\ \ N) \Downarrow V]) &\triangleq \top.
\end{aligned}$$

Fig. 3. Specification of object-level inference rules via Horn clauses.

Proof outline. Applications of inference figures correspond exactly to the selection of *prog*· clauses for backchaining over. The completeness for the treatment of the goal-reduction and backchaining steps follows from familiar completeness theorems for logic programs [27]. □

Having now described this interpreter, it is interesting to note that, although conceptually there might be important distinctions arising from using $\rhd$, provability of $M \Downarrow V$ directly from its Horn clause specification or indirectly via the use of this interpreter are essentially the same: (uniform) proofs in one setting map naturally to (uniform) proofs in the other setting. From a practical point of view, this distinction does not provide any proof search advantages.

If we leave Horn clauses for a logic with universally quantified judgments, then a difference does appear. We look at this next.

4 λ-tree syntax and generic judgments

It is a common observation that first-order terms are not expressive enough to capture rich syntactic structures declaratively. In particular, such terms do not permit a direct encoding of the syntactic category of "abstraction" and the associated notions of α-conversion and substitution.

4.1 Syntactic representation of abstractions

The encoding style called *higher-order abstract syntax* [35] views such abstractions as functional expressions that relying on the full power of β-conversion in a typed λ-calculus setting to perform substitutions. The computer systems

λProlog, Elf, Isabelle, and Coq, to name a few, all implement a form HOAS and many earlier papers have appeared exploiting this style of syntactic representation [24,25,26,34]. Since the earliest papers, however, there has been a tendency to consider richer λ-calculi as foundations for HOAS, moving away from the simply typed λ-calculus setting where it was first exploited. Trying to encode a syntactic category of abstraction by placing it within a rich function spaces can cause significant problems (undecidable unification, exotic terms, etc) that might seem rather inappropriate if one is only trying to develop a simple treatment of syntax.

The notion of *λ-tree syntax* [28,23] was introduced to work around these complexities. Here, λ-abstractions are not general functions: they can only be applied to other, internally bound variables. Substitution of general values is not part of the equality theory in the λ-term syntax approach: it must be coded as a separate judgment via logic. This weaker approach has a much simpler equality theory, yielding a unification setting (called L_λ [20] or *higher-order pattern unification* [32,33]) which is decidable and unary. The relationship between the λ-tree approach where abstractions are applied to only internal bound variable and HOAS where abstractions can be applied to general terms is rather similar to the distinctions made in the π-calculus between π_I, which only allows "internal mobility" [38] and the full π-calculus, where "external mobility" is also allowed (via general substitutions). In Section 5.2, we will see that this comparison is not accidental. In this paper, we generally view syntax as encoded using λ-trees.

4.2 Generic judgments as atomic meta-level judgments

When using HOAS or λ-tree syntax representations, inference rules of the form

$$\frac{\bigwedge x.Gx}{A} \quad ,$$

are often encountered. If one were to capture this in the interpreter described in Figure 2, there would need to be a way to interpret universally quantified goals. One is tempted to augment that earlier interpreter with the following clause:

(1) $$\rhd(\bigwedge_\sigma x.G\ x) \stackrel{\triangle}{=} \forall_\sigma x[\rhd G\ x],$$

that is, the object-level universal quantifier would be interpreted using the meta-level universal quantifier. While this is a common approach to dealing with object-level universal quantification, this encoding causes some problems when attempting to reason about logic specifications containing generic judgments.

For example, consider proving the query $\forall y_1 \forall y_2 [q\ \langle y_1, t_1\rangle\ \langle y_2, t_2\rangle\ \langle y_2, t_3\rangle]$, where $\langle\cdot,\cdot\rangle$ is used to form pairs, from the three clauses

$$q\ X\ X\ Y.$$

$$q\ X\ Y\ X.$$

$$q\ Y\ X\ X.$$

This query succeeds only if t_2 and t_3 are equal. In particular, we would like to prove the sequent

$$\triangleright(\bigwedge y_1 \bigwedge y_2 [q\ \langle y_1, t_1\rangle\ \langle y_2, t_2\rangle\ \langle y_2, t_3\rangle]) \longrightarrow t_2 = t_3,$$

where t_1, t_2, and t_3 are eigenvariables and with a definition that consists of the clause (1), those clauses in Figure 2, and the following clauses:

$$X = X \triangleq \top$$

$$\mathit{prog}\ (\bigwedge X \bigwedge Y\ q\ X\ X\ Y) \triangleq \top$$

$$\mathit{prog}\ (\bigwedge X \bigwedge Y\ q\ X\ Y\ X) \triangleq \top$$

$$\mathit{prog}\ (\bigwedge X \bigwedge Y\ q\ Y\ X\ X) \triangleq \top$$

Using these definitional clauses, this sequent reduces to

$$\triangleright(q\ \langle s_1, t_1\rangle\ \langle s_2, t_2\rangle\ \langle s_2, t_3\rangle) \longrightarrow t_2 = t_3,$$

for some terms s_1 and s_2. This latter sequent is provable only if s_1 and s_2 are chosen to be two non-unifiable terms. This style proof is quite unnatural and it also depends on the fact that the underlying type that is quantified in $\forall y_1 \forall y_2$ is non-empty.

Additionally, if we use the rule (1) then whenever $\triangleright(\bigwedge_\sigma x.G\ x)$ is provable, the meta-level atomic formula $\triangleright(G\ t)$ is provable for all terms t of type σ. While this is likely to be appropriate when the object-language is a conventional logic, it is not likely to be appropriate when one encodes something like the π-calculus where an object-level universal quantifier might be used to encode restriction but where the presence of, say, a match prefix means that general substitutions may not be applicable to judgments generally.

For these reasons, the conversion of an object-level universal quantifier into a meta-level universal quantifier in (1) must be judged inappropriate. We now look for a different approach.

Consider the rule for proving a universal formula:

$$\frac{\Gamma, c : \sigma \vdash Pc}{\Gamma \vdash \bigwedge_\sigma x.Px}$$

where c is an eigenvariable with the usual restriction that c is not free in the lower sequent. Here, P is a variable of higher type, and the context, Γ, denotes a set of distinct typed eigenvariables. If we see the judgment $\Gamma \vdash \bigwedge_\sigma x.Px$ as "atomic" and that this is the only way to prove a universally quantified formula, then this rule can be inverted: that is, if $\bigwedge_\sigma x.Px$ is provable assuming the variables in Γ are generic then Pc is provable assuming that c is also generic. Whether or not that generic c can be instantiated and yield another valid judgment is dependent on the object-level itself. In other words, we will require that this universally quantified variable acts as a bound variable but will not assume that it can be arbitrarily instantiated: using an analogy from before, we will assume that we can apply this abstraction to another abstracted variable but not to an arbitrary value.

Thus we need to encode the object-level judgment $x_1, \ldots, x_n \vdash (Px_1 \ldots x_n)$, where the variables on the left are all distinct and understood as bound entirely within this judgment, as an atomic formula in our meta-logic. We mention two ways to achieve this encoding. The first introduces a "local" binders using a family of constants, say, loc_σ of type $(\sigma \to obj) \to obj$. The above expression would be something of the form

$$loc_{\sigma_1} \lambda x_1 \ldots loc_{\sigma_n} \lambda x_n.\ Px_1 \ldots x_n.$$

While this encoding is natural, it hides the top-level structure of $Px_1 \ldots x_n$ under a prefix of varying length. Unification and matching, which are central to the functioning of the definition introduction rules, would not be able to directly access that top-level structure. The second alternative employs a coding technique used by McDowell [14,15]. Here, one abstraction, say for a variable l of type evs (eigenvariables), is always written over the judgment and is used to denote the list of distinct variables $x_1, \ldots, x_n$. Individual variables are then accessed via the projections ρ_σ of type $evs \to \sigma$ and $\hat{\rho}$ of type $evs \to evs$. For example, the judgment $x : a, y : b, z : c \vdash Pxyz$ could be encoded as either the expression $loc_a \lambda x loc_b \lambda y loc_c \lambda z.Pxyz$, or as

$$\lambda l(P(\rho_a l)(\rho_b(\hat{\rho} l))(\rho_c(\hat{\rho}(\hat{\rho} l)))).$$

In this second, preferred encoding, the abstraction l denotes a list of variables, the first variable being of type a, the second being of type b, and the third of type c.

4.3 A interpreter for generic judgments

An interpreter for generic judgments is displayed in Figure 4. This interpreter generalizes the previous interpreter by allowing for the additional abstraction over evs. Here, the three meta-level predicates $atomic\ \cdot$, $prog\ \cdot$, and $\rhd \cdot$ all have the type $(evs \to obj) \to o$ while $\cdot \lhd \cdot$ has the type $(evs \to obj) \to (evs \to obj) \to o$. Notice that the technique of replacing the abstraction $\lambda l. \bigwedge_\sigma \lambda w.(G\ l\ w))$

$$\begin{aligned}
\rhd_l((G\ l)\ \&\ (G'\ l)) &\triangleq \rhd_l(Gl) \wedge \rhd_l(G'l).\\
\rhd_l(\textstyle\bigwedge_\sigma w.(G\ l\ w)) &\triangleq \rhd_l(G(\hat\rho l)(\rho_\sigma l)).\\
\rhd_l(Al) &\triangleq \mathit{atomic}\ A \wedge \mathit{prog} D \wedge (Dl) \lhd_l (Al).\\
(Al) \lhd_l (Al) &\triangleq \mathit{atomic}\ A.\\
((G\ l) \Rightarrow (D\ l)) \lhd_l (Al) &\triangleq (Dl) \lhd_l (Al) \wedge \rhd_l(Gl).\\
(\textstyle\bigwedge_\sigma w.(D\ l)) \lhd_l (Al) &\triangleq \exists_{\mathit{evs}\to\sigma} t.(D\ l\ (t\ l) \lhd_l (Al)).
\end{aligned}$$

Fig. 4. An interpreter for simple generic judgments.

with $\lambda l.G(\hat\rho l)(\rho l))$ is really the same as replacing the judgment

$$x_1, \ldots, x_n \vdash \forall y(Pyx_1 \ldots x_n) \quad \text{with} \quad x_1, \ldots, x_n, x_{n+1} \vdash (Px_1 \ldots x_n x_{n+1}).$$

Notice that this interpreter is not in L_λ for two reasons. First, the definitional clause for interpreting $(\lambda l. \bigwedge_\sigma \lambda w.(G\ l\ w))$ contains the expression $(\lambda l.G(\hat\rho l)(\rho_\sigma l))$ and the subterms $(\hat\rho l)$ and $(\rho_\sigma l)$ are not distinct bound variables. They are, however, distinct object-level variables so it should be a rather simple matter to extend the technical definition of L_λ to also allow for this style encoding of object-level variables. A second reason that this interpreter is not in L_λ is the definitional clause for backchaining over $(\lambda l. \bigwedge_\sigma w.(D\ l\ w))$ since this clause contains the expression $(\lambda l.D\ l\ (t\ l))$, which requires applying an abstraction to a general (external) term t. Such a specification can be made into an L_λ specification by encoding object-level substitution as an explicit judgment [21]. The fact that this specification is not in L_λ simply means that when we apply the left-introduction rule for definitions, unification may not produce a most general unifier.

Proposition 4.1 *Let $n \geq -1$ and let $x_0{:}\,\sigma_0, \ldots, x_n{:}\,\sigma_n$ be distinct variables such that $(Px_0 \ldots x_n)$ is an atomic formula in which the variables $x_0, \ldots, x_n$ are not free in P. The judgment $x_0, \ldots, x_n \vdash (Px_1 \ldots x_n)$ has a proof using inference figures (admitting universally quantified premises) if and only if*

$$\underset{l}{\rhd} P\ (\rho_{\sigma_0}\ l) \cdots (\rho_{\sigma_n}\ (\hat\rho^n l))$$

has a proof using the interpreter in Figure 4 in which the inference figures are encoded as atomic $\cdot$ and prog $\cdot$ clauses (as in Figure 3).

Proof outline. Applications of inference figures correspond exactly to the selection of *prog*$\cdot$ clauses for backchaining over. Here, the use of ρ and $\hat\rho$ ensures that object-level eigenvariables are represented by new terms at the meta-level. For this encoding to work properly, we also assume that no constants at the object-level have types involving *evs*. □

Other judgments besides generic judgments can be encoded similarly. For example, in [14,15], hypothetical as well as linear logic judgments were encoded along these lines. The main objective in those papers is to encode an object-level sequent as atomic judgments in a meta-logic. We focus on generic judgments here because of their relationship to abstractions within syntax.

5 The π-calculus

To illustrate the use of this style representation of universal judgments, we turn, as many others have done [28,11,3,37], to consider encoding the π-calculus. In particular, we follow the presentation in [28] for the syntax and one-step operational semantics.

5.1 Syntax

We shall follow the presentation of the π-calculus given in [29]. We need three primitive syntactic categories: name for channels, proc for processes, and action for actions. The output prefix is the constructor out of type $\mathit{name} \to \mathit{name} \to \mathit{proc} \to \mathit{proc}$ and the input prefix is the constructor in of type $\mathit{name} \to (\mathit{name} \to \mathit{proc}) \to \mathit{proc}$: the π-calculus expressions $\bar{x}y.P$ and $x(y).P$ are represented as $(\mathit{out}\ x\ y\ P)$ and $(\mathit{in}\ x\ \lambda y.P)$, respectively. We use $\mid$ and $+$, both of type $\mathit{proc} \to \mathit{proc} \to \mathit{proc}$ and written as infix, to denote parallel composition and summation, and ν of type $(\mathit{name} \to \mathit{proc}) \to \mathit{proc}$ to denote restriction. The π-calculus expression $(x)P$ will be encoded as $\nu\lambda n.P$, which itself is abbreviated as simply $\nu x.P$. The match operator, $[\cdot = \cdot]\cdot$ is of type $\mathit{name} \to \mathit{name} \to \mathit{proc} \to \mathit{proc}$. When τ is written as a prefix, it has type $\mathit{proc} \to \mathit{proc}$. When τ is written as an action, it has type action. The symbols $\downarrow$ and $\uparrow$, both of type $\mathit{name} \to \mathit{name} \to \mathit{action}$, denote the input and output actions, respectively, on a named channel with a named value.

We shall deal with only *finite* π-calculus expression, that is, expressions without ! or defined constants. Extending this work to infinite process expressions can be done using induction, as outlined in [18] or by adding an explicit co-induction proof rule dual to the induction rule. Fortunately, the finite expressions are rich enough to illustrate the issues regarding syntax and abstractions that are the focus of this paper.

Proposition 5.1 *Let P be a finite π-calculus expression using the syntax of [29]. If the free names of P are admitted as constants in the meta-logic of type* name *then P corresponds uniquely to a $\beta\eta$-equivalence class of terms of type* proc.

5.2 One-step transitions

The transition semantics uses two predicates: $\cdot \xrightarrow{\cdot} \cdot$ of type $\mathit{proc} \to \mathit{action} \to \mathit{proc} \to \mathit{obj}$; and $\cdot \xrightharpoonup{\cdot} \cdot$ of type $\mathit{proc} \to (\mathit{name} \to \mathit{action}) \to (\mathit{name} \to \mathit{proc}) \to o$. The first of these predicates encodes transitions involving free values and the second encodes transitions involving bound values. Figure 5 specifies the one step transition system for the "core" π-calculus. Figure 6 provides the increment to the core rules to get the late transition system, and Figure 7 gives the increment to the core to get the early transition system. Note that all the rules in the core system belong to the L_λ subset of logic specifications: that is, abstractions are applied to only abstracted variables (either bound by a λ-abstraction or bound by a universally quantifier in the premise of the rule). Furthermore, note that each of the increments for the late and early systems involve at least one clause that is not in L_λ. The core system of rules has also been singled out and named π_I [38] since it only allows for "internal" mobility of names, that is, local (restricted) names only being passed to abstractions.

One advantage of this style of specification over the traditional one [29] is the absence of complicated side-conditions on variables: they are handled directly by the logical mechanisms described above.

In order for this theory of one-step transitions to be interpreted by the prover given in Figure 4, we need to take the following steps.

- Convert the inference rules of either the core, late, or early system into *prog* clauses. This is straightforward (as illustrated in Section 1).
- Axiomatize the atomic predicate, which would simply be the two definitional clauses

$$\begin{aligned} \mathit{atomic}\ \ (P \xrightarrow{A} Q) &\ \triangleq \top \\ \mathit{atomic}\ \ (P \xrightharpoonup{A} Q) &\ \triangleq \top \end{aligned}$$

- One clause for the $\triangleright\cdot$ and one clause for $\cdot\triangleleft\cdot$ are parametrized by a type σ. Here, the clause for $\triangleright\cdot$ needs just one instance for σ equal to *name* while $\cdot\triangleleft\cdot$ needs 7 different instances on each for σ set equal to *name*, *action*, *proc*, *name* $\rightarrow$ *name*, *name* $\rightarrow$ *action*, *name* $\rightarrow$ *proc*, and *name* $\rightarrow$ *name* $\rightarrow$ *proc*.

Proposition 5.2 *Let $P \xrightarrow{A} Q$ be provable in late (resp., early) transition system of [29]. If A is either τ or $\downarrow xy$ or $\uparrow xy$ then $\triangleright_l P \xrightarrow{A} Q$ is provable from the clauses for the interpreter plus the clauses encoding late (resp., early) transitions. (Here, P, A, and Q are all translated to the corresponding meta-level expression.) If A is $x(y)$ then $\triangleright_l P \xrightharpoonup{\downarrow x} R$ and if A is $\bar{x}(y)$ then $\triangleright_l P \xrightharpoonup{\uparrow x} R$. Here, R is the meta-level representation of the λ-abstraction of y over Q.*

Proof Outline. Follows almost directly from Propositions 4.1 and 5.1. The induction needs to be strengthen slightly to handle the case where the bound variable l in $\triangleright_l$ are free in the judgment, which can happen, of course, when a universally quantified goal is interpreted. $\square$

Since the types of P and P' are different in the expression $P \xrightharpoonup{A} P'$, we cannot immediately form the transitive closure of this relationship to give a notion of a sequence of transitions. It is necessary to lower the type of P' first by applying it to a term of type *name*. How this is done, depends on what we

$$\frac{}{\tau.P \xrightarrow{\tau} P}\tau \qquad \frac{P \xrightarrow{A} Q}{[x = x]P \xrightarrow{A} Q}\text{match} \qquad \frac{P \xrightharpoonup{A} Q}{[x = x]P \xrightharpoonup{A} Q}\text{match}$$

$$\frac{P \xrightarrow{A} R}{P + Q \xrightarrow{A} R}\text{sum} \qquad \frac{Q \xrightarrow{A} R}{P + Q \xrightarrow{A} R}\text{sum} \qquad \frac{P \xrightharpoonup{A} R}{P + Q \xrightharpoonup{A} R}\text{sum}$$

$$\frac{Q \xrightharpoonup{A} R}{P + Q \xrightharpoonup{A} R}\text{sum} \qquad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q}\text{par} \qquad \frac{Q \xrightarrow{A} Q'}{P|Q \xrightarrow{A} P|Q'}\text{par}$$

$$\frac{P \xrightharpoonup{A} M}{P|Q \xrightharpoonup{A} \lambda n(Mn|Q)}\text{par} \qquad \frac{Q \xrightharpoonup{A} N}{P|Q \xrightharpoonup{A} \lambda n(P|Nn)}\text{par}$$

$$\frac{\bigwedge n(Pn \xrightarrow{A} P'n)}{\nu n.Pn \xrightarrow{A} \nu n.P'n}\text{res} \qquad \frac{\bigwedge n(Pn \xrightharpoonup{A} P'n)}{\nu n.Pn \xrightharpoonup{A} \lambda m\ \nu n.(P'mn)}\text{res}$$

$$\frac{}{\mathit{out}\ x\ y\ P \xrightarrow{\uparrow xy} P}\ \text{output} \qquad \frac{}{\mathit{in}\ x\ M \xrightharpoonup{\downarrow x} M}\ \text{input}$$

$$\frac{\bigwedge y(My \xrightarrow{\uparrow xy} M'y)}{\nu y.My \xrightharpoonup{\uparrow x} M'}\ \text{open}$$

$$\frac{P \xrightharpoonup{\downarrow x} M \qquad Q \xrightharpoonup{\uparrow x} N}{P|Q \xrightarrow{\tau} \nu n.(Mn|Nn)}\ \text{close} \qquad \frac{P \xrightharpoonup{\uparrow x} M \qquad Q \xrightharpoonup{\downarrow x} N}{P|Q \xrightarrow{\tau} \nu n.(Mn|Nn)}\ \text{close}$$

Fig. 5. The core π-calculus in λ-tree syntax.

$$\frac{P \xrightharpoonup{\downarrow x} M \qquad Q \xrightarrow{\uparrow xy} Q'}{P|Q \xrightarrow{\tau} (My)|Q'}\ \text{L-com} \qquad \frac{P \xrightarrow{\uparrow xy} P' \qquad Q \xrightharpoonup{\downarrow x} N}{P|Q \xrightarrow{\tau} P'|(Ny)}\ \text{L-com}$$

Fig. 6. The additional rules for late π-calculus.

$$\frac{}{\mathit{in}\ x\ M \xrightarrow{\downarrow xy} My}\ \text{E-input}$$

$$\frac{P \xrightarrow{\uparrow xy} P' \qquad Q \xrightarrow{\downarrow xy} Q'}{P|Q \xrightarrow{\tau} P'|Q'}\ \text{E-com} \qquad \frac{Q \xrightarrow{\downarrow xy} Q' \qquad P \xrightarrow{\uparrow xy} P'}{P|Q \xrightarrow{\tau} P'|Q'}\ \text{E-com}$$

Fig. 7. The additional rules for early π-calculus.

$$\begin{aligned}
sim_l\ (Pl)\ (Ql) \triangleq{}& \forall A \forall P' [\triangleright_l (Pl \xrightarrow{Al} P'l) \supset \exists Q' \triangleright_l (Ql \xrightarrow{Al} Q'l) \wedge \\
&\qquad\qquad\qquad\qquad\qquad sim_l\ (P'l)\ (Q'l)] \wedge \\
& \forall X \forall P' [\triangleright_l (Pl \xrightharpoonup{\downarrow(Xl)} P'l) \supset \exists Q' \triangleright_l (Ql \xrightharpoonup{\downarrow(Xl)} Q'l) \wedge \\
&\qquad\qquad\qquad\qquad\qquad \forall w\ sim_l\ (P'lw)\ (Q'lw)] \wedge \\
& \forall X \forall P' [\triangleright_l (Pl \xrightharpoonup{\uparrow(Xl)} P'l) \supset \exists Q' \triangleright_l (Ql \xrightharpoonup{\uparrow(Xl)} Q'l) \wedge \\
&\qquad\qquad\qquad\qquad\qquad sim_l\ (P'(\hat{\rho} l)(\rho l))\ (Q'(\hat{\rho} l)(\rho l))]
\end{aligned}$$

Fig. 8. Definitional clause for simulation of π-calculus

are trying to model. In the next section, we consider modeling simulation.

5.3 Simulation of π-expressions

For simplicity, we shall consider only simulation and not bisimulation: extending to bisimulation is not difficult (see Figure 1) but does introduce several more cases and make our examples more difficult to read.

Figure 8 presents a definitional clause for simulation. Here, the meta-logical predicate *sim* is of type $(evs \to proc) \to (evs \to proc) \to o$ and again, we abbreviate the expression $sim(\lambda l.Pl)(\lambda l.Ql)$ as $sim_l(Pl)(Ql)$. Here, X has type $evs \to name$, P has type $evs \to proc$, and P' has two different types, $evs \to proc$ and $evs \to name \to proc$. Since the only occurrence of ρ_σ is such that σ is *name*, we shall drop the subscript on ρ. Notice also that for this set of clauses to be successfully stratified, the level of *sim* must be strictly greater than the level of all the other predicates of the interpreter (which can all be equal).

Notice also that *sim* is a meta-level predicate while $\cdot \xrightarrow{\cdot} \cdot$ and $\cdot \xrightharpoonup{\cdot} \cdot$ are object-level predicates. This is a required separation since simulation needs to encompass the provability of these one-step translation relations. This is different from the encoding in [18] (Figure 1) since no universal judgments were needed to encoded CCS and hence the object/meta-level distinction was not needed.

The first conjunct in the body of the clause in Figure 8 deals with the case where a process makes either a τ step or a free input or output action. In these cases, the variable A would be bound to either $\lambda l.\tau$ (in the first case) or $\lambda l. \downarrow (N\ l)(M\ l)$ or $\lambda l. \uparrow (N\ l)(M\ l)$, in which cases, N and M would be of the form $\lambda l.\rho(\hat{\rho}^i l)$ for some non-negative integer i.

The last two cases correspond to when a bounded input or output action is done. In the case of the bounded input (the second conjunct), a universal quantifier of the meta-logic, $\forall w$, is used to instantiate the abstractions (P' and Q'), whereas in the bounded output case (the third conjunct), a universal quantifier of the object-level is emulated: such an internal quantifier is immediately replaced by using a new variable in the context, via the use of ρ and $\hat{\rho}$. This one definition clause thus illustrates important distinctions about meta-level and object-level: in particular, the observation that simulation is encoded as a meta-level predicate and not an object-level predicate (as with the one-step predicates) and that the universal quantifiers in both logics each have their applications and should not be confused.

5.4 Modal Logics for π-calculus

To further illustrate the ease of handling and encoding binding structures, we now encode the modal logic for the for π-calculus given in [30] (of necessity, we consider only binary conjunctions instead of general, indexed conjunctions). We first introduce the new type *assert* to denote assertion forms and then introduce the following constructors of this type: $\mathit{true} : \mathit{assert}$ for true, $\cdot\ \mathit{and}\ \cdot : \mathit{assert} \to \mathit{assert} \to \mathit{assert}$ for conjunction, $\mathit{not}\,\cdot : \mathit{assert} \to \mathit{assert}$ for negation, $\langle \cdot = \cdot \rangle \cdot : \mathit{name} \to \mathit{name} \to \mathit{assert} \to \mathit{assert}$ for the match modal, $\langle \cdot \rangle \cdot : \mathit{action} \to \mathit{assert} \to \mathit{assert}$ for the possibility modal for non-binding actions, and the following four modal operators used to encode the possibility of a bound actions: $\langle \downarrow \cdot \rangle \cdot, \langle \downarrow \cdot \rangle^L \cdot, \langle \downarrow \cdot \rangle^E \cdot, \langle \uparrow \cdot \rangle \cdot : \mathit{name} \to (\mathit{name} \to \mathit{assert}) \to \mathit{assert}$. The first three of these modals are used to code the "basic", "late", and "early" versions of the bounded input prefix while the forth encodes the bounded output action. Natural numbers are encoded as the type *nat* with constants $z : \mathit{nat}$ and $\mathit{succ} : \mathit{nat} \to \mathit{nat}$. The satisfaction relation is defined using two predicates: $\models$ at type $(\mathit{evs} \to \mathit{proc}) \to (\mathit{evs} \to \mathit{assert}) \to o$ as well as at the type $\mathit{nat} \to (\mathit{evs} \to \mathit{proc}) \to (\mathit{evs} \to \mathit{assert}) \to o$. This extra argument is used to help stratify this definition in the presence of negation in the assertion language. The predicate *depth* is of type $(\mathit{evs} \to \mathit{assert}) \to \mathit{nat} \to o$ and the expression $\mathit{depth}(\lambda l.Bl)\ N$ is abbreviated as simply $\mathit{depth}_l(Bl, N)$. This predicate holds if N is an upper bound on the nesting of negations in the (Bl) formula: this number is used to pick a suitable level to start the use of the stratified version of satisfaction. As we have done before, the expression $\models (\lambda l.Pl)\ (\lambda l.Bl)$ is abbreviated as $Pl \models_l Bl$ while the expression $\models^i (\lambda l.Pl)\ (\lambda l.Bl)$ is abbreviated as $Pl \models_l^i Bl$.

To properly stratify the definition in Figure 9, a slight generalization to the definition of levels for predicates needs to be made. In particular, we need to see the expression $\models^i$ as a predicate of level i and then give the predicate $\models$ the level ω. In other words, levels need to be generalized beyond finite ordinal.

6 Related and future work

Of course, the value of this approach to encoding the π-calculus comes, in part, from the ability to automate proofs using such definitions. For example,

$$Pl \models_l Bl \triangleq \exists i. depth_l(Bl, i) \wedge Pl \models_l^i Bl$$

$$\begin{aligned}
depth_l(true, N) &\triangleq \top \\
depth_l((B_1 l) \; and \; (B_2 l), N) &\triangleq depth_l(B_1 l, N) \wedge depth_l(B_2 l, N) \\
depth_l(not(Bl), succ\; N) &\triangleq depth_l(Bl, N) \\
depth_l(\langle Xl = Yl \rangle Bl, N) &\triangleq depth_l(Bl, N) \\
depth_l(\langle Al \rangle Bl, N) &\triangleq depth_l(Bl, N) \\
depth_l(\langle \downarrow Al \rangle Bl, N) &\triangleq depth_l(B(\hat{\rho} l)(\rho l), N) \\
depth_l(\langle \downarrow Al \rangle^E Bl, N) &\triangleq depth_l(B(\hat{\rho} l)(\rho l), N) \\
depth_l(\langle \downarrow Al \rangle^L Bl, N) &\triangleq depth_l(B(\hat{\rho} l)(\rho l), N) \\
depth_l(\langle \uparrow Al \rangle Bl, N) &\triangleq depth_l(B(\hat{\rho} l)(\rho l), N)
\end{aligned}$$

$$\begin{aligned}
Pl \models_l^i true &\triangleq \top \\
Pl \models_l^i (B_1 l) \; and \; (B_2 l) &\triangleq Pl \models_l^i B_1 l \wedge P \models_l^i B_2 l \\
Pl \models_l^{i+1} not(Bl) &\triangleq (Pl \models_l^i Bl) \supset \bot \\
Pl \models_l^i \langle Xl = Xl \rangle Bl &\triangleq P \models_l^i B \\
Pl \models_l^i \langle Al \rangle Bl &\triangleq \rhd_l Pl \xrightarrow{Al} P'l \wedge P'l \models_l^i Bl \\
Pl \models_l^i \langle \downarrow Al \rangle Bl &\triangleq \exists P' \exists z (\rhd_l Pl \xrightharpoonup{Al} P'l \wedge P'lz \models_l^i Blz) \\
Pl \models_l^i \langle \downarrow Al \rangle^E Bl &\triangleq \forall z \exists P' (\rhd_l Pl \xrightharpoonup{Al} P'l \wedge P'lz \models_l^i Blz) \\
Pl \models_l^i \langle \downarrow Al \rangle^L Bl &\triangleq \exists P' \forall z (\rhd_l Pl \xrightharpoonup{Al} P'l \wedge P'lz \models_l^i Blz) \\
Pl \models_l^i \langle \uparrow Al \rangle Bl &\triangleq \exists P' (\rhd_l Pl \xrightharpoonup{Al} P'l \wedge P'(\hat{\rho} l)(\rho l) \models_l^i B(\hat{\rho} l)(\rho l))
\end{aligned}$$

Fig. 9. Definition of a process satisfying an assert formula.

one would hope that the sequent

$$assert_l(Bl), \; Pl \models_l Bl, \; sim_l(Pl)(Ql) \longrightarrow Ql \models_l Bl,$$

would have a simple, natural proof (where the predicate $assert_l(\cdot)$ describes a subset of the modal logic that would correspond to simulation). Jeremie Wajs and the author are working on a tactic-style theorem prover for a logic with induction and definitions. This system, called Iris, is written entirely in Nadathur's Teyjus implementation [31] of λProlog and appears to be the first theorem proving system to be written entirely using higher-order abstract syntax (parser, printer, top-level, tactics, tacticals, etc). Example proofs that we have done by hand come out as expected: they are rather natural and immediate, although the encoding of eigenvariable context as a single abstraction makes expressions rather awkward to read. Fortunately, simple printing and parsing conventions can improve readability greatly. Given that the interpreter in Figure 4 is based on Horn clauses definitions, it is possible to employ well known induction principles for Horn clauses to help prove such properties.

There are various other calculi, such as the join and ambient calculi, in which names and name restriction are prominent and we plan to test this style of encoding with them. Generic judgments have been used to model names for references and exceptions [22,2] so it would be interesting to see if this style of encoding can adequately help in reasoning about programming language semantics containing such features. Comparing this particular encoding of the π-calculus with those of others, for example, [11,3,37], should be done in some detail.

Finally, the approach to encoding syntax and operational semantics used here is strongly motivated by proof theoretic considerations. There has been much work lately on using a more model-theoretic or categorical-theoretic approaches for such syntactic representations, see for example [4,5,10]. Comparing those two approaches should be quite illuminating.

Acknowledgements.
Catuscia Palamidessi has made a number of comments on a draft of this paper. Anonymous reviewers also made several corrections to the presentation of this paper. This work is funded in part by NSF grants CCR-9912387, CCR-9803971, INT-9815645, and INT-9815731.

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, February 1995.

[3] Jolle Despeyroux. A higher-order specification of the π-calculus. In *Proc. of the IFIP International Conference on Theoretical Computer Science, IFIP TCS'2000, Sendai, Japan, August 17-19, 2000.*, August 2000.

[4] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, 1999.

[5] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.

[6] Jean-Yves Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.

[7] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. i. Clauses as rules. *Journal of Logic and Computation*, pages 261–283, December 1990.

[8] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. ii. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, October 1991.

[9] John Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123–152, April 1993.

[10] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.

[11] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[12] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[13] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, March 1987.

[14] Raymond McDowell. *Reasoning in a Logic with Definitions and Induction.* PhD thesis, University of Pennsylvania, December 1997.

[15] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. To appear in the ACM Transactions on Computational Logic.

[16] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.

[17] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[18] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 197(1-2), 2001. To appear.

[19] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[20] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[21] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.

[22] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, September 1996.

[23] Dale Miller. Abstract sybtax for variable binders: An overview. In John Lloyd and et. al., editors, *Computational Logic - CL 2000*, Springer LNAI 1861, pp. 239–253 (2000).

[24] Dale Miller and Gopalan Nadathur. A computational logic approach to syntax and semantics. Presented at the Tenth Symposium of the Mathematical Foundations of Computer Science, IBM Japan, May 1985.

[25] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255. Association for Computational Linguistics, Morristown, New Jersey, 1986.

[26] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[27] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[28] Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. In Pierpaolo Degano, Roberto Gorrieri, Alberto Marchetti-Spaccamela, and Peter Wegner, editors, *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, volume 31. ACM, Sep 1999.

[29] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.

[30] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[31] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[32] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *LICS91*, pages 342–349. IEEE, July 1991.

[33] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *LICS93*, pages 64–74. IEEE, June 1993.

[34] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.

[35] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[36] G. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[37] C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In *Proceedings of FOSSACS'01*, LNCS, 2001.

[38] Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

[39] Peter Schroeder-Heister. Cut-elimination in logics with definitional reflection. In D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, volume 619 of *Lecture Notes in Computer Science*, pages 146–171. Springer-Verlag, 1992.

[40] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.