Encoding Generic Judgments [⋆]

Dale Miller[1] and Alwen Tiu[2]

[1] INRIA-FUTURS and Ècole Polytechnique
dale.miller@inria.fr,
[2] Pennsylvania State University and Ècole Polytechnique
tiu@cse.psu.edu

Abstract. The operational semantics of a computation system is often presented as inference rules or, equivalently, as logical theories. Specifications can be made more declarative and high-level if syntactic details concerning bound variables and substitutions are encoded directly into the logic using term-level abstractions (λ-abstraction) and proof-level abstractions (eigenvariables). When one wishes to reason about relations defined using term-level abstractions, *generic judgment* are generally required. Care must be taken, however, so that generic judgments are not uniformly handled using proof-level abstractions. Instead, we present a technique for encoding generic judgments and show two examples where generic judgments need to be treated at the term level: one example is an interpreter for Horn clauses extended with universal quantified bodies and the other example is that of the π-calculus.

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 [Plo81]) or big-step approach (a.k.a. natural semantics [Kah87]). In either case, algebraic (first-order) terms can be used to encode the language being specified and the first-order theory of Horn can be used to formalize and interpret such semantic specifications. For example, consider the following natural semantics specification of a conditional expression for a functional programming language:

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

These two inference figures can also be seen as 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.

[⋆] An earlier draft of this paper appeared in MERLIN 2001 [Mil01].

Of course, once a specification is made, one might want to reason about it. For example, 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. Hallnäs and Schroeder-Heister [HSH91,SH93], Girard [Gir92], and more recently, McDowell and Miller [MM97,MM00] have all considered just such introduction rules for non-logical constants, using a notion of *definition*.

2 A proof theoretic notion of definitions

A *definition* is a finite collection of definition *clauses* of the form $\forall \bar{x}[H \triangleq B]$, 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 definition clauses. The atomic formula H is called the *head* of the clause, and the formula B is called the *body*. The symbol $\triangleq$ is used simply to indicate a definition 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. Let H, B, and $\bar{x}$ be restricted as above. If we assume further that the only logical connectives that may occur in B are $\exists$, $\top$, and $\wedge$, then we say that the formula $\forall \bar{x}[B \supset H]$ is a *Horn clause* and the definition clause $\forall \bar{x}[H \triangleq B]$ is a *Horn definition clause*.

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 $\forall \bar{x}[H \triangleq B]$ in the given definition such that A is equal to $H\theta$. The left-introduction rule is

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

where the variables $\bar{x}$ are chosen (via α-conversion) to be distinct from the (eigen)variables free in the lower sequent of the rule. The set $\mathit{CSU}(A, H)$ 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 [Hue75].) There are many important situations where CSUs are not only finite but are also singleton (containing a most general unifier) whenever terms are unifiable: in particular, first-order unification and the *higher-order pattern* unification [Mil91a], where the application of functional variables are restricted to distinct bound variables.

We must also restrict the use of implication in the bodies of definition clauses, otherwise cut-elimination does not hold [SH92]. 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:

1. $\mathrm{lvl}(p\,\bar{t}) = \mathrm{lvl}(p)$
2. $\mathrm{lvl}(\bot) = \mathrm{lvl}(\top) = 0$
3. $\mathrm{lvl}(B \wedge C) = \mathrm{lvl}(B \vee C) = \max(\mathrm{lvl}(B), \mathrm{lvl}(C))$
4. $\mathrm{lvl}(B \supset C) = \max(\mathrm{lvl}(B) + 1, \mathrm{lvl}(C))$
5. $\mathrm{lvl}(\forall x.B) = \mathrm{lvl}(\exists x.B) = \mathrm{lvl}(B)$.

We now require that for every definition clause $\forall \bar{x}[p\,\bar{t} \overset{\triangle}{=} B]$, $\mathrm{lvl}(B) \leq \mathrm{lvl}(p)$. (If a definition is restricted to the Horn case, then this restriction is trivial to satisfy.) Cut-elimination for this use of definition within intuitionistic logic was proved in [McD97,MM00]. In fact, that proof was for a logic that also included a formulation of induction.

Definitions are a way to introduce logical equivalences so that we do not introduce into proof search meaningless cycles: for example, if our specification contains the equivalence $H \equiv B$, 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 our first example sequent above. We first convert the two Horn clauses representing the evaluation rules for the conditional into the following definition clauses.

$$\begin{array}{l}(\mathit{if}\ B\ M\ N) \Downarrow V \overset{\triangle}{=} B \Downarrow \mathit{true} \wedge M \Downarrow V. \\ (\mathit{if}\ B\ M\ N) \Downarrow V \overset{\triangle}{=} B \Downarrow \mathit{false} \wedge N \Downarrow V.\end{array}$$

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 [MMP01], 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 labeled transition systems. In particular, assume that $P \xrightarrow{A} P'$ is defined via clauses to which are added the two

$$\mathit{sim}\ P\ Q \triangleq \forall A \forall P'.\ P \xrightarrow{A} P' \supset \exists Q'.\ Q \xrightarrow{A} Q' \wedge \mathit{sim}\ P'\ Q'$$

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

Fig. 1. Simulation and bisimulation as definitions.

clauses in Figure 1. These two clauses are a direct encoding of the closure conditions for simulation and bisimulation. (To satisfy the restrictions on levels, the level given to the predicate $\cdot \xrightarrow{\cdot} \cdot$ must be less than the level given to either predicate *sim* or *bisim*.) In [MMP01] 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 $\mathit{bisim}\ P\ Q$. Since provability characterizes the least fixed point and simulation and bisimulation are characterized using the greatest fixed point, the restriction to noetherian transition systems is necessary since noetherian guarantees that these two fixed points are the same.

In Section 4 we show how we can capture simulation and bisimulation for the (finite) π-calculus in a similar style.

3 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.

3.1 Syntactic representation of abstractions

The encoding style called *higher-order abstract syntax* [PE88] views such abstractions as functional expressions that rely 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 of HOAS and many earlier papers have appeared exploiting this style of syntactic representation [MN85,MN86,MN87,Pau89]. 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* [MP99,Mil00] 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 notion of equality gives rise to L_λ unification (also, higher-order pattern unification) [Mil91a,Nip91], which is decidable and unary. The relationship between the λ-tree approach where abstractions are applied to only internally 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" [San96] and the full π-calculus, where "external mobility" is also allowed (via general substitutions). In Section 4, we will see that this comparison is not accidental.

3.2 When no object-level abstractions are present

To help illustrate the special needs of reasoning about syntax that contains abstractions, we start with an example in which such abstractions are not present.

Consider the specification of an interpreter for object-level Horn clauses. We shall use the type o to denote meta-level logical expressions and obj to denote object-level logical expressions. The meta-logic denotes universal and existential quantification at type σ as $\forall_\sigma$ and $\exists_\sigma$ (both of type $(\sigma \to o) \to o$) and denotes conjunction and disjunction as $\wedge$ and $\supset$ (both of type $o \to o \to o$). The object-logic denotes universal and existential quantification by $\hat{\forall}_\sigma$ and $\hat{\exists}_\sigma$ (both of type $(\sigma \to obj) \to obj$), conjunction and implication by & and $\Rightarrow$ (both of type $obj \to obj \to obj$), and truth by $\hat{\top}$ (of type obj). Following usual conventions: type subscripts on quantifiers will often be dropped if they can be easily inferred or are not important, and if a quantifier is followed by a lambda abstraction, the lambda is dropped; eg, $\hat{\forall}\lambda x$ will be abbreviated as simply $\hat{\forall} x$.

To encode the provability relation for the object-logic, we exploit the completeness of goal-directed proofs for our object-logic [Mil90]. Provability can be specified as an interpreter using the following four (meta-level) predicates: provability is denoted by $\triangleright$ and has type $obj \to o$, backchaining is denoted by the infix symbol $\triangleleft$ and has type $obj \to obj \to o$, *atomic* of type $obj \to o$ decides if an object-level formula is atomic, and *prog*, also of type $obj \to o$, decides if a formula is an object-level assumption (object-level logic program). The definition in Figure 2 is an interpreter for object-level Horn clause.

Completing the specification of the interpreter requires additional definition 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 $tm \to tm \to obj$, where tm is the type of the programming language for which evaluation is being specified. It is possible to now prove the sequent

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

using the definition clauses in Figures 2 and 3. In fact, the proof of this sequent follows the same structure as the proof of this sequent when it is considered at the meta-level only (as in Section 2). That is, although the proof using the meta-level/object-level distinctions is more complex and detailed, no interesting information is uncovered by these additional complexities. The reason to present

$$\begin{array}{rlrl}
\rhd \hat{\top} & \stackrel{\triangle}{=} \top. & A \lhd A & \stackrel{\triangle}{=} \textit{atomic } A. \\
\rhd(G \mathbin{\&} G') & \stackrel{\triangle}{=} \rhd G \wedge \rhd G'. & (G \Rightarrow D) \lhd A & \stackrel{\triangle}{=} D \lhd A \wedge \rhd G. \\
\rhd(\hat{\exists}_\sigma\, G) & \stackrel{\triangle}{=} \exists_\sigma x. \rhd(Gx) & (\hat{\forall}_\sigma\, D) \lhd A & \stackrel{\triangle}{=} \exists_\sigma t.\ (D\ t \lhd A).
\end{array}$$

$$\rhd A \stackrel{\triangle}{=} \exists D(\textit{atomic } A \wedge \textit{prog } D \wedge D \lhd A).$$

Fig. 2. Interpreter for object-level specifications.

$$\begin{array}{rl}
\textit{atomic } (M \Downarrow V) & \stackrel{\triangle}{=} \top. \\
\textit{prog } (\hat{\forall} B\, \hat{\forall} M\, \hat{\forall} N\, \hat{\forall} V [B \Downarrow \textit{true} \mathbin{\&} M \Downarrow V \Rightarrow (\textit{if } B\ M\ N) \Downarrow V]) & \stackrel{\triangle}{=} \top. \\
\textit{prog } (\hat{\forall} B\, \hat{\forall} M\, \hat{\forall} N\, \hat{\forall} V [B \Downarrow \textit{false} \mathbin{\&} N \Downarrow V \Rightarrow (\textit{if } B\ M\ N) \Downarrow V]) & \stackrel{\triangle}{=} \top.
\end{array}$$

Fig. 3. Examples of object-level Horn clauses.

this interpreter here is so that we may elaborate it in the next section to help capture reasoning about generic judgments.

3.3 Generic judgments as atomic meta-level judgments

When using HOAS or λ-tree syntax representations, inference rules of the form $\dfrac{\hat{\forall}\, x.Gx}{A}$ 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 obvious such interpretation is via the clause

$$\rhd(\hat{\underset{\sigma}{\forall}}\, x.G\ x) \stackrel{\triangle}{=} \forall_\sigma x[\rhd G\ x], \tag{1}$$

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

$$\rhd(\hat{\forall}\, y_1\, \hat{\forall}\, y_2 [q\ \langle y_1, t_1\rangle\ \langle y_2, t_2\rangle\ \langle y_2, t_3\rangle]) \longrightarrow t_2 = t_3,$$

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:

$$\begin{array}{c}
X = X \stackrel{\triangle}{=} \top \\
\textit{atomic } (q\ X\ Y\ Z) \stackrel{\triangle}{=} \top \\
\textit{prog } (\hat{\forall} X\, \hat{\forall} Y\ q\ X\ X\ Y) \stackrel{\triangle}{=} \top \\
\textit{prog } (\hat{\forall} X\, \hat{\forall} Y\ q\ X\ Y\ X) \stackrel{\triangle}{=} \top \\
\textit{prog } (\hat{\forall} X\, \hat{\forall} Y\ q\ Y\ X\ X) \stackrel{\triangle}{=} \top
\end{array}$$

Using these definitional clauses, this sequent reduces to

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

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 $\forall y_1 \forall y_2$ contains at least two distinct elements.

Additionally, if we consider a broader class of computational systems, other than conventional logical systems, a similar issue appears when we try to encode their term-level abstractions at the meta-level. For systems where names can be given scope and can be compared (the π-calculus, for example), interpreting object-level abstraction via universal quantifier may not be appropriate in certain cases. In the case of π-calculus, consider encoding the one-step transition semantics for the restriction operator:

$$(x)Px \xrightarrow{A} (x)P'x \triangleq \forall x.Px \xrightarrow{A} P'x.$$

If the atomic judgment $(x)Px \xrightarrow{A} (x)P'x$ is provable, then the meta-level atomic formula $Pt \xrightarrow{A} P't$ is provable for all names t. This is certainly not true in the present of match or mismatch operator in P. (We return to the π-calculus in more detail in Section 4.)

For these reasons, the uniform analysis of an object-level universal quantifier with a universal quantifier in (1) should be judged inappropriate. We now look for a different approach for encoding object logics.

A universal formula can be proved by an inference rule such as

$$\frac{\Gamma, c : \sigma \vdash Pc}{\Gamma \vdash \hat{\forall}_\sigma\, x.Px}\ ,$$

where P is a variable of higher type, Γ is a set of distinct typed variables, and c is a variable that is not free in the Γ nor in $\hat{\forall}_\sigma\, x.Px$. When we map the judgment $\Gamma \vdash \hat{\forall}_\sigma\, x.Px$ into the meta-logic, the entire judgment will be seen as an atomic meta-level formula: that is, we do not encode just $\rhd(\hat{\forall}_\sigma\, x.Px)$ but rather the entire judgment $\rhd(\Gamma \vdash \hat{\forall}_\sigma\, x.Px)$.

We consider two encodings of the judgment $x_1, \ldots, x_n \vdash (Px_1 \ldots x_n)$, where the variables on the left are all distinct. The first encoding introduces a "local" binders using a family of constants, say, loc_σ of type $(\sigma \rightarrow obj) \rightarrow obj$. The above expression would be something of the form

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

where, for $i = 1, \ldots, n$, σ_i is the type of the variable x_i. 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, does not directly access that top-level

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

Fig. 4. An interpreter for simple generic judgments.

structure. The second alternative employs a coding technique used by McDowell [McD97,MM02]. 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.

3.4 An interpreter for generic judgments

To illustrate this style encoding of generic judgments, consider the interpreter displayed in Figure 4. This interpreter specifies provability for object-level Horn clauses in which the body of clauses may have occurrences of the universal quantifiers (as well as true, conjunction, and the existential quantifier) and generalizes the previous interpreter (Figure 2). 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 $(evs \to obj) \to (evs \to obj) \to o$. The *evs* abstractions in these predicates are abbreviated as the subscript l next to the predicates, so the expression $\rhd((\lambda l.G\ l)\ \&\ (\lambda l.G'\ l))$ is written simply as $\rhd_l((G\ l)\ \&\ (G'\ l))$. Notice that the technique of replacing the abstraction $\lambda l.\,\hat{\forall}_\sigma\, \lambda w.(G\ l\ w))$ with $\lambda l.G(\hat{\rho}l)(\rho l))$ corresponds to replacing the judgment $x_1, \ldots, x_n \vdash \forall y(Pyx_1 \ldots x_n)$ with $x_1, \ldots, x_n, x_{n+1} \vdash (Px_1 \ldots x_n x_{n+1})$.

Notice that proof search using this interpreter will generate more than L_λ unification problems (via the left and right definition rules) for two reasons. First, the definition clause for interpreting $(\lambda l.\,\hat{\forall}_\sigma\, w.(G\ l\ w))$ 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. However, the technical definition of L_λ can be extended to allow for this style of encoding of object-level variables, while still preserving the decidability and the mgu property of the unification problems [Tiu02]. A second reason that this interpreter is not in L_λ is the definition clause for backchaining over $(\lambda l.\,\hat{\forall}_\sigma\, w.(D\ l\ w))$ 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 [Mil91b]. 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.

See [MM02] for the correctness proofs of this encoding of generic judgments. Other judgments besides generic judgments can be encoded similarly. For example, in [McD97,MM02], 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.

4 The π-calculus

To illustrate the use of this style of representation of universal judgments, we turn, as many others have done [MP99,HMS01,Des00,RHB01], to consider encoding the π-calculus. In particular, we follow the encoding in [MP99] for the syntax and one-step operational semantics.

Following the presentation of the π-calculus given in [MPW92], we shall require 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 $|$ 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 [MMP01] 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 1. *Let P be a finite π-calculus expression using the syntax of [MPW92]. If the free names of P are admitted as constants of type* name *then P corresponds uniquely to a $\beta\eta$-equivalence class of terms of type proc.*

We first consider the encoding of one-step transition semantics of π-calculus. As we did with the encoding of object logics, we explicitly encode the "signatures" of π-calculus expressions (i.e., the names) as abstractions of type *evs*. The encoding uses two predicates: $\cdot \overset{\cdot}{\longrightarrow} \cdot$ of type $(\mathit{evs} \to \mathit{proc}) \to (\mathit{evs} \to \mathit{action}) \to$

$(evs \to proc) \to o$; and $\cdot \xrightharpoonup{\cdot} \cdot$ of type $(evs \to proc) \to (evs \to name \to action) \to (evs \to name \to proc) \to o$. The first of these predicates encodes transitions involving free values and the second encodes transitions involving bound values.

Figures 5, 6 and 7 present the transition semantics as Horn specification. We omit the *evs* abstraction for simplicity of presentation. We shall see later how to translate this specification into Horn definition clauses. 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.

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

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

$$\dfrac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q}\,\text{par} \qquad \dfrac{Q \xrightarrow{A} Q'}{P|Q \xrightarrow{A} P|Q'}\,\text{par}$$

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

$$\dfrac{\forall n(Pn \xrightarrow{A} P'n)}{\nu n.Pn \xrightarrow{A} \nu n.P'n}\,\text{res} \qquad \dfrac{\forall n(Pn \xrightharpoonup{A} P'n)}{\nu n.Pn \xrightharpoonup{A} \lambda m\ \nu n.(P'nm)}\,\text{res}$$

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

$$\dfrac{\forall y(My \xrightarrow{\uparrow xy} M'y)}{\nu y.My \xrightharpoonup{\uparrow x} M'}\,\text{open}$$

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

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

Sangiorgi has also motivated the same core system of rules [San96] and named them π_I: this subset of the π-calculus allows for "internal" mobility of names, that is, local (restricted) names only being passed to abstractions (via β_0 reductions), and disallows "external" mobility (via the more general β reduction rules).

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

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.

One advantage of this style of specification over the traditional one [MPW92] is the absence of complicated side-conditions on variables: they are handled directly by the treatment of abstractions by logic.

Universally quantified premises, of which there are three in Figure 5, is one of the logical features that aids in encoding abstractions. When the inference rules in these figures are written as formulas, cut-free provability of the judgments $P \xrightarrow{A} Q$ and $P \xrightharpoonup{A} Q$ corresponds to one-step transitions for the π-calculus. While the right-introduction rule for the universal quantifiers that appears in these premises is correct, the corresponding left-introduction rule (which does not appear in cut-free proofs of just one-step transitions) for universal quantifiers is not always appropriate. It is the left-hand introduction rules for universal quantifiers that allow for arbitrary substitutions for abstractions, and, as suggested in Section 3.3, this property is certainly undesirable if we wish to extend our specification of the π-calculus, for example, to include the mismatch operator. We can also encode these Horn clauses as *prog* clauses in an object-logic interpreter like the one we defined in Section 3.4. This style of encoding inherits similar substitution properties of the object-logic.

One way to address this problem of the universal quantifier is to translate these inference rules into Horn definitions using the technique described in the previous section. We first need to make explicit the object-level eigenvariables by writing down the *evs* abstractions. Then, it is almost straightforward to rewrite these inference rule into Horn definitions, except for the cases involving the restriction operator which we will show here for the rules res and open.

$$\begin{array}{ll} \text{res:} & \nu n.Pln \xrightarrow{Al}_l \nu n.P'ln \triangleq P(\hat\rho l)(\rho l) \xrightarrow{A(\hat\rho l)}_l P'(\hat\rho l)(\rho l) \\ \text{open:} & \nu y.Mly \xrightarrow{\uparrow(xl)}_l M'l \triangleq M(\hat\rho l)(\rho l) \xrightarrow{\uparrow(x(\hat\rho l))(\rho l)}_l M'(\rho l) \end{array}$$

$$\begin{aligned}
\mathit{sim}_l\ (Pl)\ (Ql) \triangleq\ & \forall A \forall P' [(Pl \xrightarrow{Al}_l P'l) \supset \exists Q'\, (Ql \xrightarrow{Al}_l Q'l) \wedge \\
& \qquad\qquad\qquad\qquad\qquad \mathit{sim}_l\ (P'l)\ (Q'l)] \wedge \\
& \forall X \forall P' [(Pl \xrightarrow{\downarrow(Xl)}_l P'l) \supset \exists Q'\, (Ql \xrightarrow{\downarrow(Xl)}_l Q'l) \wedge \\
& \qquad\qquad\qquad\qquad\qquad \forall w\ \mathit{sim}_l\ (P'l(wl))\ (Q'l(wl))] \wedge \\
& \forall X \forall P' [(Pl \xrightarrow{\uparrow(Xl)}_l P'l) \supset \exists Q'\, (Ql \xrightarrow{\uparrow(Xl)}_l Q'l) \wedge \\
& \qquad\qquad\qquad\qquad\qquad \mathit{sim}_l\ (P'(\hat{\rho}l)(\rho l))\ (Q'(\hat{\rho}l)(\rho l))]
\end{aligned}$$

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

Here, the expression $\lambda l.Pl \xrightarrow{\lambda l.Al}_l \lambda l.P'l$ is abbreviated as $Pl \xrightarrow{Al}_l P'l$.

Proposition 2. *Let $P \xrightarrow{A} Q$ be provable in late (resp., early) transition system of [MPW92]. If A is either τ or $\downarrow xy$ or $\uparrow xy$ then $Pl \xrightarrow{Al}_l Ql$ is provable from the definition clauses encoding late (resp., early) transitions. (Here, P, A, and Q are all translated to the corresponding logic-level term.) If A is $x(y)$ then $Pl \xrightarrow{\downarrow(xl)}_l Rl$ and if A is $\bar{x}(y)$ then $Pl \xrightarrow{\uparrow(xl)}_l Rl$. Here, R is the representation of the λ-abstraction of y over Q.*

Since the types of P and P' are different in the expression $P \xrightarrow{A} P'$, 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 are trying to model. To illustrate two such choices, we now consider encoding simulation.

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 definition clause for simulation. Here, the predicate *sim* is of type $(\mathit{evs} \to \mathit{proc}) \to (\mathit{evs} \to \mathit{proc}) \to o$ and again, we abbreviate the expression $\mathit{sim}\ (\lambda l.Pl)\ (\lambda l.Ql)$ as $\mathit{sim}_l(Pl)(Ql)$. Here, X has type $\mathit{evs} \to \mathit{name}$, P has type $\mathit{evs} \to \mathit{proc}$, and P' has two different types, $\mathit{evs} \to \mathit{proc}$ and $\mathit{evs} \to \mathit{name} \to \mathit{proc}$. 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 one-step transitions (which can all be equal).

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, $\forall w$, is used to instantiate the abstractions (P' and Q'), whereas in the bounded output case (the third conjunct), a term-level abstraction is emulated: such an internal abstraction 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 an important distinction between term-level and proof-level abstractions: in particular, they reflect the different behaviors of name-binding constructs in input prefix and restriction operator of π-calculus.

5 Related and future work

Of course, the value of this approach to encoding object-logics and the π-calculus comes, in part, from the ability to automate proofs using such definitions. Jeremie Wajs and Miller have been developing a tactic-style theorem prover for a logic with induction and definitions [Waj02]. This system, called Iris, is written entirely in Nadathur's Teyjus implementation [NM99] 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 object-level signature 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 properties of the object logics/systems.

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 [Mil96,Chi95] and it would be interesting to see if this style of encoding can help in reasoning about programming language semantics containing such features. Comparing this particular encoding of the π-calculus with those of others, for example, [HMS01,Des00,RHB01], 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 [FPT99,GP99,Hof99]. Comparing those two approaches should be illuminating.

Acknowledgments. Catuscia Palamidessi has made a number of comments on a draft of this paper. Anonymous reviewers for MERLIN 2001 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

[Chi95] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, February 1995.

[Des00] 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.

[FPT99] 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.

[Gir92] Jean-Yves Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.

[GP99] 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.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)inductive type theory. *TCS*, 253(2):239–285, 2001.

[Hof99] 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.

[HSH91] 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.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *TCS*, 1:27–57, 1975.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. March 1987.

[McD97] Raymond McDowell. *Reasoning in a Logic with Definitions and Induction.* PhD thesis, University of Pennsylvania, December 1997.

[Mil90] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[Mil91a] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil91b] 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.

[Mil96] Dale Miller. Forum: A multiple-conclusion specification language. *TCS*, 165(1):201–232, September 1996.

[Mil00] Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.

[Mil01] Dale Miller. Encoding generic judgments: Preliminary results. In R.L. Crole S.J. Ambler and A. Momigliano, editors, *ENTCS*, volume 58. Elsevier, 2001. Proceedings of the MERLIN 2001 Workshop, Siena.

[MM97] 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.

[MM00] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *TCS*, 232:91–119, 2000.

[MM02] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[MMP01] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *TCS*, 197(1-2), 2001. To appear.

[MN85] 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.

[MN86] 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.

[MN87] 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.

[MP99] 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.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *LICS91*, pages 342–349. IEEE, July 1991.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, ed., *CADE16*, pages 287–291, Trento, Italy, July 1999. LNCS.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[Plo81] G. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[RHB01] 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.

[San96] Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *TCS*, 167(2):235–274, 1996.

[SH92] 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 *LNCS*, pages 146–171. Springer, 1992.

[SH93] 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.

[Tiu02] Alwen F. Tiu. An extension of L-lambda unification. Draft, available via `http://www.cse.psu.edu/~tiu/llambdaext.pdf`, September 2002.

[Waj02] Jérémie D. Wajs. *Reasoning about Logic Programs Using Definitions and Induction.* PhD thesis, Pennsylvania State University, 2002.