A Proof Theory for Generic Judgments: An extended abstract

Dale Miller
INRIA/Futurs/Saclay & École polytechnique
dale.miller@inria.fr

Alwen Tiu
École polytechnique & Penn State University
tiu@cse.psu.edu

Abstract

A powerful and declarative means of specifying computations containing abstractions involves meta-level, universally quantified generic judgments. *We present a proof theory for such judgments in which signatures are associated to each sequent (used to account for eigenvariables of the sequent) and to each formula in the sequent (used to account for generic variables locally scoped over the formula). A new quantifier, ∇, is introduced to explicitly manipulate the local signature. Intuitionistic logic extended with ∇ satisfies cut-elimination even when the logic is additionally strengthened with a proof theoretic notion of definitions. The resulting logic can be used to encode naturally a number of examples involving name abstractions, and we illustrate using the π-calculus and the encoding of object-level provability.*

Keywords: proof search, reasoning about operational semantics, generic judgments, higher-order abstract syntax.

1. Eigenvariables and generic reasoning

In specifying and reasoning about computations involving abstractions, one needs to encode both the static structure of such abstractions and their dynamic structure during computation. One successful approach to such an encoding, generally called *higher-order abstract syntax* [22], uses λ-terms to encode the static structure of abstractions and universally quantified judgments to encode their dynamic structure.

There are, of course, several ways to prove a universally quantified expression, $\forall_\gamma x.B$. An approach that can be called the *extensional*, attempts to prove $B[t/x]$ for all (closed) terms t of type γ. This rule might involve an infinite number of premises if the domain of the type γ is infinite. If the type γ is defined inductively, a proof by *induction* can replace the need for infinite premises with finite premises (the *base* cases and *inductive* cases) but with the need to discover invariants. Another more *intensional* approach, however, involves introducing a new, generic variable, say, $c : \gamma$, that has not been introduced before in the proof, and to prove the formula $B[c/x]$ instead. In natural deduction and sequent calculus proofs, such new variables are called *eigenvariables*.

In Gentzen's original presentation of the sequent calculus [5], eigenvariables were immutable: reading proofs bottom-up, once an eigenvariable is introduced it is not used as a site for substitution. In other words, Gentzen's eigenvariables did not vary in proof construction: rather they acted more as fresh, scoped constants.

The generic interpretation of quantifiers generally entails the extensional interpretation: this is a simple consequence of the cut-elimination theorem as follows. Assume that the sequent $\Gamma \longrightarrow \forall x.B$ is proved using the introduction of $\forall$ on the right from the premise $\Gamma \longrightarrow B[c/x]$, where c is an eigenvariable and $\Pi(c)$ is a proof of this premise. Similarly, assume that the sequent $\Gamma', \forall x B \longrightarrow C$ is proved using the introduction of $\forall$ on the left from the premise $\Gamma', B[t/x] \longrightarrow C$, where t is some term. To reduce the rank of the cut formula $\forall x.B$ between the sequents $\Gamma \longrightarrow \forall x.B$ and $\Gamma', \forall x B \longrightarrow C$, the eigenvariable c in the sequent calculus proof $\Pi(c)$ must be substituted by t to yield a proof $\Pi(t)$ of $\Gamma \longrightarrow B[t/x]$: in this way, the cut-formula is now the smaller formula $B[t/x]$. In Gentzen, this role of c in $\Pi(c)$ as a site for substitution only takes place in the meta-theory of proofs and not in proofs themselves.

Recent years have witnessed two different developments in the role of eigenvariables in the specification of computation systems.

Eigenvariables as fresh, scoped constants Focusing on their *intensional* nature and guarantee of newness or freshness in proof search, eigenvariables have been used to encode name restrictions in the π-calculus [15], nonces in security protocols [1], reference locations in imperative programming [2, 16], and constructors hidden within abstract data-types [12]. Eigenvariables also provide an essential

$$\frac{\Sigma : (\sigma, y : \gamma) \triangleright B[y/x], \Gamma \longrightarrow \mathcal{C}}{\Sigma : \sigma \triangleright \nabla_\gamma x.B, \Gamma \longrightarrow \mathcal{C}} \ \nabla\mathcal{L}$$

$$\frac{\Sigma : \Gamma \longrightarrow (\sigma, y : \gamma) \triangleright B[y/x]}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \nabla_\gamma x.B} \ \nabla\mathcal{R}$$

Figure 1. Rules for the ∇-quantifier.

aspect of recursive programming with data encoded using higher-order abstract syntax. In this role, eigenvariables are essentially constants, scoped over part of a computation.

Eigenvariables as variables to instantiate Computation in logic programming can be seen as a (restricted) form of cut-free proof search. Cut and cut-elimination can then be used to reason directly about computation: for example, if A has a cut-free proof (that is, it can be computed) and we know that $A \supset B$ can be proved (possibly with cuts), cut-elimination allows us to conclude that B has a cut-free proof (that is, it can be computed). As we mentioned above, such direct reasoning on logic specification involves instantiations of eigenvariables. Similarly, focusing on their *extensional* nature guaranteed by cut-elimination, enrichments to the sequent calculus have been proposed by [7, 24, 6, 9] in which eigenvariables are intended as variables to be substituted. This enrichment to proof theory (discussed here in Section 4) holds promise for providing proof systems for the direct reasoning of logic specifications (see, for example, the above mentioned papers as well as [10, 11]).

These two approaches are, however, at odds with each other. Consider, for example, the problem of representing restriction of names or nonces using $\forall$ quantification. (The following example can be dualized in the event that a logical specification uses $\exists$ quantification instead of $\forall$, as in, for example, [1]). One can imagine that a proof of the expression $\forall x \forall y.P(x, y)$ involves two different fresh "names" or "nonces" whereas a proof of the expression $\forall z.P(z, z)$ involves just one such item. Of course, in logic, the implication $\forall x \forall y.P(x, y) \supset \forall z.P(z, z)$ holds, so if there is a proof with the two different names, there must be one with those names identified (via cut-elimination), and this is unlikely to be the intended meaning of such quantification. This suggests that when using eigenvariables solely to provide scope and freshness to names, one cannot reason directly with the specification using the center piece of proof theory: cut-elimination.

2. The ∇-quantifier

One approach to solving this problem of forcing one connective, the $\forall$-quantifier, to have two behaviors that are not entirely compatible, is to extend the logic with a new quantifier. In this paper, we do this by adding the ∇-quantifier: its role will be to declare variables to be new and of local scope. The syntax of the formula $\nabla_\gamma x.B$ is like that for the universal and existential quantifiers. Following Church's approach with the Simple Theory of Types [3] formulas are given the type o, and for all types γ not containing o, ∇_γ is a constant of type $(\gamma \to o) \to o$. The expression $\nabla_\gamma \lambda x.B$ is usually abbreviated as simply $\nabla_\gamma x.B$ or as $\nabla x.B$ if the type information is either simple to infer or not important.

Intuitionistic sequents without the need to account for ∇ are structures of the form

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

Here, Σ is a *signature* containing the list of all (explicitly typed) eigenvariables of the sequent. We write $\Sigma \vdash t : \gamma$ to denote that t is a simply typed λ-term of type γ in which there may appear the (fixed) logical and non-logical constants as well as those eigenvariables in Σ. We shall also say t is a Σ-term (of type γ), and, if γ is o, t is a Σ-formula. In the displayed sequent above, $n \geq 0$ and $B_0, B_1, \ldots, B_n$ are Σ-formulas. Informally, the "extensional" reading of this sequent would be that for every substitution θ that maps a variable $x : \gamma \in \Sigma$ to a term of type γ, if $B_i\theta$ holds for all $i = 1, \ldots, n$, then $B_0\theta$ holds.

To account for this new quantifier, we introduce into sequents a new element of context. Sequents will now have one *global* signature (containing the sequent's eigenvariables) and several *local* signatures, used to scope locally fresh variables. More generally, sequents have the structure

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

Here, σ_i, for $i = 0, \ldots, n$ are signatures and the other items are as above. We shall consider sequents to be binding structures in the sense that the signatures, both the global and local ones, are abstractions over their respective scopes. The variables in Σ and σ_i will admit α-conversion by systematically changing the names of variables in signatures as well as those in their scope, following the usual convention of the λ-calculus. In general, however, we will assume that the local signatures σ_i contain names different than those in the global signature Σ. The expression $\sigma \triangleright B$ is called a *generic judgment* or simply *judgment*. We use script letters $\mathcal{A}$, $\mathcal{B}$, etc. to denote judgments. We write simply B instead of $\sigma \triangleright B$ if the signature σ is empty.

The introduction rules for ∇ are given in Figure 1. The variable y must be new to the variables in σ and Σ (implicit in the definition of sequent). The expression $(\sigma, y : \gamma)$ denotes the signature containing the type declaration $y : \gamma$ appended to the end of the list σ. Notice that since the left and right rules are essentially the same, this quantifier will be self dual.

$$\frac{\Sigma : \sigma \triangleright B, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B \wedge C, \Gamma \longrightarrow \mathcal{D}} \wedge\mathcal{L} \qquad \frac{\Sigma : \sigma \triangleright C, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B \wedge C, \Gamma \longrightarrow \mathcal{D}} \wedge\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \qquad \Sigma : \Gamma \longrightarrow \sigma \triangleright C}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \wedge C} \wedge\mathcal{R}$$

$$\frac{\Sigma : \sigma \triangleright B, \Gamma \longrightarrow \mathcal{D} \qquad \Sigma : \sigma \triangleright C, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B \vee C, \Gamma \longrightarrow \mathcal{D}} \vee\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright B}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \vee C} \vee\mathcal{R} \qquad \frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright C}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \vee C} \vee\mathcal{R}$$

$$\frac{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \qquad \Sigma : \sigma \triangleright C, \Gamma \longrightarrow \mathcal{D}}{\Sigma : \sigma \triangleright B \supset C, \Gamma \longrightarrow \mathcal{D}} \supset\mathcal{L} \qquad \frac{\Sigma : \sigma \triangleright B, \Gamma \longrightarrow \sigma \triangleright C}{\Sigma : \Gamma \longrightarrow \sigma \triangleright B \supset C} \supset\mathcal{R}$$

$$\frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma : \sigma \triangleright B[t/x], \Gamma \longrightarrow \mathcal{C}}{\Sigma : \sigma \triangleright \forall_\gamma x.B, \Gamma \longrightarrow \mathcal{C}} \forall\mathcal{L} \qquad \frac{\Sigma, h : \Gamma \longrightarrow \sigma \triangleright B[(h\,\sigma)/x]}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \forall x.B} \forall\mathcal{R}$$

$$\frac{\Sigma, h : \sigma \triangleright B[(h\,\sigma)/x], \Gamma \longrightarrow \mathcal{C}}{\Sigma : \sigma \triangleright \exists x.B, \Gamma \longrightarrow \mathcal{C}} \exists\mathcal{L} \qquad \frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma : \Gamma \longrightarrow \sigma \triangleright B[t/x]}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \exists_\gamma x.B} \exists\mathcal{R}$$

$$\frac{}{\Sigma : \sigma \triangleright \bot, \Gamma \longrightarrow \mathcal{B}} \bot\mathcal{L} \qquad \frac{}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \top} \top\mathcal{R} \qquad \frac{\Sigma : \mathcal{B}, \mathcal{B}, \Gamma \longrightarrow \mathcal{C}}{\Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{C}} c\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow \mathcal{C}}{\Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{C}} w\mathcal{L}$$

$$\frac{}{\Sigma : \sigma \triangleright B, \Gamma \longrightarrow \sigma \triangleright B} \mathit{init} \qquad \frac{\Sigma : \Delta \longrightarrow \mathcal{B} \qquad \Sigma : \mathcal{B}, \Gamma \longrightarrow \mathcal{C}}{\Sigma : \Delta, \Gamma \longrightarrow \mathcal{C}} \mathit{cut}$$

Figure 2. The intuitionistic rules of $FO\lambda$.

3. An intuitionistic logic with ∇

We now consider Gentzen's LJ calculus [5] with the addition of global and local signatures and ∇. Besides this new quantifier, the other logical connectives are $\bot$, $\top$, $\wedge$, $\vee$, $\supset$, $\forall_\gamma$, and $\exists_\gamma$ (again, the type γ does not contain o) and their inference rules are given in Figure 2. Notice that no inference rule in Figure 2 requires non-empty local signatures: as a result, if all the local signatures in sequents in a derivation built from those rules are set to empty, the resulting derivation is a standard derivation in intuitionistic logic.

The interaction between the global and local signatures and the universal and existential quantifiers needs some explanations. In the rule for $\forall\mathcal{L}$ (and, dually, for $\exists\mathcal{R}$), the quantifier appears in the scope of the global signature Σ and the local signature σ. This quantifier can be instantiated (reading the rule bottom-up) with a term built from variables in both of these signatures. Similarly, in the rule for $\forall\mathcal{R}$ (and, dually, for $\exists\mathcal{L}$), the quantifier appears in the scope of the global signature Σ and the local signature σ. This quantifier can be instantiated (reading the rule bottom-up) with an eigenvariable whose intended range is over all terms built from variables in Σ and σ. Since, however, the eigenvariable h is stored in the global scope, its dependency on σ would be forgotten unless we employ some particular encoding technique. For this purpose, we use *raising* [14]: to denote a variable of type γ_0 that can range over some set of constants and over the variables in $\sigma = (x_1 : \gamma_1, \ldots, x_n : \gamma_n)$ ($n \geq 0$), we can use instead the term $(hx_1 \ldots x_n)$ where the variable h ranges over the set of constants only (the dependency on σ can be forgotten). Of course, the type of h will be $\gamma_1 \to \cdots \to \gamma_n \to \gamma_0$ instead of simply γ_0. In the inference rules of Figure 2, we write $(h\sigma)$ to denote $(hx_1 \ldots x_n)$.

For the sake of consistency with a naming convention from the papers [8, 9], we shall refer to the inference system defined with just the rules in Figure 2 as $FO\lambda$ (mnemonic for a "first-order logic for λ-expressions"). The proof system resulting from the addition of the rules for ∇ (Figure 1) is called $FO\lambda^\nabla$.

Below are some theorems of $FO\lambda^\nabla$ involving ∇. In these formulas, we use $\neg C$ to abbreviate $C \supset \bot$ and we write $B \equiv C$ to denote $(B \supset C) \wedge (C \supset B)$.

$$\begin{gathered}
\nabla x \neg Bx \equiv \neg \nabla x Bx \\
\nabla x(Bx \wedge Cx) \equiv \nabla x Bx \wedge \nabla x Cx \\
\nabla x(Bx \vee Cx) \equiv \nabla x Bx \vee \nabla x Cx \\
\nabla x(Bx \supset Cx) \equiv \nabla x Bx \supset \nabla x Cx \\
\nabla x \forall y Bxy \equiv \forall h \nabla x Bx(hx) \\
\nabla x \exists y Bxy \equiv \exists h \nabla x Bx(hx) \\
\nabla x \forall y Bxy \supset \forall y \nabla x Bxy \\
\nabla x.\top \equiv \top, \quad \nabla x.\bot \equiv \bot
\end{gathered}$$

As a result of these equivalences, ∇ can alway be given atomic scope within formulas (with the simple cost of raising the quantified variables in its scope).

Below are some non-theorems of $FO\lambda^\nabla$ involving ∇.

$$\begin{array}{cc}
\nabla x \nabla y Bxy \supset \nabla z Bzz & \nabla x Bx \supset \exists x Bx \\
\nabla z Bzz \supset \nabla x \nabla y Bxy & \nabla x Bx \supset \forall x Bx \\
\forall y \nabla x Bxy \supset \nabla x \forall y Bxy & \exists x Bx \supset \nabla x Bx \\
\nabla x B \equiv B & \forall x Bx \supset \nabla x Bx
\end{array}$$

4. Introduction rules for definitions

Introduction rules are, generally, restricted to logical connectives and quantifiers. The recent development of a proof theoretic notion of *definitions* [7, 24, 6, 9] provides left and right introduction rules also for non-logical predicate symbols, provided that they are "defined" in terms of other predicates appropriately. Given certain restrictions on the syntax of definitions, a proof system with such definition introduction rules can enjoy cut-elimination. In this section, we take the treatment of definitions from [8, 9] and extend it to handle local signatures.

Definition 1 A *definitional clause* is written $\forall \bar{x}[p\,\bar{t} \triangleq B]$, where p is a predicate constant, every free variable of the formula B is also free in at least one term in the list $\bar{t}$ of terms, and all variables free in $p\,\bar{t}$ are contained in the list $\bar{x}$ of variables. The atomic formula $p\,\bar{t}$ is called the *head* of the clause, and the formula B is called the *body*. The symbol $\triangleq$ is used simply to indicate a definitional clause: it is not a logical connective. A *definition* is a (perhaps infinite) set of definitional clauses. 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.

Although predicates are defined via mutual recursion, circularities through implications (negations) must be avoided. To do this, we stratify definitions by first associating to each predicate p a natural number $\mathrm{lvl}(p)$, the *level* of p. The notion of level is generalized to formulas as follows.

Definition 2 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. $\mathrm{lvl}(\forall x.B) = \mathrm{lvl}(\nabla x.B) = \mathrm{lvl}(\exists x.B) = \mathrm{lvl}(B)$.

We shall require that for every definitional clause $\forall \bar{x}[p\,\bar{t} \triangleq B]$, $\mathrm{lvl}(B) \leq \mathrm{lvl}(p)$. This requirement allows us to prove cut-elimination for $FO\lambda^{\Delta\nabla}$ (see Section 5 and [9, 25]).

Introduction rules for defined atoms involve the use of substitutions. We recall some basic definitions related to substitutions. A *substitution* θ is a mapping (with application written in postfix notation) from variables to terms, such that the set $\{x \mid x\theta \neq x\}$ is finite. Although substitutions are extended to mappings from terms to terms, generic judgments to generic judgments, etc, when we refer to the *domain* and the *range* of a substitution, we refer to those sets defined on this most basic function. A substitution is extended to a function from terms to terms in the usual fashion. Composition of substitutions is defined as $t(\theta \circ \sigma) = (t\theta)\sigma$, for all terms t. Two substitutions θ and σ are considered equal if for all variables x, $x\sigma =_\eta x\theta$ (equal modulo η-conversion). The empty substitution is written as ϵ. The application of a substitution θ to a generic judgment $x_1, \ldots, x_n \triangleright B$, written as $(x_1, \ldots, x_n \triangleright B)\theta$, is $x_1, \ldots, x_n \triangleright B'$, if $(\lambda x_1 \ldots \lambda x_n.B)\theta$ is equal (modulo λ-conversion) to $\lambda x_1 \ldots \lambda x_n.B'$. If Γ is a multiset of generic judgments, then $\Gamma\theta$ is the multiset $\{J\theta \mid J \in \Gamma\}$. Finally, if Σ is a signature then $\Sigma\theta$ is the signature that results from removing from Σ the variables in the domain of θ and adding the variables that are free in the range of θ.

$$\frac{\Sigma : \Gamma \longrightarrow (\sigma \triangleright B)\theta}{\Sigma : \Gamma \longrightarrow \sigma \triangleright A}\ \mathit{def}\mathcal{R},\ \text{where } \mathrm{dfn}(\Sigma, \epsilon, \sigma \triangleright A, \theta, B)$$

$$\frac{\{\Sigma\rho : (\sigma \triangleright B)\theta, \Gamma\rho \longrightarrow \mathcal{C}\rho \mid \mathrm{dfn}(\Sigma, \rho, \sigma \triangleright A, \theta, B)\}}{\Sigma : \sigma \triangleright A, \Gamma \longrightarrow \mathcal{C}}\ \mathit{def}\mathcal{L}$$

Figure 3. The definition introduction rules

The following relation will be useful for the introduction rules for defined atoms.

Definition 3 The relation $\mathrm{dfn}(\Sigma, \rho, \sigma \triangleright A, \theta, B)$ holds for the formulas A and B, the substitutions ρ and θ, and the (disjoint) signatures Σ and σ whenever the following holds: the variables $h_1, \ldots, h_n$ are distinct from the variables in Σ and σ, the signature σ is the list of variables $\bar{y} = y_1, \ldots, y_p$ ($p \geq 0$), the given definition contains a clause $\forall x_1, \ldots, x_n.[H' \triangleq B']$, the formulas B and H are "raised" versions of B' and H', that is,

$$B = B'[(h_1\ \bar{y})/x_1, \ldots, (h_n\ \bar{y})/x_n]$$

$$H = H'[(h_1\ \bar{y})/x_1, \ldots, (h_n\ \bar{y})/x_n]$$

and $(\lambda y_1 \cdots \lambda y_p A)\rho = (\lambda y_1 \cdots \lambda y_p H)\theta$.

The right and left rules for atoms are given in Figure 3. Specifying a set of sequents as the premise should be understood to mean that each sequent in the set is a premise of the rule. Notice that in the *def*$\mathcal{L}$ rule, the free variables of the conclusion can be instantiated in the premises. In particular, a variable can possibly be removed from Σ and several new variables can be added.

These rules for definitions add considerable expressive power to intuitionistic logic. For example, *def*$\mathcal{R}$ is essentially the *backchaining* rule found in logic programming, while *def*$\mathcal{L}$ is essentially a case analysis on how an atom can be proved and can be used to establish *finite failure*. Together, these two rules can be used to encode simulation and bisimulation in certain abstract transition systems [11]. Other uses involve reasoning about computational system [10].

The proof system that arises from adding together the inference rules in Figures 2 and 3 is called $FO\lambda^{\Delta}$. If we add to $FO\lambda^{\Delta}$ the rules in Figure 1, the resulting proof system is called $FO\lambda^{\Delta\nabla}$ (pronounced "fold nabla"). It is this logic that will involve us for the remainder of this abstract.

5. The meta-theory of $FO\lambda^{\Delta\nabla}$

Of course, the main meta-theorem for $FO\lambda^{\Delta\nabla}$ is cut-elimination.

Proposition 4 *Given a fixed stratified definition, a sequent has a proof in $FO\lambda^{\Delta\nabla}$ if and only if it has a cut-free proof.*

Proof Outline. The proof of cut-elimination for $FO\lambda^{\Delta\mathbb{N}}$ [9] can be adapted to this setting. The $FO\lambda^{\Delta\mathbb{N}}$ logic includes induction and hence the induction required to prove termination is much more complicated than is required for $FO\lambda^{\Delta\nabla}$, which does not incorporate induction. Here, an induction involving the heights of proofs works similarly to that done by Gentzen [5], with an additional measure involving the level of cut formulas. The stratification of definitions makes sure that the level of cut formulas decreases when permuting up cut over definition rules. Other aspects of the proof are similar. Central to the proof is the following substitution lemma about $FO\lambda^{\Delta\nabla}$ proofs: if $\Sigma : \Gamma \longrightarrow \mathcal{C}$ has a proof and θ be a substitution, then there is a derivation of $\Sigma\theta : \Gamma\theta \longrightarrow \mathcal{C}\theta$ with the same or lesser height. A complete proof of cut elimination can be found in [25].

In certain situations, the difference between ∇ and $\forall$ cannot actually be observed. More specifically, consider the following restrictions on formulas and definitions. An hc-*goal* (named for Horn clauses) is a formula built from $\top$, $\wedge$, $\vee$, and $\exists$. An $\mathsf{hc}^{\forall}$-*goal* is a formula built from $\top$, $\wedge$, $\vee$, $\exists$, and $\forall$, while an hc^{∇}-*goal* is a formula built from $\top$, $\wedge$, $\vee$, $\exists$, and ∇. A definition is an hc-*definition* (resp., $\mathsf{hc}^{\forall}$-*definition* and hc^{∇}-*definition*) if the body of all of its clauses are hc-*goals* (resp., $\mathsf{hc}^{\forall}$-*goals* and hc^{∇}-*goals*). Notice that all such definitions are trivially stratifiable. Numerous interesting computer science motivated specifications are examples of $\mathsf{hc}^{\forall}$-definitions: we consider in more detail two such examples in Sections 6 and 7. The proof of the following proposition follows by a simple induction on the structure of $FO\lambda^{\Delta\nabla}$ proofs.

Proposition 5 *Let $\mathbf{D}$ be an $\mathsf{hc}^{\forall}$-definition and let $\mathbf{D}'$ be the hc^{∇}-definition resulting from replacing all occurrences of $\forall$ in the body of clauses of $\mathbf{D}$ with ∇. Similarly, let G be an $\mathsf{hc}^{\forall}$-goal and let G' be the hc^{∇}-goal resulting from replacing all occurrences of $\forall$ in G with ∇. The sequent $\Sigma : \cdot \longrightarrow \cdot \triangleright G$ is provable using definition $\mathbf{D}$ if and only if the sequent $\cdot : \cdot \longrightarrow \Sigma \triangleright G'$ is provable using definition $\mathbf{D}'$.*

As a consequence of this proposition, the difference between $\forall$ and ∇ (or, equivalently, between the global and local signatures of a sequent) cannot be seen if one is simply attempting to "evaluate" $\mathsf{hc}^{\forall}$ logical programs by determining the goals that they can prove. A difference between these two quantifiers only starts to appear (for $\mathsf{hc}^{\forall}$-definitions) if more interesting goals are considered: for example, in Section 6, we illustrate the differences between $\forall$ and ∇ with the specification of simulation and bisimulation in the π-calculus.

A natural question to ask about ∇, in relation to its role as local binder, is whether the relative orders among consecutive ∇'s matters, or more precisely, whether the formula

$$\nabla y \nabla x Bxy \supset \nabla x \nabla y Bxy$$

is provable in $FO\lambda^{\Delta\nabla}$. Of course, this formula is not provable in the logic without definitions. Consider the following Definition and Proposition.

Definition 6 A definition $\mathbf{D}$ is *noetherian* if for every definition clause $\forall\bar{x}.[p\bar{t} \stackrel{\triangle}{=} B]$ in $\mathbf{D}$, it holds that $\mathrm{lvl}(p) > \mathrm{lvl}(B)$.

Proposition 7 *Given a noetherian definition, the sequent $\Sigma : \Gamma, \sigma \triangleright B \longrightarrow \sigma' \triangleright B$, where σ' is a permutation of σ, is provable in $FO\lambda^{\Delta\nabla}$.*

Proof By induction on the level of B with subordinate induction on the size of B. In the case where B is an atomic formula, we apply *def$\mathcal{L}$* followed by *def$\mathcal{R}$*. Since the definition is noetherian, we always get formulas of lower level as a result. A detailed proof can be found in [25]. ∎

Thus, for noetherian definitions, ∇'s can be interchanged. We conjecture that this is also true for non-noetherian definitions as well.

6. Example: the π-calculus

Operational semantics of specification languages or programming languages is often given using inference rules, following the small-step approach (a.k.a., structured operational semantic) or big-step approach (a.k.a. natural semantics). Frequently, the specification of such semantics requires new symbols to be created to be used for such things as nonces in security protocols [1], locations for reference cells [2, 16], or new communication channels [19]. Given the logic $FO\lambda^{\Delta\nabla}$, we now have the ability to scope variables within sequents either globally via $\forall$ or locally via ∇. We illustrate these choices with a specification of the π-calculus.

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

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

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

$$
\frac{\nabla n(Pn \xrightarrow{A} P'n)}{\nu n.Pn \xrightarrow{A} \nu n.P'n}\,\text{res}
\qquad
\frac{\nabla n(Pn \xrightharpoonup{A} P'n)}{\nu n.Pn \xrightharpoonup{A} \lambda m\, \nu n.(P'nm)}\,\text{res}
\qquad
\frac{\nabla y(My \xrightarrow{\uparrow xy} M'y)}{\nu y.My \xrightharpoonup{\uparrow x} M'}\,\text{open}
$$

$$
\frac{}{\mathit{out}\, x\, y\, P \xrightarrow{\uparrow xy} P}\,\text{output}
\qquad
\frac{P \xrightharpoonup{\downarrow x} M \qquad Q \xrightharpoonup{\uparrow x} N}{P \mid Q \xrightarrow{\tau} \nu n.(Mn \mid Nn)}\,\text{close}
\qquad
\frac{P \xrightharpoonup{\uparrow x} M \qquad Q \xrightharpoonup{\downarrow x} N}{P \mid Q \xrightarrow{\tau} \nu n.(Mn \mid Nn)}\,\text{close}
$$

$$
\frac{}{\mathit{in}\, x\, M \xrightharpoonup{\downarrow x} M}\,\text{input}
\qquad
\frac{P \xrightharpoonup{\downarrow x} M \qquad Q \xrightarrow{\uparrow xy} Q'}{P \mid Q \xrightarrow{\tau} (My) \mid Q'}\,\text{com}
\qquad
\frac{P \xrightarrow{\uparrow xy} P' \qquad Q \xrightharpoonup{\downarrow x} N}{P \mid Q \xrightarrow{\tau} P' \mid (Ny)}\,\text{com}
$$

Figure 4. The rules for the (late) π-calculus.

Consider encoding π-calculus [19] using higher-order abstract syntax following [17, 18]. Since we are focused here on abstractions in syntax, we shall deal with only *finite* π-calculus expression, that is, expressions without ! or defined constants. Extending this work to infinite process expressions should be possible by adding induction (as in [11]) or co-induction to our proof system. We shall require three primitive syntactic categories: *n* for channels, *p* for processes, and *a* for actions. The output prefix is the constructor *out* of type $n \to n \to p \to p$ and the input prefix is the constructor *in* of type $n \to (n \to p) \to p$: 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 $p \to p \to p$ and written as infix, to denote parallel composition and summation, and ν of type $(n \to p) \to p$ 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 $n \to n \to p \to p$. When τ is written as a prefix, it has type $p \to p$. When τ is written as an action, it has type *a*. The symbols $\downarrow$ and $\uparrow$, both of type $n \to n \to a$, denote the input and output actions, respectively, on a named channel with a named value: e.g., $\downarrow xy$ denotes the action of inputting y on channel x.

We use two predicates to encode the one-step transition semantics for the π-calculus. The predicate $\cdot \xrightarrow{\cdot} \cdot$ of type $p \to a \to p \to o$ encodes transitions involving free values and the predicate $\cdot \xrightharpoonup{\cdot} \cdot$ of type $p \to (n \to a) \to (n \to p) \to o$ encodes transitions involving bound values. Figure 4 (taken from [18]) contains the inference rules specifying the late version of the transitions for the π-calculus [19]. In these rules, capital letters (possibly primed) are used to denote schema variables for inference rules: these schema variables have primitive types such as *a*, *n*, and *p* as well as functional types such as $n \to a$ and $n \to p$. These inference rules can trivially be written as definition clauses: a few such clauses are presented in Figure 5. Here, schema variables are universally quantified (implicitly) at the top-level of such clauses. Notice that the complicated side conditions in the original specification of π-calculus are not present here as they are now part of the meta-logic. For example, the side condition that $x \neq y$ in the open rule is handled by using two different quantifier scopes for x and y and the rule of logic that substitutions cannot capture bound variables.

Let $\mathcal{L}$ be the complete definition for the one step transition for the π-calculus. Clearly, $\mathcal{L}$ is an hc^{∇}-definition. Let let $\mathcal{L}'$ be the result of replacing all occurrences of ∇ in $\mathcal{L}$ with $\forall$. Furthermore, let $\mathcal{L}''$ be the result of replacing all occurrences of the symbol $\triangleq$ in the clauses of $\mathcal{L}'$ by reverse implication: thus, $\mathcal{L}''$ is a set of formulas and is not a definition. If we are interested in only computing the one-step transitions of the late π-calculus, that is, proving the atomic formulas $P \xrightarrow{A} P'$ or $P \xrightharpoonup{A} P'$, then the following observations are easy to establish. Let B range over atomic formulas. Proposition 5 implies that $\cdot : \cdot \longrightarrow B$ is provable in $FO\lambda^{\Delta\nabla}$ using definition $\mathcal{L}$ if and only if $\cdot : \cdot \longrightarrow B$

$$\begin{array}{rcl} \nu n.Pn \xrightarrow{A} \nu n.Qn & \triangleq & \nabla n(Pn \xrightarrow{A} Qn) \\ \nu y.Py \xrightarrow{\uparrow X} Q & \triangleq & \nabla y(Py \xrightarrow{\uparrow Xy} Qy) \\ \mathit{in}\, X\ M \xrightarrow{\downarrow X} M & \triangleq & \top \\ P \mid Q \xrightarrow{\tau} S \mid (T\ Y) & \triangleq & \exists X.P \xrightarrow{\uparrow XY} S \wedge Q \xrightarrow{\downarrow X} T \end{array}$$

Figure 5. Corresponding definition clauses

$$\begin{array}{rl} \mathit{sim}\, P\ Q \triangleq & \forall A \forall P'\ [(P \xrightarrow{A} P') \supset \exists Q'.(Q \xrightarrow{A} Q') \\ & \qquad \wedge\, \mathit{sim}\ P'\ Q'] \wedge \\ & \forall X \forall P'\ [(P \xrightarrow{\downarrow X} P') \supset \exists Q'.(Q \xrightarrow{\downarrow X} Q') \\ & \qquad \wedge\, \forall w.\mathit{sim}\ (P'w)\ (Q'w)] \wedge \\ & \forall X \forall P'\ [(P \xrightarrow{\uparrow X} P') \supset \exists Q'.(Q \xrightarrow{\uparrow X} Q') \\ & \qquad \wedge\, \nabla w.\mathit{sim}\ (P'w)\ (Q'w)] \end{array}$$

Figure 6. Definition of π-calculus simulation

is provable in $FO\lambda^{\Delta}$ using definition $\mathcal{L}'$. Furthermore, a cut-free proof of $\cdot : \cdot \longrightarrow B$ in $FO\lambda^{\Delta}$ using definition $\mathcal{L}'$ does not contain occurrences of $\mathit{def}\mathcal{L}$, and, as a result, the definition mechanism itself can be replaced: the sequent $\cdot : \cdot \longrightarrow B$ is provable in $FO\lambda^{\Delta}$ with the definition $\mathcal{L}'$ if and only if the sequent $\Sigma : \mathcal{L}'' \longrightarrow B$ is provable in $FO\lambda$.

Thus, only standard logic programming (such as in λProlog) is needed to compute the one-step transitions of the π-calculus, and ∇ and definitions do not add expressive power. To see what expressive power is contributed by both ∇ and definitions in a proof system, consider the problem of computing the relationship of simulation for the π-calculus. (For simplicity, we shall consider only simulation and not bisimulation: extending to bisimulation is not difficult but does introduce several more cases and make our examples more difficult to read.)

To illustrate how ∇ and $\forall$ in the body of this definition clause differ, consider following four π-calculus expressions. (Here we are using the usual abbreviations: when the name, say z is used as a prefix, it denotes the prefix $z(w)$ where w is vacuous in its scope; when the name, $\bar{z}$ is used as a prefix, it denotes the prefix $\bar{z}a$, where a is some fixed value; the expression $\bar{x}(y).P$ abbreviates $(y)\bar{x}y.P$; and when a prefix is written without a continuation, the continuation 0 is assumed. Thus, for example, $\bar{y} \mid z$ denotes $\bar{y}a.0 \mid z(w).0$.)

$$\begin{array}{ll} P_1 = x(y).(\bar{y} \mid z) & P_2 = x(y).((\bar{y}.z) + (z.\bar{y})) \\ P_3 = \bar{x}(y).(\bar{y} \mid z) & P_4 = \bar{x}(y).((\bar{y}.z) + (z.\bar{y})) \end{array}$$

The process P_2 is simulated by P_1 but the converse is not true since after P_1 preforms an $(\downarrow xz)$, it is possible for the resulting process to take a τ step. The sequence of actions $(\downarrow\ xz)$ and τ is not possible with P_2. The processes P_3 and P_4 do, however, simulate each other (they are, in fact, bisimilar). The only difference between these pairs of processes is, of course, that the first is prefixed with a bounded input prefix while the second is prefixed with a bounded output prefix. These different bounded prefixes are handled in the simulation definition in Figure 6 using, in one case, $\forall$ and the other case ∇.

For example, consider proving the sequent

$$\cdot : \cdot \longrightarrow \mathit{sim}\ (x(y).(\bar{y} \mid z))\ (x(y).((\bar{y}.z) + (z.\bar{y}))),$$

which, as we discussed above, should fail. (For readability, we shall use π-calculus syntax directly instead of the abstract syntax on which the actual logic is based.) The free names x and z are interpreted as meta-level constants. The attempt to prove this sequent reduces (via $\mathit{def}\mathcal{R}$, $\forall$ and $\supset \mathcal{R}$) to needing to prove the three sequents (1-3) in Figure 7. A simple argument about the permutabilities of inference rules [9] shows that if a sequent with an atom on the left has a proof, it has a proof with an instance of the $\mathit{def}\mathcal{L}$ rule that introduces that atom. Thus, we can conclude that sequents (1) and (3) are trivially provable since the required unification problem in $\mathit{def}\mathcal{L}$ fails for all clauses in the definition. The second sequent is the consequence of a non-trivial occurrence of the $\mathit{def}\mathcal{L}$ rule, giving rise to the need to prove sequent (4) in Figure 7 (here, the variable N is instantiated to x and P' is instantiated to $\lambda y.(\bar{y} \mid z)$). Proving this requires making the appropriate substitution for Q' (obvious) and then proving the sequent

$$\cdot : \cdot \longrightarrow \forall w.\mathit{sim}\ (\bar{w} \mid z)\ ((\bar{w}.z) + (z.\bar{w}))$$

Similarly to our first step, proving this reduces to the three sequents (5), (6), and (7). Applying $\mathit{def}\mathcal{L}$ rule to sequents (6) and (7) produces one premise for each case, which eventually leads to proving the sequents $w, u : \cdot \longrightarrow \mathit{sim}\ \bar{w}\ \bar{w}$ and $\cdot : \cdot \longrightarrow u \triangleright \mathit{sim}\, z\ z$; both are trivially provable. A proof of (5) using $\mathit{def}\mathcal{L}$ has two premises: one with A instantiated to τ, w to z, and P' to $0 \mid 0$, and one with A instantiated to $\uparrow wa$ and P' to $0 \mid z$ (w is not instantiated). The first of these premise sequents is the sequent

$$\cdot : \cdot \longrightarrow \exists Q'[((\bar{z}.z) + (z.\bar{z})) \xrightarrow{\tau} Q' \wedge \mathit{sim}\ (0 \mid 0)\ Q']$$

This is not provable since there is no τ transition from $((\bar{z}.z) + (z.\bar{z}))$. As a result, since this sequent is not provable we may conclude that the original sequent is not provable. The reason for this failure is also clear from this attempt of a proof construction: although both P_1 and P_2 make an initial input step, the first of the resulting pair of processes can make a τ step but the second cannot.

Turning to the case of expressions P_3 and P_4, consider proving the sequent

$$\cdot : \cdot \longrightarrow \mathit{sim}\ (\bar{x}(y).(\bar{y} \mid z))\ (\bar{x}(y).((\bar{y}.z) + (z.\bar{y}))),$$

$$A, P' : (x(y).(\bar{y} \mid z)) \xrightarrow{A} P' \longrightarrow \exists Q'[(x(y).((\bar{y}.z) + (z.\bar{y}))) \xrightarrow{A} Q' \wedge \textit{sim}\; P'\; Q'] \tag{1}$$

$$N, P' : (x(y).(\bar{y} \mid z)) \xrightarrow{\downarrow N} P' \longrightarrow \exists Q'[(x(y).((\bar{y}.z) + (z.\bar{y}))) \xrightarrow{\downarrow N} Q' \wedge \forall w.\textit{sim}\; (P'w)\; (Q'w)] \tag{2}$$

$$N, P' : (x(y).(\bar{y} \mid z)) \xrightarrow{\uparrow N} P' \longrightarrow \exists Q'[(x(y).((\bar{y}.z) + (z.\bar{y}))) \xrightarrow{\uparrow N} Q' \wedge \nabla x.\textit{sim}\; (P'x)\; (Q'x)] \tag{3}$$

$$\cdot : \cdot \longrightarrow \exists Q'[(x(y).((\bar{y}.z) + (z.\bar{y}))) \xrightarrow{\downarrow x} Q' \wedge \forall w.\textit{sim}\; (\bar{w} \mid z)\; (Q'w)] \tag{4}$$

$$w, A, P' : (\bar{w} \mid z) \xrightarrow{A} P' \longrightarrow \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{A} Q' \wedge \textit{sim}\; P'\; Q'] \tag{5}$$

$$w, N, P' : (\bar{w} \mid z) \xrightarrow{\downarrow N} P' \longrightarrow \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{\downarrow N} Q' \wedge \forall u.\textit{sim}\; (P'u)\; (Q'u)] \tag{6}$$

$$w, N, P' : (\bar{w} \mid z) \xrightarrow{\uparrow N} P' \longrightarrow \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{\uparrow N} Q' \wedge \nabla u.\textit{sim}\; (P'u)\; (Q'u)] \tag{7}$$

$$\cdot : \cdot \longrightarrow \exists Q'[(\bar{x}(y).((\bar{y}.z) + (z.\bar{y}))) \xrightarrow{\uparrow x} Q' \wedge \nabla w.\textit{sim}\; (\bar{w} \mid z)\; (Q'w)] \tag{4'}$$

$$A, P' : w \triangleright (\bar{w} \mid z) \xrightarrow{(Aw)} (P'w) \longrightarrow w \triangleright \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{(Aw)} Q' \wedge \textit{sim}\; (P'w)\; Q'] \tag{5'}$$

$$N, P' : w \triangleright (\bar{w} \mid z) \xrightarrow{\downarrow (Nw)} (P'w) \longrightarrow w \triangleright \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{\downarrow (Nw)} Q' \wedge \forall u.\textit{sim}\; (P'wu)\; (Q'u)] \tag{6'}$$

$$N, P' : w \triangleright (\bar{w} \mid z) \xrightarrow{\uparrow (Nw)} (P'w) \longrightarrow w \triangleright \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{\uparrow (Nw)} Q' \wedge \nabla u.\textit{sim}\; (P'wu)\; (Q'u)] \tag{7'}$$

Figure 7. Some sequents

which, as we discussed above, should succeed. A proof attempt of this sequent proceeds similar to the previous example, yielding the sequent $(4')$ in Figure 7. Proving this reduces to the three sequents $(5')$, $(6')$, and $(7')$: notice that w is not given global scope in the sequents but local scope and that the eigenvariables (A, P', and N') are raised with respect to their counterparts in (5), (6), and (7)). Sequents $(6')$ and $(7')$ are proved as in (6) and (7). In this case, however, a proof of $(5')$ using *def$\mathcal{L}$* has exactly one premise, where A instantiated to $\lambda w.\uparrow wa$ and P' to $\lambda w.0 \mid z$. The resulting sequent is

$$\cdot : \cdot \longrightarrow w \triangleright \exists Q'[((\bar{w}.z) + (z.\bar{w})) \xrightarrow{\uparrow wa} Q' \wedge \textit{sim}\; (0 \mid z)\; Q']$$

This sequent, like all the remaining ones in this proof attempt, now have a simple proof.

Notice that although we have now encountered higher-order unification problems and higher-order substitutions, the unification problems generated from this particular example fall within the *higher-order pattern unification* or L_λ unification problems [13, 21]. This subset of the unification of simply typed λ-terms has complexity similar to that of first-order unification: it is decidable (in linear time) and has most general unifiers when unifiers exist. Proof search for a sequent that starts out with first-order quantification will remain "essentially" first-order, even though raising introduces variables of higher-order type.

The encoding of π-calculus above can also be extended to include the mismatch operator by using negation.

$$\frac{(x = y) \supset \bot \qquad P \xrightarrow{A} Q}{[x \neq y]P \xrightarrow{A} Q}\ \text{mismatch}$$

Operationally, mismatch is modeled as failure of unification at the logic level. Notice that the resulting definition is not Horn anymore since we have an implication in the body of the clause representing the above inference rule. As a consequence, Proposition 5 is not applicable to this definition.

7. Example: an object-logic encoding

Consider the problem of proving the formula

$$\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle],$$

where q is a three place predicate, $\langle \cdot, \cdot \rangle$ is used to form pairs, t_1 and t_2 are some first-order terms, and the only assumptions for the predicate q are the (universal closure of the) three atomic formulas: $q\ X\ X\ Y$, $q\ X\ Y\ X$ and $q\ Y\ X\ X$. Clearly, this query succeeds only if terms t_2 and t_3 are equal [18]. One natural way to formalizing this reasoning involves first encoding provability of an object-level first-order logic in $FO\lambda^{\Delta\nabla}$ and then to reason directly on this encoding. Let *obj* be the type of object-level formulas and let the object-level logic constants be: $\hat{\top}$ of type *obj*, $\hat{\&}$ and $\hat{\Rightarrow}$ of type $\textit{obj} \rightarrow \textit{obj} \rightarrow \textit{obj}$, and $\hat{\forall}$ and $\hat{\exists}$ be the quantifiers at type $(i \rightarrow \textit{obj}) \rightarrow \textit{obj}$ (for some fixed type i ranging over first-order object-level terms). To encode provability, we use four predicates: $\textit{pv}\ \cdot$ of type $\textit{obj} \rightarrow o$ encodes first-order provability, $\textit{bc}(\cdot, \cdot)$ of type $\textit{obj} \rightarrow \textit{obj} \rightarrow o$ encodes "backchaining", $\textit{atom}\ \cdot$ describes object-level atomic formulas, and $\textit{prog}\ \cdot$ describes object-level logic programs clauses. Figure 8 presents an encoding of provability for a first-order logic programming language that is restricted to $\mathsf{hc}^\forall$. Figure 9 contains such additional clauses for the example we are considering here.

Notice that while the object-level logic here is $\mathsf{hc}^\forall$ (since

$$
\begin{array}{rcl}
pv\,\hat{\top} & \triangleq & \top \\
pv\,(G \,\&\, G') & \triangleq & pv\,G \wedge pv\,G' \\
pv\,(\hat{\forall}\,G) & \triangleq & \nabla x.pv\,(Gx) \\
pv\,(\hat{\exists}\,G) & \triangleq & \exists x.pv\,(Gx) \\
pv\,A & \triangleq & \exists D.atom\,A \wedge prog\,D \wedge bc(D,A) \\
bc(A,A) & \triangleq & atom\,A \\
bc(G \Rightarrow D, A) & \triangleq & bc(D,A) \wedge pv\,G \\
bc(\hat{\forall}\,D, A) & \triangleq & \exists t.\ bc(D\ t, A)
\end{array}
$$

Figure 8. Interpreter for an object-level logic.

$$
\begin{array}{rcl}
X = X & \triangleq & \top \\
atom\,(q\ X\ Y\ Z) & \triangleq & \top \\
prog\,(\hat{\forall}\,X\,\hat{\forall}\,Y\ q\ X\ X\ Y) & \triangleq & \top \\
prog\,(\hat{\forall}\,X\,\hat{\forall}\,Y\ q\ X\ Y\ X) & \triangleq & \top \\
prog\,(\hat{\forall}\,X\,\hat{\forall}\,Y\ q\ Y\ X\ X) & \triangleq & \top
\end{array}
$$

Figure 9. Additional definition clauses.

we are concerned with the provability of a universally quantified formula), the meta-level definition is hc^{∇}.

The query that captures our intended example is the following formula

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

along with the definition consisting of the clauses in Figures 8 and 9. Attempting a proof of this formula leads to the following sequent (after applying some right rules and a pair of $def\mathcal{L}$ and $\nabla\mathcal{L}$ rules):

$$X, Y, Z : (s, r) \triangleright pv\,(q\ \langle s, X\rangle\ \langle r, Y\rangle\ \langle r, Z\rangle) \longrightarrow \triangleright Y = Z.$$

A series of *def*L rules will now need to be applied in order to work through the encoding for the object-level interpreter. In the end, three separate unification problems will be attempted, one for each of the three ways to prove the predicate q. In particular, the $def\mathcal{L}$ rule will attempt to unify the term $\lambda s \lambda r.(q\ \langle s, X\rangle\ \langle r, Y\rangle\ \langle r, Z\rangle)$ with each of the following three terms:

$$
\begin{array}{c}
\lambda s \lambda r.(q\ (X'\ s\ r)\ (X'\ s\ r)\ (Y'\ s\ r)) \\
\lambda s \lambda r.(q\ (X'\ s\ r)\ (Y'\ s\ r)\ (X'\ s\ r)) \\
\lambda s \lambda r.(q\ (Y'\ s\ r)\ (X'\ s\ r)\ (X'\ s\ r))
\end{array}
$$

The first two unification problems fail and hence the corresponding occurrences of $def\mathcal{L}$ succeed. The third of these unification problems is solvable, however, with X' instantiated to $\lambda s \lambda r.\langle r, Z\rangle$, Y' instantiated to $\lambda s \lambda r.\langle s, Z\rangle$, Y instantiated to Z (or vice versa), and X uninstantiated. As a result, this third premise is the sequent $\cdot : \cdot \longrightarrow Y = Y$, which is provable using $def\mathcal{R}$.

The more common approach to encoding object-logic provability into a meta-logic uses the meta-level universal quantifier instead of the ∇ for the clause encoding the provability of object-level universal quantification: that is, the clause

$$pv\,(\hat{\forall}\,x.G\ x) \triangleq \forall x[pv\,(G\ x)].$$

is used instead. In this case, attempting a proof of this formula reduces to an attempt to prove the sequent

$$X, Y, Z : \triangleright pv\,(q\ \langle s_1, X\rangle\ \langle s_2, Y\rangle\ \langle r, Z\rangle) \longrightarrow \triangleright Y = Z,$$

and were s_1 and s_2 are two terms. To complete the proof, these two terms must be chosen to be different. While this sequent can be proved, doing so requires the assumption that there are two such terms (the domain is non-empty and not a singleton). Our encoding using ∇ allows this (meta-level) proof to be completed in a more natural way without this assumption.

8. Related work and conclusion

We have maintained the approach to specification in which meta-level and proof-level abstractions are used to encode abstractions both of the static structure of expressions (e.g., using meta-level λ-abstractions to encode the input prefix in the π-calculus) and the dynamic structure of computation (e.g., name generation as eigenvariables). While this style of syntactic representation has been successfully used to enumerate judgments about operational semantics and to encode object-logic provability, proof level abstractions (eigenvariables) seem inadequate when one wishes to reason about computation directly (as outlined in Section 1). Since this style of syntactic representation is best understood declaratively within proof theory, we have explored a simple mechanism within sequent calculus to expand the notion of abstraction in the building of proofs. In [18], we provided some experiments in specification that this paper attempts to formalize using proof theory.

It is natural to ask about possible connections between the ∇-quantifier here and the new quantifier of Pitts and Gabbay [4, 23]. Both are self dual and both have similar sets of applications in mind. There are significant differences, however: ∇ has a natural proof theory with a cut-elimination theorem but has no set theoretic semantics, while Pitts and Gabbay have a model theory based on set theory but no cut-elimination result. While ∇ neither implies nor is implied by $\forall$ or $\exists$, the quantifier of Pitts and Gabbay is entailed by $\forall$ and entails $\exists$.

To work with larger examples than those shown here, one needs an implementation of $FO\lambda^{\Delta\nabla}$. The Isabelle theorem prover should provide a promising setting for building an interactive theorem prover given the work reported in [20]. A natural next step is to attempt adding directly to $FO\lambda^{\Delta\nabla}$ induction and co-induction: induction should work much as it does in $FO\lambda^{\Delta\mathbb{N}}$ [9]. Some related work on co-induction appears in [20].

Acknowledgments The authors wish to thank Catuscia Palamidessi for valuable discussions regarding our π-calculus examples. We would also like to thank the anonymous reviewers of this paper for their helpful suggestions on an earlier draft of this paper. This work has been supported in part by NSF grants CCR-9912387, INT-9815645, and INT-9815731. The second author gratefully acknowledges support from LIX at École polytechnique.

References

[1] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

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

[3] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[5] G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

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

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

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

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

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

[11] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411-437, 2003.

[12] D. Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

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

[14] D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.

[15] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1993.

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

[17] D. Miller and C. Palamidessi. Foundational aspects of syntax. In P. Degano, R. Gorrieri, A. Marchetti-Spaccamela, and P. Wegner, editors, *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, volume 31. ACM, Sep 1999.

[18] D. Miller and A. Tiu. Encoding generic judgments. In *Proceedings of FSTTCS*, number 2556 in LNCS, pages 18–32, December 2002.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[20] A. Momigliano, S. Ambler, and R. Crole. A hybrid encoding of Howe's method for establishing congruence of bisimilarity. In *LFM'02*, volume 70.2 of *ENTCS*, 2002.

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

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

[23] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*. To appear. (A preliminary version appeared in the *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software* (TACS 2001), LNCS 2215, Springer-Verlag, 2001, pp 219–242.).

[24] P. 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.

[25] A. Tiu. Cut-elimination for a logic with generic judgments. April 2003. Draft available via `http://www.cse.psu.edu/~tiu/foldn2.pdf`.