Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 174 (2007) 3–18

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Logic for Reasoning about Generic Judgments

Alwen Tiu

Australian National University and National ICT Australia

Abstract

This paper presents an extension of a proof system for encoding generic judgments, the logic $FO\lambda^{\Delta\nabla}$ of Miller and Tiu, with an induction principle. The logic $FO\lambda^{\Delta\nabla}$ is itself an extension of intuitionistic logic with fixed points and a "generic quantifier", ∇, which is used to reason about the dynamics of bindings in object systems encoded in the logic. A previous attempt to extend $FO\lambda^{\Delta\nabla}$ with an induction principle has been unsuccessful in modeling some behaviours of bindings in inductive specifications. It turns out that this problem can be solved by relaxing some restrictions on ∇, in particular by adding the axiom $B \equiv \nabla x.B$, where x is not free in B. We show that by adopting the equivariance principle, the presentation of the extended logic can be much simplified. Cut-elimination for the extended logic is stated, and some applications in reasoning about an object logic and a simply typed λ-calculus are illustrated.

Keywords: Proof theory, higher-order abstract syntax, logical frameworks.

1 Introduction

This paper aims at providing a framework for reasoning about specifications of deductive systems using *higher-order abstract syntax* [20]. Higher-order abstract syntax is a declarative approach to encoding syntax with bindings using Church's simply typed λ-calculus. The main idea is to support the notions of α-equivalence and substitutions in the object syntax by operations in λ-calculus, in particular α-conversion and β-reduction. There are at least two approaches to higher-order abstract syntax. The *functional programming* approach encodes the object syntax as a data type, where the binding constructs in the object language are mapped to functions in the functional language. In this approach, terms in the object language become values of their corresponding types in the functional language. The *proof search* approach encodes object syntax as expressions in a logic whose terms are simply typed, and functions that act on the object terms are defined via relations, i.e., logic programs. There is a subtle difference between this approach and the former; in the proof search approach, the simple types are inhabited by well-formed expressions, instead of values as in the functional approach (i.e., the abstraction

doi:10.1016/j.entcs.2007.01.016

type is inhabited by functions). The proof search approach is often referred to as λ-*tree syntax* [16], to distinguish it from the functional approach. This paper concerns the λ-tree syntax approach.

Specifications which use λ-tree syntax are often formalized using hypothetical and generic judgments in intuitionistic logic. It is enough to restrict to the fragment of first-order intuitionistic logic whose only formulas are those of hereditary Harrop formulas, which we will refer to as the HH logic. Consider for instance the problem of defining the data type for untyped λ-terms. One first introduces the following constants:

$$app : tm \to tm \to tm \qquad abs : (tm \to tm) \to tm$$

where the type tm denotes the syntactic category of λ-terms and app and abs encode application and abstraction, respectively. The property of being a λ-term is then defined via the following theory:

$$\bigwedge M \bigwedge N (lam\ M \wedge lam\ N \Rightarrow lam\ (app\ M\ N))\ \ \&$$
$$\bigwedge M ((\bigwedge x. lam\ x \Rightarrow lam\ (M\, x)) \Rightarrow lam\ (abs\ M))$$

where $\bigwedge$ is the universal quantifier and $\Rightarrow$ is implication.

Reasoning about object systems encoded in HH is reduced to reasoning about the structure of proofs in HH. McDowell and Miller formalize this kind of reasoning in the logic $FO\lambda^{\Delta\mathbb{N}}$ [10], which is an extension of first-order intuitionistic logic with fixed points and natural numbers induction. This is done by encoding the sequent calculus of HH inside $FO\lambda^{\Delta\mathbb{N}}$ and prove properties about it. We refer to HH as object logic and $FO\lambda^{\Delta\mathbb{N}}$ as meta logic. McDowell and Miller considered different styles of encoding and concluded that explicit representations of hypotheses and, more importantly, eigenvariables of the object logic are required in order to capture some statements about object logic provability in the meta logic [11]. One typical example involves the use of hypothetical and generic reasoning as follows: Suppose that the following formula is provable in HH.

$$\bigwedge x. p\, x\, s \Rightarrow \bigwedge y. p\, y\, t \Rightarrow p\, x\, t.$$

By inspection on the inference rules of HH, one observes that this is only possible if s and t are syntactically equal. This observation comes from the fact that the right introduction rule for universal quantifier, reading the rule bottom-up, introduces new constants, or eigenvariables. The quantified variables x and y will be replaced by distinct eigenvariables and hence the only matching hypothesis for $p\, x\, t$ would be $p\, x\, s$, and therefore s and t has to be equal. Let $\vdash_{HH} F$ denote the provability of the formula F in HH. Then in the meta logic, we would want to be able to prove the statement:

$$\forall s \forall t. (\vdash_{HH} \bigwedge x. p\, x\, s \Rightarrow \bigwedge y. p\, y\, t \Rightarrow p\, x\, t) \supset s = t.$$

The question is then how we would interpret the object logic eigenvariables in the meta logic. It is argued in [11] that the existing quantifiers in $FO\lambda^{\Delta\mathbb{N}}$ cannot be used to capture the behaviours of object logic eigenvariables directly. McDowell and Miller then resort to a non-logical encoding technique (in the sense that no logical connectives are used) which has some similar flavor to the use of de Bruijn indexes. The use of this encoding technique, however, has a consequence that substitutions in the object logic has to be formalized explicitly.

Motivated by the above mentioned limitation of $FO\lambda^{\Delta\mathbb{N}}$, Miller and Tiu later introduced a new quantifier ∇ to $FO\lambda^{\Delta\mathbb{N}}$ which allows one to move the binders from the object logic to the meta logic. A generic judgment in the object logic, for instance $\vdash_{HH} \bigwedge x.G\,x$ is reflected in the meta logic as $\nabla x.\vdash_{HH} G\,x$. More generally, object logic eigenvariables are ∇-quantified at the meta level. This meta logic, called $FO\lambda^{\Delta\nabla}$ [17], allows one to perform case analyses on the provability of the object logic. Tiu later extended $FO\lambda^{\Delta\nabla}$ with induction and co-induction rules, resulting in the logic Linc [25]. However, some inductive properties about the object logic are not provable in Linc, e.g, the implication

$$\vdash_{HH} \bigwedge x.G\,x \supset \forall t.\vdash_{HH} G\,t \tag{1}$$

which states the extensional property of object logic universal quantification.

The inductive proof of the formula (1) would require an induction hypothesis that quantifies over object logic signatures, i.e., it is a statement of the sort

$$\text{“for all” } \vec{z},\ \forall H\nabla\vec{z}(\vdash_{HH} \bigwedge x.H\,\vec{z}\,x \supset \forall t.\vdash_{HH} H\,\vec{z}\,t)$$

where $\vec{z}$ is a list of object logic eigenvariables occurring in the object sequents. An obvious extension to Linc to formalize this statement would be to allow for quantification over arbitrary lists of variables which act like variable contexts to the object logic. However this is technically non-trivial and may require complicated proof theory. In this paper we follow an easier but weaker approach, which is expressive enough to allow for inductive reasoning over object specifications involving bindings. Instead of having explicit quantification over variable contexts, we require every proposition to hold in any variable context. This effectively translates to admitting the following axiom in $FO\lambda^{\Delta\nabla}$:

$$B \supset \nabla x.B, \qquad x \text{ is not free in } B, \tag{2}$$

which is not provable in $FO\lambda^{\Delta\nabla}$. Extensions to $FO\lambda^{\Delta\nabla}$ have been previously proposed in a couple of previous works [5,2]. In both works, it is suggested that adding the following axioms

$$\nabla x\nabla y.B\,x\,y \supset \nabla y\nabla x.B\,x\,y \quad \text{and} \quad B \equiv \nabla x.B, \tag{3}$$

where x is not free in B in the second scheme, to $FO\lambda^{\Delta\nabla}$ would result in a natural semantics for the extended logic. As it turns out, admitting these axioms would give a simpler proof theory too, compared to just having (2). We therefore adopt the axioms (3) in the extension of $FO\lambda^{\Delta\nabla}$ discussed in the paper. This extended logic, called LG^{ω}, is obtained by extending $FO\lambda^{\Delta\nabla}$ with natural number induction and with the axiom schemes (3). We show that inductive properties of λ-tree syntax specifications can be stated directly and in a purely logical fashion, and proved in LG^{ω}.

Relation to nominal logic

To guarantee good proof search behavior and syntactic consistency of the logic LG^ω (i.e., cut-elimination), the axiom schemes (3) need to be absorbed into the rules of the proof system of LG^ω. There are at least a couple of ways of achieving this. One way is to extend the proof system of $FO\lambda^{\Delta\nabla}$ with some structural rules corresponding to the axioms (3). The other is to adopt the notion of *equivariant predicates* as in nominal logic [21], that is, provability of a predicate is invariant under permutations of *names*. We show here the second approach, which is simpler. The equivalent of the two formulations can be found in an extended version of the paper [26]. The equivariant principle is technically enforced by introducing a countably infinite set of name constants into the logic, and change the identity rule of the logic to allow equivalence under permutations of name constants:

$$\frac{\pi.B = \pi'.B'}{\Gamma, B \longrightarrow B'}\ id$$

where π and π' are permutations on names. LG^ω is in fact very close to nominal logic, when we consider only the behaviours of logical connectives. In particular, the quantifier ∇ in LG^ω shares the same properties, in relation to other connectives of the logic, with the И quantifier in nominal logic. However, there are two important differences in our approach. First, we do not attempt to redefine α-conversion and substitutions in LG^ω in terms of permutations (or *swapping*) and the notion of *freshness* as in nominal logic. Name swapping and freshness constraints are not part of the syntax of LG^ω. These notions are present only in the meta theory of the logic. In LG^ω, for example, variables are always considered to have empty support, that is, $\pi.x = x$ for every permutation π. This is because we restrict substitutions to the "closed" ones, in the sense that no name constants can appear in the substitutions. A restricted form of open substitutions can be recovered indirectly at the meta theory of LG^ω. The fact that variables have empty support allows one to work with permutation free formulas and terms. So in LG^ω, we can prove that $p\ x\ a \supset p\ x\ b$, where a and b are names, without using explicit axioms of permutations and freshness. In nominal logic, one would prove this by using the swapping axiom $p\ x\ a \supset p\ ((a\ b).x)\ ((a\ b).b)$, where $(a\ b)$ denotes a swapping of a and b, and then show that $(a\ b).x = x$. The latter might not be valid if x is substituted by a, for example. The validity of this formula in nominal logic would therefore depend on the assumption on the support of x.

The second difference between LG^ω and nominal logic is that LG^ω allows closed terms (again, in the sense that no name constants appear in them) of type name, while in nominal logic, allowing such terms would lead to inconsistency [21]. As an example, the type tm in the encoding of λ-terms mentioned previously can be treated as a nominal type in LG^ω. This has an important consequence that we do not need to redefine the notion of substitutions for the encoded λ-terms, which is instead mapped to β-reduction in the meta language of LG^ω.

The rest of this paper is organized as follows. In Section 2 we introduce a proof system for LG^ω. Section 3 states some meta theories of LG^ω, in particular cut-elimination and a translation from LG^ω without fixed points and induction to $FO\lambda^\nabla$ with the axioms (3). Section 4 shows an encoding of HH logic in LG^ω and how some properties of the object logic can be formalized in LG^ω. Section 5 illustrates the use of HH to specify the typing judgments of λ-calculus and the evaluation relation on λ-terms. It also shows an example of reasoning about the encoded λ-calculus, by induction on the provability of the typing judgments in the object logic HH. Section 6 discusses some related and future work. The proofs of the main results in this paper can be in an extended version of the paper [26].

2 A logic for generic judgments

We first define the core fragment of the logic LG^ω which does not have fixed point rules or induction. The starting point is the logic $FO\lambda^\nabla$ introduced in [17]. $FO\lambda^\nabla$ is an extension of a subset of Church's Simple Theory of Types in which formulas are given the type o. The core fragment of LG^ω, which we refer to as LG, shares the same set of connectives as $FO\lambda^\nabla$, namely, $\bot$, $\top$, $\wedge$, $\vee$, $\supset$, $\forall_\tau$, $\exists_\tau$ and ∇_τ. The type τ in the quantifiers is restricted to that which does not contain the type o. Hence the logic is essentially first-order. We abbreviate $(B \supset C) \wedge (C \supset B)$ as $B \equiv C$.

To enforce equivariant reasoning, we introduce a distinguished set of base types, called *nominal types*, which is denoted with $\mathcal{N}$. Nominal types are ranged over by ι. We restrict the ∇ quantifier to nominal types. For each nominal type $\iota \in \mathcal{N}$, we assume an infinite number of constants of that type. These constants are called *nominal constants*. We denote the family of nominal constants by $\mathcal{C}_\mathcal{N}$. The role of the nominal constants is to enforce the notion of equivariance: provability of formulas is invariant under permutations of nominal constants. Depending on the application, we might also assume a set of non-nominal constants, which is denoted by $\mathcal{K}$.

We assume the usual notion of capture-avoiding substitutions. Substitutions are ranged over by θ and ρ. Application of substitutions is written in a postfix notation, e.g., $t\theta$ is an application of θ to the term t. Given two substitutions θ and θ', we denote their composition by $\theta \circ \theta'$ which is defined as $t(\theta \circ \theta') = (t\theta)\theta'$. A *typing context* is a set of typed variables or constants. The judgment $\Delta \vdash t : \tau$ denotes the fact that the term t has type the simple type τ, given the typing context Δ. Its operational semantics is the usual type system for Church's simple type theory. A *signature* is a set of variables. A substitution θ respects a given signature Σ if there exists a set of typed variables Σ' such that for every $x : \tau \in \Sigma$ which is in the domain of θ, it holds that $\mathcal{K} \cup \Sigma' \vdash \theta(x) : \tau$. We denote by $\Sigma\theta$ the minimal set of variables satisfying the above condition. The substitution θ in this case is called a Σ-*substitution.* We assume that variables, free or bound, are of a different syntactic category from constants.

Definition 2.1 A permutation on $\mathcal{C}_\mathcal{N}$ is a bijection from $\mathcal{C}_\mathcal{N}$ to $\mathcal{C}_\mathcal{N}$. The permutations on $\mathcal{C}_\mathcal{N}$ are ranged over by π. Application of a permutation π to a nominal constant a is denoted with $\pi(a)$. We shall be concerned only with permutations which respect types, i.e., for every $a : \iota$, $\pi(a) : \iota$. Further, we shall also restrict to permutations which are finite, that is, the set $\{a \mid \pi(a) \neq a\}$ is finite. Application of a permutation to an arbitrary term (or formula), written $\pi.t$, is defined as follows:

$$\pi.a = \pi(a), \text{ if } a \in \mathcal{C}_{\mathcal{N}}. \qquad \pi.c = c, \quad \text{if } c \notin \mathcal{C}_{\mathcal{N}}. \qquad \pi.x = x$$
$$\pi.(M\ N) = (\pi.M)\ (\pi.N) \qquad \pi.(\lambda x.M) = \lambda x.(\pi.M)$$

A permutation involving only two nominal constants is called *swapping*. We use $(a\ b)$, where a and b are constants of the same type, to denote the swapping $\{a \mapsto b, b \mapsto a\}$.

The *support* of a term (or formula) t, written $supp(t)$, is the set of nominal constants appearing in it. It is clear from the above definition that if $supp(t)$ is empty, then $\pi.t = t$ for all π. The definition of Σ-substitution implies that for every θ and for every $x \in \Sigma$, $\theta(x)$ has empty support. Therefore Σ-substitutions and permutations commute, that is, $(\pi.t)\theta = \pi.(t\theta)$.

A sequent in LG^ω is an expression of the form $\Sigma; \Gamma \vdash C$ where Σ is a signature and the formulas in $\Gamma \cup \{C\}$ are in $\beta\eta$-normal form. The free variables of Γ and C are among the variables in Σ. The inference rules for the core fragment of LG^ω, i.e., the logic LG, are given in Figure 1.

In the $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$ rules, a denotes a nominal constant. In the $\exists\mathcal{L}$ and $\forall\mathcal{R}$ rules, we use *raising* [14] to encode the dependency of the quantified variable on the support of B, since we do not allow Σ-substitutions to mention any nominal constants. In the rules, the variable h has its type raised in the following way: suppose $\vec{c}$ is the list $c_1 : \iota_1, \ldots, c_n : \iota_n$ and the quantified variable x is of type τ. Then the variable h is of type: $\iota_1 \to \iota_2 \to \ldots \to \iota_n \to \tau$. This raising technique is similar to that of $FO\lambda^{\Delta\nabla}$, and is used to encode explicitly the minimal support of the quantified variable. Its use prevents one from mixing the scopes of $\forall$ (dually, $\exists$) and ∇. That is, it prevents the formula $\forall x \nabla y. p\, x\, y \equiv \nabla y \forall x. p\, x\, y$, and its dual, to be proved.

Looking at the introduction rules for $\forall$ and $\exists$, one might notice the asymmetry between the left and the right introduction rules. The left rule for $\forall$ allows instantiations with terms containing any nominal constants while the raised variable in the right introduction rule of $\forall$ takes into account only those which are in the support of the quantified formula. However, we will see that we can extend the dependency of the raised variable to an arbitrary number of fresh nominal constants not in the support without affecting the provability of the sequent (see Lemma 3.5 and Lemma 3.6).

We now extend the logic LG with a proof theoretic notion of equality and fixed points, following on works by Hallnas and Schroeder-Heister [7,23], Girard [6] and McDowell and Miller [10]. The equality rules are as follows:

$$\frac{\{\Sigma\theta; \Gamma\theta \vdash C\theta \mid (\lambda\vec{c}.t)\theta =_{\beta\eta} (\lambda\vec{c}.s)\theta\}}{\Sigma; \Gamma, s = t \vdash C}\ \mathrm{eq}\mathcal{L} \qquad \frac{}{\Sigma; \Gamma \vdash t = t}\ \mathrm{eq}\mathcal{R}$$

where $supp(s = t) = \{\vec{c}\}$ and θ is a Σ-substitution in the eq$\mathcal{L}$ rule. In the eq$\mathcal{L}$ rule,

$$\frac{\pi.B = \pi'.B'}{\Sigma;\Gamma, B \vdash B'}\ id_\pi \qquad \frac{\Sigma;\Gamma \vdash B \quad \Sigma; B, \Delta \vdash C}{\Sigma;\Gamma,\Delta \vdash C}\ cut \qquad \frac{\Sigma;\Gamma, B, B \vdash C}{\Sigma;\Gamma, B \vdash C}\ c\mathcal{L}$$

$$\frac{}{\Sigma;\Gamma,\bot \vdash C}\ \bot\mathcal{L} \qquad \frac{}{\Sigma;\Gamma \vdash \top}\ \top\mathcal{R}$$

$$\frac{\Sigma;\Gamma, B_i \vdash C}{\Sigma;\Gamma, B_1 \wedge B_2 \vdash C}\ \wedge\mathcal{L}, i \in \{1,2\} \qquad \frac{\Sigma;\Gamma \vdash B \quad \Sigma;\Gamma \vdash C}{\Sigma;\Gamma \vdash B \wedge C}\ \wedge\mathcal{R}$$

$$\frac{\Sigma;\Gamma, B \vdash C \quad \Sigma;\Gamma, D \vdash C}{\Sigma;\Gamma, B \vee D \vdash C}\ \vee\mathcal{L} \qquad \frac{\Sigma;\Gamma \vdash B_i}{\Sigma;\Gamma \vdash B_1 \vee B_2}\ \vee\mathcal{R}, i \in \{1,2\}$$

$$\frac{\Sigma;\Gamma \vdash B \quad \Sigma;\Gamma, D \vdash C}{\Sigma;\Gamma, B \supset D \vdash C}\ \supset\mathcal{L} \qquad \frac{\Sigma;\Gamma, B \vdash C}{\Sigma;\Gamma \vdash B \supset C}\ \supset\mathcal{R}$$

$$\frac{\Sigma, \mathcal{K}, \mathcal{C}_\mathcal{N} \vdash t : \tau \quad \Sigma;\Gamma, B[t/x] \vdash C}{\Sigma;\Gamma, \forall_\tau x.B \vdash C}\ \forall\mathcal{L} \qquad \frac{\Sigma, h;\Gamma \vdash B[h\,\vec{c}/x]}{\Sigma;\Gamma \vdash \forall x.B}\ \forall\mathcal{R}, h \notin \Sigma, supp(B) = \{\vec{c}\}$$

$$\frac{\Sigma;\Gamma, B[a/x] \vdash C}{\Sigma;\Gamma, \nabla x.B \vdash C}\ \nabla\mathcal{L}, a \notin supp(B) \qquad \frac{\Sigma;\Gamma \vdash B[a/x]}{\Sigma;\Gamma \vdash \nabla x.B}\ \nabla\mathcal{R}, a \notin supp(B)$$

$$\frac{\Sigma, h;\Gamma, B[h\,\vec{c}/x] \vdash C}{\Sigma;\Gamma, \exists x.B \vdash C}\ \exists\mathcal{L}, h \notin \Sigma, supp(B) = \{\vec{c}\} \qquad \frac{\Sigma, \mathcal{K}, \mathcal{C}_\mathcal{N} \vdash t : \tau \quad \Sigma;\Gamma \vdash B[t/x]}{\Sigma;\Gamma \vdash \exists_\tau x.B}\ \exists\mathcal{R}$$

Fig. 1. The inference rules of LG

the substitution θ is a *unifier* of $\lambda\vec{c}.s$ and $\lambda\vec{c}.t$. Note that the λ-abstraction on $\vec{c}$ in eq$\mathcal{L}$ is quite redundant, since Σ-substitutions cannot mention nominal constants and it will be equally valid to say that θ is a unifier of t and s. The use of λ's in the rule is just to make it clear that the unification problem that arises in the rule is the usual higher-order unification and to conform with the formulations of equality rules in Linc [18,25].

We specify the premise of the rule as a set to mean that every element of the set is a premise. Since the terms s and t can be arbitrary higher-order terms, in general the set of their unifiers can be infinite. However, in some restricted cases, e.g., when $\lambda\vec{c}.s$ and $\lambda\vec{c}.t$ are *higher-order pattern* terms [13,19], if both terms are unifiable, then there exists a most general unifier. The applications we are considering are those which satisfy the higher-order pattern restrictions.

Definition 2.2 To each atomic formula, we associate a fixed point equation, or a *definition clause*, following the terminology of $FO\lambda^{\Delta\nabla}$. A definition clause is written $\forall\vec{x}.p\,\vec{x} \stackrel{\triangle}{=} B$ where the free variables of B are among $\vec{x}$. The predicate $p\,\vec{x}$ is called the *head* of the definition clause, and B is called the *body*. A *definition* is a set of definition clauses. We often omit the outer quantifiers when referring to a definition clause.

The introduction rules for defined atoms are as follows:

$$\frac{\Sigma;\Gamma, B[\vec{t}/\vec{x}] \vdash C}{\Sigma;\Gamma, p\,\vec{t} \vdash C}\ def\mathcal{L}, p\,\vec{x} \stackrel{\triangle}{=} B \qquad \frac{\Sigma;\Gamma \vdash B[\vec{t}/\vec{x}]}{\Sigma;\Gamma \vdash p\,\vec{t}}\ def\mathcal{R}, p\,\vec{x} \stackrel{\triangle}{=} B$$

In order to prove the cut-elimination theorem and the consistency of LG^ω, we allow only definition clauses which satisfy an *equivariance preserving* condition and a certain positivity condition, so as to guarantee the existence of fixed points.

Definition 2.3 We associate with each predicate symbol p a natural number, the *level* of p. Given a formula B, its *level* $lvl(B)$ is defined as follows:

(i) $lvl(p\,\bar{t}) = lvl(p)$

(ii) $lvl(\bot) = lvl(\top) = 0$

(iii) $lvl(B \wedge C) = lvl(B \vee C) = \max(lvl(B), lvl(C))$

(iv) $lvl(B \supset C) = \max(lvl(B) + 1, lvl(C))$

(v) $lvl(\forall x.B) = lvl(\nabla x.B) = lvl(\exists x.B) = lvl(B)$.

A definition clause $p\,\vec{x} \stackrel{\triangle}{=} B$ is stratified if $lvl(B) \leq lvl(p)$ and $supp(B) = \emptyset$. We consider only definition clauses which are stratified.

An example that violates the first restriction in Definition 2.3 is the definition $p \stackrel{\triangle}{=} p \supset \bot$. In [23], Schroeder-Heister shows that admitting this definition in a logic with contraction leads to inconsistency. To see why we need the second restriction on name constants, consider the definition $q\,x \stackrel{\triangle}{=} (x = a)$, where a is a nominal constant. Let b be a nominal constant different from a. Then $q\,b$ is both true, since it is equivariant to $q\,a$ and false, by the definition of fixed point.

In examples and applications, we often express definition clauses with patterns in the heads. Let us consider, for example, a definition clause for lists. We first introduce a type lst to denote lists of elements of type α, and the constants

$$nil : lst \qquad :: \;\; : \alpha \to lst \to lst$$

which denote the empty list and a constructor to build a list from an element of type α and another list. The latter will be written in the infix notation. The definition clause for *lists* is as follows.

$$list\ L \stackrel{\triangle}{=} L = nil \vee \exists_\alpha A \exists_{lst} L'.L = (A :: L') \wedge list\ L'.$$

Using patterns, the above definition of lists can be rewritten as

$$list\ nil \stackrel{\triangle}{=} \top. \qquad list\ (A :: L) \stackrel{\triangle}{=} list\ L.$$

We shall often work directly with this patterned notation for definition clauses. For this purpose, we introduce the notion of *patterned definitions*. A *patterned definition clause* is written $\forall \vec{x}.H \stackrel{\triangle}{=} B$ where the free variables of H and B are among $\vec{x}$. The stratification of definitions in Definition 2.3 applies to patterned definitions as well. Since the patterned definition clauses are not allowed to have free occurrences of nominal constants, in matching the heads of the clauses with an atomic formula in a sequent, we need to raise the variables of the clauses to account for nominal constants that are in the support of the introduced formula. Given a patterned definition clause $\forall x_1 \dots \forall x_n.H \stackrel{\triangle}{=} B$ its raised clause with respect to the list of constants $c_1 : \iota_1 \dots c_n : \iota_n$ is

$$\forall h_1 \dots \forall h_n.H[h_1\ \vec{c}/x_1, \dots, h_n\ \vec{c}/x_n] \stackrel{\triangle}{=} B[h_1\ \vec{c}/x_1, \dots, h_n\ \vec{c}/x_n].$$

The introduction rules for patterned definitions are

$$\frac{\{\Sigma\theta; B\theta, \Gamma\theta \longmapsto C\theta\}_\theta}{\Sigma; A, \Gamma \longmapsto C}\ \mathit{def}\mathcal{L} \qquad \frac{\Sigma; \Gamma \longmapsto B\theta}{\Sigma; \Gamma \longmapsto A}\ \mathit{def}\mathcal{R}$$

In the *def*$\mathcal{L}$ rule, B is the body of the raised patterned clause $\forall x_1 \ldots \forall x_n.H \overset{\triangle}{=} B$ and $(\lambda\vec{c}.H)\theta = (\lambda\vec{c}.A)\theta$ where $\{\vec{c}\}$ is the support of A. In the *def*$\mathcal{R}$ rule, we match A with the head of the clause, i.e., $\lambda\vec{c}.A = (\lambda\vec{c}.H)\theta$. These patterned rules can be derived using the non-patterned definition rules and the equality rules, as shown in [25].

Natural number induction.

We introduce a type *nt* to denote natural numbers, with the usual constants $z : nt$ (zero) and $s : nt \to nt$ (the successor function), and a special predicate $nat : nt \to o$. The rules for natural number induction are the same as those in $FO\lambda^{\Delta\mathbb{N}}$ [10], which are the introduction rules for the predicate *nat*.

$$\frac{\longmapsto D\,z \quad j; D\,j \longmapsto D\,(s\,j) \quad \Sigma; \Gamma, D\,I \longmapsto C}{\Sigma; \Gamma, nat\,I \longmapsto C}\ nat\mathcal{L}$$

$$\frac{}{\Sigma; \Gamma \longmapsto nat\ z}\ nat\mathcal{R} \qquad \frac{\Sigma; \Gamma \longmapsto nat\,I}{\Sigma; \Gamma \longmapsto nat\,(s\,I)}\ nat\mathcal{R}$$

The logic *LG* extended with the equality, definitions and induction rules is referred to as LG^ω.

3 The meta theory of LG^ω

In this section we investigate some properties of the logic LG^ω. We first look at the properties of the ∇ quantifier in relation to other connectives. The proof of the following proposition is straightforward by inspection on the rules of *LG*.

Proposition 3.1 *The following formulas are provable in LG:*

(i) $\nabla x.(Bx \wedge Cx) \equiv \nabla x.Bx \wedge \nabla x.Cx$.

(ii) $\nabla x.(Bx \supset Cx) \equiv \nabla x.Bx \supset \nabla x.Cx$.

(iii) $\nabla x.(Bx \vee Cx) \equiv \nabla x.Bx \vee \nabla x.Cx$.

(iv) $\nabla x.B \equiv B$, *provided that x is not free in B.*

(v) $\nabla x \nabla y.Bxy \equiv \nabla y \nabla x.Bxy$.

(vi) $\forall x.Bx \supset \nabla x.Bx$.

(vii) $\nabla x.Bx \supset \exists x.Bx$.

The formulas (i) – (iii) are provable in $FO\lambda^\nabla$. The proposition is true also in nominal logic with ∇ replaced by И.

The following properties concern the transformation of derivations. Provability is preserved under Σ-substitutions, permutations and a restricted form of name substitutions.

Lemma 3.2 Substitutions. *Let Π be a proof of $\Sigma; \Gamma \vdash C$ and let θ be a Σ-substitution. Then there exists a proof Π' of $\Sigma\theta; \Gamma\theta \vdash C\theta$.*

Lemma 3.3 Permutations. *Let Π be a proof of $\Sigma; B_1, \dots, B_n \vdash B_0$. Then there exists a proof Π' of $\Sigma; \pi_1.B_1, \dots, \pi_n.B_n \vdash \pi_0.B_0$.*

Lemma 3.4 Restricted name substitutions. *Let Π be a proof of*

$$\Sigma, x : \iota; B_1, \dots, B_n \vdash B_0.$$

Then there exists a proof of Π' of $\Sigma; B_1[a_1/x], \dots, B_n[a_n/x] \vdash B_0[a_0/x]$, where $a_i \notin supp(B_i)$ for each $i \in \{0, \dots, n\}$.

The next two lemmas are crucial to the cut-elimination proof: they allow one to reintroduce the symmetry between $\forall\mathcal{L}$ and $\forall\mathcal{R}$, and dually, between $\exists\mathcal{L}$ and $\exists\mathcal{R}$ rules.

Lemma 3.5 Support extension. *Let Π be a proof of $\Sigma, h; \Gamma \vdash B[h\ \vec{a}/x]$ where $\{\vec{a}\} = supp(B)$, $h \notin \Sigma$ and h is not free in Γ and B. Let $\vec{c}$ be a finite list of nominal constants not in the support of B. Then there exists a proof Π' of $\Sigma, h'; \Gamma \vdash B[h'\ \vec{a}\vec{c}/x]$.*

Lemma 3.6 Support extension. *Let Π be a proof of $\Sigma, h; B[h\ \vec{a}/x], \Gamma \vdash C$ where $\{\vec{a}\} = supp(B)$, $h \notin \Sigma$ and h is not free in Γ, B and C. Let $\vec{c}$ be a finite list of nominal constants not in the support of B. Then there exists a proof Π' of $\Sigma, h'; B[h'\ \vec{a}\vec{c}/x], \Gamma \vdash C$ where $h' \notin \Sigma$.*

The main result on the meta theory of LG^ω is the cut-elimination theorem, from which the consistency of the logic follows.

Theorem 3.7 *The cut rule is admissible in LG^ω.*

Corollary 3.8 *The logic LG^ω is consistent, i.e., it is not the case that both A and $A \supset \bot$ are provable.*

Finally, we show that the formulation of LG is equivalent to $FO\lambda^\nabla$ extended with the axiom schemes of name permutations and weakening.

Theorem 3.9 *Let F be a formula which contains no occurrences of nominal constants. Then F is provable in $FO\lambda^\nabla$ extended with the axiom schemes $B \equiv \nabla x.B$ and $\nabla x \nabla y.B\,x\,y \supset \nabla y \nabla x.B\,x\,y$ if and only if F is provable in LG.*

4 Encoding an object logic

We now consider an encoding of the logic HH mentioned in the introduction in LG^ω. The encoding of this object logic has been done in $FO\lambda^{\Delta\mathbb{N}}$ by McDowell and

$$\begin{array}{lcl}
seq_I\ L\ tt & \stackrel{\triangle}{=} & \top. \\
seq_I\ L\ \langle A \rangle & \stackrel{\triangle}{=} & elem\ A\ L. \\
seq_{(s\,I)}\ L\ (A \,\&\, B) & \stackrel{\triangle}{=} & seq_I\ L\ A \wedge seq_I\ L\ B. \\
seq_{(s\,I)}\ L\ (A \Rightarrow B) & \stackrel{\triangle}{=} & seq_I\ (A :: L)\ B. \\
seq_{(s\,I)}\ L\ (\bigwedge x.Gx) & \stackrel{\triangle}{=} & \nabla x.seq_I\ L\ Gx. \\
seq_{(s\,I)}\ L\ \bigvee x.Gx & \stackrel{\triangle}{=} & \exists x.seq_I\ L\ Gx. \\
seq_{(s\,I)}\ L\ \langle A \rangle & \stackrel{\triangle}{=} & \exists B.prog\ A\ B \wedge seq_I\ L\ B.
\end{array}$$

Fig. 2. Definition of an object logic.

Miller [11]. The formalization of the object logic properties in this section follows closely the $FO\lambda^{\Delta\mathbb{N}}$ encoding. The only major difference is that we do not need an explicit encoding of eigenvariables; eigenvariables are mapped to nominal constants in the meta logic LG^{ω}.

The object logic formulas are generated by the following grammar.

$$\begin{array}{l}
D ::= A \mid G \Rightarrow A \mid \bigwedge_{\tau} x.D \\
G ::= A \mid tt \mid G \,\&\, G \mid A \Rightarrow G \mid \bigwedge_{\iota} x.G \mid \bigvee_{\tau} .G
\end{array}$$

where A ranges over atomic (object-level) formula, $\Rightarrow$, $\&$, $\bigwedge$ and $\bigvee$ denote implication, conjunction, universal quantifier and existential quantifier, respectively. D and G represent definite clauses and goal formulas, respectively. Notice that in goal formulas, universal quantification is restricted to nominal types. The sequent rules for HH are the standard right introduction rules for the logical connectives plus the *backchaining* rule:

$$\frac{\Gamma, \bigwedge \vec{x}.G \supset A \longrightarrow G\theta}{\Gamma, \bigwedge \vec{x}.G \supset A \longrightarrow A'}\ bc, A\theta = A'$$

This sequent system is complete for the HH fragment of intuitionistic logic, as a consequence of *uniform provability* of intuitionistic logic [15].

In order to encode the object-logic formulas into LG^{ω}, we first introduce some types and constants. The object logic formulas are given the type *prp*, while atomic formulas are given the type *atm*. The formulas of HH are encoded using the following constants:

$$\begin{array}{lll}
\langle\,\rangle : atm \to prp & \& : prp \to prp \to prp & \bigwedge_{\tau} : (\iota \to prp) \to prp \\
tt : prp & \Rightarrow : atm \to prp \to prp & \bigvee_{\tau} : (\tau \to prp) \to prp
\end{array}$$

We denote the encoding of an object level formula A in LG^{ω} with $[\![A]\!]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{X, I_2; seq_{I_2}\ [p\ X]\ \langle p\ a\rangle \vdash \bot}{X, I_2; \nabla y.seq_{I_2}\ [p\ X]\ \langle p\ y\rangle \vdash \bot}\,\nabla\mathcal{L}}{X, I_1; seq_{I_1}\ [p\ X]\ (\bigwedge y.\langle p\ y\rangle) \vdash \bot}\,def\mathcal{L}}{X, I; seq_I\ nil\ (p\ X \Rightarrow \bigwedge y.\langle p\ y\rangle) \vdash \bot}\,def\mathcal{L}}{\vdash \forall X \forall I.(seq_I\ nil\ (p\ X \Rightarrow \bigwedge y.\langle p\ y\rangle) \supset \bot)}\,\forall\mathcal{R}; \supset\mathcal{R}$$

Fig. 3. A derivation in LG^ω.

Since the set of definite clauses in the sequents does not change in the proofs in HH, we will not put them explicitly in the HH sequents in their encoding in LG^ω. Hence hypotheses of HH sequents are lists of atomic formulas. The object logic sequent is represented using the predicate $seq : nt \to atmlist \to prp \to o$ where $atmlist$ is the type for lists of atomic formulas, with the usual constructors nil and $::$. The natural number in the encoding of sequents will be used as a measure of the length of object logic proofs. Inductive properties about the provability in HH will be proved using this measure. An object sequent $\Gamma \longrightarrow A$ is represented as the atomic formula $(seq_I\ [\![\Gamma]\!]\ [\![A]\!])$ in LG^ω. We encode definite clauses using a predicate called $prog : atm \to prp \to o$. A definite clause $\bigwedge \vec{x}.G \Rightarrow A$ is encoded as the definition clause $\forall \vec{x}.prog\ A\ G \stackrel{\triangle}{=} \top$. The patterned definition of the sequent rules of HH is given in Figure 2. It uses the following definition clauses.

$$\begin{array}{llll}
list_i\ nil & \stackrel{\triangle}{=} \top. & list_{(s\ i)}\ (A :: L) & \stackrel{\triangle}{=} list_i\ L. \\
list\ L & \stackrel{\triangle}{=} \exists i.nat\ i \wedge list_i\ L. & & \\
elem\ A\ (A :: L) & \stackrel{\triangle}{=} \top. & elem\ A\ (B :: L) & \stackrel{\triangle}{=} elem\ A\ L.
\end{array}$$

We refer to this definition together with the definition in Figure 2 as $\mathcal{D}(HH)$ and any additional definite clauses with $\mathcal{D}(prog)$.

Example:

The formula $p\ X \Rightarrow \bigwedge y.p\ y$ is not provable in the empty theory, whatever the value of X is. This fact is formalized in LG^ω as the formula

$$\forall X \forall I.(seq_I\ nil\ (p\ X \Rightarrow \bigwedge y.\langle p\ y\rangle) \supset \bot).$$

A partial derivation of this formula in LG^ω is shown in Figure 3. In the figure the notation $[p\,X]$ stands for the list $(p\,X\ ::\ nil)$. The derivation is completed by applying $def\mathcal{L}$ to the topmost sequent, resulting in two matching cases: the identity rule and the backchaining rule. Since we assume no definite clauses, this leaves us with proving the sequent: $X, I_2; elem\ (p\ X)\ (p\ a :: nil) \vdash \bot$. Applying $def\mathcal{L}$ to this sequent results in the sequent $X, I_2; elem\ (p\ X)\ nil \vdash \bot$, since $\lambda a.p\ X$ and $\lambda a.p\ a$ are not unifiable. Another application of $def\mathcal{L}$ gives us empty premise and hence the sequent is provable. □

It is straightforward to see that the structure of the HH proofs corresponds to the structure of proofs of its encoding in LG^ω; in particular, the backchaining rule in HH corresponds to the $def\mathcal{R}$ rule (for the patterned definition) in LG^ω. We now state some properties of the encoding of HH in LG^ω.

Theorem 4.1 *Let $\mathcal{D}(prog)$ be a definition corresponding to a set of definite clauses $\mathcal{P}$. Then the sequent $\mathcal{P}, \Gamma \longrightarrow G$ is derivable in HH if and only if $seq_i\ [\![\Gamma]\!]\ [\![G]\!]$ is derivable in LG^ω with the definition $\mathcal{D}(prog) \cup \mathcal{D}(HH)$ for some natural number i.*

Theorem 4.2 *The following formulas are provable in LG^ω with the definition of the object logic HH:*

(i) *Structural rules:* $\forall L \forall L' \forall G \forall i.\, nat\, i \supset list\ L \supset list\ L'$

$$(\forall A.elem\ A\ L \supset elem\ A\ L') \supset seq_i\ L\ G \supset seq_i\ L'\ G.$$

(ii) *Atomic cut:* $\forall L \forall G \forall A.list\ L \supset\ \exists i.(nat\ i \wedge seq_i\ L\ (A \Rightarrow G)) \supset$

$$\exists i.(nat\ i \wedge seq_i\ L\ \langle A \rangle) \supset \exists i.nat\ i \wedge seq_i\ L\ G.$$

(iii) *Specialization:* $\forall L \forall G \forall i.nat\ i \supset list\ L \supset seq_{(s\,i)}\ L\ (\bigwedge G) \supset \forall x.seq_i\ L\ (G\,x).$

We conclude this section by a remark that ∇ is strictly speaking not necessary for capturing object logic provability, as Theorem 4.2 (3) shows, rather it is the use of nominal constants to model eigenvariables that allows that. The use of ∇, however, results in a more natural correspondence between the encoding of HH and its actual sequent proofs.

5 Reasoning about operational semantics

Following McDowell and Miller [11], we use the encoding of HH in LG^ω to specify and reason about the operational semantics of simply typed λ-calculus. Reasoning about more complicated languages like PCF can be done as well using a similar approach (see [11]).

We introduce a type ty to denote object-level types. The type tm denotes the object-level λ-terms and is considered a nominal type. The language of the (object-level) λ-terms is encoded using the following constants:

$$app : tm \to tm \to tm \qquad abs : ty \to (tm \to tm) \to tm$$

which denote application and abstraction, respectively. The object-level type constructor, i.e., the ‘arrow’, is encoded via the constant $ar : ty \to ty \to ty$. Object-level base types are ranged over by α.

The evaluation relation and the typing judgments of the simply typed calculus are given as definite clauses below.

$$eval\ (abs\ T\ M)\ (abs\ T\ M) \Leftarrow tt.$$

$$eval\ (app\ M\ N)\ V \Leftarrow \bigvee P \bigvee T. eval\ M\ (abs\ P\ T)\ \&\ eval\ (P\ N)\ V.$$

$$typeof\ (abs\ T\ M)\ (ar\ T\ T') \Leftarrow \bigwedge x. typeof\ x\ T \Rightarrow typeof\ (Mx)\ T'.$$

$$typeof\ (app\ M\ N)\ T \Leftarrow \bigvee T'. typeof\ M\ (ar\ T'\ T)\ \&\ typeof\ N\ T'.$$

It is straightforward to translate these clauses to *prog* clauses.

We state a couple of properties here as formulas in LG^ω. In the following theorems, we use the notation $L \triangleright G$ to denote the formula $\exists i. nat\ i \wedge seq_i\ L\ G$. If L is *nil* we simply write $\triangleright G$.

Theorem 5.1 Subject reduction. *The following formula is provable*

$$\forall M \forall V \forall T.\ \triangleright \langle eval\ M\ V\rangle \wedge \triangleright\langle typeof\ M\ T\rangle \supset \triangleright\langle typeof\ V\ T\rangle.$$

A proof of a similar theorem is given in [11] for the untyped λ-term in the logic $FO\lambda^{\Delta\mathbb{N}}$. This proof can be adapted straightforwardly to give a proof for the above theorem. A more interesting property is the determinacy of type assignments, provided that the typing context is well-formed, that is, each variable in the context is assigned a unique type. The well-formedness of a typing context L is specified as the formula:

$$\forall X \forall T_1 \forall T_2. elem\ (typeof\ X\ T_1)\ L \supset elem\ (typeof\ X\ T_2)\ L \supset T_1 = T_2.$$

The above formula will be denoted by *ctx L*.

Theorem 5.2 *The following formula is provable:*

$$\forall L \forall X \forall T_1 \forall T_2.\ list\ L \supset ctx\ L \supset L \triangleright \langle typeof\ X\ T_1\rangle \supset L \triangleright \langle typeof\ X\ T_2\rangle \supset T_1 = T_2.$$

6 Related and future work

There have been many previous related works in providing frameworks for higher-order abstract syntax, or more generally abstract syntax with bindings. A non-exhaustive list includes encoding in proof assistants like Coq [4], HOL [27], Isabelle [28], and Twelf [24], categorical frameworks [8], *the theory of contexts* [9], nominal logic [21], and proof search frameworks [11,25]. The approach taken here is similar to the latter; the novelty of our work lies in the use of equivariance principle within the usual style of higher-order abstract syntax specifications. An immediate future work will be to implement the logic LG^ω, possibly on top of an existing proof assistant, and to perform large case studies, in particular, the problem sets put out in the POPLMark Challenge [1].

In the current work we show only the treatment of natural number induction. Extensions to iterated (co-)inductive definitions can be done in a similar way as in [18,25].

Semantics of LG. There have been several attempts at giving a semantics for the logic $FO\lambda^{\nabla}$: Cheney and Gabbay proposed an encoding into nominal logic [5,2], Miculan and Yemane [12] gave a categorical semantics and Schöpp [22] gave a Tarski-style semantics and a categorical semantics for a classical version of $FO\lambda^{\Delta\nabla}$. In these works, it is suggested that extending $FO\lambda^{\nabla}$ with the axiom schemes (3) would result in a natural semantics for ∇. The works by Miculan and Yemane and Schöpp seem closer to the logic LG and could very well serve as a basis for finding a categorical model for LG. There are some similarities between LG and Nominal Logic, but the treatment of substitutions and the addition of closed terms of type name in LG make it not obvious whether the support models of Nominal Logic can be used for LG. We leave the investigation of support models for LG (or a classical version of LG), such as the ones in [21,3], as a future work.

Acknowledgement

The author would like to thank James Cheney for his many helpful remarks and suggestions, in particular those related to Nominal Logic, the anonymous referees for their useful and detailed comments, and also David Baelde, Alberto Momigliano, Michael Norrish and Brigitte Pientka for their comments on earlier drafts of the paper.

References

[1] Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich and S. Zdancewic, *Mechanized metatheory for the masses: the* POPLMARK *challenge*, in: J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, Lecture Notes in Computer Science (2005), pp. 50–65.

[2] Cheney, J., *A simpler proof theory for nominal logic*, in: *Proc. FOSSACS'05*, Lecture Notes in Computer Science **3441** (2005), pp. 379–394.

[3] Cheney, J., *Completeness and Herbrand theorems for nominal logic*, Journal of Symbolic Logic **7** (2006), pp. 299–320.

[4] Despeyroux, J., A. Felty and A. Hirschowitz, *Higher-order abstract syntax in Coq*, in: *Second International Conference on Typed Lambda Calculi and Applications*, 1995, pp. 124–138.

[5] Gabbay, M. J. and J. Cheney, *A sequent calculus for nominal logic*, in: *Proc. 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, 2004, pp. 139–148.

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

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

[8] Hofmann, M., *Semantical analysis of higher-order abstract syntax*, in: *14th Annual Symposium on Logic in Computer Science* (1999), pp. 204–213.

[9] Honsell, F., M. Miculan and I. Scagnetto, *An axiomatic approach to metareasoning on systems in higher-order abstract syntax*, in: *Proc. ICALP'01*, number 2076 in LNCS (2001), pp. 963–978.

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

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

[12] Miculan, M. and K. Yemane, *A unifying model of variables and names*, in: *Proc. FOSSACS'05*, Lecture Notes in Computer Science **3441** (2005), pp. 170 – 186.

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

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

[15] Miller, D., G. Nadathur, F. Pfenning and A. Scedrov, *Uniform proofs as a foundation for logic programming*, Annals of Pure and Applied Logic **51** (1991), pp. 125–157.

[16] Miller, D. and C. Palamidessi, *Foundational aspects of syntax.*, ACM Comput. Surv. **31** (1999), p. 11.

[17] Miller, D. and A. Tiu, *A proof theory for generic judgments*, ACM Trans. Comput. Logic **6** (2005), pp. 749–783.

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

[19] Nipkow, T., *Functional unification of higher-order patterns*, in: M. Vardi, editor, *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS 1993)* (1993), pp. 64–74.

[20] Pfenning, F. and C. Elliott, *Higher-order abstract syntax*, in: *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation* (1988), pp. 199–208.

[21] Pitts, A. M., *Nominal logic, a first order theory of names and binding*, Information and Computation **186** (2003), pp. 165–193.

[22] Schöpp, U., *Modelling generic judgments*, in: *Proceedings of Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, 2006, pp. 1–16.

[23] Schroeder-Heister, P., *Cut-elimination in logics with definitional reflection*, in: D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, LNCS **619** (1992), pp. 146–171.

[24] Schürmann, C., "Automating the Meta Theory of Deductive Systems," Ph.D. thesis, Carnegie Mellon University (2000).

[25] Tiu, A., "A Logical Framework for Reasoning about Logical Specifications," Ph.D. thesis, Pennsylvania State University (2004).

[26] Tiu, A., *A logic for reasoning about generic judgments* (2006), extended version. Available on: http://users.rsise.anu.edu.au/~tiu/lgext.pdf.

[27] Urban, C. and M. Norrish, *A formal treatment of the Barendregt Variable Convention in rule inductions*, in: *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (2005), pp. 25–32.

[28] Urban, C. and C. Tasson, *Nominal techniques in Isabelle/HOL*, in: R. Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE)*, LNCS **3632** (2005), pp. 38–53.