Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 228 (2009) 135–150
www.elsevier.com/locate/entcs

On the Role of Names in Reasoning about λ-tree Syntax Specifications

Alwen Tiu

Computer Sciences Laboratory
The Australian National University
Canberra ACT 0200, Australia
Email: Alwen.Tiu@rsise.anu.edu.au

Abstract

Lambda tree syntax (a variant of HOAS) and nominal techniques are two approaches to representing and reasoning about languages containing bindings. Although they are based on separate foundations, recent advances in the proof theory of generic judgments have shown that one may be able to incorporate some aspects of nominal techniques (i.e., the equivariant principle) to simplify reasoning about λ-tree syntax specifications, while still maintaining the crucial aspects of λ-tree syntax. In this paper, we present a logic, called LGn^{ω}, which incorporates a notion of generic judgments and equivariant reasoning. The logic LGn^{ω} is a simple extension of a logic called LG^{ω} by Tiu, and can be seen as a special case of the logic $\mathcal{G}$ by Gacek, Miller and Nadathur. A central idea of LGn^{ω} is the representation of a data type for names (represented by a predicate). Although the data type is inhabited by infinitely many elements, the judgments of the logic only ever use finitely many of them, and more importantly, validity of these judgments are preserved under arbitrary permutation of names, i.e., they are equivariant judgments. This finite support of judgments allows for tractable introduction rules of the name predicate. We illustrate with two examples how this simple extension can be used for reasoning about specifications involving bindings. In the first example, we show how one can represent the data type for λ-terms, and derive a structural induction principle for inductive reasoning over λ-terms. In the second example, we re-examine previous known encodings of open and late bisimulations for the π-calculus. We show that the difference between open and late bisimulation can be characterized by the choice of the encodings of names: the "untyped" version (for the former) versus the "typed" one (for the latter).

Keywords: generic judgments, nominal techniques, logical framework, pi-calculus, bisimulation

1 Introduction

Lambda-tree syntax [10] is an abstract syntax for representing syntactic structures involving bindings using Church's simply typed λ-calculus. It is a weak variant of higher-order abstract syntax (HOAS) and shares two main aspects of the HOAS encoding style, namely, the use of λ-abstraction to represent bindings in object-level syntactic structures, and the use of applications to represent object-level substitutions. The representation language in λ-tree syntax is intentionally weak, so as to avoid certain problems with adequacy of representations. To compensate for this weakness in representation, reasoning about properties of the encodings are

doi:10.1016/j.entcs.2008.12.122

delegated to a meta-logic (typically first-order or higher-order logics), whose term structures are those of the simply typed λ-calculus.

In reasoning about λ-tree syntax specifications of object systems, say, type systems for functional languages, one is often faced with the problem of representing object-level typing contexts faithfully in the meta level. Such a representation requires an interpretation of object-level "names" or "variables". The roles of these names or variables, in the object level, are often confined to providing distinct identifiers. Thus, when encoding the typing judgments at the meta level, one may want to ensure that this notion of distinctness is represented faithfully. This is one of the motivations behind the design of the proof theory for generic judgments [12,21], on which we base our meta logic designs.

One observation that one can make about many of the operational semantics of modern programming languages and type systems is that most of the judgments of interest, such as typing judgments, evaluations, bisimulation, etc., are invariant under injective variable renaming, and more specifically, under arbitrary permutations of the variables. In other words, these judgments are *equivariant judgments*. This notion of equivariance is first formalised by Gabbay and Pitts [6] using FM-sets, and is later given a first-order axiomatizations by Pitts [15], resulting in an extension of first-order logic called nominal logic. It is adopted into a meta logic based on λ-tree syntax, called LG^ω, in [22]. To capture this notion of equivariance in LG^ω, we introduce a set of base types, called *nominal types*, each of which is inhabited by infinitely many constants, called *nominal constants* (or just *names*). The role of these constants is to enforce the equivariant principle. The meta logic is designed in a way such that provability of judgments is invariant under arbitrary permutation of names. Names in the meta logic have a similar role to eigenvariables; they act as proof level binders for a new quantifier, called ∇.

The logic LG^ω was introduced as a first attempt to address a certain gap in inductive reasoning involving generic judgments in its predecessor, the logic LINC [21], which also features the ∇-quantifier. In LINC there is no interplay between the induction rule and the variable context surrounding the judgment being proved. This makes it unsuitable for certain applications, such as reasoning about type systems, which heavily involves reasoning about variables in typing contexts. Although LG^ω is more expressive than LINC (for reasoning about bindings), the gap is essentially still there. This gap is finally closed in the recently introduced logic $\mathcal{G}$ by Gacek, Miller and Nadathur [4]. The logic $\mathcal{G}$ makes a simple, but crucial, extension to LG^ω: it allows one to specify a fixed point definition which carries around a variable context, encoded via ∇. This extension is surprisingly powerful, allowing one to specify and reason, quite naturally, about a variety of properties not previously expressible directly in LINC or LG^ω (see [5]). A main motivation for the present work is to understand better the expressivity of $\mathcal{G}$ in a minimal setting, that is, the minimal extension to LG^ω that is still powerful enough to reason about inductive properties of λ-tree syntax specifications. The minimal extension proposed here, called LGn^ω, is essentially $\mathcal{G}$ with only one extended fixed point definition, denoting a datatype for names. This extension is also motivated by its applications in formalising the

π-calculus [11], in which the notion of names plays a central role in its meta theory. As it turns out, we can still do a variety of reasoning tasks that seem to be the core novel features of $\mathcal{G}$ using LGn^ω (see Section 5 for a discussion).

Although LGn^ω is motivated by a specialization of $\mathcal{G}$, we follow a different approach to the design of the logic. Instead of using the extended fixed point definition as in $\mathcal{G}$ to define a datatype for names, we take as primitive a notion of datatype for names that arises naturally from its intended interpretation. We introduce a built-in predicate called *name* of single arity, which recognizes its argument as being a name. That is, the proposition *name* a is derivable in the logic if and only if a is a nominal constant. Since we assume an infinite set of names (nominal constants), the case analysis rule for this datatype is simply an enumeration of all possible nominal constants. In a simplified form, this rule can be presented as an infinite branching rule:

$$\frac{\Gamma \Longrightarrow P\ a_1 \quad \Gamma \Longrightarrow P\ a_2 \quad \cdots \quad \Gamma \Longrightarrow P\ a_n \quad \cdots}{\Gamma \Longrightarrow \forall x.name\ x \supset P\ x}$$

However, thanks to the equivariant property of the logic, in using the rule, we only need to consider a finite number of cases: those in which the name x is instantiated to one which occurs already in P, plus an additional one where the name is "fresh" with respect to P. It thus allows us to derive a version of the rule which uses only a finite number of names, similar to that of $\mathcal{G}$, while at the same time makes it easier for us to establish the meta theory of LGn^ω using the simpler, infinitary rule.

We show with a couple of examples how the predicate *name* can be used. In the first example (Section 3), we show how one can encode the data type for untyped λ-terms, along with a structural induction principle. We define some standard relations on λ-terms, namely, the notion of freshness of a name with respect to a term, abstraction of a name from a term and substitutions of a name in a term with another term. We then prove some properties about these relations that show: (a) the notion of freshness is implicitly supported by meta-level scoping restrictions and the ∇ quantifier, and (b) meta-level applications co-incide with the inductively defined substitution predicate. In the second example (Section 4), we re-examine a previous known encoding of the π-calculus [24] and the notions open and late bisimulations [20]. As shown in [24], the difference between open and late bisimulation can be characterized by the presence or absence of the assumption about the decidability of name equality, i.e.,

$$\forall x \forall y.name\ x \supset name\ y \supset x = y \vee x \neq y.$$

Name equality happens to be decidable in LGn^ω. Therefore we obtain a new, simpler characterization of the difference between open and late bisimulation: In late bisimulation, names are typed, whereas in open bisimulation, names are untyped.

Some of the proofs are omitted from the main text and can be found in the appendix of an extended version of the paper. The proofs concerning the examples in Section 3 have been mechanically verified in the proof assistant Abella [2] and

can be found on the web. [1]

2 The logic LGn^ω

In this section, we present the proof system for LGn^ω. Since LGn^ω is just a small extension of LG^ω, a significant part of this section is an overview of LG^ω. LGn^ω is based on a subset of Church's Simple Theory of Types. Following Church, we designate a special type o to denote formulas. The core fragment of LGn^ω, shares the same set of connectives as LG^ω, 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, denoted by $\mathcal{C}_\iota$. These constants are called *nominal constants*. We denote the family of nominal constants by $\mathcal{C}_\mathcal{N}$. Provability of formulas in LGn^ω is invariant under permutations of nominal constants. The set of non-nominal constants is denoted by $\mathcal{K}$.

We assume the usual notion of capture-avoiding and type preserving substitutions. Substitutions are ranged over by θ, σ 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 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. We denote by $\Sigma\theta$ the signature obtained by replacing every $x \in \Sigma$ with the free variables in $\theta(x)$.

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:

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

Notice that the permutation action on variables is an identity. Implicit in this definition is the assumption that variables always have empty support. This design feature allows us to omit explicit representation of permutations at the term level. Dependency of a variable on names can be encoded via a technique called *raising*, to be explained shortly.

[1] URL: http://rsise.anu.edu.au/~tiu/papers/names.thm

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

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

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

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

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

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

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

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

Fig. 1. The core inference rules of LGn^ω.

$$\frac{\{\Sigma\theta; \Gamma\theta \Longrightarrow C\theta \ \mid\ t\theta = s\theta,\ supp(\theta) = \emptyset\}}{\Sigma; \Gamma, s = t \Longrightarrow C}\ \mathrm{eq}\mathcal{L} \qquad \frac{}{\Sigma; \Gamma \Longrightarrow t = t}\ \mathrm{eq}\mathcal{R}$$

$$\frac{\{\Sigma\theta; \Gamma\theta \Longrightarrow C\theta \mid\ t\theta \in \mathcal{C}_\iota \text{ and } supp(t, \Gamma, C)\#\theta\}}{\Sigma; name_\iota\ t, \Gamma \Longrightarrow C}\ name\mathcal{L} \qquad \frac{a \in \mathcal{C}_\iota}{\Sigma; \Gamma \Longrightarrow name_\iota\ a}\ name\mathcal{R}$$

Fig. 2. The inference rules for equality and names.

The *support* of a term (or formula) t, written $supp(t)$, is the set of nominal constants appearing in it. The support of a substitution, written $supp(\theta)$ is the set of nominal constants appearing in the range of the substitutions. A substitution is a *closed substitution* if its support is empty. Given a list of nominal constant $\vec{c}$ and a term t, we say that $\vec{c}$ *is fresh for* t, written $\vec{c}\#t$, if $\{\vec{c}\} \cap supp(t) = \emptyset$. Similarly, given a list of nominal constants $\vec{c}$ and a substitution θ, we say that $\vec{c}$ *is fresh for* θ, written $\vec{c}\#\theta$, if $supp(\theta) \cap \{\vec{c}\} = \emptyset$.

A sequent in LGn^ω is an expression of the form $\Sigma; \Gamma \Longrightarrow C$ 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 LGn^ω 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* [8] to encode the dependency of the quantified variable on the support of B. 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: $\iota_1 \to \iota_2 \to \ldots \to \iota_n \to \tau$. Raising is used to encode explicitly the minimal support of the quantified variable. As we shall see later, provability is preserved under support extensions.

We now extend the core logic with a proof theoretic notion of names, equality,

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

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

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

Fig. 3. Fixed points and induction

fixed points and natural number induction. The latter three are the same as in LG^{ω}. The rules for fixed points are the standard ones, and have been considered in many previous work [7,19,3,9]. We first look at the equality rules, given in Figure 2. In eq$\mathcal{L}$, we specify the premise of the rule as a set to mean that every element of the set is a premise. What the rule does, reading it bottom-up, is essentially computing a set of *unifiers* for s and t. Notice that the substitution θ is a closed substitution, therefore solvability of the equation $s = t$ is the same as solvability of the equation: $\lambda\vec{c}.s =_{\beta\eta} \lambda\vec{c}.t$, where $\vec{c}$ is the support of $s = t$, and θ can be computed using standard higher-order unification algorithms.

The datatype for names is encoded via a family of special predicates $name_{\iota} : \iota \to o$. We shall often omit the subscript ι in $name_{\iota}$ when the type ι is not important or when it can be inferred from context. The introduction rules for *name* are given in Figure 2. The right introduction rule simply recognizes that a constant belongs to the set of nominal constants. The more interesting rule is *name*$\mathcal{L}$, which considers all possible substitutions to a term t such that the resulting term $t\theta$ is a nominal constant. If t is headed with a non-nominal constant, then the rule simply produces an empty premise, in which case the lower sequent is proved trivially. Notice that in *name*$\mathcal{L}$ we allow substitutions that mention nominal constants, as long as these constants are fresh with respect to the conclusions. The rule can be infinitary, since the set of nominal constants is infinite. We shall see later that it can be restricted to a version which uses only a finite number of names.

We now introduce a proof theoretic notion of *definitions*.

Definition 2.2 To each atomic formula, we associate a fixed point equation, or a *definition clause*. 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.

We adopt a style of definitions with no patterns in the heads, but as it has been shown in [21], allowing patterns in the head does not add any expressive power, and both styles of definitions are interchangable in the presence of equality. However, when we discuss examples, we shall use patterned definitions. The introduction rules for defined atoms are given in Figure 3. Certain monotonicity restrictions need to be imposed on definition clauses so as to guarantee cut elimination. These restritions are the same as the ones for LG^{ω} (see [22] for details).

The rules for natural numbers are given in Figure 3. 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 $nat : nt \to o$. These rules are the same as those in $FO\lambda^{\Delta\mathbb{N}}$ [9]. In $nat\mathcal{L}$, we restrict the invariant D to a closed term such that $supp(D) = \emptyset$. We do not gain any expressive power by allowing nominal constants in D, since these constants can be introduced via the ∇ quantifier.

The cut elimination proof for LGn^ω follows much of the cut elimination proof of LG^ω[23]. One subtle difference is in the proof transformation involving substitutions, stated in the following proposition.

Proposition 2.3 *Let Π be a derivation of the sequent $\Sigma; \Gamma \Longrightarrow C$ in LGn^ω. Let $\vec{c}$ be the list of nominal constants occuring in Γ and C. Let θ be a substitution with such that $\vec{c}\#\theta$. Then there exists a derivation Π' of $\Sigma\theta; \Gamma\theta \Longrightarrow C\theta$ in LGn^ω such that the height of Π' is less or equal to the height of Π.*

Note that, unlike, LG^ω, proof-level substitutions can mention nominal constants, as long as they are fresh with respect to the sequents in the proofs.

Theorem 2.4 *Cut elimination holds for LGn^ω.*

A version of $name\mathcal{L}$ with finite names

The rule $name\mathcal{L}$ as given in Figure 2 can have infinitely many premises. For example, applying the rule to a sequent like

$$x : \iota; name\ x, \Gamma(x) \Longrightarrow C(x)$$

results in infinitely many premises, each of which replaces the variable x with a name $a \in \mathcal{C}_\mathcal{N}$. We now show that one can restrict the rule to one which uses finitely many names, without losing soundness. This rule is given below.

$$\frac{\{\Sigma'\theta; \Gamma'\theta \Longrightarrow C'\theta \mid t'\theta \in supp(t, \Gamma, C) \cup \{a\}, a\#(t, \Gamma, C) \text{ and } supp(\theta) = \emptyset\}}{\Sigma; name_\iota\ t, \Gamma \Longrightarrow C}\ name_{fL}$$

Here Σ', t', Γ' and C' are obtained as follows. Let $\vec{c} = c_1 : \iota_1, \ldots, c_n : \iota_n$ be the support of (t, Γ, C) and let $a : \iota$ be a new nominal constant not in $\vec{c}$. Define a substitution σ as follows:

$$\sigma = \{(h'\ a) \mid h \in \Sigma \text{ and } h' \text{ is a variable of suitable type that is not in } \Sigma\}.$$

Then Σ', t', Γ' and C' are defined as $\Sigma\sigma$, $t\sigma$, $\Gamma\sigma$ and $C\sigma$, respectively. We call this substitution σ a *raising substitution* of the rule. This rule essentially reduces the extension of the support of the conclusion sequent to one in which only one new name is used, and since the judgments of the logic are equivariant, it does not matter which name we choose, as long as it is fresh with respect to the current support. There is another potential infinity in the rule because we consider arbitrary matching substitution θ. Since θ in this case is a closed substitution, the problem of matching t' with a name $b \in supp(t, \Gamma, C) \cup \{a\}$ reduces to a specific case of

$$tm\ I\ X \triangleq name\ X. \qquad tm\ (s\ I)\ (app\ M\ N) \triangleq tm\ I\ M \wedge tm\ I\ N.$$
$$tm\ (s\ I)\ (lam\ M) \triangleq \nabla x.tm\ I\ (M\ x). \qquad term\ M \triangleq \exists I.nat\ I \wedge tm\ I\ M.$$

$$fresh\ A\ B \triangleq name\ A \wedge name\ B \wedge A \neq B.$$
$$fresh\ A\ (app\ M\ N) \triangleq fresh\ A\ M \wedge fresh\ A\ N.$$
$$fresh\ A\ (lam\ M) \triangleq \nabla x.fresh\ A\ (M\ x).$$

$$abstract\ A\ A\ (\lambda x.x) \triangleq name\ A.$$
$$abstract\ A\ B\ (\lambda x.B) \triangleq name\ A \wedge name\ B \wedge (A \neq B).$$
$$abstract\ A\ (app\ M\ N)\ (\lambda x.app\ (R\ x)\ (T\ x)) \triangleq abstract\ A\ M\ R \wedge abstract\ A\ N\ T.$$
$$abstract\ A\ (lam\ M)\ (\lambda x.lam\ (T\ x)) \triangleq \nabla y.abstract\ A\ (M\ y)\ (\lambda x.T\ x\ y).$$

$$subst\ X\ M\ X\ M \triangleq name\ X.$$
$$subst\ X\ M\ Y\ Y \triangleq name\ X \wedge name\ Y \wedge X \neq Y.$$
$$subst\ X\ M\ (app\ R\ S)\ (app\ U\ V) \triangleq subst\ X\ M\ R\ U \wedge subst\ X\ M\ S\ V.$$
$$subst\ X\ M\ (lam\ N)\ (lam\ R) \triangleq \nabla x.subst\ X\ M\ (N\ x)\ (R\ x).$$

Fig. 4. A data type for λ-terms and some relations over λ-terms

higher-order matching: $\lambda\vec{c}\lambda a.t' =_{\beta\eta} \lambda\vec{c}\lambda a.b$ where $\vec{c} = supp(t, \Gamma, C)$. Readers who are familar with Huet's higher-order unification algorithm will notice that in solving this matching problem, one needs only use the projection step. This matching problem can be shown to be decidable and there exists a finite complete set of unifiers (CSU), if it is solvable. Hence, in practice one needs only to consider a finite number of premises generated by this CSU.

We refer to the logic LGn^{ω} with the $name\mathcal{L}$ rule replaced by $name_{fL}$ as LGn^{ω}_{f}.

Proposition 2.5 *Let Π be a derivation of $\Sigma; \Gamma \Longrightarrow C$ in LGn^{ω}_{f}. Then there exists a derivation Π' of the same sequent in LGn^{ω}.*

3 λ-terms, freshness and substitutions

In this section, we show a few simple examples of representing and reasoning about an encoding of untyped λ-terms. These examples serve only to illustrate how one can reason inductively about data structures with bindings. We prove some basic properties related to freshness and substitutions, which are basic ingredients for more complicated reasoning tasks. We encode these structures directly as definitions in LGn^{ω}, and derive a structural induction principle for λ-terms. In the following, we assume there is one nominal type for representing expressions, denoted by *exp*.

3.1 A structural induction rule for λ-terms

The data structure representing λ-terms is encoded as the definition clause for *term* given in Figure 4. The syntactic types of the constructors are as expected, namely, $app : exp \to exp \to exp$ and $lam : (exp \to exp) \to exp$. Notice that we need to index the predicates *tm* with a natural number since we intend to perform induction over

$$\frac{\begin{array}{c}\{.;. \Longrightarrow P\ (\lambda\vec{n}.\ a) \mid a \in \{\vec{n}\} \cup \{b\}, b \notin \vec{n}\} \\ M, N; P\ M \wedge P\ N \Longrightarrow P\ (\lambda\vec{n}.(app\ (M\ \vec{n})\ (N\ \vec{n}))) \\ M; \nabla a.P\ (\lambda\vec{n}.M\ \vec{n}\ a) \Longrightarrow P\ (\lambda\vec{n}.lam\ (M\ \vec{n})) \\ \Sigma; P\ (\lambda\vec{n}.t), \Gamma \Longrightarrow C\end{array}}{\Sigma; term\ t, \Gamma \Longrightarrow C}\ term\mathcal{L}$$

Fig. 5. A structural induction rule for λ-terms. In the rule, $\vec{n} = supp(t)$.

the structure of terms, and since in LGn^{ω} we allow only natural number induction.

Proving inductive properties of terms using natural number induction can be quite cumbersome. We shall derive a more user-friendly rule for induction over *term*. In reasoning about *term* t, we often need to take into account the support of t. The set $supp(t)$ can be seen as some sort of context for the term t. To make this context explicit in the invariants of the induction, we use (meta-level) λ-abstraction to encode this context into the invariants. Therefore, in the derived rule, the invariants can be abstractions of arbitrary arity, depending on the support of t. This rule is given in Figure 5. In the rule, $\vec{n} = supp(t)$ and P is a closed term of type $exp \to \cdots \to exp \to o$. For the base cases, we consider all the cases where the term is a name, i.e., the cases where it is among the support of t and another case where it is a new name.

Proposition 3.1 *The rule term$\mathcal{L}$ is derivable in LGn^{ω}.*

3.2 Freshness and scoping

The notion of freshness of a name with respect to a term is encoded via a predicate called $fresh$, given in Figure 4. In the figure, we abbreviate $(X = Y) \supset \bot$ as $X \neq Y$. Scoping restrictions at the meta level can be shown to imply the derivability of the freshness relation, as given in the following theorem.

Theorem 3.2 Freshness and scopes. *The formula $\forall M \nabla x.term\ M \supset fresh\ x\ M$ is derivable in LGn^{ω}.*

3.3 Abstractions

In this example, we show how one can abstract a name from a term. This relation is defined via the predicate *abstract* in Figure 4. As in the case with freshness, scoping restrictions and ∇-quantification at the meta level imply derivability of an abstraction.

Theorem 3.3 *The following formulas are derivable in LGn^{ω}.*

(i) $\forall M \nabla x.term\ (M\ x) \supset abstract\ x\ (M\ x)\ M$.

(ii) $\forall M \nabla x \forall N.term\ (M\ x) \supset abstract\ x\ (M\ x)\ N \supset M = N$.

As expected, an abstracted name is fresh with respect to the abstracted term, as stated in the following theorem.

$$\begin{array}{rl} \tau\ P \xrightarrow{\tau} P \triangleq \top & \\ in\ X\ M \xrightarrow{\downarrow X} M \triangleq \top & match\ x\ x\ P \xrightarrow{A} Q \triangleq P \xrightarrow{A} Q \\ out\ x\ y\ P \xrightarrow{\uparrow xy} P \triangleq \top & match\ x\ x\ P \xrightarrow{A} Q \triangleq P \xrightarrow{A} Q \end{array}$$

$$\begin{array}{rlrl} P+Q \xrightarrow{A} R \triangleq P \xrightarrow{A} R & & P \mid Q \xrightarrow{A} P' \mid Q \triangleq P \xrightarrow{A} P' & \\ P+Q \xrightarrow{A} R \triangleq Q \xrightarrow{A} R & & P \mid Q \xrightarrow{A} P \mid Q' \triangleq Q \xrightarrow{A} Q' & rcl \\ P+Q \xrightarrow{A} R \triangleq P \xrightarrow{A} R & P \mid Q \xrightarrow{A} \lambda n(M\,n \mid Q) \triangleq P \xrightarrow{A} M & & \\ P+Q \xrightarrow{A} R \triangleq Q \xrightarrow{A} R & P \mid Q \xrightarrow{A} \lambda n(P \mid N\,n) \triangleq Q \xrightarrow{A} N. & & \end{array}$$

$$\begin{array}{rl} \nu n.Pn \xrightarrow{A} \nu n.Qn \triangleq \nabla n(Pn \xrightarrow{A} Qn) & \\ \nu n.Pn \xrightarrow{A} \lambda m\ \nu n.P'nm \triangleq \nabla n(Pn \xrightarrow{A} P'n) & \\ \nu y.My \xrightarrow{\uparrow X} M' \triangleq \nabla y(My \xrightarrow{\uparrow Xy} M'y) & \\ P \mid Q \xrightarrow{\tau} \nu y.(My \mid Ny) \triangleq \exists X.P \xrightarrow{\downarrow X} M \wedge Q \xrightarrow{\uparrow X} N & \\ P \mid Q \xrightarrow{\tau} \nu y.(My \mid Ny) \triangleq \exists X.P \xrightarrow{\uparrow X} M \wedge Q \xrightarrow{\downarrow X} N & \\ P \mid Q \xrightarrow{\tau} MY \mid Q' \triangleq \exists X.P \xrightarrow{\downarrow X} M \wedge Q \xrightarrow{\uparrow XY} Q' & \\ P \mid Q \xrightarrow{\tau} P' \mid NY \triangleq \exists X.P \xrightarrow{\uparrow XY} P' \wedge Q \xrightarrow{\downarrow X} N & \end{array}$$

Fig. 6. A specification of the operational semantics of the π-calculus.

Theorem 3.4 *The following formula is derivable in LGn^{ω}:*

$$\forall A \forall M \forall N.name\ A \supset term\ M \supset abstract\ A\ M\ N \supset fresh\ A\ (lam\ N).$$

3.4 Substitutions and meta-level applications

Substitutions of a name for a term can be encoded as the predicate *subst* in Figure 4. This explicit encoding of substitutions co-incides with the implicit one using meta-level applications.

Theorem 3.5 *The following formula is derivable in LGn^{ω}.*

$$\forall M \forall N \nabla y \forall R.term\ M \wedge term\ (N\ y) \supset subst\ y\ M\ (N\ y)\ R \equiv (R = (N\ M))$$

4 The π-calculus and bisimulation

We now consider a specification of the π-calculus and its associated notions of bisimulation. We consider here only the finite fragment of the π-calculus, given by the following grammar:

$$\mathtt{P} ::= 0 \mid \tau.\mathtt{P} \mid \bar{x}y.\mathtt{P} \mid x(y).\mathtt{P} \mid (\nu x)\mathtt{P} \mid [x = y]\mathtt{P} \mid \mathtt{P}|\mathtt{P} \mid \mathtt{P} + \mathtt{P}.$$

We use the symbols $\mathtt{P}$, $\mathtt{Q}$, $\mathtt{R}$, $\mathtt{S}$, and $\mathtt{T}$ to denote processes and lower case letters, *e.g.*, a, b, c, d, x, y, z to denote names. The occurrence of y in the process $x(y).\mathtt{P}$ and $(y)\mathtt{P}$ is a binding occurrence, with $\mathtt{P}$ as its scope. We consider processes to be syntactical equivalent up to renaming of bound names.

The encoding of the operational semantics of the π-calculus in λ-tree syntax has been done in several previous work, e.g., [10,24,12], so we shall not go into every formal detail of the encoding. Three primitive syntactic categories are used to encode the π-calculus expressions: n for names, p for processes, and a for actions. The type n is a nominal type, and it is the only nominal type we consider in this section. The process expressions are encoded into λ-tree syntax using the following constructors:

$$0 : p \qquad \tau : p \to p \qquad \mathit{out} : n \to n \to p \to p \qquad \mathit{in} : n \to (n \to p) \to p$$
$$+ : p \to p \to p \qquad | : p \to p \to p \qquad \mathit{match} : n \to n \to p \to p \qquad \nu : (n \to p) \to p$$

We shall write the constructors $+$ and $|$ using the infix notation. The mapping between π-calculus processes and λ-terms of type p is defined in a straightforward way, where the input prefix $(x(y).P)$ maps to $(\mathit{in}\ x\ \lambda y.P)$, output prefix $(\bar{x}(y).P)$ maps to $\mathit{out}\ x\ y\ P$, and the match operator $[. = .]$ maps to match.

There are three kinds of one-step transition relations for the late version of the π-calculus: the free transition $P \xrightarrow{\alpha} Q$, where α is either a *silent action*, or an output action (of the form $\bar{x}y$), the *bound output* transition $P \xrightarrow{\bar{x}(y)} Q$ and the *bound input* transition $P \xrightarrow{x(y)} Q$. In the bound input and bound output transitions, the name y is a binder whose scope is Q. The encoding of these transitions in λ-tree syntax are given in the following:

$$P \xrightarrow{\tau} Q \quad P \xrightarrow{\uparrow xy} Q \qquad P \xrightarrow{\uparrow x} (\lambda y.Q) \qquad P \xrightarrow{\downarrow x} (\lambda y.Q)$$

representing, respectively, a silent transition, a free output transition, a bound output transition and a bound input transition. Here, the constructors $\uparrow$ and $\downarrow$ have the type $n \to n \to a$. The encoding of the operational semantics of the finite π-calculus is given in Figure 6.

We now look at specifications of bisimulation for the π-calculus. We consider two variants of bisimulation: the strong late bisimulation and the strong open bisimulation [20]. Given a relation $\mathcal{R}$ on processes, we write $P\ \mathcal{R}\ Q$ to denote $(P, Q) \in \mathcal{R}$. Given a process P, we denote with $\mathrm{fn}(P)$ the set of free names in P. This notation extends to free names of sets of processes in the obvious way.

Definition 4.1 A process relation $\mathcal{R}$ is a *strong late bisimulation* if $\mathcal{R}$ is symmetric and whenever $\mathtt{P}\ \mathcal{R}\ \mathtt{Q}$,

(i) if $\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'$ and α is a free action, then there is $\mathtt{Q}'$ such that $\mathtt{Q} \xrightarrow{\alpha} \mathtt{Q}'$ and $\mathtt{P}'\ \mathcal{R}\ \mathtt{Q}'$;

(ii) if $\mathtt{P} \xrightarrow{x(z)} \mathtt{P}'$ and $z \notin \mathrm{fn}(\mathtt{P}, \mathtt{Q})$ then there is $\mathtt{Q}'$ such that $\mathtt{Q} \xrightarrow{x(z)} \mathtt{Q}'$ and, for every name y, $\mathtt{P}'[y/z]\ \mathcal{R}\ \mathtt{Q}'[y/z]$; and

(iii) if $\mathtt{P} \xrightarrow{\bar{x}(z)} \mathtt{P}'$ and $z \notin \mathrm{fn}(\mathtt{P}, \mathtt{Q})$ then there is $\mathtt{Q}'$ such that $\mathtt{Q} \xrightarrow{\bar{x}(z)} \mathtt{Q}'$ and $\mathtt{P}'\ \mathcal{R}\ \mathtt{Q}'$.

$$
\begin{aligned}
\text{lbisim } P\ Q \triangleq\ & \forall A \forall P' \, [P \xrightarrow{A} P' \supset \exists Q'.\ Q \xrightarrow{A} Q' \wedge \text{lbisim } P'\ Q'] \wedge \\
& \forall A \forall Q' \, [Q \xrightarrow{A} Q' \supset \exists P'.\ P \xrightarrow{A} P' \wedge \text{lbisim } Q'\ P'] \wedge \\
& \forall X \forall P' \, [P \xrightarrow{\downarrow X} P' \supset \exists Q'.\ Q \xrightarrow{\downarrow X} Q' \wedge \forall w.name\ w \supset \text{lbisim } (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q' \, [Q \xrightarrow{\downarrow X} Q' \supset \exists P'.\ P \xrightarrow{\downarrow X} P' \wedge \forall w.name\ w \supset \text{lbisim } (Q'w)\ (P'w)] \wedge \\
& \forall X \forall P' \, [P \xrightarrow{\uparrow X} P' \supset \exists Q'.\ Q \xrightarrow{\uparrow X} Q' \wedge \nabla w.\text{lbisim } (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q' \, [Q \xrightarrow{\uparrow X} Q' \supset \exists P'.\ P \xrightarrow{\uparrow X} P' \wedge \nabla w.\text{lbisim } (Q'w)\ (P'w)]
\end{aligned}
$$

$$
\begin{aligned}
\text{obisim } P\ Q \triangleq\ & \forall A \forall P' \, [P \xrightarrow{A} P' \supset \exists Q'.\ Q \xrightarrow{A} Q' \wedge \text{obisim } P'\ Q'] \wedge \\
& \forall A \forall Q' \, [Q \xrightarrow{A} Q' \supset \exists P'.\ P \xrightarrow{A} P' \wedge \text{obisim } Q'\ P'] \wedge \\
& \forall X \forall P' \, [P \xrightarrow{\downarrow X} P' \supset \exists Q'.\ Q \xrightarrow{\downarrow X} Q' \wedge \forall w.\text{obisim } (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q' \, [Q \xrightarrow{\downarrow X} Q' \supset \exists P'.\ P \xrightarrow{\downarrow X} P' \wedge \forall w.\ \text{obisim } (Q'w)\ (P'w)] \wedge \\
& \forall X \forall P' \, [P \xrightarrow{\uparrow X} P' \supset \exists Q'.\ Q \xrightarrow{\uparrow X} Q' \wedge \nabla w.\ \text{obisim } (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q' \, [Q \xrightarrow{\uparrow X} Q' \supset \exists P'.\ P \xrightarrow{\uparrow X} P' \wedge \nabla w.\ \text{obisim } (Q'w)\ (P'w)]
\end{aligned}
$$

Fig. 7. Specification of strong late, *lbisim*, and open, *obisim*, bisimulations.

The processes $\mathtt{P}$ and $\mathtt{Q}$ are *strong late bisimilar*, written $\mathtt{P} \sim_l \mathtt{Q}$, if there is a strong late bisimulation $\mathcal{R}$ such that $\mathtt{P}\ \mathcal{R}\ \mathtt{Q}$.

Notice that in the late bisimulation, in the clause concerning input transitions, one needs to perform a case analysis on the input name of the process pair before checking the bisimilarity of their continuations. Also, implicit in the definition is the idea that free names in processes are constants. As a consequence, late bisimilarity is not closed under input prefix. Open bisimulation, on the other hand, treats names more like variables. But to enforce the "freshness" of names introduced by the bound output prefix, one needs to augment the definition of bisimulation with a notion of *distinctions* among names.

Definition 4.2 A *distinction* D is a finite symmetric and irreflexive relation on names. A substitution θ *respects* a distinction D if $(x, y) \in D$ implies $x\theta \neq y\theta$. We refer to the substitution θ as a *D-substitution*. Given a distinction D and a D-substitution θ, the result of applying θ to all variables in D, written $D\theta$, is another distinction. We denote with $\text{fn}(D)$ the set of names occurring in D.

Definition 4.3 The set $\mathcal{S} = \{\mathcal{S}_D\}_D$ of process relations is an indexed open bisimulation if for each D, $\mathcal{S}_D$ is symmetric and for every θ that respects D, $\mathtt{P}\ \mathcal{S}_D\ \mathtt{Q}$ implies:

(i) if $\mathtt{P}\theta \xrightarrow{\alpha} \mathtt{P}'$ and α is a free action, then there is $\mathtt{Q}'$ such that $\mathtt{Q}\theta \xrightarrow{\alpha} \mathtt{Q}'$ and $\mathtt{P}'\mathcal{S}_{D\theta}\mathtt{Q}'$,

(ii) if $\mathtt{P}\theta \xrightarrow{x(z)} \mathtt{P}'$ and $z \notin \text{fn}(\mathtt{P}\theta, \mathtt{Q}\theta)$ then there is $\mathtt{Q}'$ such that $\mathtt{Q}\theta \xrightarrow{x(z)} \mathtt{Q}'$ and $\mathtt{P}'\ \mathcal{S}_{D\theta}\ \mathtt{Q}'$,

(iii) if $\mathtt{P}\theta \xrightarrow{\bar{x}(z)} \mathtt{P}'$ and $z \notin \text{fn}(\mathtt{P}\theta, \mathtt{Q}\theta)$ then there is $\mathtt{Q}'$ such that $\mathtt{Q}\theta \xrightarrow{\bar{x}(z)} \mathtt{Q}'$ and $\mathtt{P}'\ \mathcal{S}_{D'}\ \mathtt{Q}'$ where $D' = D\theta \cup (\{z\} \times \text{fn}(\mathtt{P}\theta, \mathtt{Q}\theta, D\theta))$.

The processes P and Q are *strong open D-bisimilar*, written $\mathtt{P} \sim_o^D \mathtt{Q}$, if there is an indexed open bisimulation $\mathcal{S}$ such that $P\ \mathcal{S}_D\ \mathtt{Q}$. The processes P and Q are *strong open bisimilar* if $\mathtt{P} \sim_o^{\emptyset} \mathtt{Q}$.

Notice that the definition of open bisimulation uses quantification over substitutions. A direct encoding would therefore formalize this name substitution explicitly. However, we can obtain a more concise encoding by using quantifiers of logic. This encoding is not new and has been used in a previous work [24]. We refer the reader to this work for the details concerning the use of quantifier alternations to encode distinctions and to avoid explicit encodings of substitutions.

The specifications of late and open bisimulations in LGn^ω are given in Figure 7. The specification of open bisimulation is the same as in [24]. In that work, actually both open and late bisimulation are defined using the same definition (in the logic $FO\lambda^{\Delta\nabla}$), i.e., the predicate obisim in this case. Their difference appears in the statement of adequacy. For late bisimulation, $P \sim_l Q$ corresponds to the provability of the $FO\lambda^{\Delta\nabla}$ formula $\mathcal{E} \supset \nabla\vec{x}.obisim\ P\ Q$ where $\vec{x}$ are the free names of P and Q and $\mathcal{E}$ is a set of formulas of the form $F = \nabla\vec{y}\forall u\forall v.u = v \vee u \neq v$. This extra assumption is needed to guarantee the completeness of the encoding, and it is used to perform case analysis on names. Such a case analysis is needed in proving certain bisimilar processes, in particular, when non-deterministic choice is presence in the processes (see [17] for an example).

As noted earlier, equality between names is decidable in LGn^ω, provided we explicitly annotate each name variable as belonging to the set of names, using the *name* predicate. That is, the formula F above, modified slightly with explicit name predicates, i.e., $\nabla\vec{y}\forall u\forall v.name\ u \supset name\ v \supset u = v \vee u \neq v$, is a theorem in LGn^ω, for any $\vec{y}$. To get a complete encoding of late bisimulation, we then need only to explicitly "type" every name that is introduced in the body of the definition, as shown in Figure 7.

We now state the adequacy statements for the encoding of open and late bisimulations. The proof for the adequacy of open bisimulation is basically the same as the one done for its encoding in $FO\lambda^{\Delta\nabla}$. We state the theorem here without proofs; interested readers can refer to [24] for an outline, and [25] for more detailed proofs. The adequacy result of late bisimulation is, however, new and requires a different proof than that in [24,25].

Theorem 4.4 Adequacy of open bisimulation. *Let* P *and* Q *be two processes and let* $\bar{n}$ *be the free names in* P *and* Q. *Then* $\mathtt{P} \sim_o \mathtt{Q}$ *if and only if* $\forall\bar{n}.obisim\ \mathtt{P}\ \mathtt{Q}$ *is derivable in* LGn^ω.

Notice that we universally quantify the free names of P and Q in the above theorem, to capture the idea that these names can be identified with each other. The late bisimulation encoding requires ∇-quantification of free names, since these are supposed to be pairwise distinct constants.

Theorem 4.5 Adequacy of late bisimulation. *Let* P *and* Q *be two processes and let* $\bar{n}$ *be the free names in* P *and* Q. *Then* $\mathtt{P} \sim_l \mathtt{Q}$ *iff* $\nabla\bar{n}.lbisim\ \mathtt{P}\ \mathtt{Q}$ *is derivable in* LGn^ω.

5 Conclusion and related work

We have shown a proof theoretic treatment of a notion of names and the equivariant principle in a logical framework based on λ-tree syntax. Most of the foundations needed to build this framework have been done in LG^{ω}[22]. However, with a small, but crucial extension involving an explicit data type for names, one can do a variety of reasoning tasks involving names and bindings in a rather clean way. We have also shown that one can derive a structural induction rule for λ-terms, in which the context of induction is encoded via abstractions. The case analysis rule on *name* also allows us to prove the decidability of name equality. This results in a simpler characterization of the difference between late and open bisimulation for the π-calculus, as the difference between typed and untyped encodings of names.

The logic LGn^{ω}, with the $name\mathcal{L}$-rule replaced by a version with finite number of names (see Section 2), can be seen as a specific instance of the logic $\mathcal{G}$ [4]. The logic $\mathcal{G}$ allows definition clauses with ∇-quantifiers in the head of the clauses. The predicate *name* of LGn^{ω} can be seen as the extended definition clause

$$(\nabla x.name\ x) \stackrel{\triangle}{=} \top$$

of $\mathcal{G}$. The scoping of variables and alternation of quantifiers in the head of a definition clause in $\mathcal{G}$ enforces the notion of freshness of a name with respect to a term. For example, in $\mathcal{G}$, the $fresh$ predicate would be encoded simply as

$$\forall M(\nabla x.fresh\ x\ M) \stackrel{\triangle}{=} \top,$$

from which one can prove $\forall M \nabla x.fresh\ x\ M$. Most of the examples studied in $\mathcal{G}$ so far seem to make use of only this aspect of ∇ in the head of definitions. We have seen that we can encode this notion of freshness, provided we explicitly type the term M above. Therefore it seems that, in principle, we can do most of the examples done in $\mathcal{G}$ in LGn^{ω}, with perhaps extra inductive definitions. Note that by restricting to LGn^{ω}, the unification problem that arises from rule applications ($name\mathcal{L}$ and eq$\mathcal{L}$) is a higher-order unification problem, whereas the more general fixed point rules of $\mathcal{G}$ require also permutation of names on top of higher-order unification. The latter can be seen as the higher-order version of the equivariant unification problem [1]. However, equivariant unification can still arises in proof search since the *id* rule requires checking equality modulo permutations.

The work on LG^{ω} and LGn^{ω} is closely related to nominal logic. We have adapted the notion of names and equivariant principle from nominal logic. However, the related notions of freshness and permutations are not taken as primitives of the logic. As a result, we do not have to deal explicitly with freshness and permutations, which is possible through the use of raising to encode explicitly the support of a term. As a consequence, we are able to obtain a rather simple induction rule for λ-terms. Similar induction principles have been obtained in other works, e.g., [13,26], but explicit assumptions about freshness and permutations are a part of the induction scheme. Note that these works aim at formalising the "informal" (meaning, not

machine-checked) practice often used in mathematic proofs, e.g., the Barendregt variable convention. Our approach is essentially HOAS-based, so it would require familiarity with this particular style of encodings, which is relatively more removed from the usual informal practice in mathematics. In [26], the authors impose some conditions on the schematic rules for induction, which is aimed at ruling out faulty reasoning with variables and binders. It is interesting to investigate whether there is a parallel to their rule conditions in our setting.

The idea of induction and/or recursion over open terms in a context has also been studied in the type theoretic setting, see e.g., [18] and more recently, [16,14]. The use of nominal constants seems very similar to the notion of parameters used in [16] to provide some form of implicit variable context, in which a ∇-like quantifier is used to type patterns involving these parameters. In [14], the variable context in an open term is represented explicitly; such a term-in-context is typed using a contextual type system. In these works, a distinction is drawn between computation types and representation types. An analog of this separation is the two-level encoding used in [5], where the object-logic can be seen as the "representation" level and the meta logic the "computation" level.

Acknowledgement. The author thanks Andrew Gacek for his help with the proof assistant Abella. The author also thanks him and the anonymous referees for their comments on an earlier draft of the paper. This work is supported by a project funded by the Australian Research Council.

References

[1] James Cheney. Equivariant unification. In *Proceedings of RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2005.

[2] Andrew Gacek. System description: Abella – A system for reasoning about computations. Accepted to IJCAR 2008. Available from http://arxiv.org/abs/0803.2305, 2008.

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

[4] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *Proceedings of LICS 2008*. IEEE Computer Society Press, 2008. To appear.

[5] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. Accepted to LFMTP 2008, April 2008.

[6] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *Proceedings of LICS 1999*, pages 214–224. IEEE Computer Society Press, 1999.

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

[8] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

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

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

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

[12] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[13] Michael Norrish. Mechanising lambda-calculus using a classical first order theory of terms with permutations. *Higher-Order and Symbolic Computation*, 19(2-3):169–195, 2006.

[14] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of POPL*, pages 371–382, 2008.

[15] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[16] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of ESOP 2008*, volume 4960 of *LNCS*, pages 93–107. Springer, 2008.

[17] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.

[18] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, October 2000.

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

[20] Davide Sangiorgi and David Walker. *π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[21] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.

[22] Alwen Tiu. A logic for reasoning about generic judgments. *Electr. Notes Theor. Comput. Sci.*, 174(5):3–18, 2007.

[23] Alwen Tiu. Cut elimination for a logic with generic judgments and induction. Technical report, January 2008. Extended version of LFMTP'06 paper. Available from http://arxiv.org/abs/0801.3065.

[24] Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, September 2004.

[25] Alwen Tiu and Dale Miller. Proof search specifications for bisimulation and modal logics for the π-calculus. Submitted. Available via http://arXiv.org/abs/0805.2785, 2008.

[26] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of CADE-21*, pages 35–50, 2007.