Nominal Abstraction

Gacek Andrew, Dale Miller, Gopalan Nadathur

Nominal Abstraction

Andrew Gacek[*,a], Dale Miller[b], Gopalan Nadathur[a]

[a] *Department of Computer Science and Engineering, University of Minnesota*
[b] *INRIA Saclay - Île-de-France & LIX/École polytechnique*

Abstract

Recursive relational specifications are commonly used to describe the computational structure of formal systems. Recent research in proof theory has identified two features that facilitate direct, logic-based reasoning about such descriptions: the interpretation of atomic judgments through recursive definitions and an encoding of binding constructs via generic judgments. However, logics encompassing these two features do not currently allow for the definition of relations that embody dynamic aspects related to binding, a capability needed in many reasoning tasks. We propose a new relation between terms called *nominal abstraction* as a means for overcoming this deficiency. We incorporate nominal abstraction into a rich logic also including definitions, generic quantification, induction, and co-induction that we then prove to be consistent. We present examples to show that this logic can provide elegant treatments of binding contexts that appear in many proofs, such as those establishing properties of typing calculi and of arbitrarily cascading substitutions that play a role in reducibility arguments.

Key words: generic judgments, higher-order abstract syntax, λ-tree syntax, proof search, reasoning about operational semantics

*4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455
Email addresses: `agacek@cs.umn.edu` (Andrew Gacek), `dale.miller@inria.fr` (Dale Miller), `gopalan@cs.umn.edu` (Gopalan Nadathur)

Preprint submitted to Information and Computation *September 22, 2010*

1. Introduction

This paper contributes to an increasingly important approach to using relational specifications for formalizing and reasoning about a wide class of computational systems. This approach, whose theoretical underpinnings are provided by recent ideas from proof theory and proof search, has been used with success in codifying within a logical setting the methods of structural operational semantics that are often employed in describing aspects such as the evaluation and type assignment characteristics of programming languages. The main ingredients of this approach are the use of terms to represent the syntactic objects that are of interest in the relevant systems and the reflection of their dynamic aspects into judgments over such terms.

One common application of the method has utilized recursive relational specifications or judgments over algebraic terms. We highlight three stages of development in the kinds of judgments that have been employed in this context, using the transition semantics for CCS as a motivating example [1]:

(1) *Logic programming, may behavior* Logic programming languages allow for a natural encoding and animation of relational specifications. For example, Horn clauses provide a simple and immediate encoding of CCS labeled transition systems and unification and backtracking provide a means for exploring what is *reachable* from a given process. An early system based on this observation was Centaur [2], which used Prolog to animate the operational semantics and typing judgments of programming languages. Traditional logic programming is, however, limited to describing only *may* behavior judgments. For example, using it, we are not able to prove that a given CCS process P *cannot* make a transition. Since this negative property is logically equivalent to proving that P is bisimilar to the null process 0, such systems cannot also capture bisimulation.

(2) *Model checking, must behavior* One way to account for must behavior is to allow for the unfolding of specifications in both positive and negative settings. Proof theoretic techniques that provided for such a treatment were developed in the early 1990s [3, 4] and extended in subsequent work [5]. In the basic form, these techniques require an unfolding until termination, and are therefore applicable to *recursive definitions* that are *noetherian*. Specifications that meet this restriction and, hence, to which this method is applicable, include bisimulation for finite processes and many model checking problems. As an example, bisimulation for finite CCS can be given an immediate and declarative treatment using these techniques [6].

(3) *Theorem proving, infinite behavior* Reasoning about all members of a domain or about possibly infinite executions requires the addition of induction and co-induction to the above framework of recursive definitions. Incorporating induction in proof theory goes back to Gentzen. The work in [5, 7, 8] provides induction and co-induction rules associated with recursive relational specifications. In such a setting, one can prove, for example, that (strong) bisimulation in CCS is a congruence.

The systems that are to be specified and reasoned about often involve terms that use names and binding. An elegant way to treat such terms is to encode them as λ-terms and equate them using the theory of α, β, and η-conversion. The three stages discussed above need to be extended to treat representations based on such terms. The manner in which this has been done is illustrated next using the relational specification of the π-calculus [9].

(1) *Logic programming, λ-tree syntax* Higher-order generalizations of logic programming, such as *higher-order hereditary Harrop formulas* [10] and the dependently typed LF [11], adequately capture may behavior for terms containing bindings. In particular, the presence of hypothetical and universal judgments supports the λ-tree syntax [12] approach to higher-order abstract syntax [13]. The logic programming languages λProlog [14] and Twelf [15] support such syntax representations and can be used to provide simple specifications of, for example, reachability in the π-calculus.

(2) *Model checking, ∇-quantification* While the notions of universal quantification and *generic judgment* are often conflated, a satisfactory treatment of must behavior requires splitting apart these concepts. The ∇-quantifier [16] was introduced to encode generic judgments directly. To illustrate the need for this split, consider the formula $\forall w.\neg(\lambda x.x = \lambda x.w)$. If we think of λ-terms as denoting abstracted syntax (terms modulo α-conversion), this formula should be provable (variable capture is not allowed in logically sound substitution). On the other hand, if we think of λ-terms as describing functions, then the equation $\lambda y.t = \lambda y.s$ is equivalent to $\forall y.t = s$. But then our example formula is equivalent to $\forall w.\neg\forall x.x = w$, which should not be provable since it is not true in a model with a single element domain. To think of λ-terms syntactically, we treat $\lambda y.t = \lambda y.s$ as equivalent not to $\forall y.t = s$ but, rather, to $\nabla y.t = s$. Our example formula then becomes equivalent to $\forall w.\neg\nabla x.x = w$, which *is* provable [16]. Using a representation based on this new quantifier, the π-calculus process $(\nu x).[x = w].\bar{w}x$ can be proved to be bisimilar to 0. Bedwyr [17] is a model checker that treats such generic judgments.

(3) *Theorem proving, equality of generic judgments* When there is only finite behavior, logics for recursive definitions do not need the cut or initial rules, and, consequently, there is no need to know when two judgments are the same. On the other hand, the treatment of induction and co-induction relies on the ability to make such identifications: *e.g.*, when carrying out an inductive argument over natural numbers, one must be able to recognize when the case for $i+1$ has been reduced to the case for i. This identity question is complicated by the presence of the ∇-quantifier: for example, the proof search treatment of such quantifiers involves instantiation with generic objects whose choice of name is arbitrary and this must be factored into assessments of equality. The LG^ω proof system [18] provides a way to address this issue and uses this to support inductive reasoning over recursive definitions. Using LG^ω encodings extended with co-induction (as described in this paper), one can prove, for instance, that (open) bisimulation is a π-calculus congruence.

The key observation underlying this paper is that logics like LG^ω are still missing an ingredient that is important to many reasoning tasks. Within these logics, the ∇-quantifier can be used to control the structure of terms relative to the generic judgments in which they occur. However, these logics do not possess a complementary device for simply and precisely characterizing such structure *within* the logic. Consider, for example, the natural way to specify typing of λ-terms in this setting [19]. The representation of λ-terms within this approach uses (meta-level) abstracted variables to encode object-level bound variables and

∇-bound variables (also called here *nominal constants*) to encode object-level free variables. Conceptually, the type specification uses recursion over the representation of λ-terms, transforming abstracted variables into nominal constants, and building a context that associates nominal constants with types. Now suppose that the list $[\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle]$ represents a particular context. The semantics of the ∇-quantifier ensures that each x_i in this list is a *unique* nominal constant. This property is important to the integrity of the type assignment. Moreover, making it explicit can also be important to the reasoning process; for example, a proof of the uniqueness of type assignment would draw critically on this fact. Unfortunately, LG^ω and related logics do not possess a succinct and general way to express such a property.

This paper describes a way of realizing this missing feature, thereby yielding a logic that represents a natural endpoint to this line of development. The particular means for overcoming the deficiency is a relation between terms called a *nominal abstraction*. In its essence, nominal abstraction is an extension of the equality relation between terms that allows for the characterization also of occurrences of nominal constants in such terms. Combining this relation with definitions, we will, for instance, be able to specify a property of the form

$$\nabla x_1 \cdots \nabla x_n.\ \mathit{cntx}\,[\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle]$$

which effectively asserts that *cntx* is true of a list of type assignments to n distinct nominal constants. By exploiting the recursive structure of definitions, *cntx* can further be defined so that the length of the list is arbitrary. We integrate nominal abstraction into a broader logical context that includes also the ability to interpret definitions inductively and co-inductively. The naturalness of nominal abstraction is clear from the modular way in which we are able to define this extended logic and to prove it consistent. We present examples of specification and reasoning to bring out the usefulness of the resulting logic, focusing especially on the capabilities resulting from nominal abstraction.[1]

One of the features desired for the logic presented in this paper is that it support the λ-tree approach to the treatment of syntax. As discussed earlier in this section, such a treatment is typically based on permitting λ-bindings into terms and using universal and hypothetical judgments in analyzing these terms. Hypothetical judgments force an "open-world" assumption; in the setting of interest, we use them to assert new properties of the constants that are introduced in treating bound variables. However, our desire to be able to reason inductively about predicate definitions provides a contradictory tension: the definition of predicates must be fixed once and for all in order to state induction principles. This tension is relieved in the logic we describe by disallowing hypothetical judgments and instead using lists (as illustrated above) to implicitly encode contexts needed in syntactic analyses. This approach is demonstrated in greater detail through the examples in Section 7.2.

The rest of this paper is structured as follows. We develop a logic called $\mathcal{G}$, that is a rather rich logic, in the next three sections. Section 2 presents the rules for the core fragment

[1]While there might appear to be a similarity between nominal abstraction and "atom-abstraction" in *nominal logic* [20] from this discussion, these two concepts are technically quite different and should not be confused. Section 8.2 contains a comparison between $\mathcal{G}$ and nominal logic that should make the differences clear.

of $\mathcal{G}$ that is inherited from LG^ω. Section 3 introduces the nominal abstraction relation with its associated inference rules. Finally, Section 4 completes the framework by adding the mechanism of recursive definitions together with the possibility of interpreting these inductively or co-inductively. A central technical result of this paper is the cut-elimination theorem for $\mathcal{G}$, which is presented in Section 5: an immediate consequence of this theorem is the consistency of $\mathcal{G}$. Section 6 introduces a more flexible and suggestive style for recursive definitions that allows one to directly define generic judgments: such definitions allow for the use of "∇ in the head." We show that this style of definition can be accounted for by using the nominal abstraction predicate. Section 7 presents a collection of examples that illustrate the expressiveness of nominal abstraction in $\mathcal{G}$; a reader who is interested in seeing motivating examples first might peruse this section before digesting the detailed proofs in the earlier sections. Section 8 compares the development in this paper with recent related work on specification and reasoning techniques.

This paper extends the conference paper [21] in two important ways. First, nominal abstraction is used here as a more general and modular method for obtaining the benefits of allowing ∇-quantification in the "heads of definitions." Second, the modularity provided by nominal abstraction is exploited to allow recursive definitions to be read inductively and co-inductively. The logic in [21] was also called $\mathcal{G}$: this name is reused here for a richer logic. The logic developed in this paper has been implemented in the Abella system [22]. Abella has been used successfully in formalizing the proofs of theorems in a number of areas [19].

2. A Logic with Generic Quantification

The core logic underlying $\mathcal{G}$ is obtained by extending an intuitionistic and predicative subset of Church's Simple Theory of Types with a treatment of generic judgments. The encoding of generic judgments is based on the quantifier called ∇ (pronounced nabla) introduced by Miller and Tiu [16] and further includes the structural rules associated with this quantifier in the logic LG^ω described by Tiu [18]. While it is possible to develop a classical variant of $\mathcal{G}$ as well, we do not follow that path here, observing simply that the choice between an intuitionistic and a classical interpretation can lead to interesting differences in the meaning of specifications written in the logic. For example, it has been shown that the specification of bisimulation for the π-calculus within this logic corresponds to *open bisimulation* under an intuitionistic reading and to *late bisimulation* under a classical reading [23].

2.1. The basic syntax

Following Church [24], terms are constructed from constants and variables using abstraction and application. All terms are assigned types using a monomorphic typing system; these types also constrain the set of well-formed expressions in the expected way. The collection of types includes o, a type that corresponds to propositions. Well-formed terms of this type are also called formulas. We assume that o does not appear in the argument types of any nonlogical constant. Two terms are considered to be equal if one can be obtained from the other by a sequence of applications of the α-, β- and η-conversion rules, *i.e.*, the λ-conversion rules. This notion of equality is henceforth assumed implicitly wherever there is a need to compare terms. Logic is introduced by including special constants representing the propositional connectives $\top$, $\bot$, $\wedge$, $\vee$, $\supset$ and, for every type τ that does not contain o, the constants $\forall_\tau$ and $\exists_\tau$ of type $(\tau \to o) \to o$. The binary propositional connectives are written as usual in infix form and the expressions $\forall_\tau x.B$ and $\exists_\tau x.B$ abbreviate the formulas $\forall_\tau \lambda x.B$ and $\exists_\tau \lambda x.B$, respectively. Type subscripts will be omitted from quantified formulas when they can be inferred from the context or are not important to the discussion. We also use a shorthand for iterated quantification: if $\mathcal{Q}$ is a quantifier, the expression $\mathcal{Q}x_1, \ldots, x_n.P$ will abbreviate $\mathcal{Q}x_1 \ldots \mathcal{Q}x_n.P$.

The usual inference rules for the universal quantifier can be seen as equating it to the conjunction of all of its instances: that is, this quantifier is treated extensionally. There are several situations where one wishes to treat an expression such as "$B(x)$ holds for all x" as a statement about the existence of a uniform argument for every instance rather than the truth of a particular property for each instance [16]; such situations typically arise when one is reasoning about the binding structure of formal objects represented using the *λ-tree syntax* [12] version of *higher-order abstract syntax* [13]. The ∇-quantifier serves to encode judgments that have this kind of a "generic" property associated with them. Syntactically, this quantifier corresponds to including a constant ∇_τ of type $(\tau \to o) \to o$ for each type τ not containing o.[2] As with the other quantifiers, $\nabla_\tau x.B$ abbreviates $\nabla_\tau \lambda x.B$ and the type subscripts are often suppressed for readability.

2.2. Generic judgments and ∇-quantification

Towards understanding the ∇-quantifier, let us consider the rule for typing abstractions in the simply-typed λ-calculus as an example of something that we might want to encode within $\mathcal{G}$. This rule has the form

$$\frac{\Gamma, x\!:\!\alpha \vdash t : \beta}{\Gamma \vdash (\lambda x\!:\!\alpha.t) : \alpha \to \beta}\ x \notin dom(\Gamma)$$

In the conclusion of this rule, the variable x is bound and its scope is clearly delimited by the abstraction that binds it. It appears that x is free in the premise of the rule, but it is in fact implicitly bound over the judgment whose subcomponents, specifically Γ, also constrain its identity. One way to precisely encode this rule in a meta-logic is to introduce an explicit quantifier over x in the upper judgment; in a proof search setting, the encoding of the rule can then be understood as one that moves a term level binding to a formula level binding. However, the quantifier that is used must have special properties. First, it should enforce a property of genericity on proofs: we want the associated typing judgment to have a derivation that is independent of the choice of term for x. Second, we should be able to assume and to use the property that instantiation terms chosen for x are distinct from other terms appearing in the judgment, in particular, in Γ.

[2] We may choose to allow ∇-quantification at fewer types in particular applications; such a restriction may be useful in adequacy arguments for reasons we discuss later.

$$\frac{B \approx B'}{\Sigma : \Gamma, B \vdash B'}\ id \qquad \frac{\Sigma : \Gamma \vdash B \quad \Sigma : B, \Delta \vdash C}{\Sigma : \Gamma, \Delta \vdash C}\ cut \qquad \frac{\Sigma : \Gamma, B, B \vdash C}{\Sigma : \Gamma, B \vdash C}\ c\mathcal{L}$$

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

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

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

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

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

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

Figure 1: The core rules of $\mathcal{G}$

Neither the existential nor the universal quantifier have quite the characteristics needed for x in the encoding task considered. Miller and Tiu [16] therefore introduced the ∇-quantifier for this purpose. Using this quantifier, the typing rule can be represented by a formula like $\forall\Gamma, t, \alpha, \beta.(\nabla x.(\Gamma, x : \alpha \vdash t\,x : \beta)) \supset (\Gamma \vdash (\lambda x : \alpha.t\,x) : \alpha \to \beta)$ where t has a higher-order type which allows its dependency on x to be made explicit. The inference rules associated with the ∇-quantifier are designed to ensure the adequacy of such an encoding: the formula $\nabla x.F$, also called a *generic judgment*, must be established by deriving F assuming x to be a completely generic variable and in deriving $\nabla x \nabla y.F$ it is assumed that the instantiations for x and y are distinct. In the logic $\mathcal{G}$, we shall assume two further "structural" properties for the ∇-quantifier which flow naturally from the application domains of interest. First, we shall allow for ∇*-strengthening*, *i.e.*, we will take $\nabla x.F$ and F to be equivalent if x does not appear in F. Second, we shall take the relative order of ∇-quantifiers to be irrelevant, *i.e.*, we shall permit a ∇*-exchange* principle; the formulas $\nabla x \nabla y.F$ and $\nabla y \nabla x.F$ will be considered to be equivalent. These assumptions facilitate a simplification of the inference rules, allowing us to realize generic judgments through a special kind of constants called *nominal constants*.

2.3. A sequent calculus presentation of the core logic

The logic $\mathcal{G}$ assumes that the collection of constants is partitioned into the set $\mathcal{C}$ of nominal constants and the set $\mathcal{K}$ of usual, non-nominal constants. We assume the set $\mathcal{C}$

contains an infinite number of nominal constants for each type at which ∇ quantification is permitted. We define the *support* of a term (or formula), written $\mathrm{supp}(t)$, as the set of nominal constants appearing in it. A permutation of nominal constants is a type-preserving bijection π from $\mathcal{C}$ to $\mathcal{C}$ such that $\{x \mid \pi(x) \neq x\}$ is finite. The application of a permutation π to a term t, denoted by $\pi.t$, is defined as follows:

$$\begin{array}{ll} \pi.a = \pi(a),\ \text{if } a \in \mathcal{C} & \qquad \pi.c = c,\ \text{if } c \notin \mathcal{C} \text{ is atomic} \\ \pi.(\lambda x.M) = \lambda x.(\pi.M) & \qquad \pi.(M\ N) = (\pi.M)\ (\pi.N) \end{array}$$

We extend the notion of equality between terms to encompass also the application of permutations to nominal constants appearing in them. Specifically, the relation $B \approx B'$ holds if there is a permutation π such that B λ-converts to $\pi.B'$. Since λ-convertibility is an equivalence relation and permutations are invertible and composable, it follows that $\approx$ is an equivalence relation.

The rules defining the core of $\mathcal{G}$ are presented in Figure 1. Sequents in this logic have the form $\Sigma : \Gamma \vdash C$ where Γ is a multiset and the signature Σ contains all the free variables of Γ and C. In keeping with our restriction on quantification, we assume that o does not appear in the type of any variable in Σ. The expression $B[t/x]$ in the quantifier rules denotes the capture-avoiding substitution of t for x in the formula B. In the $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$ rules, a denotes a nominal constant of an appropriate type. In the $\exists\mathcal{L}$ and $\forall\mathcal{R}$ rule we use raising [25] to encode the dependency of the quantified variable on the support of B; the expression $(h\ \vec{c})$ in which h is a fresh eigenvariable is used in these two rules to denote the (curried) application of h to the constants appearing in the sequence $\vec{c}$. The $\forall\mathcal{L}$ and $\exists\mathcal{R}$ rules make use of judgments of the form $\Sigma, \mathcal{K}, \mathcal{C} \vdash t : \tau$. These judgments enforce the requirement that the expression t instantiating the quantifier in the rule is a well-formed term of type τ constructed from the eigenvariables in Σ and the constants in $\mathcal{K} \cup \mathcal{C}$. Notice that in contrast the $\forall\mathcal{R}$ and $\exists\mathcal{L}$ rules seem to allow for a dependency on only a restricted set of nominal constants. This asymmetry is not, however, significant: a consequence of Corollary 20 in Section 5 is that the dependency expressed through raising in the latter rules can be extended to any number of nominal constants that are not in the relevant support set without affecting the provability of sequents.

Equality modulo λ-conversion is built into the rules in Figure 1, and also into later extensions of this logic, in a fundamental way: in particular, proofs are preserved under the replacement of formulas in sequents by ones to which they λ-convert. A more involved observation is that we can replace a formula B in a sequent by another formula B' such that $B \approx B'$ without affecting the provability of the sequent or even the very structure of the proof. As a particular example, if a and b are nominal constants, then the following three sequents are all derivable: $P\ a \vdash P\ a$, $P\ b \vdash P\ b$, and $P\ a \vdash P\ b$. The last of these examples makes clear that nominal constants represent implicit quantification whose scope is limited to *individual formulas* in a sequent rather than ranging over the entire sequent. For the core logic, this observation follows from the form of the *id* rule and the fact that permutations distribute over logical structure. We shall prove this property explicitly for the full logic in Section 5.

The treatment of ∇-quantification via nominal constants also validates the ∇-exchange and ∇-strengthening principles discussed earlier. It is interesting to note that the latter principle implies that every type at which one is willing to use ∇-quantification is non-empty and, in fact, contains an unbounded number of members. For example, the formula $\exists_\tau x.\top$ is always provable, even if there are no closed terms of type τ because this formula is equivalent to $\nabla_\tau y.\exists_\tau x.\top$, which is provable. Similarly, for any given $n \geq 1$, the following formula is provable

$$\exists_\tau x_1 \ldots \exists_\tau x_n. \left[\bigwedge_{1 \leq i,j \leq n, i \neq j} x_i \neq x_j \right].$$

3. Characterizing Occurrences of Nominal Constants

We are interested in adding to our logic the capability of characterizing occurrences of nominal constants within terms and also of analyzing the structure of terms with respect to such occurrences. For example, we may want to define a predicate called *name* that holds of a term exactly when that term is a nominal constant. Similarly, we might need to identify a binary relation called *fresh* that holds between two terms just in the case that the first term is a nominal constant that does not occur in the second term. Towards supporting such possibilities, we define in this section a special binary relation called *nominal abstraction* and then present proof rules that incorporate an understanding of this relation into the logic. A formalization of these ideas requires a careful treatment of substitution. In particular, this operation must be defined to respect the intended formula level scope of nominal constants. We begin our discussion with an elaboration of this aspect.

3.1. Substitutions and their interaction with nominal constants

The following definition reiterates a common view of substitutions in logical contexts.

Definition 1. *A substitution is a type preserving mapping from variables to terms that is the identity at all but a finite number of variables. The domain of a substitution is the set of variables that are not mapped to themselves and its range is the set of terms resulting from applying it to the variables in its domain. We write a substitution as* $\{t_1/x_1, \ldots, t_n/x_n\}$ *where* $x_1, \ldots, x_n$ *is a list of variables that contains the domain of the substitution and* $t_1, \ldots, t_n$ *is the value of the map on these variables. The support of a substitution* θ*, written as* $\mathrm{supp}(\theta)$*, is the set of nominal constants that appear in the range of* θ*. The restriction of a substitution* θ *to the set of variables* Σ*, written as* $\theta \uparrow \Sigma$*, is a mapping that is like* θ *on the variables in* Σ *and the identity everywhere else.*

A substitution essentially calls for the replacement of variables by their associated terms in any context to which it is applied. A complicating factor is that we will want to consider substitutions in which nominal constants appear in the terms that are to replace particular variables. Such a substitution will typically be determined relative to one formula in a sequent but may then have to be applied to other formulas in the same sequent. In doing this, we have to take into account the fact that the scopes of the implicit quantifiers over nominal constants are restricted to individual formulas. Thus, the logically correct application of a substitution should be accompanied by a renaming of these nominal constants in the term being substituted into so as to ensure that they are not confused with the ones appearing in the range of the substitution. For example, consider the formula $p\ a\ x$ where a is a nominal constant and x is a variable; this formula is intended to be equivalent to $\nabla a.p\ a\ x$. If we were to substitute $f\ a$ for x naively into it, we would obtain the formula $p\ a\ (f\ a))$. However, this results in an unintended capture of a nominal constant by an (implicit) quantifier as a result of a substitution. To carry out the substitution in a way that avoids such capture, we should first rename the nominal constant a in $p\ a\ x$ to some other nominal constant b and then apply the substitution to produce the formula $p\ b\ (f\ a)$.

Definition 2. *The ordinary application of a substitution θ to a term B is denoted by $B[\theta]$ and corresponds to the replacement of the variables in B by the terms that θ maps them to, making sure, as usual, to avoid accidental binding of the variables appearing in the range of θ. More precisely, if $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$, then $B[\theta]$ is the term $(\lambda x_1 \ldots \lambda x_n.B)\ t_1\ \ldots\ t_n$; this term is, of course, considered to be equal to any other term to which it λ-converts. By contrast, the* nominal capture avoiding application *of θ to B is written as $B[\![\theta]\!]$ and is defined as follows. Assuming that π is a permutation of nominal constants that maps those appearing in* $\mathrm{supp}(B)$ *to ones not appearing in* $\mathrm{supp}(\theta)$*, let $B' = \pi.B$. Then $B[\![\theta]\!] = B'[\theta]$.*

The notation $B[\theta]$ generalizes the one used in the quantifier rules in Figure 1. This ordinary notion of substitution is needed to define such rules and it is used in the proof theory. As we will see in Section 5, however, it is nominal capture avoiding substitution that is the logically correct notion of substitution for $\mathcal{G}$ since it preserves the provability of sequents. For this reason, when we speak of the application of a substitution in an unqualified way, we shall mean the nominal capture avoiding form of this notion. It is interesting to note that as the treatment of syntax becomes richer and more abstract, the natural notions of equality of expressions and of substitution also change. When the syntax of terms is encoded as trees, term equality is tree equality and substitution corresponds to "grafting." When syntax involves binding operators (as in first-order formulas or λ-terms), then it is natural for equality to become λ-convertibility and for substitutions to be "capture-avoiding" in the usual sense. Here, we have introduced into syntax the additional notion of nominal constants, for which we need to upgrade equality to the $\approx$-relation and substitution to the one which avoids the capture of nominal constants.

The definition of the nominal capture avoiding application of a substitution is ambiguous in that we do not uniquely specify the permutation to be used. We resolve this ambiguity by deeming as acceptable *any* permutation that avoids conflicts. As a special instance of the lemma below, we see that for any given formula B and substitution θ, all the possible values for $B[\![\theta]\!]$ are equivalent modulo the $\approx$ relation. Moreover, as we show in Section 5, formulas that are equivalent under $\approx$ are interchangeable in the contexts of proofs.

Lemma 3. *If $t \approx t'$ then $t[\![\theta]\!] \approx t'[\![\theta]\!]$.*

Proof. Let t be λ-convertible to $\pi_1.t'$, let $t[\![\theta]\!] = (\pi_2.t)[\theta]$ where $\mathrm{supp}(\pi_2.t) \cap \mathrm{supp}(\theta) = \emptyset$, and let $t'[\![\theta]\!]$ be λ-convertible to $(\pi_3.t')[\theta]$ where $\mathrm{supp}(\pi_3.t') \cap \mathrm{supp}(\theta) = \emptyset$. Then we define a function π partially by the following rules:

1. $\pi(c) = \pi_2.\pi_1.\pi_3^{-1}(c)$ if $c \in \mathrm{supp}(\pi_3.t')$ and
2. $\pi(c) = c$ if $c \in \mathrm{supp}(\theta)$.

Since $\mathrm{supp}(\pi_3.t') \cap \mathrm{supp}(\theta) = \emptyset$, these rules are not contradictory, *i.e.*, this (partial) function is well-defined. The range of the first rule is $\mathrm{supp}(\pi_2.\pi_1.\pi_3^{-1}.\pi_3.t') = \mathrm{supp}(\pi_2.\pi_1.t') = \mathrm{supp}(\pi_2.t)$ which is disjoint from the range of the second rule, $\mathrm{supp}(\theta)$. Since the mapping in each rule is determined by a permutation, these rules together define a one-to-one partial mapping that can be extended to a bijection on $\mathcal{C}$. We take any such extension to be the complete definition of π that must therefore be a permutation.

To prove that $t[\![\theta]\!] \approx t'[\![\theta]\!]$ it suffices to show that if t is λ-convertible to $\pi_1.t'$ then $(\pi_2.t)[\theta]$ is λ-convertible to $\pi.((\pi_3.t')[\theta])$. We will prove this by induction on the structure of t'. Permutations and substitutions distribute over the structure of terms, thus the cases for when t' is an abstraction or application follow directly from the induction hypothesis. If t' is a nominal constant c then $(\pi_2.t)[\theta]$ must be λ-convertible to $(\pi_2.\pi_1.c)[\theta] = \pi_2.\pi_1.c$. Also, $\pi.((\pi_3.t')[\theta])$ must be λ-convertible to $\pi.\pi_3.c$. Further, in this case the first rule for π applies which means $\pi.\pi_3.c = \pi_2.\pi_1.\pi_3^{-1}.\pi_3.c = \pi_2.\pi_1.c$. Thus $(\pi_2.t)[\theta]$ is λ-convertible to $\pi.((\pi_3.t')[\theta])$. Finally, suppose t' is a variable x. In this case t must be λ-convertible to x so that we must show $x[\theta]$ λ-converts to $\pi.(x[\theta])$. If x does not have a binding in θ then both terms are equal. Alternatively, if $x[\theta] = s$ then $\pi.s = s$ follows from an inner induction on s and the second rule for π. Thus $(\pi_2.t)[\theta]$ λ-converts to $\pi.((\pi_3.t')[\theta])$, as is required. $\Box$

We shall need to consider the composition of substitutions later in this section. The definition of this notion must also pay attention to the presence of nominal constants.

Definition 4. *Given a substitution θ and a permutation π of nominal constants, let $\pi.\theta$ denote the substitution that is obtained by replacing each t/x in θ with $(\pi.t)/x$. Given any two substitutions θ and ρ, let $\theta \circ \rho$ denote the substitution that is such that $B[\theta \circ \rho] = B[\theta][\rho]$. In this context, the* nominal capture avoiding composition *of θ and ρ is written as $\theta \bullet \rho$ and defined as follows. Let π be a permutation of nominal constants such that* $\mathrm{supp}(\pi.\theta)$ *is disjoint from* $\mathrm{supp}(\rho)$. *Then $\theta \bullet \rho = (\pi.\theta) \circ \rho$.*

The notation $\theta \circ \rho$ in the above definition represents the usual composition of θ and ρ and can, in fact, be given in an explicit form based on these substitutions. Thus, $\theta \bullet \rho$ can also be presented in an explicit form. Notice that our definition of nominal capture avoiding composition is, once again, ambiguous because it does not fix the permutation to be used, accepting instead any one that satisfies the constraints. However, as before, this ambiguity is harmless. To understand this, we first extend the notion of equivalence under permutations to substitutions.

Definition 5. *Two substitutions θ and ρ are considered to be permutation equivalent, written $\theta \approx \rho$, if and only if there is a permutation of nominal constants π such that $\theta = \pi.\rho$. This notion of equivalence may also be parameterized by a set of variables Σ as follows: $\theta \approx_\Sigma \rho$ just in the case that $\theta \uparrow \Sigma \approx \rho \uparrow \Sigma$.*

It is easy to see that all possible choices for $\theta \bullet \rho$ are permutation equivalent and that if $\varphi_1 \approx \varphi_2$ then $B[\![\varphi_1]\!] \approx B[\![\varphi_2]\!]$ for any term B. Thus, if our focus is on provability, the ambiguity in Definition 4 is inconsequential by a result to be established in Section 5. As a further observation, note that $B[\![\theta \bullet \rho]\!] \approx B[\![\theta]\!][\![\rho]\!]$ for any B. Hence our notion of nominal capture avoiding composition of substitutions is sensible.

The composition operation can be used to define an ordering relation between substitutions:

Definition 6. *Given two substitutions ρ and θ, we say ρ is* less general than *θ, denoted by $\rho \leq \theta$, if and only if there exists a σ such that $\rho \approx \theta \bullet \sigma$. This relation can also be parametrized by a set of variables: ρ is less general than θ relative to Σ, written as $\rho \leq_\Sigma \theta$, if and only if $\rho \uparrow \Sigma \leq \theta \uparrow \Sigma$.*

The notion of generality between substitutions that is based on nominal capture avoiding composition has a different flavor from that based on the traditional form of substitution composition. For example, if a is a nominal constant, the substitution $\{a/x\}$ is strictly less general than $\{a/x, y'a/y\}$ relative to Σ for any Σ which contains x and y. To see this, note that we can compose the latter substitution with $\{(\lambda z.y)/y'\}$ to obtain the former, but the naive attempt to compose the former with $\{y'a/y\}$ yields $\{b/x, y'a/y\}$ where b is a nominal constant distinct from a. In fact, the "most general" solution relative to Σ containing $\{a/x\}$ will be $\{a/x\} \cup \{z'a/z \mid z \in \Sigma \backslash \{x\}\}$.

3.2. Nominal Abstraction

The nominal abstraction relation allows implicit formula level bindings represented by nominal constants to be moved into explicit abstractions over terms. The following notation is useful for defining this relationship.

Notation 7. *Let t be a term, let $c_1, \ldots, c_n$ be distinct nominal constants that possibly occur in t, and let $y_1, \ldots, y_n$ be distinct variables not occurring in t and such that, for $1 \leq i \leq n$, y_i and c_i have the same type. Then we write $\lambda c_1 \ldots \lambda c_n.t$ to denote the term $\lambda y_1 \ldots \lambda y_n.t'$ where t' is the term obtained from t by replacing c_i by y_i for $1 \leq i \leq n$.*

There is an ambiguity in the notation introduced above in that the choice of variables $y_1, \ldots, y_n$ is not fixed. However, this ambiguity is harmless: the terms that are produced by acceptable choices are all equivalent under a renaming of bound variables.

Definition 8. *Let $n \geq 0$ and let s and t be terms of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and τ, respectively; notice, in particular, that s takes n arguments to yield a term of the same type as t. Then the expression $s \unrhd t$ is a formula that is referred to as a nominal abstraction of degree n or simply as a nominal abstraction. The symbol $\unrhd$ is used here in an overloaded way in that the degree of the nominal abstraction it participates in can vary. The nominal abstraction $s \unrhd t$ of degree n is said to hold just in the case that s λ-converts to $\lambda c_1 \ldots c_n.t$ for some nominal constants $c_1, \ldots, c_n$.*

Clearly, nominal abstraction of degree 0 is the same as equality between terms based on λ-conversion, and we will therefore use $=$ to denote this relation in that situation. In the more general case, the term on the left of the operator serves as a pattern for isolating occurrences of nominal constants. For example, if p is a binary constructor and c_1 and c_2 are nominal constants, then the nominal abstractions of the following first row hold while those of the second do not.

$$\begin{array}{lll} \lambda x.x \unrhd c_1 & \lambda x.p\ x\ c_2 \unrhd p\ c_1\ c_2 & \lambda x.\lambda y.p\ x\ y \unrhd p\ c_1\ c_2 \\ \lambda x.x \ntrianglerighteq p\ c_1\ c_2 & \lambda x.p\ x\ c_2 \ntrianglerighteq p\ c_2\ c_1 & \lambda x.\lambda y.p\ x\ y \ntrianglerighteq p\ c_1\ c_1 \end{array}$$

The symbol $\unrhd$ corresponds, at the moment, to a mathematical relation that holds between pairs of terms as explicated by Definition 8. We now overload this symbol by treating it also as a binary predicate symbol of $\mathcal{G}$. In the next subsection we shall add inference rules to make the mathematical understanding of $\unrhd$ coincide with its syntactic use as a predicate in sequents. It is, of course, necessary to be able to determine when we mean to use $\unrhd$ in the mathematical sense and when as a logical symbol. When we write an expression such as $s \unrhd t$ without qualification, this should be read as a logical formula whereas if we say that "$s \unrhd t$ holds" then we are referring to the abstract relation from Definition 8. We might also sometimes use an expression such as "$(s \unrhd t)[\![\theta]\!]$ holds." In this case, we first treat $s \unrhd t$ as a formula to which we apply the substitution θ in a nominal capture avoiding way to get a (syntactic) expression of the form $s' \unrhd t'$. We then read $\unrhd$ in the mathematical sense, interpreting the overall expression as the assertion that "$s' \unrhd t'$ holds." Note in this context that $s \unrhd t$ constitutes a single formula when read syntactically and hence the expression $(s \unrhd t)[\![\theta]\!]$ is, in general, *not* equivalent to the expression $s[\![\theta]\!] \unrhd t[\![\theta]\!]$.

In the proof-theoretic setting, nominal abstraction will be used with terms that contain free occurrences of variables for which substitutions can be made. The following definition is relevant to this situation.

Definition 9. *A substitution θ is said to be a solution to the nominal abstraction $s \unrhd t$ just in the case that $(s \unrhd t)[\![\theta]\!]$ holds.*

Solutions to a nominal abstraction can be used to provide rich characterizations of the structures of terms. For example, consider the nominal abstraction $(\lambda x.\mathit{fresh}\ x\ T) \unrhd S$ in which T and S are variables and *fresh* is a binary predicate symbol. Any solution to this problem requires that S be substituted for by a term of the form $\mathit{fresh}\ a\ R$ where a is a nominal constant and R is a term in which a does not appear, *i.e.*, a must be "fresh" to R.

An important property of solutions to a nominal abstraction is that these are preserved under permutations to nominal constants. We establish this fact in the lemma below; this lemma will be used later in showing the stability of the provability of sequents with respect to the replacement of formulas by ones they are equivalent to modulo the $\approx$ relation.

Lemma 10. *Suppose $(s \unrhd t) \approx (s' \unrhd t')$. Then $s \unrhd t$ and $s' \unrhd t'$ have exactly the same solutions. In particular, $s \unrhd t$ holds if and only if $s' \unrhd t'$ holds.*

$$\frac{\{\Sigma\theta : \Gamma[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \text{ is a solution to } (s \trianglerighteq t)\}_\theta}{\Sigma : \Gamma, s \trianglerighteq t \vdash C} \trianglerighteq\mathcal{L} \qquad \frac{}{\Sigma : \Gamma \vdash s \trianglerighteq t} \trianglerighteq\mathcal{R},\ s \trianglerighteq t \text{ holds}$$

Figure 2: Nominal abstraction rules

$$\frac{\{\Sigma\theta : \Gamma[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \in \mathit{CSNAS}(\Sigma, s, t)\}_\theta}{\Sigma : \Gamma, s \trianglerighteq t \vdash C} \trianglerighteq\mathcal{L}_{\mathit{CSNAS}}$$

Figure 3: A variant of $\trianglerighteq\mathcal{L}$ based on *CSNAS*

Proof. We prove the particular result first. It suffices to show it in the forward direction since $\approx$ is symmetric. Let π be a permutation such that the expression $s' \trianglerighteq t'$ λ-converts to $\pi.(s \trianglerighteq t)$. Now suppose $s \trianglerighteq t$ holds since s λ-converts to $\lambda\vec{c}.t$. Then an inner induction on t' shows that s' λ-converts to $\lambda(\pi.\vec{c}).t'$ where $\pi.\vec{c}$ is the result of applying π to each element in the sequence $\vec{c}$. Thus $s' \trianglerighteq t'$ holds.

For the general result it again suffices to show it in one direction, *i.e.*, that all the solutions of $s \trianglerighteq t$ are solutions to $s' \trianglerighteq t'$. Let θ be a substitution such that $(s \trianglerighteq t)[\![\theta]\!]$ holds. By Lemma 3, $(s \trianglerighteq t)[\![\theta]\!] \approx (s' \trianglerighteq t')[\![\theta]\!]$. When the substitutions are carried out, this relation has the same form as the particular result from the first half of this proof, and thus $(s' \trianglerighteq t')[\![\theta]\!]$ holds. $\square$

3.3. Proof rules for nominal abstraction

We now add the left and right introduction rules for $\trianglerighteq$ that are shown in Figure 2 to link its use as a predicate symbol to its mathematical interpretation. The expression $\Sigma\theta$ in the $\trianglerighteq\mathcal{L}$ rule denotes the application of a substitution $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$ to the signature Σ that is defined to be the signature that results when removing from Σ the variables $\{x_1, \ldots, x_n\}$ and then adding every variable that is free in any term in $\{t_1, \ldots, t_n\}$. Notice also that in the same inference rule the operator $[\![\theta]\!]$ is applied to a multiset of formulas in the natural way: $\Gamma[\![\theta]\!] = \{B[\![\theta]\!] \mid B \in \Gamma\}$. Note that the $\trianglerighteq\mathcal{L}$ rule has an *a priori* unspecified number of premises that depends on the number of substitutions that are solutions to the relevant nominal abstraction. If $s \trianglerighteq t$ expresses an unsatisfiable constraint, meaning that it has no solutions, then the premise of $\trianglerighteq\mathcal{L}$ is empty and the rule provides an immediate proof of its conclusion.

The $\trianglerighteq\mathcal{L}$ and $\trianglerighteq\mathcal{R}$ rules capture nicely the intended interpretation of nominal abstraction. However, there is an obstacle to using the former rule in derivations: this rule has an infinite number of premises any time the nominal abstraction $s \trianglerighteq t$ has a solution. We can overcome this difficulty by describing a rule that includes only a few of these premises but in such way that their provability ensures the provability of all the other premises. Since the provability of $\Gamma \vdash C$ implies the provability of $\Gamma[\![\theta]\!] \vdash C[\![\theta]\!]$ for any θ (a property established formally in Section 5), if the first sequent is a premise of an occurrence of the $\trianglerighteq\mathcal{L}$ rule, the second does not need to be used as a premise of that same rule occurrence. Thus, we can limit the set of premises to be considered if we can identify with any given nominal abstraction a (possibly finite) set of solutions from which any other solution can be obtained through composition with a suitable substitution. The following definition formalizes the idea of such a "covering set."

Definition 11. *A* complete set of nominal abstraction solutions *(CSNAS) of s and t on Σ is a set S of substitutions such that*

1. *each $\theta \in S$ is a solution to $s \unrhd t$, and*
2. *for every solution ρ to $s \unrhd t$, there exists a $\theta \in S$ such that $\rho \leq_\Sigma \theta$.*

We denote any such set by $\mathit{CSNAS}(\Sigma, s, t)$.

Using this definition we present an alternative version of $\unrhd\mathcal{L}$ in Figure 3. Note that if we can find a finite complete set of nominal abstraction solutions then the number of premises to this rule will be finite.

Theorem 12. *The rules $\unrhd\mathcal{L}$ and $\unrhd\mathcal{L}_{\mathit{CSNAS}}$ are inter-admissible.*

Proof. Suppose we have the following arbitrary instance of $\unrhd\mathcal{L}$ in a derivation:

$$\frac{\{\Sigma\theta : \Gamma[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \text{ is a solution to } (s \unrhd t)\}_\theta}{\Sigma : \Gamma, s \unrhd t \vdash C} \unrhd\mathcal{L}$$

This rule can be replaced with a use of $\unrhd\mathcal{L}_{\mathit{CSNAS}}$ instead if we could be certain that, for each $\rho \in \mathit{CSNAS}(\Sigma, s, t)$, it is the case that $\Sigma\rho : \Gamma[\![\rho]\!] \vdash C[\![\rho]\!]$ is included in the set of premises of the shown rule instance. But this must be the case: by the definition of *CSNAS*, each such ρ is a solution to $s \unrhd t$.

In the other direction, suppose we have the following arbitrary instance of $\unrhd\mathcal{L}_{\mathit{CSNAS}}$.

$$\frac{\{\Sigma\theta : \Gamma[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \in \mathit{CSNAS}(\Sigma, s, t)\}_\theta}{\Sigma : \Gamma, s \unrhd t \vdash C} \unrhd\mathcal{L}_{\mathit{CSNAS}}$$

To replace this rule with a use of the $\unrhd\mathcal{L}$ rule instead, we need to be able to construct a derivation of $\Sigma\rho : \Gamma[\![\rho]\!] \vdash C[\![\rho]\!]$ for each ρ that is a solution to $s \unrhd t$. By the definition of *CSNAS*, we know that for any such ρ there exists a $\theta \in \mathit{CSNAS}(\Sigma, s, t)$ such that $\rho \leq_\Sigma \theta$, *i.e.*, such that there exists a σ for which $\rho \uparrow \Sigma \approx (\theta \uparrow \Sigma) \bullet \sigma$. Since we are considering the application of these substitutions to a sequent all of whose eigenvariables are contained in Σ, we can drop the restriction on the substitutions and suppose that $\rho \approx \theta \bullet \sigma$. Now, we shall show in Section 5 that if a sequent has a derivation then the result of applying a substitution to it in a nominal capture-avoiding way produces a sequent that also has a derivation. Using this observation, it follows that $\Sigma\theta\sigma : \Gamma[\![\theta]\!][\![\sigma]\!] \vdash C[\![\theta]\!][\![\sigma]\!]$ has a proof. But this sequent is permutation equivalent to $\Sigma\rho : \Gamma[\![\rho]\!] \vdash C[\![\rho]\!]$ which must, again by a result established explicitly in Section 5, also have a proof. □

Theorem 12 allows us to choose which of the left rules we wish to consider in any given context. We shall assume the $\unrhd\mathcal{L}$ rule in the formal treatment in the rest of this paper, leaving the use of the $\unrhd\mathcal{L}_{\mathit{CSNAS}}$ rule to practical applications of the logic.

3.4. Computing complete sets of nominal abstraction solutions

For the $\trianglerighteq\mathcal{L}_{CSNAS}$ rule to be useful, we need an effective way to compute restricted complete sets of nominal abstraction solutions. We show here that the task of finding such complete sets of solutions can be reduced to that of finding complete sets of unifiers (*CSU*) for higher-order unification problems [26]. In the straightforward approach to finding a solution to a nominal abstraction $s \trianglerighteq t$, we would first identify a substitution θ that we apply to $s \trianglerighteq t$ to get $s' \trianglerighteq t'$ and we would subsequently look for nominal constants to abstract from t' to get s'. To relate this problem to the usual notion of unification, we would like to invert this order: in particular, we would like to consider all possible ways of abstracting over nominal constants first and only later think of applying substitutions to make the terms equal. The difficulty with this second approach is that we do not know which nominal constants might appear in t' until after the substitution is applied. However, there is a way around this problem. Given the nominal abstraction $s \trianglerighteq t$ of degree n, we first consider substitutions for the variables occurring in it that introduce n new nominal constants in a completely general way. Then we consider all possible ways of abstracting over the nominal constants appearing in the altered form of t and, for each of these cases, we look for a complete set of unifiers.

The idea described above is formalized in the following definition and associated theorem. We use the notation $\mathit{CSU}(s,t)$ in them to denote an arbitrary but fixed selection of a complete set of unifiers for the terms s and t.

Definition 13. *Let s and t be terms of type $\tau_1 \to \ldots \to \tau_n \to \tau$ and τ, respectively. Let $c_1, \ldots, c_n$ be n distinct nominal constants disjoint from $\mathrm{supp}(s \trianglerighteq t)$ such that, for $1 \le i \le n$, c_i has the type τ_i. Let Σ be a set of variables and for each $h \in \Sigma$ of type τ', let h' be a distinct variable not in Σ that has type $\tau_1 \to \ldots \to \tau_n \to \tau'$. Let $\sigma = \{h'\ c_1\ \ldots\ c_n/h \mid h \in \Sigma\}$ and let $s' = s[\sigma]$ and $t' = t[\sigma]$. Let*

$$C = \bigcup_{\vec{a}} \mathit{CSU}(\lambda\vec{b}.s', \lambda\vec{b}.\lambda\vec{a}.t')$$

where $\vec{a} = a_1, \ldots, a_n$ ranges over all selections of n distinct nominal constants from $\mathrm{supp}(t) \cup \{\vec{c}\}$ such that, for $1 \le i \le n$, a_i has type τ_i and $\vec{b}$ is some corresponding listing of all the nominal constants in s' and t' that are not included in $\vec{a}$. Then we define

$$S(\Sigma, s, t) = \{\sigma \bullet \rho \mid \rho \in C\}$$

The use of the substitution σ above represents another instance of the application of the general technique of raising that allows certain variables (the h variables in this definition) whose substitution instances might depend on certain nominal constants ($c_1, \ldots, c_n$ here) to be replaced by new variables of higher type (the h' variables) whose substitution instances are not allowed to depend on those nominal constants. This technique was previously used in the $\exists\mathcal{L}$ and $\forall\mathcal{R}$ rules presented in Section 2.

An important observation concerning Definition 13 is that it requires us to consider all possible (ordered) selections $a_1, \ldots, a_n$ of distinct nominal constants from $\mathrm{supp}(t) \cup \{\vec{c}\}$. The set of such selections is potentially large, having in it at least $n!$ members. However, in

$$\frac{\Sigma : \Gamma, B\ p\ \vec{t} \vdash C}{\Sigma : \Gamma, p\ \vec{t} \vdash C}\ \mathit{def}\mathcal{L} \qquad\qquad \frac{\Sigma : \Gamma \vdash B\ p\ \vec{t}}{\Sigma : \Gamma \vdash p\ \vec{t}}\ \mathit{def}\mathcal{R}$$

Figure 4: Introduction rules for atoms whose predicate is defined as $\forall \vec{x}.\ p\ \vec{x} \triangleq B\ p\ \vec{x}$

the uses that we have seen of $\mathcal{G}$ in reasoning tasks, n is typically small, often either 1 or 2. Moreover, in these reasoning applications, the cardinality of the set $\mathrm{supp}(t) \cup \{\vec{c}\}$ is also usually small.

Theorem 14. *$S(\Sigma, s, t)$ is a complete set of nominal abstraction solutions for $s \unrhd t$ on Σ.*

Proof. First note that $\mathrm{supp}(\sigma) \cap \mathrm{supp}(s \unrhd t) = \emptyset$ and thus $(s \unrhd t)[\![\theta]\!]$ is equal to $(s' \unrhd t')$. Now we must show that every element of $S(\Sigma, s, t)$ is a solution to $s \unrhd t$. Let $\sigma \bullet \rho \in S(\Sigma, s, t)$ be an arbitrary element where σ is as in Definition 13, ρ is from $\mathit{CSU}(\lambda\vec{b}.s', \lambda\vec{b}.\lambda\vec{a}.t')$, and $s' = s[\sigma]$ and $t' = t[\sigma]$. By the definition of CSU we know $(\lambda\vec{b}.s' = \lambda\vec{b}.\lambda\vec{a}.t')[\rho]$. This means $(s' = \lambda\vec{a}.t')[\![\rho]\!]$ holds and thus $(s' \unrhd t')[\![\rho]\!]$ holds. Rewriting s' and t' in terms of s and t this means $(s \unrhd t)[\![\sigma]\!][\![\rho]\!]$. Thus $\sigma \bullet \rho$ is a solution to $s \unrhd t$.

In the other direction, we must show that if θ is a solution to $s \unrhd t$ then there exists $\sigma \bullet \rho \in S(\Sigma, s, t)$ such that $\theta \leq_\Sigma \sigma \bullet \rho$. Let θ be a solution to $s \unrhd t$. Then we know $(s \unrhd t)[\![\theta]\!]$ holds. The substitution θ may introduce some nominal constants which are abstracted out of the right-hand side when determining equality, so let us call these the *important* nominal constants. Let $\sigma = \{h'\ c_1\ \ldots\ c_n/h \mid h \in \Sigma\}$ be as in Definition 13 and let π' be a permutation which maps the important nominal constants of θ to nominal constants from $c_1, \ldots, c_n$. This is possible since n nominal constants are abstract from the right-hand side and thus there are at most n important nominal constants. Then let $\theta' = \pi'.\theta$, so that $(s \unrhd t)[\![\theta']\!]$ holds and it suffices to show that $\theta' \leq_\Sigma \sigma \bullet \rho$. Note that all we have done at this point is to rename the important nominal constants of θ so that they match those introduced by σ. Now we define $\rho' = \{\lambda c_1 \ldots \lambda c_n.r/h' \mid r/h \in \theta'\}$ so that $\theta' = \sigma \bullet \rho'$. Thus $(s \unrhd t)[\![\sigma]\!][\![\rho']\!]$ holds. By construction, σ shares no nominal constants with s and t, thus we know $(s' \unrhd t')[\![\rho']\!]$ where $s' = s[\sigma]$ and $t' = t[\sigma]$. Also by construction, ρ' contains no important nominal constants and thus $(s' = \lambda\vec{a}.t')[\![\rho]\!]$ holds for some nominal constants $\vec{a}$ taken from $\mathrm{supp}(t) \cup \{\vec{c}\}$. If we let $\vec{b}$ be a listing of all nominal constants in s' and t' but not in $\vec{a}$, then $(\lambda\vec{b}.s' = \lambda\vec{b}.\lambda\vec{a}.t')[\![\rho]\!]$ holds. At this point the inner equality has no nominal constants and thus the substitution ρ can be applied without renaming: $(\lambda\vec{b}.s' = \lambda\vec{b}.\lambda\vec{a}.t')[\rho']$ holds. By the definition of CSU, there must be a $\rho \in \mathit{CSU}(\lambda\vec{b}.s', \lambda\vec{b}.\lambda\vec{a}.t')$ such that $\rho' \leq \rho$. Thus $\sigma \bullet \rho' \leq_\Sigma \sigma \bullet \rho$ as desired. □

4. Definitions, Induction, and Co-induction

The sequent calculus rules presented in Figure 1 treat atomic judgments as fixed, unanalyzed objects. We now add the capability of defining such judgments by means of formulas, possibly involving other predicates. In particular, we shall assume that we are given a fixed, finite set of *clauses* of the form $\forall \vec{x}.\ p\ \vec{x} \triangleq B\ p\ \vec{x}$ where p is a predicate constant that takes a number of arguments equal to the length of $\vec{x}$. Such a clause is said to define p and the entire collection of clauses is called a *definition*. The expression B, called the *body* of the clause, must be a term that does not contain p or any of the variables in $\vec{x}$ and must have a type such that $B\ p\ \vec{x}$ has type o. Definitions are also restricted so that a predicate is defined by at most one clause. The intended interpretation of a clause $\forall \vec{x}.\ p\ \vec{x} \triangleq B\ p\ \vec{x}$ is that the atomic formula $p\ \vec{t}$, where $\vec{t}$ is a list of terms of the same length and type as the variables in $\vec{x}$, is true if and only if $B\ p\ \vec{t}$ is true. This interpretation is realized by adding to the calculus the rules $def\mathcal{L}$ and $def\mathcal{R}$ shown in Figure 4 for unfolding predicates on the left and the right of sequents using their defining clauses.

Definitions can have a recursive structure. In particular, the predicate p can appear free in the body $B\ p\ \vec{x}$ of a clause of the form $\forall \vec{x}.\ p\ \vec{x} \triangleq B\ p\ \vec{x}$. A fixed-point interpretation is intended for definitions with clauses that are recursive in this way. Additional restrictions are needed to ensure that fixed points actually exist in this setting and that their use is compatible with the embedding logic. Two particular constraints suffice for this purpose. First, the body of a clause must not contain any nominal constants. This restriction can be justified from another perspective as well: as we see in Section 5, it helps in establishing that $\approx$ is a provability preserving equivalence between formulas. Second, definitions should be *stratified* so that clauses, such as $a \triangleq (a \supset \bot)$, in which a predicate has a negative dependency on itself, are forbidden. While such stratification can be enforced in different ways, we use a simple approach to doing this in this paper. This approach is based on associating with each predicate p a natural number that is called its *level* and that is denoted by $\mathrm{lvl}(p)$. This measure is then extended to arbitrary formulas by the following definition.

Definition 15. *Given an assignment of levels to predicates, the function* lvl *is extended to all formulas in λ-normal form as follows:*

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

In general, the level of a formula B, written as $\mathrm{lvl}(B)$*, is the level of its λ-normal form.*

A definition is *stratified* if we can assign levels to predicates in such a way that $\mathrm{lvl}(B\ p\ \vec{x}) \leq \mathrm{lvl}(p)$ for each clause $\forall \vec{x}.\ p\ \vec{x} \triangleq B\ p\ \vec{x}$ in that definition.

The $def\mathcal{L}$ and $def\mathcal{R}$ rules do not discriminate between any of the fixed points of a definition. We now allow for the selection of least and greatest fixed points so as to support inductive and co-inductive definitions of predicates. Specifically, we denote an inductive clause by $\forall \vec{x}.\ p\ \vec{x} \stackrel{\mu}{=} B\ p\ \vec{x}$ and a co-inductive one by $\forall \vec{x}.\ p\ \vec{x} \stackrel{\nu}{=} B\ p\ \vec{x}$. As a refinement of the earlier restriction on definitions, a predicate may have at most one defining clause that is designated to be inductive, co-inductive or neither. The $def\mathcal{L}$ and $def\mathcal{R}$ rules may be used with clauses in any one of these forms. Clauses that are inductive admit additionally the left rule $\mathcal{IL}$ shown in Figure 5. This rule is based on the observation that the least fixed point of

$$\frac{\vec{x} : B\ S\ \vec{x} \vdash S\ \vec{x} \qquad \Sigma : \Gamma, S\ \vec{t} \vdash C}{\Sigma : \Gamma, p\ \vec{t} \vdash C}\ \mathcal{IL}$$

provided p is defined as $\forall \vec{x}.\ p\ \vec{x} \stackrel{\mu}{=} B\ p\ \vec{x}$ and S is a term that has the same type as p and does not contain nominal constants

$$\frac{\Sigma : \Gamma \vdash S\ \vec{t} \qquad \vec{x} : S\ \vec{x} \vdash B\ S\ \vec{x}}{\Sigma : \Gamma \vdash p\ \vec{t}}\ \mathcal{CIR}$$

provided p is defined as $\forall \vec{x}.\ p\ \vec{x} \stackrel{\nu}{=} B\ p\ \vec{x}$ and S is a term that has the same type as p and does not contain nominal constants

Figure 5: The induction left and co-induction right rules

a monotone operator is the intersection of all its pre-fixed points; intuitively, anything that follows from any pre-fixed point should then also follow from the least fixed point. In a proof search setting, the term corresponding to the schema variable S in this rule functions like the induction hypothesis and is accordingly called the invariant of the induction. Clauses that are co-inductive, on the other hand, admit the right rule $\mathcal{CIR}$ also presented in Figure 5. This rule reflects the fact that the greatest fixed point of a monotone operator is the union of all the post-fixed points; any member of such a post-fixed point must therefore also be a member of the greatest fixed point. The substitution that is used for S in this rule is called the co-invariant or the simulation of the co-induction. Just like the restriction on the body of clauses, in both $\mathcal{IL}$ and $\mathcal{CIR}$, the (co-)invariant S must not contain any nominal constants.

As a simple illustration of the use of these rules, consider the clause $p \stackrel{\mu}{=} p$. The desired inductive reading of this clause implies that p must be false. In a proof-theoretic setting, we would therefore expect that the sequent $\cdot : p \vdash \bot$ can be proved. This can, in fact, be done by using $\mathcal{IL}$ with the invariant $S = \bot$. On the other hand, consider the clause $q \stackrel{\nu}{=} q$. The co-inductive reading intended here implies that q must be true. The logic $\mathcal{G}$ satisfies this expectation: the sequent $\cdot : \cdot \vdash q$ can be proved using $\mathcal{CIR}$ with the co-invariant $S = \top$.

The addition of inductive and co-inductive forms of clauses and the mixing of these forms in one setting requires a stronger stratification condition to guarantee consistency. One condition that suffices and that is also practically acceptable is the following that is taken from [27]: in a clause of any of the forms $\forall \vec{x}.\ p\ \vec{x} \triangleq B\ p\ \vec{x}$, $\forall \vec{x}.\ p\ \vec{x} \stackrel{\mu}{=} B\ p\ \vec{x}$ or $\forall \vec{x}.\ p\ \vec{x} \stackrel{\nu}{=} B\ p\ \vec{x}$, it must be that $\mathrm{lvl}(B\ (\lambda \vec{x}.\top)\ \vec{x}) < \mathrm{lvl}(p)$. This disallows any mutual recursion between clauses, a restriction which can easily be overcome by merging mutually recursive clauses into a single clause. We henceforth assume that all definitions satisfy all three conditions described for them in this section. Corollary 22 in Section 5 establishes the consistency of the logic under these restrictions.

5. Some Properties of the Logic

We have now described the logic $\mathcal{G}$ completely: in particular, its proof rules consist of the ones in Figures 1, 2, 4 and 5. This logic combines and extends the features of several logics such as $FO\lambda^{\Delta\mathbb{N}}$ [5], $FO\lambda^{\Delta\nabla}$ [16], LG^{ω} [28] and Linc^{-} [27]. The relationship to Linc^{-} is of special interest to us below: $\mathcal{G}$ is a conservative extension to this logic that is obtained by adding a treatment of the ∇ quantifier and the associated nominal constants and by generalizing the proof rules pertaining to equality to ones dealing with nominal abstraction. This correspondence will allow the proof of the critical meta-theoretic property of cut-elimination for Linc^{-} to be lifted to $\mathcal{G}$.

We shall actually establish three main properties of $\mathcal{G}$ in this section. First, we shall show that the provability of a sequent is unaffected by the application of permutations of nominal constants to formulas in the sequent. This property consolidates our understanding that nominal constants are quantified implicitly at the formula level; such quantification also renders irrelevant the particular names chosen for such constants. Second, we show that the application of substitution in a nominal capture-avoiding way preserves provability; by contrast, ordinary application of substitution does not have this property. Finally, we show that the *cut* rule can be dispensed with from the logic without changing the set of provable sequents. This implies that the left and right rules of the logic are balanced and, moreover, that the logic is consistent. This is the main result of this section and its proof uses the earlier two results together with the argument for cut-elimination for Linc^{-}.

Several of our arguments will be based on induction on the heights of proofs. This measure is defined formally below. Notice that the height of a proof can be an infinite ordinal because the $\trianglerighteq\mathcal{L}$ rule can have an infinite number of premises. Thus, we will be using a transfinite form of induction.

Definition 16. *The* height *of a derivation* Π*, denoted by* $\mathrm{ht}(\Pi)$*, is* 1 *if* Π *has no premise derivations and is the least upper bound of* $\{\mathrm{ht}(\Pi_i)+1\}_{i\in\mathcal{I}}$ *if* Π *has the premise derivations* $\{\Pi_i\}_{i\in\mathcal{I}}$ *where* $\mathcal{I}$ *is some index set. Note that the typing derivations in the rules* $\forall\mathcal{L}$ *and* $\exists\mathcal{R}$ *are not considered premise derivations in this sense.*

Many proof systems, such as Linc^{-}, include a weakening rule that allows formulas to be dropped (reading proofs bottom-up) from the left-hand sides of sequents. While $\mathcal{G}$ does not include such a rule directly, its effect is captured in a strong sense as we show in the lemma below. Two proofs are to be understood here and elsewhere as having the same structure if they are isomorphic as trees, if the same rules appear at corresponding places within them and if these rules pertain to formulas that can be obtained one from the other via a renaming of eigenvariables and nominal constants.

Lemma 17. *Let* Π *be a proof of* $\Sigma : \Gamma \vdash B$ *and let* Δ *be a multiset of formulas whose eigenvariables are contained in* Σ*. Then there exists a proof of* $\Sigma : \Delta, \Gamma \vdash B$ *which has the same structure as* Π*. In particular* $\mathrm{ht}(\Pi) = \mathrm{ht}(\Pi')$ *and* Π *and* Π' *end with the same rule application.*

Proof. The lemma can be proved by an easy induction on $\mathrm{ht}(\Pi)$. We omit the details. $\square$

The following lemma shows a strong form of the preservation of provability under permutations of nominal constants appearing in formulas, the first of our mentioned results.

Lemma 18. *Let Π be a proof of $\Sigma : B_1, \ldots, B_n \vdash B_0$ and let $B_i \approx B_i'$ for $i \in \{0, 1, \ldots, n\}$. Then there exists a proof Π' of $\Sigma : B_1', \ldots, B_n' \vdash B_0'$ which has the same structure as Π. In particular $\mathrm{ht}(\Pi) = \mathrm{ht}(\Pi')$ and Π and Π' end with the same rule application.*

Proof. The proof is by induction on $\mathrm{ht}(\Pi)$ and proceeds specifically by considering the last rule used in Π. When this is a left rule, we shall assume without loss of generality that it operates on B_n.

The argument is easy to provide when the last rule in Π is one of $\bot\mathcal{L}$ or $\top\mathcal{R}$. If this rule is an *id*, *i.e.*, if Π is of the form

$$\frac{B_j \approx B_0}{\Sigma : B_1, \ldots, B_n \vdash B_0} \; id$$

then, since $\approx$ is an equivalence relation, it must be the case that $B_j' \approx B_0'$. Thus, we can let Π' be the derivation

$$\frac{B_j' \approx B_0'}{\Sigma : B_1', \ldots, B_n' \vdash B_0'} \; id$$

If the last rule is a $\trianglerighteq\mathcal{R}$ applied to a nominal abstraction $s \trianglerighteq t$ then the result follows immediately from Lemma 10.

In the remaining cases we shall show that the last rule in Π can also have $\Sigma : B_1', \ldots, B_n' \vdash B_0'$ as a conclusion with the premises in this application of the rule being related via permutations in the way required by the lemma to the premises of the rule application in Π. The lemma then follows from the induction hypothesis.

In the case when the last rule in Π pertains to a binary connective—*i.e.*, when the rule is one of $\lor\mathcal{L}$, $\lor\mathcal{R}$, $\land\mathcal{L}$, $\land\mathcal{R}$, $\supset\mathcal{L}$ or $\supset\mathcal{R}$—the desired conclusion follows naturally from the observation that permutations distribute over the connective. The proof can be similarly completed when a $\exists\mathcal{L}$, $\exists\mathcal{R}$, $\forall\mathcal{L}$ or $\forall\mathcal{R}$ rule ends the derivation, once we have noted that the application of permutations can be moved under the $\exists$ and $\forall$ quantifiers. For the *cut* and $c\mathcal{L}$ rules, we have to show that permutations can be extended to include the newly introduced formula in the upper sequent(s). This is easy: for the *cut* rule we use the identity permutation and for $c\mathcal{L}$ we replicate the permutation used to obtain B_n' from B_n.

The two remaining rules from the core logic are $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$. The argument in these cases are similar and we consider only the later in detail. In this case, the last rule in Π is of the form

$$\frac{\Sigma : B_1, \ldots, B_n \vdash C[a/x]}{\Sigma : B_1, \ldots, B_n \vdash \nabla x.C} \; \nabla\mathcal{R}$$

where $a \notin \mathrm{supp}(C)$. Obviously, $B_0' = \nabla x.C'$ for some C' such that $C \approx C'$. Let d be a nominal constant such that $d \notin \mathrm{supp}(C)$ and $d \notin \mathrm{supp}(C')$. Such a constant must exist since both sets are finite. Then $C[a/x] \approx C[d/x] \approx C'[d/x]$. Thus the following

$$\frac{\Sigma : B_1', \ldots, B_n' \vdash C'[d/x]}{\Sigma : B_1', \ldots, B_n' \vdash \nabla x.C'} \; \nabla\mathcal{R}$$

is also an instance of the $\nabla\mathcal{R}$ rule and its upper sequent has the desired form.

When the last rule in Π is $\trianglerighteq\mathcal{L}$, it has has the structure

$$\frac{\{\Sigma\theta : B_1[\![\theta]\!], \ldots, B_{n-1}[\![\theta]\!] \vdash B_0[\![\theta]\!] \mid \theta \text{ is a solution to } s \trianglerighteq t\}}{\Sigma : B_1, \ldots, s \trianglerighteq t \vdash B_0} \trianglerighteq\mathcal{L}$$

Here we know that B'_n is a nominal abstraction $s' \trianglerighteq t'$ that, by Lemma 10, has the same solutions as $s \trianglerighteq t$. Further, by Lemma 3, $B_i[\![\theta]\!] \approx B'_i[\![\theta]\!]$ for any substitution θ. Thus

$$\frac{\left\{\Sigma\theta : B'_1[\![\theta]\!], \ldots, B'_{n-1}[\![\theta]\!] \vdash B'_0[\![\theta]\!] \mid \theta \text{ is a solution to } s' \trianglerighteq t'\right\}}{\Sigma : B'_1, \ldots, s' \trianglerighteq t' \vdash B'_0} \trianglerighteq\mathcal{L}$$

is also an instance of the $\trianglerighteq\mathcal{L}$ rule and its upper sequents have the required property.

The arguments for the rules $\mathit{def}\mathcal{L}$ and $\mathit{def}\mathcal{R}$ are similar and we therefore only consider the case for the former rule in detail. Here, B_n must be of the form $p\,\vec{t}$ where p is a predicate symbol and the upper sequent must be identical to the lower one except for the fact that B_n is replaced by a formula of the form $B\,p\,\vec{t}$ where B contains no nominal constants. Further, B'_n is of the form $p\,\vec{s}$ where $p\,\vec{t} \approx p\,\vec{s}$. From this it follows that $B\,p\,\vec{t} \approx B\,p\,\vec{s}$ and hence that $\Sigma : B'_1, \ldots, B'_n \vdash B'_0$ can be the lower sequent of a rule whose upper sequent is related in the desired way via permutations to the upper sequent of the last rule in Π.

The only remaining rules to consider are $\mathcal{IL}$ and $\mathcal{CIR}$. Once again, the arguments in these cases are similar and we therefore consider only the case for $\mathcal{IL}$ in detail. Here, Π ends with a rule of the form

$$\frac{\vec{x} : B\,S\,\vec{x} \vdash S\,\vec{x} \qquad \Sigma : B_1, \ldots, S\,\vec{t} \vdash B_0}{\Sigma : B_1, \ldots, p\,\vec{t} \vdash B_0} \mathcal{IL}$$

where p is a predicate symbol defined by a clause of the form $\forall \vec{x}.\, p\,\vec{x} \stackrel{\mu}{=} B\,p\,\vec{x}$ and S contains no nominal constants. Now, B'_n must be of the form $p\,\vec{r}$ where $p\,\vec{t} \approx p\,\vec{r}$. Noting the proviso on S, it follows that $S\,\vec{t} \approx S\,\vec{r}$. But then the following

$$\frac{\vec{x} : B\,S\,\vec{x} \vdash S\,\vec{x} \qquad \Sigma : B'_1, \ldots, S\,\vec{r} \vdash B'_0}{\Sigma : B'_1, \ldots, p\,\vec{r} \vdash B'_0} \mathcal{IL}$$

is also an instance of the $\mathcal{IL}$ rule and its upper sequents are related in the manner needed to those of the $\mathcal{IL}$ rule used in Π. $\square$

Several rules in $\mathcal{G}$ require the selection of eigenvariables and nominal constants. Lemma 18 shows that we obtain what is essentially the same proof regardless of how we choose nominal constants in such rules so long as the local non-occurrence conditions are satisfied. A similar observation with regard to the choice of eigenvariables is also easily verified. We shall therefore identify below proofs that differ only in the choices of eigenvariables and nominal constants.

We now turn to the second of our desired results, the preservation of provability under substitutions.

Lemma 19. *Let* Π *be a proof of* $\Sigma : \Gamma \vdash C$ *and let* θ *be a substitution. Then there is a proof* Π' *of* $\Sigma\theta : \Gamma[\![\theta]\!] \vdash C[\![\theta]\!]$ *such that* $\mathrm{ht}(\Pi') \leq \mathrm{ht}(\Pi)$*.*

Proof. We show how to transform the proof Π into a proof Π' for the modified sequent. The transformation is by recursion on $\mathrm{ht}(\Pi)$, the critical part of it being a consideration of the last rule in Π. The transformation is, in fact, straightforward in all cases other than when this rule is $\trianglerighteq\mathcal{L}$, $\forall\mathcal{R}$, $\exists\mathcal{L}$, $\exists\mathcal{R}$, $\forall\mathcal{L}$, $\mathcal{IL}$ and $\mathcal{CIR}$. In these cases, we simply apply the substitution in a nominal capture avoiding way to the lower and any possible upper sequents of the rule. It is easy to see that the resulting structure is still an instance of the same rule and its upper sequents are guaranteed to have proofs (of suitable heights) by induction.

Suppose that the last rule in Π is an $\trianglerighteq\mathcal{L}$, *i.e.*, it is of the form

$$\frac{\{\Sigma\rho : \Gamma[\![\rho]\!] \vdash C[\![\rho]\!] \mid \rho \text{ is a solution to } s \trianglerighteq t\}}{\Sigma : \Gamma, s \trianglerighteq t \vdash C} \trianglerighteq\mathcal{L}$$

Then the following

$$\frac{\{\Sigma(\theta \bullet \rho') : \Gamma[\![\theta \bullet \rho']\!] \vdash C[\![\theta \bullet \rho']\!] \mid \rho' \text{ is a solution to } (s \trianglerighteq t)[\![\theta]\!]\}}{\Sigma\theta : \Gamma[\![\theta]\!], (s \trianglerighteq t)[\![\theta]\!] \vdash C[\![\theta]\!]} \trianglerighteq\mathcal{L}$$

is also an $\trianglerighteq\mathcal{L}$ rule. Noting that if ρ' is a solution to $(s \trianglerighteq t)[\![\theta]\!]$, then $\theta \bullet \rho'$ is a solution to $s \trianglerighteq t$, we see that the upper sequents of this rule are contained in the upper sequents of the rule in Π. It follows that we can construct a proof of the lower sequent whose height is less than or equal to that of Π.

The argument is similar in the cases when the last rule in Π is a $\forall\mathcal{R}$ or a $\exists\mathcal{L}$ so we consider only the former in detail. In this case the rule has the form

$$\frac{\Sigma, h : \Gamma \vdash B[h\ \vec{c}/x]}{\Sigma : \Gamma \vdash \forall x.B} \forall\mathcal{R}$$

where $\{\vec{c}\} = \mathrm{supp}(\forall x.B)$. Let $\{\vec{a}\} = \mathrm{supp}((\forall x.B)[\![\theta]\!])$. Further, let h' be a new variable name. We assume without loss of generality that neither h nor h' appear in the domain or range of θ. Letting $\rho = \theta \cup \{\lambda\vec{c}.h'\ \vec{a}/h\}$, consider the structure

$$\frac{(\Sigma, h)\rho : \Gamma[\![\rho]\!] \vdash B[h\ \vec{c}/x][\![\rho]\!]}{\Sigma\theta : \Gamma[\![\theta]\!] \vdash (\forall x.B)[\![\theta]\!]}$$

The upper sequent here is equivalent under λ-conversion to $\Sigma\theta, h' : \Gamma[\![\theta]\!] \vdash (B[\![\theta]\!])[h'\ \vec{a}/x]$ so this structure is, in fact, also an instance of the $\forall\mathcal{R}$ rule. Moreover, its upper sequent is obtained via substitution from the upper sequent of the rule in Π. The lemma then follows by induction.

The arguments for the cases when the last rule is an $\exists\mathcal{R}$ or an $\forall\mathcal{L}$ are similar and so we provide it explicitly only for the former. In this case, we have the rule

$$\frac{\Sigma, \mathcal{K}, \mathcal{C} \vdash t : \tau \quad \Sigma : \Gamma \vdash B[t/x]}{\Sigma : \Gamma \vdash \exists_\tau x.B} \exists\mathcal{R}$$

ending Π. Let π be a permutation such that $\text{supp}(\pi.(B[t/x])) \cap \text{supp}(\theta) = \emptyset$. We assume without loss of generality that x does not appear in the domain or range of θ. Then consider the structure

$$\frac{\Sigma\theta, \mathcal{K}, \mathcal{C} \vdash (\pi.t)[\theta] : \tau \quad \Sigma\theta : \Gamma[\![\theta]\!] \vdash (\pi.B)[\theta][(\pi.t)[\theta]/x]}{\Sigma\theta : \Gamma[\![\theta]\!] \vdash (\exists_\tau x.B)[\![\theta]\!]}$$

The typing derivation here is well-formed since permutations and substitutions are type preserving. Additionally, $\text{supp}(B) \subseteq \text{supp}(B[t/x])$ implies $\text{supp}(\pi.B) \cap \text{supp}(\theta) = \emptyset$, and so the conclusion of the lower sequent is equivalent to $\exists_\tau x.(\pi.B)[\theta]$. Thus this structure is an instance of the $\exists\mathcal{R}$ rule. The term $(\pi.B)[\theta][(\pi.t)[\theta]/x]$ is equal to $(\pi.(B[t/x]))[\theta]$ which is equivalent to $(B[t/x])[\![\theta]\!]$. Thus the upper right sequent is obtained via substitution from the upper right sequent of the rule in Π. The lemma then follows by induction.

The only remaining cases for the last rule are $\mathcal{IL}$ and $\mathcal{CIR}$. The arguments in these cases are, yet again, similar and it suffices to make only the former explicit. In this case, the end of Π has the form

$$\frac{\vec{x} : B\ S\ \vec{x} \vdash S\ \vec{x} \quad \Sigma : \Gamma, S\ \vec{t} \vdash C}{\Sigma : \Gamma, p\ \vec{t} \vdash C}\ \mathcal{IL}$$

But then the following

$$\frac{\vec{x} : B\ S\ \vec{x} \vdash S\ \vec{x} \quad \Sigma\theta : \Gamma[\![\theta]\!], (S\ \vec{t})[\![\theta]\!] \vdash C[\![\theta]\!]}{\Sigma\theta : \Gamma[\![\theta]\!], (p\ \vec{t})[\![\theta]\!] \vdash C[\![\theta]\!]}$$

is also an instance of the $\mathcal{IL}$ rule. Moreover, the same proof as in Π can be used for the left upper sequent and the right upper sequent has the requisite form for using the induction hypothesis. $\square$

The proof of Lemma 19 effectively defines a transformation of a derivation Π based on a substitution θ. We shall use the notation $\Pi[\![\theta]\!]$ to denote the transformed derivation. Note that $\text{ht}(\Pi[\![\theta]\!])$ can be less than $\text{ht}(\Pi)$. This may happen because the transformed version of a $\trianglerighteq\mathcal{L}$ rule can have fewer upper sequents.

Corollary 20. *The following rules are admissible.*

$$\frac{\Sigma, h : \Gamma \vdash B[h\ \vec{a}/x]}{\Sigma : \Gamma \vdash \forall x.B}\ \forall\mathcal{R}^* \qquad\qquad \frac{\Sigma, h : \Gamma, B[h\ \vec{a}/x] \vdash C}{\Sigma : \Gamma, \exists x.B \vdash C}\ \exists\mathcal{L}^*$$

where $h \notin \Sigma$ and $\vec{a}$ is any listing of distinct nominal constants which contains $\text{supp}(B)$.

Proof. Let Π be a derivation for $\Gamma \vdash B[h\ \vec{a}/x]$, let h' be a variable that does not appear in Π, and let $\{\vec{c}\} = \text{supp}(B)$. By Lemma 19, $\Pi[\![\lambda\vec{a}.h'\ \vec{c}/h]\!]$ is a valid derivation. Since $\vec{a}$ contains $\vec{c}$, no nominal constants appear in the substitution $\{\lambda\vec{a}.h'\ \vec{c}/h\}$. It can now be seen that the last sequent in $\Pi[\![\lambda\vec{a}.h'\ \vec{c}/h]\!]$ has the form $\Sigma, h' : \Gamma' \vdash B'$ where $B' \approx B[h'\ \vec{c}/h]$ and Γ' results from replacing some of the formulas in Γ by ones that they are equivalent to under $\approx$. But then, by Lemma 18, there must be a derivation for $\Sigma, h' : \Gamma \vdash B[h'\ \vec{c}/h]$. Using a $\forall\mathcal{R}$ rule below this we get a derivation for $\Sigma : \Gamma \vdash \forall x.B$, verifying the admissibility of $\forall\mathcal{R}^*$. The argument for $\exists\mathcal{L}^*$ is analogous. $\square$

We now turn to the main result of this section, the redundancy from a provability perspective of the *cut* rule in $\mathcal{G}$. The usual approach to proving such a property is to define a set of transformations called cut reductions on derivations that leave the end sequent unchanged but that have the effect of pushing occurrences of *cut* up the proof tree to the leaves where they can be immediately eliminated. The difficult part of such a proof is showing that these cut reductions always terminate. In simpler sequent calculi such as the one for first-order logic, this argument can be based on an uncomplicated measure such as the size of the cut formula. However, the presence of definitions in a logic like $\mathcal{G}$ renders this measure inadequate. For example, the following is a natural way to define a cut reduction between a *def*$\mathcal{L}$ and a *def*$\mathcal{R}$ rule that work on the cut formula:

$$\dfrac{\dfrac{\begin{array}{c}\Pi' \\ \Sigma : \Gamma \vdash B\ p\ \vec{t}\end{array}}{\Sigma : \Gamma \vdash p\ \vec{t}}\ \mathit{def}\mathcal{R} \qquad \dfrac{\begin{array}{c}\Pi'' \\ \Sigma : B\ p\ \vec{t}, \Delta \vdash C\end{array}}{\Sigma : p\ \vec{t}, \Delta \vdash C}\ \mathit{def}\mathcal{L}}{\Sigma : \Gamma, \Delta \vdash C}\ \mathit{cut} \quad \Rightarrow \quad \dfrac{\begin{array}{c}\Pi' \\ \Sigma : \Gamma \vdash B\ p\ \vec{t}\end{array} \quad \begin{array}{c}\Pi'' \\ \Sigma : B\ p\ \vec{t}, \Delta \vdash C\end{array}}{\Sigma : \Gamma, \Delta \vdash C}\ \mathit{cut}$$

Notice that $B\ p\ \vec{t}$, the cut formula in the new cut introduced by this transformation, could be more complex than $p\ \vec{t}$, the old cut formula. To overcome this difficulty, a more complicated argument based on the idea of reducibility in the style of Tait [29] is often used. Tiu and Momigliano [27] in fact formulate a notion of parametric reducibility for derivations that is based on the Girard's proof of strong normalizability for System F [30] and that works in the presence of the induction and co-induction rules for definitions. Our proof makes extensive use of this notion and the associated argument structure.

Theorem 21. *The cut rule can be eliminated from $\mathcal{G}$ without affecting the provability relation.*

Proof. The relationship between $\mathcal{G}$ and the logic Linc$^-$ treated by Tiu and Momigliano can be understood as follows: Linc$^-$ does not treat the ∇ quantifier and therefore has no rules for it. Consequently, it does not have nominal constants, it does not use raising over nominal constants in the rules $\forall\mathcal{R}$ and $\exists\mathcal{L}$, it has no need to consider permutations in the *id* (or initial) rule and has equality rules in place of nominal abstraction rules. The rules in $\mathcal{G}$ other than the ones for ∇, including the ones for definitions, induction, and co-induction, are essentially identical to the ones in Linc$^-$ except for the additional attention to nominal constants.

Tiu and Momigliano's proof can be extended to $\mathcal{G}$ in a fairly direct way since the addition of nominal constants and their treatment in the rules is quite modular and does not create any new complexities for the reduction rules. The main issues in realizing this extension is building in the idea of identity under permutations of nominal constants and lifting the Linc$^-$ notion of substitution on terms, sequents, and derivations to a form that avoids capture of nominal constants. The machinery for doing this has already been developed in Lemmas 18 and 19. In the rest of this proof we assume a familiarity with the argument for cut-elimination for Linc$^-$ and discuss only the changes to the cut reductions of Linc$^-$ to accommodate the differences.

The id rule in $\mathcal{G}$ identifies formulas which are equivalent under $\approx$ which is more permissive than equality under λ-convertability that is used in the Linc$^-$ initial rule. Correspondingly, we have to be a bit more careful about the cut reductions associated with the id (initial) rule. For example, consider the following reduction:

$$\dfrac{\dfrac{B \approx B'}{\Sigma : \Gamma, B \vdash B'}\, id \qquad \begin{array}{c}\Pi' \\ \Sigma : B', \Delta \vdash C\end{array}}{\Sigma : B, \Gamma, \Delta \vdash C}\, cut \qquad \Rightarrow \qquad \begin{array}{c}\Pi' \\ \Sigma : B', \Delta \vdash C\end{array}$$

This reduction has not preserved the end sequent. However, we know $B \approx B'$ and so we can now use Lemma 18 to replace Π' with a derivation of $\Sigma : B, \Delta \vdash C$. Then we can use Lemma 17 to produce a derivation of $\Sigma : B, \Gamma, \Delta \vdash C$ as desired. The changes to the cut reduction when id applies to the right upper sequent of the cut rule are similar.

The $\forall\mathcal{R}$ and $\exists\mathcal{L}$ rules of $\mathcal{G}$ extend the corresponding rules of Linc$^-$ by raising over nominal constants in the support of the quantified formula. The $\forall\mathcal{L}$ and $\exists\mathcal{R}$ rules of $\mathcal{G}$ also extend the corresponding rules in Linc$^-$ by allowing instantiations which contain nominal constants. Despite these changes, the cut reductions involving these quantifier rules remain unchanged for $\mathcal{G}$ except for the treatment of essential cuts that involve an interaction between $\forall\mathcal{R}$ and $\forall\mathcal{L}$ and, similarly, between $\exists\mathcal{R}$ and $\exists\mathcal{L}$. The first of these is treated as follows:

$$\dfrac{\dfrac{\begin{array}{c}\Pi' \\ \Sigma, h : \Gamma \vdash B[h\ \vec{c}/x]\end{array}}{\Sigma : \Gamma \vdash \forall x.B}\, \forall\mathcal{R} \qquad \dfrac{\begin{array}{c}\Pi'' \\ \Sigma : \Delta, B[t/x] \vdash C\end{array}}{\Sigma : \Delta, \forall x.B \vdash C}\, \forall\mathcal{L}}{\Sigma : \Gamma, \Delta \vdash C}\, cut \ \Rightarrow\ \dfrac{\begin{array}{c}\Pi'[\![\lambda\vec{c}.t/h]\!] \\ \Sigma : \Gamma \vdash B[t/x]\end{array} \quad \begin{array}{c}\Pi'' \\ \Sigma : \Delta, B[t/x] \vdash C\end{array}}{\Sigma : \Gamma, \Delta \vdash C}\, cut$$

The existence of the derivation $\Pi'[\![\lambda\vec{c}.t/h]\!]$ (with height at most that of Π') is guaranteed by Lemma 19. The end sequent of this derivation is $\Sigma : \Gamma[\![\lambda\vec{c}.t/h]\!] \vdash B[h\ \vec{c}/x][\![\lambda\vec{c}.t/h]\!]$. However, $\Gamma[\![\lambda\vec{c}.t/h]\!] \approx \Gamma$ because h is new to Γ and $B[h\ \vec{c}/x][\![\lambda\vec{c}.t/h]\!] \approx B[t/x]$ because $\{\vec{c}\} = \mathrm{supp}(B)$ and so $\lambda\vec{c}.t$ has no nominal constants in common with $\mathrm{supp}(B)$. Thus, by Lemma 18 and by an abuse of notation, we may consider $\Pi'[\![\lambda\vec{c}./h]\!]$ to also be a derivation of $\Sigma : \Gamma \vdash B[t/x]$. The reduction for a cut involving an interaction between an $\exists\mathcal{R}$ and an $\exists\mathcal{L}$ rule is analogous.

The logic $\mathcal{G}$ extends the equality rules in Linc$^-$ to treat the more general case of nominal abstraction. Our notion of nominal capture-avoiding substitution correspondingly generalizes the Linc$^-$ notion of substitution, and we have shown in Lemma 19 that this preserves provability. Thus the reductions for nominal abstraction are the same as for equality, except that we use nominal capture-avoiding substitution in place of regular substitution. For example, the essential cut involving an interaction between an $\trianglerighteq\mathcal{R}$ and an $\trianglerighteq\mathcal{L}$ rule is treated as follows:

$$\dfrac{\dfrac{}{\Sigma : \Gamma \vdash s \trianglerighteq t}\, \trianglerighteq\mathcal{R} \qquad \dfrac{\left\{\begin{array}{c}\Pi_\theta \\ \Sigma\theta : \Delta[\![\theta]\!] \vdash C[\![\theta]\!]\end{array}\right\}}{\Sigma : \Delta, s \trianglerighteq t \vdash C}\, \trianglerighteq\mathcal{L}}{\Sigma : \Gamma, \Delta \vdash C}\, cut \qquad \Rightarrow \qquad \begin{array}{c}\Pi_\epsilon \\ \Sigma : \Delta \vdash C\end{array}$$

Here we know $s \trianglerighteq t$ holds and thus ϵ, the identity substitution, is a solution to this nominal abstraction. Therefore we have the derivation Π_ϵ as needed. We can then apply Lemma 17 to weaken this derivation to one for $\Sigma : \Gamma, \Delta \vdash C$. For the other cuts involving nominal abstraction, we make use of the fact proved in Lemma 19 that nominal capturing avoiding substitution preserves provability. This allows us to commute other rules with $\trianglerighteq\mathcal{L}$. For example, consider the following occurrence of a cut where the upper right derivation uses an $\trianglerighteq\mathcal{L}$ on a formula different from the cut formula:

$$\dfrac{\dfrac{\Pi'}{\Sigma : \Gamma \vdash B} \qquad \dfrac{\left\{ \dfrac{\Pi_\theta}{\Sigma\theta : B[\![\theta]\!], \Delta[\![\theta]\!] \vdash C[\![\theta]\!]} \right\}}{\Sigma : B, \Delta, s \trianglerighteq t \vdash C} \trianglerighteq\mathcal{L}}{\Sigma : \Gamma, \Delta, s \trianglerighteq t \vdash C} \, cut$$

Cut reduction produces from this the following derivation:

$$\dfrac{\left\{ \dfrac{\dfrac{\Pi'[\![\theta]\!]}{\Sigma\theta : \Gamma[\![\theta]\!] \vdash B[\![\theta]\!]} \quad \dfrac{\Pi_\theta}{\Sigma\theta : B[\![\theta]\!], \Delta[\![\theta]\!] \vdash C[\![\theta]\!]}}{\Sigma\theta : \Gamma[\![\theta]\!], \Delta[\![\theta]\!] \vdash C[\![\theta]\!]} \, cut \right\}}{\Sigma : \Gamma, \Delta, s \trianglerighteq t \vdash C} \trianglerighteq\mathcal{L}$$

Finally, $\mathcal{G}$ has new rules for treating the ∇-quantifier. The only reduction rule which deals specifically with either the $\nabla\mathcal{L}$ or $\nabla\mathcal{R}$ rule is the essential cut between both rules which is treated as follows:

$$\dfrac{\dfrac{\dfrac{\Pi'}{\Sigma : \Gamma \vdash B[a/x]}}{\Sigma : \Gamma \vdash \nabla x.B} \nabla\mathcal{R} \quad \dfrac{\dfrac{\Pi''}{\Sigma : B[a/x], \Delta \vdash C}}{\Sigma : \nabla x.B, \Delta \vdash C} \nabla\mathcal{L}}{\Sigma : \Gamma, \Delta \vdash C} \, cut \quad \Rightarrow \quad \dfrac{\dfrac{\Pi'}{\Sigma : \Gamma \vdash B[a/x]} \quad \dfrac{\Pi''}{\Sigma : B[a/x], \Delta \vdash C}}{\Sigma : \Gamma, \Delta \vdash C} \, cut.$$

With these changes, the cut-elimination argument for Linc$^-$ extends to $\mathcal{G}$, *i.e.*, $\mathcal{G}$ admits cut-elimination.

□

The consistency of $\mathcal{G}$ is an easy consequence of Theorem 21.

Corollary 22. *The logic $\mathcal{G}$ is consistent,* i.e.*, not all sequents are provable in it.*

Proof. The sequent $\vdash \bot$ has no cut-free proof and, hence, no proof in $\mathcal{G}$. □

The cut-elimination theorem is important for more reasons than showing the consistency of $\mathcal{G}$. As one example, using the cut-rule in constructing proofs in $\mathcal{G}$ involves the invention of relevant cut formulas that function as *lemmas*. Thus, knowing that this kind of creative step is not essential is helpful in designing automatic theorem provers that are both practical and complete.

6. A Pattern-Based Form for Definitions

When presenting a definition for a predicate, it is often convenient to write this as a collection of clauses whose applicability is also constrained by patterns appearing in the head. For example, in logics that support equality but not nominal abstraction, list membership may be defined by the two pattern based clauses shown below.

$$\mathit{member}\ X\ (X :: L) \triangleq \top \qquad\qquad \mathit{member}\ X\ (Y :: L) \triangleq \mathit{member}\ X\ L$$

These logics also include rules for directly treating definitions presented in this way. In understanding these rules, use may be made of the translation of the extended form of definitions to a version that does not use patterns in the head and in which there is at most one clause for each predicate. For example, the definition of the list membership predicate would be translated to the following form:

$$\mathit{member}\ X\ K \triangleq (\exists L.\ K = (X :: L)) \lor (\exists Y \exists L.\ K = (Y :: L) \land \mathit{member}\ X\ L)$$

The treatment of patterns and multiple clauses can now be understood in terms of the rules for definitions using a single clause and the rules for equality, disjunction, and existential quantification.

In the logic $\mathcal{G}$, the notion of equality has been generalized to that of nominal abstraction. This allows us also to expand the pattern-based form of definitions to use nominal abstraction in determining the selection of clauses. By doing this, we would allow the head of a clausal definition to describe not only the term structure of the arguments, but also to place restrictions on the occurrences of nominal constants in these arguments. For example, suppose we want to describe the contexts in typing judgments by lists of the form $\langle c_1, T_1\rangle :: \langle c_2, T_2\rangle :: \ldots :: \mathit{nil}$ with the further proviso that each c_i is a distinct nominal constant. We will allow this to be done by using the following pattern-based form of definition for the predicate cntx :

$$\mathit{cntx}\ \mathit{nil} \triangleq \top \qquad\qquad (\nabla x.\mathit{cntx}\ (\langle x, T\rangle :: L)) \triangleq \mathit{cntx}\ L$$

Intuitively, the ∇ quantifier in the head of the second clause imposes the requirement that, to match it, the argument of cntx should have the form $\langle x, T\rangle :: L$ where x is a nominal constant that does not occur in either T or L. To understand this interpretation, we could think of the earlier definition of cntx as corresponding to the following one that does not use patterns or multiple clauses:

$$\mathit{cntx}\ K \triangleq (K = \mathit{nil}) \lor (\exists T \exists L.\ (\lambda x.\langle x, T\rangle :: L) \unrhd K \land \mathit{cntx}\ L)$$

Our objective in the rest of this section is to develop machinery for allowing the extended form of definitions to be used directly. We do this by presenting its syntax formally, by describing rules that allow us to employ such definitions and, finally, by justifying the new rules by means of a translation of the kind indicated above.

$$\frac{\Sigma : \Gamma \vdash (B\ p\ \vec{x})[\theta]}{\Sigma : \Gamma \vdash p\ \vec{s}}\ \mathit{def}\mathcal{R}^p$$

for any clause $\forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x}$ in $\mathcal{D}$ and any θ
such that $\mathit{range}(\theta) \cap \Sigma = \emptyset$ and $(\lambda \vec{z}.p\ \vec{t})[\theta] \trianglerighteq p\ \vec{s}$ holds

$$\frac{\left\{ \Sigma\theta : \Gamma[\![\theta]\!], (B\ p\ \vec{x})[\![\theta]\!] \vdash C[\![\theta]\!] \;\middle|\; \begin{array}{l} \forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x} \in \mathcal{D} \text{ and} \\ \theta \text{ is a solution to } ((\lambda \vec{z}.p\ \vec{t}) \trianglerighteq p\ \vec{s}) \end{array} \right\}}{\Sigma : \Gamma, p\ \vec{s} \vdash C}\ \mathit{def}\mathcal{L}^p$$

Figure 6: Introduction rules for a pattern-based definition $\mathcal{D}$

Definition 23. *A* pattern-based definition *is a finite collection of clauses of the form*

$$\forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x}$$

where $\vec{t}$ is a sequence of terms that do not have occurrences of nominal constants in them, p is a constant such that $p\ \vec{t}$ is of type o and B is a term devoid of occurrences of p, $\vec{x}$ and nominal constants and such that $B\ p\ \vec{t}$ is of type o. Further, we expect such a collection of clauses to satisfy a stratification condition: there must exist an assignment of levels to predicate symbols such that for any clause $\forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x}$ occurring in the set, assuming p has arity n, it is the case that $\mathrm{lvl}(B\ (\lambda \vec{x}.\top)\ \vec{x}) < \mathrm{lvl}(p)$. *Notice that we allow the collection to contain more than one clause for any given predicate symbol.*

The logical rules for treating pattern-based definitions are presented in Figure 6. These rules encode the idea of matching an instance of a predicate with the head of a particular clause and then replacing the predicate with the corresponding clause body. The kind of matching involved is made precise through the construction of a nominal abstraction after replacing the ∇ quantifiers in the head of the clause by abstractions. The right rule embodies the fact that it is enough if an instance of any one clause can be used in this way to yield a successful proof. In this rule, the substitution θ that results from the matching must be applied in a nominal capture avoiding way to the body. However, since B does not contain nominal constants, the ordinary application of the substitution also suffices. To accord with the treatment in the right rule, the left rule must consider all possible ways in which an instance of an atomic assumption $p\ \vec{s}$ can be matched by a clause and must show that a proof can be constructed in each such case.

The soundness of these rules is the content of the following theorem whose proof also makes explicit the intended interpretation of the pattern-based form of definitions.

Theorem 24. *The pattern-based form of definitions and the associated proof rules do not add any new power to the logic. In particular, the* def$\mathcal{L}^p$ *and* def$\mathcal{R}^p$ *rules are admissible under the intended interpretation via translation of the pattern-based form of definitions.*

Proof. Let p be a predicate whose clauses in the definition being considered are given by the following set of clauses.

$$\{\forall \vec{x}_i.\ (\nabla \vec{z}_i.p\ \vec{t}_i) \triangleq B_i\ p\ \vec{x}_i\}_{i\in 1..n}$$

Let p' be a new constant symbol with the same argument types as p. Then the intended interpretation of the definition of p in a setting that does not allow the use of patterns in the head and that limits the number of clauses defining a predicate to one is given by the clause

$$\forall \vec{y}.p\ \vec{y} \triangleq \bigvee_{i\in 1..n} \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x}_i$$

in which the variables $\vec{y}$ are chosen such that they do not appear in the terms $\vec{t}_i$ for $1 \leq i \leq n$. Note also that we are using the term constructor p' here so as to be able to match the entire head of a clause at once, thus ensuring that the ∇-bound variables in the head are assigned a consistent value for all arguments of the predicate.

Based on this translation, we can replace an instance of $def\mathcal{R}^p$,

$$\frac{\Gamma \vdash (B_i\ p\ \vec{x}_i)[\theta]}{\Gamma \vdash p\ \vec{s}}\ def\mathcal{R}^p$$

with the following sequence of rules, where a double inference line indicates that a rule is used multiple times.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \vdash (\lambda \vec{z}_i.p'\ \vec{t}_i)[\theta] \trianglerighteq p'\ \vec{s}}\ \trianglerighteq\mathcal{R} \qquad \Gamma \vdash (B_i\ p\ \vec{x}_i)[\theta]}{\Gamma \vdash ((\lambda \vec{z}_i.p'\ \vec{t}_i)[\theta] \trianglerighteq p'\ \vec{s}) \wedge (B_i\ p\ \vec{x}_i)[\theta]}\ \wedge\mathcal{R}}{\Gamma \vdash \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}) \wedge B_i\ p\ \vec{x}_i}\ \exists\mathcal{R}}{\Gamma \vdash \bigvee_{i\in 1..n} \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}) \wedge B_i\ p\ \vec{x}_i}\ \vee\mathcal{R}}{\Gamma \vdash p\ \vec{s}}\ def\mathcal{R}$$

Note that we have made use of the fact that θ instantiates only the variables x_i and thus has no effect on $\vec{s}$. Further, the side condition associated with the $def\mathcal{R}^p$ rule ensures that the $\trianglerighteq\mathcal{R}$ rule that appears as a left leaf in this derivation is well-formed.

Similarly, we can replace an instance of $def\mathcal{L}^p$,

$$\frac{\left\{\Sigma\theta : \Gamma[\![\theta]\!], (B_i\ p\ \vec{x}_i)[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \text{ is a solution to } ((\lambda \vec{z}.p\ \vec{t}_i) \trianglerighteq p\ \vec{s})\right\}_{i\in 1..n}}{\Sigma : \Gamma, p\ \vec{s} \vdash C}\ def\mathcal{L}^p$$

with the following sequence of rules

$$\cfrac{\cfrac{\left\{\cfrac{\cfrac{\cfrac{\{\ \Gamma[\![\theta]\!], (B_i\ p\ \vec{x}_i)[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \text{ is a solution to } ((\lambda \vec{z}.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s})\ \}}{\Gamma, (\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}, B_i\ p\ \vec{x}_i \vdash C}\ \trianglerighteq\mathcal{L}}{\Gamma, ((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}) \wedge B_i\ p\ \vec{x}_i \vdash C}\ \wedge\mathcal{L}^*}{\Gamma, \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}) \wedge B_i\ p\ \vec{x}_i \vdash C}\ \exists\mathcal{L}\right\}_{i\in 1..n}}{\Gamma, \bigvee_{i\in 1..n} \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}) \wedge B_i\ p\ \vec{x}_i \vdash C}\ \vee\mathcal{L}}{\Gamma, p\ \vec{s} \vdash C}\ def\mathcal{L}$$

$$\frac{\left\{\vec{x}_i : B_i\ S\ \vec{x}_i \vdash \nabla\vec{z}_i.S\ \vec{t}_i\right\}_{i\in 1..n} \qquad \Sigma : \Gamma, S\ \vec{s} \vdash C}{\Sigma : \Gamma, p\ \vec{s} \vdash C}\ \mathcal{IL}^p$$

assuming p is defined by the set of clauses $\{\forall\vec{x}_i.(\nabla\vec{z}_i.p\ \vec{t}_i) \stackrel{\mu}{=} B_i\ p\ \vec{x}_i\}_{i\in 1..n}$

Figure 7: Induction rule for pattern-based definitions

Here $\wedge\mathcal{L}^*$ is an application of $c\mathcal{L}$ followed by $\wedge\mathcal{L}_1$ and $\wedge\mathcal{L}_2$ on the contracted formula. It is easy to see that the solutions to $(\lambda\vec{z}.p\ \vec{t}_i) \trianglerighteq p\ \vec{s}$ and $(\lambda\vec{z}.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{s}$ are identical and hence the leaf sequents in this partial derivation are exactly the same as the upper sequents of the instance of the $\mathit{def}\mathcal{L}^p$ rule being considered. $\square$

A weak form of a converse to the above theorem also holds. Suppose that the predicate p is given by the following clauses

$$\{\forall\vec{x}_i.\ (\nabla\vec{z}_i.p\ \vec{t}_i) \triangleq B_i\ p\ \vec{x}_i\}_{i\in 1..n}$$

in a setting that uses pattern-based definitions and that has the $\mathit{def}\mathcal{L}^p$ and $\mathit{def}\mathcal{R}^p$ but not the $\mathit{def}\mathcal{L}$ and $\mathit{def}\mathcal{R}$ rules. In such a logic, it is easy to see that the following is provable:

$$\forall\vec{y}.\left[(p\ \vec{y} \supset \bigvee_{i\in 1..n} \exists\vec{x}_i.((\lambda\vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x}_i) \wedge \right.$$
$$\left. (\bigvee_{i\in 1..n} \exists\vec{x}_i.((\lambda\vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x}_i \supset p\ \vec{y})\right]$$

Thus, in the presence of *cut*, the $\mathit{def}\mathcal{L}$ and $\mathit{def}\mathcal{R}$ rules can be treated as derived rules relative to the translation interpretation of pattern-based definitions.

We would like also to allow patterns to be used in the heads of clauses when writing definitions that are intended to pick out the least and greatest fixed points, respectively. Towards this end we admit in a definition also clauses of the form $\forall\vec{x}.(\nabla\vec{z}.p\ \vec{t}) \stackrel{\mu}{=} B\ p\ \vec{x}$ and $\forall\vec{x}.(\nabla\vec{z}.p\ \vec{t}) \stackrel{\nu}{=} B\ p\ \vec{x}$ with the earlier provisos on the form of B and $\vec{t}$ and the types of B and p and with the additional requirement that all the clauses for any given predicate are unannotated or annotated uniformly with either μ or ν. Further, a definition must satisfy stratification conditions as before. In reasoning about the least or greatest fixed point forms of definitions, we may use the translation into the earlier, non-pattern form together with the rules $\mathcal{IL}$ and $\mathcal{CIR}$. It is possible to formulate an induction rule that works directly from pattern-based definitions using the idea that to show S to be an induction invariant for the predicate p, one must show that every clause of p preserves S. A rule that is based on this intuition is presented in Figure 7. The soundness of this rule is shown in the following theorem.

Theorem 25. *The $\mathcal{IL}^p$ rule is admissible under the intended translation of pattern-based definitions.*

Proof. Let the clauses for p in the pattern-based definition be given by the set

$$\{\forall \vec{x}_i.(\nabla \vec{z}_i.p\ \vec{t}_i) \stackrel{\mu}{\triangleq} B_i\ p\ \vec{x}_i\}_{i\in 1..n}$$

in which case the translated form of the definition for p would be

$$\forall \vec{y}.p\ \vec{y} \stackrel{\mu}{\triangleq} \bigvee_{i\in 1..n} \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x}_i.$$

In this context, the rightmost upper sequents of the $\mathcal{IL}^p$ and the $\mathcal{IL}$ rules that are needed to derive a sequent of the form $\Sigma : \Gamma, p\ \vec{s} \vdash C$ are identical. Thus, to show that $\mathcal{IL}^p$ rule is admissible, it suffices to show that the left upper sequent in the $\mathcal{IL}$ rule can be derived in the original calculus from all but the rightmost upper sequent in an $\mathcal{IL}^p$ rule. Towards this end, we observe that we can construct the following derivation:

$$\dfrac{\left\{ \dfrac{\dfrac{\dfrac{\{\ (\vec{y},\vec{x}_i)\theta : (B_i\ p\ \vec{x}_i)[\![\theta]\!] \vdash (S\ \vec{y})[\![\theta]\!] \mid \theta \text{ is a solution to } ((\lambda \vec{z}.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y})\ \}}{\vec{y},\vec{x}_i : (\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}, B_i\ S\ \vec{x}_i \vdash S\ \vec{y}}\trianglerighteq\mathcal{L}}{\vec{y},\vec{x}_i : ((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x}_i \vdash S\ \vec{y}}\wedge\mathcal{L}^*}{\vec{y} : \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ S\ \vec{x}_i \vdash S\ \vec{y}}\exists\mathcal{L} \right\}_{i\in 1..n}}{\vec{y} : \bigvee_{i\in 1..n} \exists \vec{x}_i.((\lambda \vec{z}_i.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ S\ \vec{x}_i \vdash S\ \vec{y}}\vee\mathcal{L}$$

Since the variables $\vec{y}$ are distinct and do not occur in $\vec{t}_i$, the solutions to $(\lambda \vec{z}.p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}$ have a simple form. In particular, let $\vec{t}_i'$ be the result of replacing in $\vec{t}_i$ the variables $\vec{z}$ with distinct nominal constants. Then $\vec{y} = \vec{t}_i'$ will be a most general solution to the nominal abstraction. Thus the upper sequents of this derivation will be

$$\vec{x}_i : B_i\ p\ \vec{x}_i \vdash S\ \vec{t}_i'$$

which are derivable if and only if the sequents

$$\vec{x}_i : B_i\ p\ \vec{x}_i \vdash \nabla \vec{z}_i.S\ \vec{t}_i$$

are derivable. □

We do not introduce a co-induction rule for pattern-based definitions largely because we have encountered few interesting co-inductive definitions that require patterns and multiple clauses.

7. Examples

We now provide some examples to illuminate the properties of nominal abstraction and its usefulness in both specification and reasoning tasks; while $\mathcal{G}$ has many more features, their characteristics and applications have been exposed in other work (*e.g.*, see [7, 8, 31, 32]). In the examples that are shown, use will be made of the pattern-based form of definitions described in Section 6. We will also adopt the convention that tokens given by capital letters denote variables that are implicitly universally quantified over the entire clause.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{B[c/z] \vdash \mathit{fresh}\ c\ \vec{x}}\ \mathit{def}\mathcal{R}^p \quad \dfrac{}{B[c/z] \vdash B[c/z]}\ \mathit{id}}{B[c/z] \vdash \mathit{fresh}\ c\ \vec{x} \wedge B[c/z]}\ \wedge\mathcal{R}}{B[c/z] \vdash \exists z.(\mathit{fresh}\ z\ \vec{x} \wedge B)}\ \exists\mathcal{R}}{\nabla z.B \vdash \exists z.(\mathit{fresh}\ z\ \vec{x} \wedge B)}\ \nabla\mathcal{L}
$$

Figure 8: The proof of an entailment involving ∇ and the *fresh* predicate

7.1. Properties of ∇ and freshness

We can use nominal abstraction to gain a better insight into the behavior of the ∇ quantifier. Towards this end, let the *fresh* predicate be defined by the following clause.

$$(\nabla x.\mathit{fresh}\ x\ E) \triangleq \top$$

We have elided the type of *fresh* here; it will have to be defined at each type that it is needed in the examples we consider below. Alternatively, we can "inline" the definition by using nominal abstraction directly, *i.e.*, by replacing occurrences of of *fresh* t_1 t_2 with $\exists E.(\lambda x.\langle x, E\rangle \trianglerighteq \langle t_1, t_2\rangle)$ for a suitably typed pairing construct $\langle\cdot,\cdot\rangle$.

Now let B be a formula whose free variables are among $z, x_1, \ldots, x_n$, and let $\vec{x} = x_1 :: \ldots :: x_n :: \mathit{nil}$ where :: and nil are constructors in the logic.[3] Then the following formulas are provable from one another in $\mathcal{G}$.

$$\nabla z.B \qquad \exists z.(\mathit{fresh}\ z\ \vec{x} \wedge B) \qquad \forall z.(\mathit{fresh}\ z\ \vec{x} \supset B)$$

Note that the type of z allows it to be an arbitrary term in the last two formulas, but its occurrence as the first argument of *fresh* will restrict it to being a nominal constant (even when $\vec{x} = \mathit{nil}$). Figure 8 shows a derivation for one of these entailments. Similar proofs can be constructed for the other entailments.

In the original presentation of the ∇ quantifier [33], it was shown that one can move a ∇ quantifier inwards over universal and existential quantifiers by using raising to encode an explicit dependency. To illustrate this, let B be a formula with two variables abstracted out, and let $C \equiv D$ be shorthand for $(C \supset D) \wedge (D \supset C)$. The following formulas are provable in the logic.

$$\nabla z.\forall x.(B\ z\ x) \equiv \forall h.\nabla z.(B\ z\ (h\ z)) \qquad \nabla z.\exists x.(B\ z\ x) \equiv \exists h.\nabla z.(B\ z\ (h\ z))$$

In order to move a ∇ quantifier outwards over universal and existential quantifiers, one would need a way to make non-dependency (*i.e.*, freshness) explicit. This is now possible using nominal abstraction as shown by the following equivalences.

$$\forall x.\nabla z.(B\ z\ x) \equiv \nabla z.\forall x.(\mathit{fresh}\ z\ x \supset B\ z\ x)$$
$$\exists x.\nabla z.(B\ z\ x) \equiv \nabla z.\exists x.(\mathit{fresh}\ z\ x \wedge B\ z\ x)$$

[3] We are, once again, finessing typing issues here in that the x_i variables may not all be of the same type. However, this problem can be solved by surrounding each of them with a constructor that yields a term with a uniform type.

$$\frac{x : a \in \Gamma}{\Gamma \vdash x : a} \qquad \frac{\Gamma \vdash t_1 : a \to b \quad \Gamma \vdash t_2 : a}{\Gamma \vdash (t_1\ t_2) : b} \qquad \frac{\Gamma, x : a \vdash t : b}{\Gamma \vdash (\lambda x{:}a.\ t) : a \to b}\ x \notin dom(\Gamma)$$

Figure 9: Type assignment for λ-terms

$$\begin{aligned}
&\mathit{member}\ P\ (P :: L) \stackrel{\mu}{=} \top \\
&\mathit{member}\ P\ (Q :: L) \stackrel{\mu}{=} \mathit{member}\ P\ L \\[1ex]
&\mathit{of}\ L\ X\ A \stackrel{\mu}{=} \mathit{member}\ \langle X, A\rangle\ L \\
&\mathit{of}\ L\ (\mathit{app}\ M\ N)\ B \stackrel{\mu}{=} \exists A. \mathit{of}\ L\ M\ (\mathit{arr}\ A\ B) \wedge \mathit{of}\ L\ N\ A \\
&\mathit{of}\ L\ (\mathit{abs}\ A\ R)\ (\mathit{arr}\ A\ B) \stackrel{\mu}{=} \nabla x. \mathit{of}\ (\langle x, A\rangle :: L)\ (R\ x)\ B
\end{aligned}$$

Figure 10: Encoding of type assignment for λ-terms

Finally, we note that the two sets of equivalences for moving the ∇ quantifier interact nicely. Specifically, starting with a formula like $\nabla z.\forall x.(B\ z\ x)$ we can push the ∇ quantifier inwards and then outwards to obtain $\nabla z.\forall h.(\mathit{fresh}\ z\ (h\ z) \supset B\ z\ (h\ z))$. Here $\mathit{fresh}\ z\ (h\ z)$ will only be satisfied if h does not use its first argument, as expected.

7.2. Type uniqueness for the simply-typed λ-calculus

As a more complete example, we consider the problem of showing the uniqueness of type assignment for the simply-typed λ-calculus. The typing rules used in the assignment are shown in Figure 9. We introduce the type tp to denote the collection of simple types and the constants $i : tp$ to represent the (single) atomic type and $arr : tp \to tp \to tp$ to represent the function type constructor. Representations of λ-terms will have the type tm and will be constructed using the constants $app : tm \to tm \to tm$ and $abs : ty \to (tm \to tm) \to tm$ that are chosen to represent application and abstraction, respectively. Finally we introduce a type a for typing assumptions together with the constant $\langle \cdot, \cdot \rangle : tm \to tp \to a$, and the type $alist$ for lists of typing assumptions constructed from the constants $nil : alist$ and the infix constant $::$ of type $a \to alist \to alist$. We define the predicate $member$ of type $a \to alist \to o$ and encode the simple typing of λ-terms in the definition of a predicate of with type $alist \to tm \to tp \to o$ as shown in Figure 10. Note here that the side-condition on the rule for typing abstractions is subsumed by the treatment of ∇ in the logic.

Given this encoding of simple typing, the task of showing the uniqueness of type assignment reduces to proving the following formula:

$$\forall t, a, b.(\mathit{of}\ nil\ t\ a \wedge \mathit{of}\ nil\ t\ b) \supset a = b.$$

While the theorem that is ultimately of interest is stated with a nil context, it is not difficult to see that in an inductive proof we will have to consider the more general case where this context is not empty. However, the typing context is not entirely arbitrary. It must have the

$$\mathit{cntx}\,\mathit{nil} \stackrel{\mu}{=} \top$$

$$(\nabla x.\mathit{cntx}\ (\langle x, A\rangle :: L)) \stackrel{\mu}{=} \mathit{cntx}\ \ L$$

Figure 11: *cntx* in $\mathcal{G}$

$$\begin{aligned}
&\mathit{cntx}\,\mathit{nil} \stackrel{\mu}{=} \top \\
&\mathit{cntx}\ (\langle X, A\rangle :: L) \stackrel{\mu}{=} (\forall M, N.X = \mathit{app}\ M\ N \supset \bot) \wedge \\
&\qquad\qquad\qquad\qquad (\forall R, B.X = \mathit{abs}\ B\ R \supset \bot) \wedge \\
&\qquad\qquad\qquad\qquad (\forall B.\mathit{member}\ \langle X, B\rangle\ L \supset \bot) \wedge \\
&\qquad\qquad\qquad\qquad \mathit{cntx}\ L
\end{aligned}$$

Figure 12: *cntx* in $\mathcal{LG}^\omega$

form $\langle x_1, a_1\rangle :: \ldots :: \langle x_n, a_n\rangle :: \mathit{nil}$ where each x_i is unique and atomic (a nominal constant). If we assume a predicate *cntx* which restricts the structure of typing contexts in this way, then we can state our generalized result as follows.

$$\forall \ell, t, a, b.(\mathit{cntx}\ \ell \wedge \mathit{of}\ \ell\ t\ a \wedge \mathit{of}\ \ell\ t\ b) \supset a = b$$

This is now provable by a straightforward induction on either of the typing assumptions.

We turn now to the question of defining a suitable *cntx* predicate. Using nominal abstraction, we can define *cntx* directly and succinctly as shown in Figure 11. An instance of the second clause must replace x with a nominal constant and A and L by terms which do not contain that nominal constant. The atomicity and distinct properties of typing assumptions follow naturally from this. To better appreciate the elegance of this approach, consider how one would enforce atomicity and distinctness without nominal abstraction. In a logic such as $\mathcal{LG}^\omega$, the restrictions imposed by *cntx* would have to be encoded via negative information as shown in Figure 12. This description of typing contexts is cumbersome and non-modular. For example, if we were to add a new constructor for λ-terms and a typing rule associated with this constructor then, even though the structure of typing contexts has not changed, we would need to change *cntx* to rule out this constructor from occurring in typing contexts. We will use the definition of *cntx* with nominal abstraction going forward.

When proving the generalized type uniqueness property, the typing context becomes important at two points: when considering the base case where a typing assumption is looked up in the context, and when extending the context with a new typing assumption. When a typing assumption is found in the context, we must show that it is unique. The definition of *cntx* describes the structure of typing assumptions that occur at the head of a context, and the following lemma uses induction to generalize this to arbitrary elements of the context.

$$\forall \ell, m, a, b.(\mathit{cntx}\ \ell \wedge \mathit{member}\ \langle m, a\rangle\ \ell \wedge \mathit{member}\ \langle m, b\rangle\ \ell) \supset a = b$$

This property can be shown by induction on *cntx* followed by case analysis on the *member* hypotheses. The interesting case is when we have $\ell = \langle m, a\rangle :: \ell'$ and $\mathit{member}\ \langle m, b\rangle\ \ell'$. Applying $\mathit{def}\mathcal{R}^p$ to $\mathit{cntx}\ (\langle m, a\rangle :: \ell')$ in this case replaces m with a nominal constant that ℓ' cannot contain. The assumption that $\mathit{member}\ \langle m, b\rangle\ \ell'$ then leads to a contradiction, thus eliminating this case. Moving on to the second point, when adding a typing assumption to the context, we need to show that the resulting context still satisfies the *cntx* predicate. This boils down to showing the following.

$$\forall \ell, a.(\mathit{cntx}\ \ell \supset \nabla x.\mathit{cntx}\ (\langle x, a\rangle :: \ell))$$

This follows directly from applying $\mathit{def}\mathcal{R}^p$ to *cntx*. With these issues taken care of, the rest of the type uniqueness proof is straightforward.

In order for the above reasoning to be meaningful, we must show that our encoding of the simply-typed λ-calculus is adequate. The crux of this is showing that $\Gamma \vdash t : a$ holds in the simply-typed λ-calculus if and only if $\vdash \mathit{of}\ \ulcorner\Gamma\urcorner\ \ulcorner t\urcorner\ \ulcorner a\urcorner$ is provable in $\mathcal{G}$. Here $\ulcorner\cdot\urcorner$ is a bijective mapping between objects of the simply-typed λ-calculus and their representation in $\mathcal{G}$. Since $\mathcal{G}$ admits cut-elimination, it is straightforward to analyze how $\vdash \mathit{of}\ \ulcorner\Gamma\urcorner\ \ulcorner t\urcorner\ \ulcorner a\urcorner$ might be proved in the logic. Then the only subtlety in showing adequacy is that the first clause for *of* allows the type of any object to be looked up in the context while the first typing rule for simply-typed λ-calculus only allows the type of variables to be looked up. This is resolved by noting that typing contexts only contain bindings for variables. Alternatively, using nominal abstraction, it is possible to give a definition of typing which is closer to the original rules (Figure 9) by replacing the first clause of *of* with the following.

$$(\nabla x.\mathit{of}\ (L\ x)\ x\ A) \stackrel{\mu}{=} \nabla x.\mathit{member}\ \langle x, A\rangle\ (L\ x)$$

An additional benefit of this encoding is that in proofs such as for type uniqueness we no longer need to consider spurious cases where the type of a term such as $\mathit{app}\ m\ n$ is looked up in the typing context.

We can now put everything together to establish the type uniqueness result for the simply-typed λ-calculus. Suppose $\Gamma \vdash t : a$ and $\Gamma \vdash t : b$ are judgments in the simply-typed λ-calculus. Then by adequacy we know $\vdash \mathit{of}\ \ulcorner\Gamma\urcorner\ \ulcorner t\urcorner\ \ulcorner a\urcorner$ and $\vdash \mathit{of}\ \ulcorner\Gamma\urcorner\ \ulcorner t\urcorner\ \ulcorner b\urcorner$ are provable in $\mathcal{G}$. Using these assumptions, the *cut* rule, and the type uniqueness result proved earlier in $\mathcal{G}$, we know that $\vdash \ulcorner a\urcorner = \ulcorner b\urcorner$ has a proof in $\mathcal{G}$. Thus it also has a cut-free proof. This proof must end with with $\trianglerighteq\mathcal{R}$ which means that $\ulcorner a\urcorner$ is equal to $\ulcorner b\urcorner$. Finally, since $\ulcorner\cdot\urcorner$ is bijective, a must equal b.

7.3. Polymorphic type generalization

In addition to reasoning, nominal abstraction can also be useful in providing declarative specifications of computations. We consider the context of a type inference algorithm that is also discussed in [34] to illustrate such an application. In this setting, we might need a predicate *spec* that relates a polymorphic type σ, a list of distinct variables $\vec{\alpha}$ (represented by nominal constants) and a monomorphic type τ just in the case that $\sigma = \forall\vec{\alpha}.\tau$. Using nominal abstraction, we can define this predicate as follows.

$$\begin{aligned} &\mathit{spec}\ (\mathit{monoTy}\ T)\ \mathit{nil}\ T \stackrel{\mu}{=} \top \\ (\nabla x.\mathit{spec}\ (\mathit{polyTy}\ P)\ (x :: L)\ (T\ x)) &\stackrel{\mu}{=} \nabla x.\mathit{spec}\ (P\ x)\ L\ (T\ x). \end{aligned}$$

Note that we use ∇ in the head of the second clause to associate the variable x at the head of the list L with its occurrences in the type $(T\ x)$. We then use ∇ in the body of this clause to allow for the recursive use of *spec*.

7.4. Arbitrarily cascading substitutions

Many reducibility arguments, such as Tait's proof of normalization for the simply typed λ-calculus [29], are based on judgments over closed terms. During reasoning, however, one has often to work with open terms. To accommodate this requirement, the closed term judgment is extended to open terms by considering all possible closed instantiations of the open terms. When reasoning with $\mathcal{G}$, open terms are denoted by terms with nominal constants representing free variables. The general form of an open term is thus $M\ c_1\ \cdots\ c_n$, and we want to consider all possible instantiations $M\ V_1\ \cdots\ V_n$ where the V_i are closed terms. This type of arbitrary cascading substitutions is difficult to realize in reasoning systems where variables are given a simple type since M would have an arbitrary number of abstractions but the type of M would *a priori* fix that number of abstractions.

We can define arbitrary cascading substitutions in $\mathcal{G}$ using nominal abstraction. In particular, we can define a predicate which holds on a list of pairs $\langle c_i, V_i\rangle$, a term with the form $M\ c_1\ \cdots\ c_n$ and a term of the form $M\ V_1\ \cdots\ V_n$. The idea is to iterate over the list of pairs and for each pair $\langle c, V\rangle$ use nominal abstraction to abstract c out of the first term and then substitute V before continuing. The following definition of the predicate *subst* is based on this idea.

$$\begin{aligned} &\mathit{subst}\ \mathit{nil}\ T\ T \stackrel{\mu}{=} \top \\ (\nabla x.\mathit{subst}\ (\langle x, V\rangle :: L)\ (T\ x)\ S) &\stackrel{\mu}{=} \mathit{subst}\ L\ (T\ V)\ S \end{aligned}$$

The ideas in this substitution predicate have been used to formalize Tait's logical relations argument for the weak normalization of the simply-typed λ-calculus in a logic similar to $\mathcal{G}$ [19]. Here, an important property of arbitrary cascading substitutions is that they act compositionally. For instance, taking the slightly simpler example of the untyped λ-calculus, we can show that *subst* acts compositionally via the following lemmas.

$$\forall \ell, t, r, s.\ \mathit{subst}\ \ell\ (\mathit{app}\ t\ r)\ s \supset \exists u, v.(s = \mathit{app}\ u\ v \wedge \mathit{subst}\ \ell\ t\ u \wedge \mathit{subst}\ \ell\ r\ v)$$
$$\forall \ell, t, r.\ \mathit{subst}\ \ell\ (\mathit{abs}\ t)\ r \supset \exists s.(r = \mathit{abs}\ s \wedge \nabla z.\mathit{subst}\ \ell\ (t\ z)\ (s\ z))$$

Both of these lemmas have straightforward proofs by induction on *subst*.

8. Related Work

We structure the discussion of related work into three parts: the previously existing framework that $\mathcal{G}$ builds on, alternative proposals for treating binding in syntax and different approaches for relating specifications of formal systems and reasoning about them.

8.1. The precursors for $\mathcal{G}$

The logic $\mathcal{G}$ that we have described in this paper provides a framework for intuitionistic reasoning that is characterized by its use of typed λ-terms for representing objects, of a fixed-point notion of definitions with associated principles of induction and co-induction, of the special ∇-quantifier to express generic judgments and of nominal abstraction for making explicit the properties of objects captured by the ∇-quantifier. All these features except the last derive from previously described logics. The style in which definitions are treated originates from work by Schroeder-Heister [35] and Girard [3]. McDowell and Miller used this idea within a fragment of the Simple Theory of Types and added to this also a treatment of induction over natural numbers [5]. The resulting logic, called $FO\lambda^{\Delta\mathbb{N}}$, provides a means for reasoning about specifications of computations over objects involving abstractions in which universally quantified judgments are used to capture the dynamic aspects of such abstractions. While such an encoding suffices for many purposes, Miller and Tiu discovered its inadequacy in, for example, treating the distinctness of names in arguments relating to the π-calculus and they developed the logic $FO\lambda^{\Delta\nabla}$ with the new ∇-quantifier as a vehicle for overcoming this deficiency [16]. Tiu then showed how to incorporate inductive and co-inductive forms of definitions into this context [8]. However, the properties initially assumed for the ∇-quantifier were too weak to support sophisticated forms of reasoning based on (co-)induction, and this led to the addition of the ∇-strengthening and ∇-exchange principles [18]. The logic that is a composite of all these features still lacks the ability, often needed in inductive arguments, to make explicit in a systematic way properties such as the freshness and distinctness of nominal constants (*i.e.*, the variables bound by the ∇-quantifier). Nominal abstraction, whose study has been the main focus of this paper, provides a natural means for reflecting such properties into definitions and as such represents a culmination of this line of development.

The exchange property assumed for the ∇-quantifier appears to have a natural justification. On the other hand, the strengthening property, while useful in many reasoning contexts, brings with it the implicit requirement that the types at which ∇-quantifiers are used be inhabited by an unbounded number of members. This assumption may complicate the process of showing the adequacy of an encoding, an important part of using a logical framework in formalizing the properties of a computational system. The observation concerning adequacy has led Baelde to develop an alternative approach to enriching the structure provided by $FO\lambda^{\Delta\nabla}$ [36, 37]. Specifically, he has proposed treating the ∇-quantifier as a defined symbol, including in its definition also the ability to lift its predicative effect over types. The exchange property for the quantifier follows from this enrichment, while the properties $(\nabla x.P) \supset P$ and $P \supset (\nabla x.P)$ where x does not occur in P are shown to hold for certain syntactic classes of formulas. The resulting logic has a domain of application that overlaps with that of $\mathcal{G}$ but, in our opinion, may not be as convenient to use in actual reasoning tasks. A detailed consideration of this issue and also the quantification of the real differences in adequacy arguments are left for future investigation.

8.2. Nominal logic

The ∇-quantifier of $\mathcal{G}$ bears several similarities to the И-quantifier contained in nominal logic. As presented in [20], nominal logic is, in essence, a variant of first-order logic whose defining characteristics are that it distinguishes certain domains as those of atoms or names and takes as primitive a freshness predicate—denoted by the infix operator #—between atoms and other objects and a swapping operation involving a pair of names and a term. The logic then formalizes certain properties of the swapping operation (referred to as equivariance properties) and of freshness. One of the freshness axioms leads to the availability of an unbounded supply of names, an aspect that is reminiscent of the consequence of the strengthening rule associated with the ∇-quantifier. Letting ϕ be a formula whose free variables are $a, x_1, \ldots, x_n$ where a is of atom type, another consequence of the swapping and freshness axioms is the following equivalence:

$$\exists a.(a\#x_1 \wedge \ldots \wedge a\#x_n \wedge \phi) \quad \equiv \quad \forall a.(a\#x_1 \wedge \ldots \wedge a\#x_n \supset \phi)$$

The И-quantifier can be defined in this setting by translating И$a.\phi$ into either one of the formulas shown in this equivalence. In our presentation of $\mathcal{G}$, we have taken the ∇-quantifier to be primitive and we have shown that we can define a *fresh* predicate using nominal abstraction. As we have seen in Section 7.1, we then get a set of equivalences between ∇, the traditional quantifiers and *fresh* that is reminiscent of the one discussed here involving the И-quantifier.

At a deeper level, there appears to be some convergence in the treatment of syntax between the nominal logic approach and the one supported by $\mathcal{G}$ using λ-terms. For example, both make use of self-dual quantifiers to manage names and both provide predicates for freshness, equality, and inequality relating to names. Probably the most fruitful way to compare these approaches in detail is via their respective proof theories: see [38, 39] for some proof theory developments for nominal logics. To illustrate such a convergence, we note that nominal logic has inspired a variant to logic programming in the form of the αProlog language [34]. The specifications written in αProlog have a Horn clause like structure with the important difference that the И-quantifier is permitted to appear in the head. Clauses of this kind bear a resemblance to the pattern-based form of definitions discussed in Section 6 in which the ∇-quantifier may appear at the front of clauses. In fact, it is shown in [40] that the former can be directly translated to the latter. The animation of such definitions in $\mathcal{G}$ through the $\mathit{def}\mathcal{R}^p$ rule requires the solution of nominal abstraction problems that is similar in several respects to the equivariant unification [41] needed in an interpreter for αProlog.

These similarities notwithstanding, the intrinsic structures of nominal logic and $\mathcal{G}$ are actually quite different. The former logic is first-order in spirit and does not include a binding construct at the outset. While it is possible to define a (first-order) binding constructor in nominal logic that obeys the principle of α-equivalence, the resulting binder is not capable of directly supporting λ-tree syntax. In particular, β-equivalence is not internalized with these terms: as a consequence, term-level substitution has to be explicitly formalized and its formal properties need to be established on a case-by-case basis. While such a first-order encoding has some drawbacks from the perspective of treating binding structure, it also has the benefit that it can be more easily formalized within the logic of existing theorem provers such as Coq and Isabelle/HOL [42, 43, 44].

8.3. Separation of specification and reasoning logics

An important envisaged use of $\mathcal{G}$ is in realizing the *two-level approach* to reasoning about the operational semantics of programming languages and process calculi. The first step in this approach is to use a *specification logic* to encode such operational semantics as well as assortments of other properties such as typing. The second step involves embedding provability of this first logic into a second logic, called the *reasoning logic*. This two level-logic approach, pioneered by McDowell and Miller [31, 45], offers several benefits, such as the ability to internalize into the reasoning logic properties about derivations in the specification logic and to use these uniformly in reasoning about the specifications of particular systems. For example, cut-elimination for the specification logic can be used to prove substitution lemmas in the reasoning logic. Another benefit is that λ-tree syntax is available for both logics since the specification logic is a simple definition within the reasoning logic. Part of our motivation for $\mathcal{G}$ was for it to play the role of a powerful reasoning logic. In particular, $\mathcal{G}$ has been provided an implementation in the Abella system [22]. Given the richer expressiveness of $\mathcal{G}$, it was been possible to redo the example proofs in [31] in a much more understandable way [22, 46].

Pfenning and Schürmann [47] also describe a two-level approach in which the terms and types of a dependently typed λ-calculus called LF [11] are used as specifications and a logic called $\mathcal{M}_2$ is used for the reasoning logic. Schürmann's PhD thesis [48] further extended that reasoning logic to one called $\mathcal{M}_2^+$. This framework is realized in the Twelf system [15], which also provides a related style of meta-reasoning based on mode, coverage, and termination checking over higher-order judgments in LF. This approach makes use of λ-tree syntax at both the specification and reasoning levels and goes beyond what is available with $\mathcal{G}$ in that it exploits the sophistication of dependent types that also provides for the encoding of proof objects. On the other hand, the kinds of meta-level theorems that can be proved in this setting are structurally weaker than those that can be proved in $\mathcal{G}$. For example, implication and negation are not present in $\mathcal{M}_2^+$ and cannot be encoded in higher-order LF judgments. Concretely, this means that properties such as bisimulation for CCS or the π-calculus are not provable in this approach.

A key component in $\mathcal{M}_2^+$ and in the higher-order LF judgment approach to meta-reasoning is the ability to specify invariants related to the structure of meta-logical contexts. These invariants are called *regular worlds* and their analogue in our system is judgments such as *cntx* which explicitly describe the structure of contexts. While the approach to proving properties in Twelf is powerful and convenient for many applications, it may be preferable to have the ability to define invariants such as *cntx* explicitly rather than relying on regular worlds, since this allows more general judgments over contexts to be described, such as in the example of arbitrary cascading substitutions (Section 7.4) where the *subst* predicate actively manipulates the context of a term.

9. Acknowledgements

We are grateful to Alwen Tiu whose observations with respect to the earlier formulation of our ideas in terms of ∇-quantifiers in the heads of clauses eventually led us to a recasting using the nominal abstraction predicate. Useful comments were also received from David Baelde and the reviewers of an earlier version of this paper and our related LICS'08 paper. This work has been supported by the National Science Foundation grants CCR-0429572 and CCF-0917140 and by INRIA through the "Equipes Associées" Slimmer. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Milner, Communication and Concurrency, Prentice-Hall International, 1989.

[2] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, Centaur: the system, in: Third Annual Symposium on Software Development Environments (SDE3), Boston, 1988, pp. 14–24.

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

[4] P. Schroeder-Heister, Rules of definitional reflection, in: M. Vardi (Ed.), Eighth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, IEEE, 1993, pp. 222–232.

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

[6] R. McDowell, D. Miller, C. Palamidessi, Encoding transition systems in sequent calculus, Theoretical Computer Science 294 (3) (2003) 411–437.

[7] A. Momigliano, A. Tiu, Induction and co-induction in sequent calculus, in: M. Coppo, S. Berardi, F. Damiani (Eds.), Post-proceedings of TYPES 2003, no. 3085 in LNCS, 2003, pp. 293–308.

[8] A. Tiu, A logical framework for reasoning about logical specifications, Ph.D. thesis, Pennsylvania State University (May 2004).

[9] R. Milner, Communicating and Mobile Systems : The π-calculus, Cambridge University Press, New York, NY, USA, 1999.

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

[11] R. Harper, F. Honsell, G. Plotkin, A framework for defining logics, Journal of the ACM 40 (1) (1993) 143–184.

[12] D. Miller, Abstract syntax for variable binders: An overview, in: J. Lloyd, *et al.* (Eds.), Computational Logic - CL 2000, no. 1861 in LNAI, Springer, 2000, pp. 239–253.

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

[14] G. Nadathur, D. Miller, An Overview of λProlog, in: Fifth International Logic Programming Conference, MIT Press, Seattle, 1988, pp. 810–827.

[15] F. Pfenning, C. Schürmann, System description: Twelf — A meta-logical framework for deductive systems, in: H. Ganzinger (Ed.), 16th Conf. on Automated Deduction (CADE), no. 1632 in LNAI, Springer, Trento, 1999, pp. 202–206.

[16] D. Miller, A. Tiu, A proof theory for generic judgments, ACM Trans. on Computational Logic 6 (4) (2005) 749–783.

[17] D. Baelde, A. Gacek, D. Miller, G. Nadathur, A. Tiu, The Bedwyr system for model checking over syntactic expressions, in: F. Pfenning (Ed.), 21th Conf. on Automated Deduction (CADE), no. 4603 in LNAI, Springer, 2007, pp. 391–397.

[18] A. Tiu, A logic for reasoning about generic judgments, in: A. Momigliano, B. Pientka (Eds.), Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06), Vol. 173 of ENTCS, 2006, pp. 3–18.

[19] A. Gacek, D. Miller, G. Nadathur, Reasoning in Abella about structural operational semantics specifications, in: A. Abel, C. Urban (Eds.), International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008), no. 228 in ENTCS, 2008, pp. 85–100.

[20] A. M. Pitts, Nominal logic, A first order theory of names and binding, Information and Computation 186 (2) (2003) 165–193.

[21] A. Gacek, D. Miller, G. Nadathur, Combining generic judgments with recursive definitions, in: F. Pfenning (Ed.), 23th Symp. on Logic in Computer Science, IEEE Computer Society Press, 2008, pp. 33–44.

[22] A. Gacek, The Abella interactive theorem prover (system description), in: A. Armando, P. Baumgartner, G. Dowek (Eds.), Fourth International Joint Conference on Automated Reasoning, Vol. 5195 of LNCS, Springer, 2008, pp. 154–161.

[23] A. Tiu, D. Miller, A proof search specification of the π-calculus, in: 3rd Workshop on the Foundations of Global Ubiquitous Computing, Vol. 138 of ENTCS, 2004, pp. 79–101.

[24] A. Church, A formulation of the simple theory of types, J. of Symbolic Logic 5 (1940) 56–68.

[25] D. Miller, Unification under a mixed prefix, Journal of Symbolic Computation 14 (4) (1992) 321–358.

[26] G. Huet, A unification algorithm for typed λ-calculus, Theoretical Computer Science 1 (1975) 27–57.

[27] A. Tiu, A. Momigliano, Induction and co-induction in sequent calculus, available from `http://arxiv.org/abs/0812.4727` (2009).

[28] A. Tiu, Cut elimination for a logic with generic judgments and induction, Tech. rep., CoRR, extended version of LFMTP'06 paper. Available from `http://arxiv.org/abs/0801.3065` (Jan. 2008).

[29] W. W. Tait, Intensional interpretations of functionals of finite type I, J. of Symbolic Logic 32 (2) (1967) 198–212.

[30] J.-Y. Girard, P. Taylor, Y. Lafont, Proofs and Types, Cambridge University Press, 1989.

[31] R. McDowell, D. Miller, Reasoning with higher-order abstract syntax in a logical framework, ACM Trans. on Computational Logic 3 (1) (2002) 80–136.

[32] A. Tiu, D. Miller, Proof search specifications of bisimulation and modal logics for the π-calculus, ACM Trans. on Computational Logic 11 (2).

[33] D. Miller, A. Tiu, A proof theory for generic judgments: An extended abstract, in: P. Kolaitis (Ed.), 18th Symp. on Logic in Computer Science, IEEE, 2003, pp. 118–127.

[34] J. Cheney, C. Urban, Nominal logic programming, ACM Trans. Program. Lang. Syst. 30 (5) (2008) 1–47. doi:http://doi.acm.org/10.1145/1387673.1387675.

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

[36] D. Baelde, A linear approach to the proof-theory of least and greatest fixed points, Ph.D. thesis, Ecole Polytechnique (Dec. 2008).

[37] D. Baelde, On the expressivity of minimal generic quantification, in: A. Abel, C. Urban (Eds.), International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008), no. 228 in ENTCS, 2008, pp. 3–19.

[38] J. Cheney, A simpler proof theory for nominal logic, in: 8th International Conference on the Foundations of Software Science and Computational Structures (FOSSACS), Vol. 3441 of LNCS, Springer, 2005, pp. 379–394.

[39] M. Gabbay, Fresh logic: proof-theory and semantics for FM and nominal techniques, J. Applied Logic 5 (2) (2007) 356–387.

[40] A. Gacek, Relating nominal and higher-order abstract syntax specifications, in: Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming, ACM, 2010, pp. 177–186.

[41] J. Cheney, Equivariant unification, Journal of Automated Reasoning(published online).

[42] A. M. Pitts, Alpha-structural recursion and induction, J. ACM 53 (3) (2006) 459–506.

[43] B. Aydemir, A. Bohannon, S. Weirich, Nominal reasoning techniques in Coq, in: International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP), Seattle, WA, USA, 2006, pp. 69–77.

[44] C. Urban, Nominal reasoning techniques in Isabelle/HOL, Journal of Automated Reasoning 40 (4) (2008) 327–356.

[45] R. McDowell, Reasoning in a logic with definitions and induction, Ph.D. thesis, University of Pennsylvania (Dec. 1997).

[46] A. Gacek, A framework for specifying, prototyping, and reasoning about computational systems, Ph.D. thesis, University of Minnesota (2009).

[47] C. Schürmann, F. Pfenning, Automated theorem proving in a simple meta-logic for LF, in: C. Kirchner, H. Kirchner (Eds.), 15th Conf. on Automated Deduction (CADE), Vol. 1421 of Lecture Notes in Computer Science, Springer, 1998, pp. 286–300.

[48] C. Schürmann, Automating the meta theory of deductive systems, Ph.D. thesis, Carnegie Mellon University, CMU-CS-00-146 (Oct. 2000).