Combining generic judgments with recursive definitions

Andrew Gacek
Department of CS&E
University of Minnesota

Dale Miller
INRIA Saclay - Île-de-France
& LIX/École polytechnique

Gopalan Nadathur
Department of CS&E
University of Minnesota

Abstract

Many semantical aspects of programming languages are specified through calculi for constructing proofs: consider, for example, the specification of structured operational semantics, labeled transition systems, and typing systems. Recent proof theory research has identified two features that allow direct, logic-based reasoning about such descriptions: the treatment of atomic judgments as fixed points (recursive definitions) and an encoding of binding constructs via generic judgments. However, the logics encompassing these two features have thus far treated them orthogonally. In particular, they have not contained the ability to form definitions of object-logic properties that themselves depend on an intrinsic treatment of binding. We propose a new and simple integration of these features within an intuitionistic logic enhanced with induction over natural numbers and we show that the resulting logic is consistent. The pivotal part of the integration allows recursive definitions to define generic judgments *in general and not just the simpler* atomic judgments *that are traditionally allowed. The usefulness of this logic is illustrated by showing how it can provide elegant treatments of object-logic contexts that appear in proofs involving typing calculi and arbitrarily cascading substitutions in reducibility arguments.*

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

1. Introduction

An important approach to specifying and reasoning about computations is based on *proof theory* and *proof search* principles. We discuss below three kinds of judgments about computational systems that one might want to capture and the proof theoretic techniques that have been used to capture them. We divide this discussion into two parts: the first part deals with judgments over *algebraic terms* and the second with judgments over *terms-with-binders*. We then exploit this overview to describe the new features of the logic we are presenting in this paper.

1.1. Judgments involving algebraic terms

We overview features of proof theory that support recursive definitions about first-order (algebraic) terms and, using CCS as an example, we illustrate the judgments about computations that can be encoded through such definitions.

(1) Logic programming, may behavior Logic programming languages allow for a natural specification and animation of operational semantics and typing judgments: this observation goes back to at least the Centaur project and its animation of Typol specifications using Prolog [3]. 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. However, traditional logic programming is limited to *may* behavior judgments: using it, we cannot prove that a given CCS process P cannot make a transition and, since this negative property is logically equivalent to proving that P is bisimilar to 0 (the null process), such systems cannot capture bisimulation.

(2) Model checking, must behavior Proof theoretic techniques for *must* behaviors (such as bisimulation and many model checking problems) have been developed in the early 1990's [4, 19] and further extended in [6]. Since these techniques work by unfolding computations until termination, they are applicable to *recursive definitions* that are *noetherian*. As an example, bisimulation for finite CCS can be given an immediate and declarative specification [8].

(3) Theorem proving, infinite behavior Reasoning about all members of a domain or about possibly infinite executions requires induction and coinduction. Incorporating induction in proof theory goes back to Gentzen. The work in [6, 14, 21] provides induction and coinduction rules associated with the above-mentioned recursive definitions. In such a setting, one can prove, for example, that (strong) bisimulation in CCS is a congruence.

1.2. Judgments involving bindings

The proof theoretic treatment of binding in terms has echoed the three stages of development described above.

We switch from CCS to the π-calculus to illustrate the different kinds of judgments that these support.

(1) Logic programming, λ-tree syntax Higher-order generalizations of logic programming, such as *higher-order hereditary Harrop formulas* [12] and the dependently typed LF [5], adequately capture may behavior for terms containing bindings. In particular, the presence of hypothetical and universal judgments supports the λ-tree syntax [11] approach to higher-order abstract syntax [16]. In these systems, reachability in the π-calculus is simple to specify. Both λProlog [15] and Twelf [17] are implementations of such logic programming.

(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 was introduced in [13] to encode generic judgments directly. To illustrate the issues here, 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). 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 instead, we treat $\lambda y.t = \lambda y.s$ as equivalent to $\nabla y.t = s$. In this case, our example formula is equivalent to $\forall w.\neg\nabla x.x = w$, which is provable [13]. Using this idea, the π-calculus process $(\nu x).[x = w].\bar{w}x$ can be encoded such that it is provably bisimilar to 0. Bedwyr [2] is a model checker that treats such generic judgments.

(3) Theorem proving, LG^ω When there is only finite behavior, logics for recursive definitions do not need the cut or initial rules, and, consequently, they do not need to answer the question "When are two generic judgments equal?" On the other hand, induction or coinduction does need an answer to this question: *e.g.*, when doing induction over natural numbers, one must be able to recognize that the case for $i + 1$ has been reduced to the case for i. The LG^ω proof system of [22] provides a natural setting for answering this question. Using LG^ω encodings, one can prove that (open) bisimulation is a π-calculus congruence.

1.3. Allowing definitions of generic judgments

In all the preceding developments, recursive definitions are only possible for *atomic* judgments. In many structure analysis problems, the behavior of binding constructs is characterized by building up a local context that attributes properties to the objects they bind. For example the usual type assignment calculus for λ-terms treats abstractions by adding assumptions about the type of the bound variables to the context of the typing judgment. To model this behavior, we might use the ∇-quantifier to build up a property of the form *cntx* $[\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle]$ that assigns types to variables and then use this in a hypothetical judgment that determines the type of a term. While this approach suffices for certain purposes, it has shortcomings when we want to prove properties of the process described. For example, suppose that we want to show the determinacy of type assignment. In such a proof it is important also to know that a context assigns types only to bound variables and that it assigns at most one type to each of them.

To solve this problem—which is actually endemic to many structure analysis tasks—we extend recursive definitions to apply not only to atomic but also to *generic* judgments. Using this device relative to type assignment, we will, for instance, be able to define the judgment

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

that forces *cntx* to assign types only to variables and at most one to each. These properties can be used in an inductive proof, provided we can verify that the contexts that are built up during type analysis recursively satisfy the definition. We present rules that support this style of argument.

The rest of the paper is structured as follows. Section 2 describes the logic $\mathcal{G}$ that allows for the extended form of definitions and Section 3 establishes its consistency. The extension has significant consequences for writing and reasoning about logical specifications. We provide a hint of this through a few examples in Section 4; as discussed later, many other applications such as solutions to the POPLmark challenge problems [1], cut-elimination for sequent calculi, and an encoding of Tait's logical relations based proof of strong normalization for the simply typed λ-calculus [20] have been successfully developed or are being encoded using the Abella system that implements $\mathcal{G}$. We conclude the paper with a comparison to related work and an indication of future directions.

2. A logic with generalized definitions

The logic $\mathcal{G}$ is obtained by extending an intuitionistic and predicative subset of Church's Simple Theory of Types with fixed point definitions, natural number induction, and a new quantifier for encoding generic judgments. Its main components are elaborated in the subsections below.

2.1. The basic syntax

Following Church, terms are constructed using abstraction and application from constants and (bound) variables. All terms are typed using a monomorphic typing system; these types also constrain the set of well-formed expressions in the expected way. The provability relation concerns well-formed terms of the distinguished type o that are also called formulas. 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 expression $\forall_\tau x.B$ ($\exists_\tau x.B$) abbreviates the formula $\forall_\tau \lambda x.B$ (respectively, $\exists_\tau \lambda x.B$). Type subscripts will be omitted from quantified formulas when they can be inferred from the context or are not important to the discussion.

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. As is argued in [13], there are a number of situations where one wishes to have a generic treatment of a statement like "$B(x)$ holds for all x": in these situations, the *form* of the argument is important and not the argument's behavior on all its possible instances. To encode such generic judgments, we use the quantifier ∇ introduced in [13]. Syntactically, this quantifier corresponds to including a constant ∇_τ of type $(\tau \to o) \to o$ for each type τ (not containing o). 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

Sequents in intuitionistic logic are usually written as

$$\Sigma : B_1, \ldots, B_n \vdash B_0 \qquad (n \geq 0)$$

where Σ is the "global signature" for the sequent: in particular, it contains the eigenvariables of the sequent proof. We shall think of Σ in this prefix position as being a binding operator for each variable it contains. The $FO\lambda^{\Delta\nabla}$ logic of [13] introduced "local signatures" for each formula in the sequent: that is, sequents are written instead as

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

where each $\sigma_0, \ldots, \sigma_n$ is a list of variables that are bound locally in the formula adjacent to it. Such local signatures within proofs reflect bindings in formulas using the ∇-quantifier: in particular, the judgment and formula

$$x_1, \ldots, x_n \triangleright B \quad \text{and} \quad \nabla x_1 \cdots \nabla x_n.B \qquad (n \geq 0)$$

have the same proof-theoretic force.

The $FO\lambda^{\Delta\nabla}$ logic of [13] (and its partial implementation in the Bedwyr logic programming/model checking system [2]) eschewed atomic formulas for explicit fixed point (recursive) definitions, along with inference rules to unfold them. In such a system, both the cut-rule and the initial rules can be eliminated and checking the equality of two generic judgments is not necessary. As we have already mentioned, when one is proving more ambitious theorems involving induction and coinduction, equality of generic judgments becomes important.

2.3. LG^ω and structural rules for ∇-quantification

There are two equations for ∇ that we seem forced to include when we consider proofs by induction. In a sense, these equations play the role of structural rules for the local, generic context. Written at the level of formulas, they are the ∇-*exchange rule* $\nabla x \nabla y.F = \nabla y \nabla x.F$ and the ∇-*strengthening rule* $\nabla x.F = F$, provided x is not free in F. The LG^ω proof system of Tiu [22] is essentially $FO\lambda^{\Delta\nabla}$ extended with these two structural rules for ∇. The move from the weaker $FO\lambda^{\Delta\nabla}$ to the stronger LG^ω logic has at least two important additional consequences.

First, the strengthening rule implies that every type at which one is willing to use ∇-quantification is not only non-empty but contains an unbounded number of members. For example, the formulas $\exists_\tau x.\top$ 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, as will be clear from the proof system given in Figure 1. Similarly, for any given $n \geq 1$, the following formula is provable

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

Second, the validity of the strengthening and exchange rules mean that all local contexts can be made equal. As a result, the local binding can now be considered as an (implicit) global binder. In such a setting, the collection of globally ∇-bound variables can be replaced with *nominal constants*. Of course, in light of the exchange rule, we must consider atomic judgments as being identical if they differ by only permutations of such constants.

We shall follow the LG^ω approach to treating ∇. Thus, for every type we assume an infinite collection of nominal constants. The collection of all nominal constants is denoted by $\mathcal{C}$; these constants are to be distinguished from the collection of usual, non-nominal constants that we denote by $\mathcal{K}$. 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 bijection π from $\mathcal{C}$ to $\mathcal{C}$ such that $\{x \mid \pi(x) \neq x\}$ is finite and π preserves types. Permutations will be extended to terms (and formulas), written $\pi.t$, as follows:

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

The core fragment of $\mathcal{G}$ is 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 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 [10] to encode the dependency of the quantified variable on the support of B. The $\forall\mathcal{L}$ and $\exists\mathcal{R}$ rules

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

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

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

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

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

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

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

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

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 variables 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. However, this asymmetry is not significant: 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.

2.4. Recursive definitions

The structure of definitions in $\mathcal{G}$ is, in a sense, its distinguishing characteristic. To motivate their form and also to understand their expressiveness, we consider first the definitions that are permitted in LG^ω. In that setting, a definitional clause has the form $\forall \vec{x}.H \triangleq B$ where H is an atomic formula all of whose free variables are contained in $\vec{x}$ and B is an arbitrary formula all of whose free variables must also be free in H. In a clause of this sort, H is called the *head* and B is called the *body* and a (possibly infinite) collection of clauses constitutes a definition. Now, there are two properties of such definitional clauses that should be noted. First, H and B are restricted in that no nominal constants are permitted to appear in them. Second, the interpretation of such a clause permits the variables in $\vec{x}$ to be instantiated with terms containing *any* nominal constant; intuitively, the quantificational structure at the head of the definition has a $\nabla\forall$ form, with the (implicit) ∇ quantification being over arbitrary sequences of nominal constants. These two properties actually limit the power of definitions: (subparts of) terms satisfying the relations they identify cannot be forced to be nominal constants and, similarly, specific (sub)terms cannot be stipulated to be independent of such constants.

These shortcomings are addressed in $\mathcal{G}$ by allowing definitional clauses to take the form $\forall \vec{x}.(\nabla \vec{z}.H) \triangleq B$ where all the free variables in $\nabla \vec{z}.H$ must appear in $\vec{x}$ and all the free variables in B must also be free in $\nabla \vec{z}.H$. The intended interpretation of the ∇ quantification over H is that particular terms appearing in the relation being defined must be identified as nominal constants although specific names may still not be assigned to these constants. Moreover, the location of this quantifier changes the prefix over the head from a $\nabla\forall$ form to a more general $\nabla\forall\nabla$ form. Concretely, the explicit ∇ quantification over $\vec{z}$ forces the instantiations for the externally $\forall$ quantified variables $\vec{x}$ to be independent of the nominal constants used for $\vec{z}$.

One illustration of the definitions permitted in $\mathcal{G}$ is provided by the following clause:

$$(\nabla n.\mathit{name}\ n) \triangleq \top.$$

An atomic predicate *name* N would satisfy this clause provided that it can be matched with its head. For this to be possible, N must be a nominal constant. Thus, *name* is a predicate that recognizes such constants. As another example, consider the clause

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

In this case the atomic formula *fresh* N T will satisfy the clause just in case N is a nominal constant and T is a term that does not contain this constant (the impossibility of variable capture ensures this constraint). Thus, this clause expresses the property of a name being "fresh" to a given term.

$$\frac{\{\Sigma'\theta : (\pi.B')\theta, \Gamma'\theta \vdash C'\theta\}}{\Sigma : A, \Gamma \vdash C}\ \mathit{def}\mathcal{L} \qquad \frac{\Sigma' : \Gamma' \vdash (\pi.B')\theta}{\Sigma : \Gamma \vdash A}\ \mathit{def}\mathcal{R}$$

Figure 2. Rules for definitions

Further illustrations of the new form of definitions and their use in reasoning tasks are considered in Section 4.

Definitions impact the logical system through introduction rules for atomic judgments. Formalizing these rules involves the use of substitutions. A *substitution* θ is a type-preserving mapping (whose application is written in postfix notation) from variables to terms, such that the set $\{x \mid x\theta \neq x\}$ is finite. Although a substitution is extended to a mapping from terms to terms, formulas to formulas, *etc*, when we refer to its *domain* and *range*, we mean these sets for this most basic function. A substitution is extended to a function from terms to terms in the usual fashion. If Γ is a multiset of formulas then $\Gamma\theta$ is the multiset $\{J\theta \mid J \in \Gamma\}$. If Σ is a signature then $\Sigma\theta$ is the signature that results from removing from Σ the variables in the domain of θ and adding the variables that are free in the range of θ.

To support the desired interpretation of a definitional clause, when matching the head of $\forall\vec{x}.(\nabla\vec{z}.H) \triangleq B$ with an atomic judgment, we must permit the instantiations for $\vec{x}$ to contain the nominal constants appearing in that judgment. Likewise, we must consider instantiations for the eigenvariables appearing in the judgment that possibly contain the nominal constants chosen for $\vec{z}$. Both possibilities can be realized via raising. Given a clause $\forall x_1, \ldots, x_n.(\nabla\vec{z}.H) \triangleq B$, we define a version of it raised over the nominal constants $\vec{a}$ and away from a signature Σ as

$$\forall\vec{h}.(\nabla\vec{z}.H[h_1\ \vec{a}/x_1, \ldots, h_n\ \vec{a}/x_n]) \triangleq B[h_1\ \vec{a}/x_1, \ldots, h_n\ \vec{a}/x_n],$$

where $h_1, \ldots, h_n$ are distinct variables of suitable type that do not appear in Σ. Given the sequent $\Sigma : \Gamma \vdash C$ and the nominal constants $\vec{c}$ that do not appear in the support of Γ or C, let σ be any substitution of the form

$$\{h'\ \vec{c}/h \mid h \in \Sigma \text{ and } h' \text{ is a variable of suitable type that is not in } \Sigma\}.$$

Then the sequent $\Sigma\sigma : \Gamma\sigma \vdash C\sigma$ constitutes a version of $\Sigma : \Gamma \vdash C$ raised over $\vec{c}$.

The introduction rules based on definitions are presented in Figure 2. The *def*$\mathcal{L}$ rule has a set of premises that is generated by considering each definitional clause of the form $\forall\vec{x}.(\nabla\vec{z}.H) \triangleq B$ in the following fashion. Let $\vec{c}$ be a list of distinct nominal constants equal in length to $\vec{z}$ such that none of these constants appear in the support of Γ, A or C and let $\Sigma' : A', \Gamma' \vdash C'$ denote a version of the lower sequent raised over $\vec{c}$. Further, let H' and B' be obtained by taking the head and body of a version of the clause being considered raised over $\vec{a} = \mathrm{supp}(A)$ and away from Σ' and applying the substitution $[\vec{c}/\vec{z}]$ to them. Then the set of premises arising from this clause are obtained by considering all permutations π of $\vec{a}\vec{c}$ and all substitutions θ such that $(\pi.H')\theta = A'\theta$, with the proviso that the range of θ may not contain any nominal constants.

The *def*$\mathcal{R}$ rule has exactly one premise that is obtained by using any one definitional clause. B' and H' are generated from this clause as in the *def*$\mathcal{L}$ case, but π is now taken to be any one permutation of $\vec{a}\vec{c}$ and θ is taken to be any one substitution such that $(\pi.H')\theta = A'$, again with the proviso that the range of θ may not contain any nominal constants.

In summary, the definition rules are based on raising the sequent over the nominal constants picked for the ∇ variables from the definition, raising the definition over nominal constants from the sequent, and then unifying the chosen atomic judgment and the head of the definition under various permutations of the nominal constants. As it is stated, the set of premises in the *def*$\mathcal{L}$ rule arising from any one definitional clause is potentially infinite because of the need to consider every unifying substitution. It is possible to restrict these substitutions instead to the members of a complete set of unifiers. In the situations where there is a single most general unifier, as is the case when we are dealing with the higher-order pattern fragment [9], the number of premises arising from each definition clause is bounded by the number of permutations. In practice, this number can be quite small as illustrated in Section 4.

Two restrictions must be placed on definitional clauses to ensure consistency of the logic. The first is that no nominal constants may appear in such a clause; this requirement also enforces an equivariance property for definitions. The second is that such clauses must be *stratified* so as to guarantee the existence of fixed points. To do this we associate with each predicate p a natural number $\mathrm{lvl}(p)$, the *level* of p. The notion is generalized to formulas as follows.

Definition 1. *Given a formula B, its level* $\mathrm{lvl}(B)$ *is defined as follows:*

1. $\mathrm{lvl}(p\ \bar{t}) = \mathrm{lvl}(p)$
2. $\mathrm{lvl}(\bot) = \mathrm{lvl}(\top) = \mathrm{lvl}(c) = 0$*, where c is a constant*
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. $\mathrm{lvl}(\forall x.B) = \mathrm{lvl}(\nabla x.B) = \mathrm{lvl}(\exists x.B) = \mathrm{lvl}(B)$

For every definitional clause $\forall\vec{x}.(\nabla\vec{z}.H) \triangleq B$, we require $\mathrm{lvl}(B) \leq \mathrm{lvl}(H)$. See [6, 22] for a description of why these properties lead to consistency.

$$\frac{\vdash I\ z \qquad x : I\ x \vdash I\ (s\ x) \qquad \Sigma : \Gamma, I\ N \vdash C}{\Sigma : \Gamma, \mathit{nat}\ N \vdash C}\ \mathit{nat}\mathcal{L}$$

$$\frac{}{\Sigma : \Gamma \vdash \mathit{nat}\ z}\ \mathit{nat}\mathcal{R} \qquad \frac{\Sigma : \Gamma \vdash \mathit{nat}\ N}{\Sigma : \Gamma \vdash \mathit{nat}\ (s\ N)}\ \mathit{nat}\mathcal{R}$$

Figure 3. Rules for natural number induction

2.5. Induction over natural numbers

The final component of $\mathcal{G}$ is an encoding of natural numbers and rules for carrying out induction over these numbers. This form of induction is useful in reasoning about specifications of computations because it allows us to induct on the height of object-logic proof trees that encode the lengths of computations. Specifically, we introduce the type nt and corresponding constructors $z : nt$ and $s : nt \to nt$. Use of induction is controlled by the distinguished predicate $\mathit{nat} : nt \to o$. The rules for this predicate are presented in Figure 3. The rule $\mathit{nat}\mathcal{L}$ is actually a rule schema, parametrized by the induction invariant I.

3. Cut-elimination and consistency for $\mathcal{G}$

The consistency of $\mathcal{G}$ is an immediate consequence of the cut-elimination result for this logic. Cut-elimination is proved for LG^ω [23] by a generalization of the approach used for $FO\lambda^{\Delta\mathbb{N}}$ [6]. The main aspect of this generalization is recognizing and utilizing the fact that certain transformations of sequents preserve provability and also do not increase (minimum) proof height. The particular transformations that are considered in the case of LG^ω have to do with weakening of hypotheses, permutations of nominal constants and substitutions for eigenvariables. We can use this framework to show that cut can be eliminated from $\mathcal{G}$ by adding one more transformation to this collection. This transformation pertains to the raising of sequents that is needed in the introduction rules based on the extended form of definitional clauses. We motivate this transformation by sketching the structure of the argument as it concerns the use of such clauses below.

The critical part of the cut-elimination argument is the reduction of what are called the essential cases of the use of the *cut* rule, *i.e.*, the situations where the last rule in the derivation is a *cut* and the last rules in the derivations of its premises introduce the cut formula. Now, the only rules of $\mathcal{G}$ that are different from those of LG^ω are $\mathit{def}\mathcal{L}$ and $\mathit{def}\mathcal{R}$. Thus, we have to consider a different argument only when these rules are the last ones used in the premise derivations in an essential case of a *cut*. In this case, the overall derivation has the form

$$\cfrac{\cfrac{\begin{matrix}\Pi_1 \\ \Sigma' : \Gamma' \vdash (\pi.B')\theta\end{matrix}}{\Sigma : \Gamma \vdash A}\ \mathit{def}\mathcal{R} \qquad \cfrac{\left\{\begin{matrix}\Pi_2^{\rho,\pi',B''} \\ \Sigma''\rho : (\pi'.B'')\rho, \Delta''\rho \vdash C''\rho\end{matrix}\right\}}{\Sigma : A, \Delta \vdash C}\ \mathit{def}\mathcal{L}}{\Sigma : \Gamma, \Delta \vdash C}\ \mathit{cut}$$

where Π_1 and $\Pi_2^{\rho,\pi',B''}$ represent derivations of the relevant sequents. Let $\Sigma' : \Gamma' \vdash A'$ be the raised version of $\Sigma : \Gamma \vdash A$ and let H' and B' be the head and body of the version of the definitional clause raised over $\mathrm{supp}(A)$ and away from Σ' used in the $\mathit{def}\mathcal{R}$ rule. From the definition of this rule, we know that θ is substitution such that $(\pi.H')\theta = A'$. Let θ' be the restriction of θ to the free variables of H'. Clearly $(\pi.H')\theta = (\pi.H')\theta'$ and $(\pi.B')\theta = (\pi.B')\theta'$. Further, since the free variables of H' are distinct from the variables in Σ', θ' has no effect on Σ', Δ', C', or A'. Thus, it must be the case that $(\pi.H')\theta' = A'\theta'$. From this it follows that

$$\begin{matrix}\Pi_2^{\theta',\pi,B'} \\ \Sigma' : (\pi.B')\theta', \Delta' \vdash C'\end{matrix}$$

is included in the set of derivations above the lower sequent of the $\mathit{def}\mathcal{L}$ rule. We can therefore reduce the *cut* in question to the following:

$$\frac{\begin{matrix}\Pi_1 \\ \Sigma' : \Gamma' \vdash (\pi.B')\theta'\end{matrix} \quad \begin{matrix}\Pi_2^{\theta',\pi,B'} \\ \Sigma' : (\pi.B')\theta', \Delta' \vdash C'\end{matrix}}{\Sigma' : \Gamma', \Delta' \vdash C'}$$

The proof of cut-elimination for LG^ω is based on induction over the height of the right premise in a *cut*, therefore this *cut* can be further reduced and eliminated. The essential properties we need to complete the proof at this point are that $\Sigma' : \Gamma', \Delta' \vdash C'$ is provable if and only if $\Sigma : \Gamma, \Delta \vdash C$ is provable, and that both proofs have the same height in this case. We formalize these in the lemma below.

Definition 2 (Proof height)**.** *Give a derivation* Π *with premise derivations* $\{\Pi_i\}_{i \in \mathcal{I}}$ *where* $\mathcal{I}$ *is some index set, the measure* $ht(\Pi)$*, the height of* Π*, is defined as the least upper bound of* $\{ht(\Pi_i) + 1\}_{i \in \mathcal{I}}$.

Lemma 3 (Raising)**.** *Let* $\Sigma : \Gamma \vdash C$ *be a sequent, let* $\vec{c}$ *be a list of nominal constants not in the support of* Γ *or* C*, and let* $\Sigma' : \Gamma' \vdash C'$ *be a version of* $\Sigma : \Gamma \vdash C$ *raised over* $\vec{c}$*. Then* $\Sigma : \Gamma \vdash C$ *has a proof of height* h *if and only if* $\Sigma' : \Gamma' \vdash C'$ *has a proof of height* h.

With this lemma in place, the following theorem and its corollary follow.

Theorem 4. *The cut rule can be eliminated from* $\mathcal{G}$ *without affecting the provability relation.*

Corollary 5. *The logic* $\mathcal{G}$ *is consistent,* i.e.*, it is not the case that both* A *and* $A \supset \bot$ *are provable.*

$$\mathit{member}\ B\ L \triangleq \exists n.\mathit{nat}\ n \wedge \mathit{element}_n\ B\ L$$
$$\mathit{element}_z\ B\ (B :: L) \triangleq \top$$
$$\mathit{element}_{(s\ N)}\ B\ (C :: L) \triangleq \mathit{element}_N\ B\ L$$

Figure 4. List membership

Cut-elimination is also useful in designing theorem provers and its counterpart, cut-admissibility, allows one to reason richly about the properties of such proof procedures.

4. Examples

We will often suppress the outermost universal quantifiers of displayed definition and assume that variables written with a capital letter are implicitly universally quantified.

Freshness In Section 2 we showed how the property of freshness could be defined in $\mathcal{G}$ by the definitional clause

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

This clause ensures that the atomic judgment *fresh* X E holds if and only if X is a nominal constant which does not appear anywhere in the term E. To see the simplicity and directness of this definition, consider how we might define freshness in a system like LG^ω which allows for definitions only of atomic judgments. In this situation, we will have to verify that X is a nominal constant by ruling out the possibility that it is a term of one of the other permitted forms. Then, checking that X does not appear in E will require an explicit walking over the structure of E. In short, such a definition would have to have the specific structure of terms coded into it and would also use (a mild form of) negative judgments.

To illustrate how the definition in $\mathcal{G}$ can be used in a reasoning task, consider proving the following lemma

$$\forall x, e, \ell.(\mathit{fresh}\ x\ \ell \wedge \mathit{member}\ e\ \ell) \supset \mathit{fresh}\ x\ e$$

where *member* is defined in Figure 4. This lemma is useful in constructing arguments such as type uniqueness where one must know that a list does not contain a typing judgment for a particular variable. The proof of this lemma proceeds by induction on the natural number n quantified in the body of *member*. The key results which must be shown for the base and inductive step are the following.

$$\forall x, b, \ell.\mathit{fresh}\ x\ (b :: \ell) \supset \mathit{fresh}\ x\ b$$
$$\forall x, b, \ell.\mathit{fresh}\ x\ (b :: \ell) \supset \mathit{fresh}\ x\ \ell$$

We shall consider the proof of the first statement, though the second one has the same structure.

The proof of the first statement reduces to proving the sequent

$$x, b, \ell : \mathit{fresh}\ x\ (b :: \ell) \vdash \mathit{fresh}\ x\ b.$$

Consider how $\mathit{def}\mathcal{L}$ acts on the hypothesis *fresh* x $(b :: \ell)$ in this sequent. First the clause for *fresh* is raised over the support of the hypothesis, but this is empty so no raising is done. Second, the sequent is raised over some new nominal constant c corresponding to the ∇ in the head of the definition for *fresh*. The last step is to consider all permutations π of the set $\{c\}$ and all solutions θ of

$$(\pi.\mathit{fresh}\ c\ e)\theta = (\mathit{fresh}\ (x'\ c)\ ((b'\ c) :: (\ell'\ c)))\theta.$$

Here there is a most general unifier $\theta = [x' \to (\lambda x.x), b' \to (\lambda x.b''), \ell' \to (\lambda x.\ell''), e \to (b'' :: \ell'')]$. The resulting sequent is

$$b'', \ell'' : \top \vdash \mathit{fresh}\ c\ b''$$

The next step in this proof is to apply $\mathit{def}\mathcal{R}$ to the conclusion. To do this we first raise the clause for *fresh* over the support of the conclusion which is $\{c\}$. Then we raise the sequent over a new nominal constant c' corresponding to the ∇ in the head of the definition. Finally we need to find a permutation π of $\{c, c'\}$ and a solution θ to $(\pi.\mathit{fresh}\ c'\ (e'\ c))\theta = \mathit{fresh}\ c\ (b'''\ c')$. Here we find the permutation which swaps c and c' and the solution θ which unifies e' and b'''. The resulting sequent is then

$$b''', \ell''' : \top \vdash \top$$

which is trivially provable.

Typing contexts We can use $\mathcal{G}$ to reason about specifications by encoding them as definitions. Often, however, we are interested both in executing and reasoning about specifications. To achieve these dual goals, we can use the logic of hereditary Harrop formulas. Specifications encoded in this style have a natural executable interpretation based on the logic programming paradigm [12]. Moreover, we can specify the second-order logic of hereditary Harrop formulas as a definition in $\mathcal{G}$ and then reason about specifications through this definition.

We encode the second-order hereditary Harrop logic as a three-place definition $\mathit{seq}_N\ L\ G$ where L denotes the context of hypothetical (assumed) atomic judgments and G denotes the goal formula. The argument N corresponds to the height of the proof tree and is used for inductive arguments; we write this argument in subscript to downplay its significance. The definition of *seq* is presented in Figure 5. The constructor $\langle \cdot \rangle$ is used to inject atomic judgments into formulas. The naturalness of the definition of *seq* comes from the ability to reflect the object level universal quantifier using the generic quantifier ∇. Backchaining is realized by the last clause of *seq*. Here it is assumed that

$$seq_N\ L\ \langle A\rangle \triangleq member\ A\ L$$
$$seq_{(s\ N)}\ L\ (B \wedge C) \triangleq seq_N\ L\ B \wedge seq_N\ L\ C$$
$$seq_{(s\ N)}\ L\ (A \supset B) \triangleq seq_N\ (A :: L)\ B$$
$$seq_{(s\ N)}\ L\ (\forall B) \triangleq \nabla x.seq_N\ L\ (B\ x)$$
$$seq_{(s\ N)}\ L\ \langle A\rangle \triangleq \exists b.prog\ A\ b \wedge seq_N\ L\ b$$

Figure 5. Second-order hereditary Harrop logic in $\mathcal{G}$

$$\forall m, n, t, u[of\ m\ (arr\ u\ t) \wedge of\ n\ u\ \supset\ of\ (app\ m\ n)\ t]$$
$$\forall r, t, u[\forall x[of\ x\ t \supset of\ (r\ x)\ u] \supset of\ (abs\ t\ r)\ (arr\ t\ u)]$$

Figure 6. Simple typing of λ-terms

the specification of interest has been encoded in the definition of *prog*. To simplify notation, we write $L \Vdash P$ for $\exists n.nat\ n \wedge seq_n\ L\ P$. When L is nil we write simply $\Vdash P$.

An example specification using hereditary Harrop formulas is the typing rules for the simply typed λ-calculus. These rules are encoded in Figure 6. Note that no explicit context of typing assumptions is used in these rules, rather the hypothetical judgment of hereditary Harrop formulas is used to track such assumptions. This context is made explicit only when reasoning about this specification via the *seq* definition.

Consider proving the type uniqueness property for the simply typed λ-calculus using the *seq* encoding. We can state the theorem as

$$\forall m, t, s.(\Vdash \langle of\ m\ t\rangle \wedge \Vdash \langle of\ m\ s\rangle) \supset t = s$$

where $=$ is defined by the single clause $\forall x.x = x \triangleq \top$. To prove this theorem, we must generalize the context in which the type judgments are formed to account for descending underneath abstractions in the rule for typing *abs*. A suitably generalized form of the theorem, then, is

$$\forall \ell, m, t, s.(cntx\ \ell \wedge \ell \Vdash \langle of\ m\ t\rangle \wedge \ell \Vdash \langle of\ m\ s\rangle) \supset t = s.$$

Here *cntx* must be defined to restrict the structure of contexts so that this theorem is true. For example, *cntx* must ensure that no variable in the context has two distinct typing assumptions. In truth, the contexts we deal with have the form $of\ c_1\ T_1 :: \ldots :: of\ c_n\ T_n :: nil$ where $c_1 \ldots c_n$ are distinct nominal constants. The challenge then, is providing a definition of *cntx* which accurately describes this structure. In particular, the definition must ensure that the first arguments to *of* are nominal constants and not some other piece of syntax, and it must also ensure that each such constant is distinct from all others.

$$\begin{aligned} &cntx\ nil \triangleq \top \\ &cntx\ (of\ X\ A :: L) \triangleq (\forall M, N.X = app\ M\ N \supset \bot) \wedge \\ &\qquad (\forall M, B.X = abs\ B\ M \supset \bot) \wedge \\ &\qquad (\forall B.member\ (of\ X\ B)\ L \supset \bot) \wedge \\ &\qquad cntx\ L \end{aligned}$$

Figure 7. *cntx* in LG^ω

$$cntx\ nil \triangleq \top$$
$$(\nabla x.cntx\ (of\ x\ A :: L)) \triangleq cntx\ \ L$$

Figure 8. *cntx* in $\mathcal{G}$

In LG^ω, *cntx* can be defined by explicitly restricting each element of the context as shown in Figure 7. This definition checks that the first argument to *of* is a nominal constant by explicitly ruling out all other possibilities for it. Then, to ensure distinctness of arguments, the rest of the list is traversed using *member*. This definition is evidently complex and the complexity carries over also into the process of reasoning based on it.

In $\mathcal{G}$ we can give a direct and concise definition of *cntx* using ∇ quantification in the head of a definition as is done in Figure 8. The occurrence of the ∇-bound variable x in the first argument of *of* codifies the fact that type assignments are only made for nominal constants. The uniqueness of such nominal constants is enforced by the quantification structure of *cntx*: the variable L cannot contain any occurrences of x. With this definition of *cntx*, the generalized theorem of type uniqueness is provable. Use of *def*$\mathcal{L}$ on the hypothesis of *cntx* ℓ will allow only the possibility of type assignments for nominal constants, while use of *def*$\mathcal{R}$ will align with new nominal constants introduced by *seq*.

Arbitrarily cascading substitutions Reducibility arguments, such as Tait's proof of normalization for the simply typed λ-calculus [20], are based on judgments over closed terms. During reasoning, however, one is often working with open terms. To compensate, 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 based on λ-tree syntax since M would have an arbitrary number of abstractions.

We can define arbitrary cascading substitutions in $\mathcal{G}$ us-

$$\mathit{subst}_z \; nil \; T \; T \triangleq \top$$

$$(\nabla x.\mathit{subst}_{(s\ N)} \; ((x, V) :: L) \; (T \; x) \; S) \triangleq \mathit{subst}_N \; L \; (T \; V) \; S$$

Figure 9. Arbitrary cascading substitutions

ing the unique structure of definitions. In particular, we can define a predicate which holds on a list of pairs (c_i, V_i), 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 (c, V) use ∇ in the head of a definition to abstract c out of the first term and then substitute V before continuing. This is the motivation for *subst* defined in Figure 9. Note that we have also added a natural number argument to be used for inductive proofs.

Given the definition of *subst* one may then show that arbitrary cascading substitutions have many of the same properties as normal higher-order substitutions. For instance, in the domain of the untyped λ-calculus, we can show that *subst* acts compositionally via the following lemmas.

$$\forall n, \ell, t, r, s.(\mathit{nat}\; n \wedge \mathit{subst}_n \; \ell \; (\mathit{app}\; t \; r) \; s) \supset \exists u, v.s = \mathit{app}\; u \; v \wedge \mathit{subst}_n \; \ell \; t \; u \wedge \mathit{subst}_n \; \ell \; r \; v$$

$$\forall n, \ell, t, r.(\mathit{nat}\; n \wedge \mathit{subst}_n \; \ell \; (\mathit{abs}\; t) \; r) \supset \exists s.r = \mathit{abs}\; s \wedge \nabla z.\mathit{subst}_n \; \ell \; (t \; z) \; (s \; z)$$

Both of these lemmas have straightforward proofs: induct on n, use $\mathit{def}\mathcal{L}$ on the assumption of *subst*, apply the inductive hypothesis and use $\mathit{def}\mathcal{R}$ to complete the proof.

5. Related work

Nominal logic Nominal logic approaches reasoning about syntax with binders by extending first-order syntax with primitives for managing variable names [18]. In particular, it assumes an infinite supply of names (similar to nominal constants) and primitives for name swapping (permutation of just two names) and for freshness. In addition, nominal logic introduces a new quantifier И (similar to ∇) which quantifies over fresh names.

Despite the similarities, there are many differences between nominal logic and our approach. First, we treat syntax as higher-order and thus obtain substitution via meta-level β-conversion, whereas in nominal logic substitution has to be explicitly defined for each syntactic category. In nominal logic, names must belong to their own type that cannot be the target type of any constructor. We have no such restrictions, *e.g.*, we allow constants such as $\mathit{app} : \mathit{tm} \to \mathit{tm} \to \mathit{tm}$ while at the same time admitting nominal constants of type *tm*. Finally, and most specifically related to $\mathcal{G}$, we can derive freshness through an explicit definition of *fresh* rather than having to take such a notion as primitive.

Twelf Twelf [17] is a system for specifying and reasoning with λ-tree syntax based on the LF logical framework [5]. Reasoning in Twelf is realized through a judgments-as-types principle, *i.e.*, meta-theorems are implemented as predicates which operate on judgments. The input to such predicates can be viewed as universally quantified and the outputs as existentially quantified. Correctness of a meta-theorem is then established by extra-logical checks of predicate properties such as termination and coverage.

Invariants related to the structure of meta-logical contexts are described within Twelf using annotations called *regular worlds*. These annotations are then used by the extra-logical coverage checker as invariants in a kind of inductive proof. The analogue of regular worlds in our system are 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, there are at least two reasons one might prefer defining explicit judgments, such as *cntx*, over the use of regular worlds. First, the *cntx* predicate and invariants related to it are part of $\mathcal{G}$, whereas Twelf uses regular worlds in an extra-logical way: as a consequence, proof objects of meta-theorems in $\mathcal{G}$ are available and these can be checked independently. Second, our approach allows more general judgments over contexts, such as in the example of arbitrary cascading substitutions where the *subst* predicate actively manipulates the context of a term.

Implementation We have implemented $\mathcal{G}$ in a soon to be released system called Abella. This system provides an interactive tactics-based interface to proof construction. The primary focus of Abella is on reasoning about specifications written in hereditary Harrop formulas and treated via the definition of *seq*. Through this approach, Abella is able to take advantage of meta-level properties of the logic of hereditary Harrop formulas (*e.g.*, cut and instantiation properties) while never having to reason outside of $\mathcal{G}$.

Abella has been used in many applications, including all the examples mentioned in this paper. First-order results include reasoning on structures such as natural numbers and lists. Taking advantage of λ-tree syntax, application domains such as the simply typed λ-calculus are directly accessible. Particular results include equivalence of big-step and small-step evaluation, preservation of typing for both forms of evaluation, and determinacy for both forms of evaluation. More advanced results which make use of generic judgments for describing contexts include type uniqueness and disjoint partitioning of λ-terms into normal and non-normal form. Larger applications include challenges 1a and 2a of the POPLmark challenge [1], a task which involves reasoning about the contexts of subtyping judgments for $F_{<:}$, a λ-calculus with bounded subtype polymorphism.

6. Future Work

We are presently investigating the extension of $\mathcal{G}$ with a general treatment of induction over definitions as in the closely related logic Linc [21]. This extension would simplify many inductive arguments by obviating explicit measures in induction; thus, natural numbers encoding computation lengths would not be needed in the definitions of the *element* and *subst* predicates considered in Section 4 if we can induct directly on the unfolding of their definitions. Another benefit of this approach to induction is that it has a naturally dual rule for coinduction over coinductive definitions. This rule has been found useful in Linc, for example, in proving properties of systems such as the π-calculus.

At a practical level, we are continuing to develop Abella as a theorem proving system and to explore its use in complex reasoning tasks. In the latter category, we are presently developing Tait's proof of strong normalization for the simply typed λ-calculus [20]: since Tait's argument uses arbitrarily cascading substitutions, $\mathcal{G}$ is particularly well positioned for this task. We expect also to use Abella to provide more elegant proofs of the many meta-logical theorems found in [7], which include cut-elimination theorems, type preservation, and determinacy of typing and evaluation. Finally, if the previously mentioned work on coinduction is completed, Abella can be used to explore the role of generic definitions in a coinductive setting.

7. Acknowledgements

We thank David Baelde and Alwen Tiu for discussions involving these ideas. This work has been supported by INRIA through the "Equipes Associées" Slimmer, by the NSF Grants OISE-0553462 (IRES-REUSSI) and CCR-0429572 and by a grant from Boston Scientific. 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] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer-Verlag, 2005.

[2] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conference on Automated Deduction*, LNAI 4603, 391–397. Springer, 2007.

[3] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SIGSOFT'88: Third Annual Symposium on Software Development Environments (SDE3)*, Boston, 1988.

[4] J.-Y. Girard. A fixpoint theorem in linear logic. An email posting to linear@cs.stanford.edu, Feb. 1992.

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

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

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

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

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

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

[11] D. Miller. Abstract syntax for variable binders: An overview. In J. Lloyd and et. al., editors, *Computational Logic - CL 2000*, LNAI 1861, 239–253. Springer, 2000.

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

[13] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Comp. Logic*, 6(4):749–783, Oct. 2005.

[14] A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In M. C. S. Berardi and F. Damiani, editors, *Post-proceedings of TYPES 2003*, LNCS 3085, 293–308, Jan. 2003.

[15] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Aug. 1988. MIT Press.

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

[17] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conference on Automated Deduction*, LNAI 1632, 202–206, Trento, 1999. Springer.

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

[19] P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Symposium on Logic in Computer Science*, 222–232. IEEE Computer Society Press, June 1993.

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

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

[22] A. Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP'06)*, 2006.

[23] A. Tiu. A logic for reasoning about generic judgments. Extended version of LFMTP'06 paper., May 2007.