Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 228 (2009) 85–100
www.elsevier.com/locate/entcs

Reasoning in Abella about Structural Operational Semantics Specifications

Andrew Gacek[1] Dale Miller[2] Gopalan Nadathur[1]

Abstract

The approach to reasoning about structural operational semantics style specifications supported by the Abella system is discussed. This approach uses λ-tree syntax to treat object language binding and encodes binding related properties in generic judgments. Further, object language specifications are embedded directly into the reasoning framework through recursive definitions. The treatment of binding via generic judgments implicitly enforces distinctness and atomicity in the names used for bound variables. These properties must, however, be made explicit in reasoning tasks. This objective can be achieved by allowing recursive definitions to also specify generic properties of atomic predicates. The utility of these various logical features in the Abella system is demonstrated through actual reasoning tasks. Brief comparisons with a few other logic based approaches are also made.

Keywords: Structural operational semantics, Abella, λ-tree syntax, object language binding

1 Introduction

This paper concerns reasoning about the descriptions of systems that manipulate formal objects such as programs and their specifications. A common approach to modelling the dynamic and static semantics of these systems is to use a syntax-driven rule-based presentation. These presentations can be naturally encoded as theories within a simple, intuitionistic logic. If the intuitionistic logic supports λ-terms and the quantification of variables ranging over such terms, then it also provides a convenient means for capturing binding notions in the syntactic objects of interest; in particular, it facilitates the use of the λ-tree approach to abstract syntax. A further benefit to using such a logic to encode semantic specifications is that an immediate and effective animation of them is provided by logic programming systems such as λProlog [13] and Twelf [18].

[1] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455.
[2] INRIA Saclay - Île-de-France & LIX/École polytechnique, Palaiseau, France

[3] This work has been supported by INRIA through the "Equipes Associées" Slimmer, and by the NSF Grant CCR-0429572 which includes funding for 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.

doi:10.1016/j.entcs.2008.12.118

Given a logic-based specification of a formal system, establishing properties of the system reduces to answering questions about what is provable in the logic encoding the specification. Different approaches can be adopted for this task. At one end, the specification logic can be formalized and reasoned about within a general purpose theorem-proving framework such as that provided by Coq [2] or Isabelle [15]. At the other end, one can develop another logic, often called a *meta-logic*, that is explicitly tuned to reasoning about the specification logic. It is the latter approach that we examine here. In particular, we expose its practical use within the context of a specific theorem-proving system called Abella [4].

The design of a logic that can act as a powerful and expressive meta-logic has been the subject of much recent research [3,5,10,12,22]. The logics emanating from these studies share a common theme: they all provide recursive definitions as a means for encoding specification logics and some form of generic reasoning for modelling binding notions at the meta level. We expose here an expressive and flexible logic called $\mathcal{G}$ within this framework. Abella is based on $\mathcal{G}$ but also provides special support for the ways in which $\mathcal{G}$ is intended to be used in meta-reasoning tasks. Our presentation pays attention to the novel features of both $\mathcal{G}$ and Abella from this perspective. Concreteness is provided by considering proofs of evaluation, typing, and normalization properties of the λ-calculus.

This paper is organized as follows. The logic $\mathcal{G}$ is summarized in Section 2 and its particular realization in Abella is discussed in Section 3. Section 4 illustrates the use of Abella in a significant theorem-proving task, that of formalizing a Tait-style proof of normalizability in the λ-calculus. Section 5 points out limitations of the currently implemented system. Finally, in Section 6 we compare Abella-style reasoning with some other approaches to the same kind of reasoning tasks.

2 The Logical Foundation

The logic $\mathcal{G}$ [5] which we use to formalize arguments about structural operational semantics is based on an intuitionistic and predicative subset of Church's Simple Theory of Types. Terms in $\mathcal{G}$ are monomorphically typed and are constructed using abstraction and application from constants and (bound) variables. The provability relation concerns 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 are typically omitted from quantified formulas when their identities do not aid the discussion.

The standard treatment of the universal quantifier accords it an extensional interpretation. When treating λ-tree syntax it is often necessary to give importance to the form of the argument for a statement like "$B(x)$ holds for all x" rather than focusing on whether or not every instance of $B(x)$ is true. The ∇ quantifier [12] is used to encode such generic judgments. Specifically, we include the constants ∇_τ of

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

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

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

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

Fig. 1. The core rules of $\mathcal{G}$: the introduction rules for the propositional connectives are not displayed.

type $(\tau \to o) \to o$ for each type τ (not containing o). As with the other quantifiers, $\nabla_\tau x.B$ abbreviates $\nabla_\tau \lambda x.B$.

The $FO\lambda^{\Delta\nabla}$ logic [12] incorporates ∇ quantification into a sequent calculus presentation of intuitionistic proof by attaching a local signature to every formula occurrence in a sequent. We are interested here in considering also proofs that use induction. In this situation, we are led naturally to including certain structural rules pertaining to local signatures [22]. Written at the level of formulas, these are the ∇-*exchange rule* $\nabla x \nabla y.F \equiv \nabla y \nabla x.F$ and the ∇-*strengthening rule* $\nabla x.F \equiv F$, provided x is not free in F. If we adopt these rules, we can make all local signatures equal and hence representable by an (implicit) global binder. We shall refer to these globally ∇-bound variables as *nominal constants*. Intuitively, one can think of nominal constants as denoting arbitrary, unique names. Notice that the exchange rule requires us to consider atomic judgments as being identical if they differ by only permutations of nominal constants.

The logic $\mathcal{G}$ uses the above treatment of the ∇ quantifier that was first introduced in the LG^ω system [22]. Specifically, an infinite collection of nominal constants are assumed for each type. The set of all nominal constants is denoted by $\mathcal{C}$. These constants are distinct from the collection of usual, non-nominal constants denoted by $\mathcal{K}$. We define the *support* of a term (or formula) t, 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. Permutations are extended to terms (and formulas), written $\pi.t$, as follows:

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

Figure 1 presents a subset of the core rules for $\mathcal{G}$; the standard rules for the propositional connectives have been omitted for brevity. Sequents in this logic have the form $\Sigma : \Gamma \vdash C$ where Γ is a set and the signature Σ contains all the free variables of Γ and C. In the rules, Γ, F denotes $\Gamma \cup \{F\}$. In the $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$ rules, a denotes a nominal constant of appropriate type. In the $\exists\mathcal{L}$ and $\forall\mathcal{R}$ rules, $\bar{c}$ is a listing of the variables in $\mathrm{supp}(B)$ and $h\ \bar{c}$ represents the application of h to these constants; raising is used here to encode the dependency of the quantified variable on $\mathrm{supp}(B)$ [8]. The judgment $\Sigma, \mathcal{K}, \mathcal{C} \vdash t : \tau$ that appears in the $\forall\mathcal{L}$ and $\exists\mathcal{R}$ rules enforces the requirement that the expression t instantiating the quantifier in the

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

Fig. 2. Rules for definitions

rule is a well-formed term of type τ constructed from the variables in Σ and the constants in $\mathcal{K} \cup \mathcal{C}$.

Atomic judgments in $\mathcal{G}$ are defined recursively by a set of clauses of the form $\forall\bar{x}.(\nabla\bar{z}.H) \triangleq B$: here H is an atomic formula all of whose free variables are contained in either $\bar{x}$ or in $\bar{z}$ and B is an arbitrary formula all of whose free variables are also free in $\nabla\bar{z}.H$. The atom H is the *head* of such a clause and B is its *body*. No nominal constant is permitted to appear in either of these formulas. A clause of this form provides part of the definition of a relation named by H using B. The ∇ quantifiers over H may be instantiated by distinct nominal constants. The variables $\bar{x}$ that are bound by the $\forall$ quantifiers may be instantiated by terms that depend on any nominal constant except those chosen for the variables in $\bar{z}$.

Certain auxiliary notions are needed in formalizing the rules for definitions in $\mathcal{G}$. A *substitution* θ is a type-preserving mapping from variables to terms such that the set $\{x \mid x\theta \neq x\}$, the *domain* of θ, is finite. A substitution is extended to a function from terms to terms in the usual fashion and we write its application using a postfix notation. If Γ is a set of formulas then $\Gamma\theta$ is the set $\{J\theta \mid J \in \Gamma\}$. 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. Given a clause $\forall x_1, \ldots, x_n.(\nabla\bar{z}.H) \triangleq B$, we define a version of it raised over the nominal constants $\bar{a}$ and away from a signature Σ as

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

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

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

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

The introduction rules for atomic judgments 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 $\forall\bar{x}.(\nabla\bar{z}.H) \triangleq B$ in the following fashion. Let $\bar{c}$ be a list of distinct nominal constants equal in length to $\bar{z}$ 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 $\bar{c}$. Further, let H' and B' be obtained by taking the head and body of a version of the clause being considered raised over $\bar{a} = \mathrm{supp}(A)$ and away from Σ' and applying the substitution $[\bar{c}/\bar{z}]$ to them. Then the set of premises arising from this clause are obtained by considering all permutations π of $\bar{a}\bar{c}$ 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, by contrast, 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 $\bar{a}\bar{c}$ 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.

Some of the expressiveness arising from the quantificational structure permitted in definitions in $\mathcal{G}$ is demonstrated by the following definitional clauses:

$$(\nabla x.name\ x) \triangleq \top \qquad\qquad \forall E.(\nabla x.fresh\ x\ E) \triangleq \top$$

The ∇ quantifier in the first clause ensures that *name* holds only for nominal constants. Similarly, the relative scopes of $\forall$ and ∇ in the second clause force *fresh* to hold only between a nominal constant and a term not containing that constant.

When $\mathcal{G}$ is used in applications, bound variables in syntactic objects will be represented either explicitly, by term-level, λ-bound variables, or implicitly, by nominal constants. The equivariance principle for nominal constants realizes alpha convertibility in the latter situation. Encoding bound variables by λ-terms ensures that substitution is built-in and that dependencies of subterms on bindings is controlled; specific dependencies can be realized by using the device of raising. Definitions with ∇ in the head allow for a similar control over dependencies pertaining to nominal constants and raising can be used to similar effect with these as well.

The consistency of $\mathcal{G}$ requires some kind of stratification condition to govern the possible negative uses of predicates in the body of definitions. There are several choices for such a condition. Rather than picking one in an *a priori* fashion, we will note relevant such conditions as needed.

The final capability of interest is induction over natural numbers. These numbers are encoded in $\mathcal{G}$ using the type nt and the constructors $z : nt$ and $s : nt \to nt$. Use of induction is controlled by the distinguished predicate $nat : nt \to o$ which is treated by specific introduction rules. In particular, the left introduction rule for *nat* corresponds to natural number induction.

3 The Architecture of Abella

Abella is an interactive theorem prover for the logic $\mathcal{G}$. The structure of Abella is influenced considerably by a two-level logic approach to specifying and reasoning about computations. There is a logic—the intuitionistic theory of second-order hereditary Harrop formulas that we call hH^2 here—that provides a convenient vehicle for formulating structural, rule-based characterizations of a variety of properties such as evaluation and type assignment. An especially useful feature of such encodings is that derivations within this "specification" logic reflect the structure of derivations in the object logic.[4] Now, the specification logic can be embedded into $\mathcal{G}$ through the medium of definitions. When used in this manner, $\mathcal{G}$ plays the role of a reasoning or meta logic: formulas in $\mathcal{G}$ can be used to encapsulate properties of derivations in the specification logic and, hence, of computations in the object logic. By keeping the correspondences simple, reasoning within $\mathcal{G}$ can be made to directly reflect the structure of informal arguments relative to the object logics.

[4] Since hH^2 is a subset of λProlog [13], it turns out that such specifications can also be compiled and executed effectively [14].

$$\frac{x : a \in \Gamma}{\Gamma \vdash x : a} \qquad \frac{\Gamma \vdash m : (a \to b) \quad \Gamma \vdash n : a}{\Gamma \vdash m\ n : b} \qquad \frac{\Gamma, x : a \vdash r : b}{\Gamma \vdash (\lambda x{:}a.r) : (a \to b)}\ x \text{ not in } \Gamma$$

Fig. 3. Rules for relating a λ-term to a simple type

$$\forall m, n, a, b[of\, m\ (arr\ a\ b) \wedge of\, n\ a\ \supset\ of\, (app\ m\ n)\ b]$$

$$\forall r, a, b[\forall x[of\, x\ a \supset of\, (r\ x)\ b] \supset of\, (abs\ a\ r)\ (arr\ a\ b)]$$

Fig. 4. Second-order hereditary Harrop formulas (hH^2) encoding simply typing

This two-level logic approach was enunciated by McDowell and Miller already in the context of the logic $FO\lambda^{\Delta\mathbb{N}}$ [10]. Abella realizes this idea using a richer logic that is capable of conveniently encoding more properties of computations. As a theorem prover, Abella also builds in particular properties arising out of the encoding of the specification logic. We discuss these aspects in more detail below.

The specification logic The formulas of hH^2 are given by the following mutually recursive definitions:

$$G\ =\ A \mid A \supset G \mid \forall_\tau x.G \mid G \wedge G \qquad\qquad D\ =\ A \mid G \supset D \mid \forall_\tau x.D$$

In these definitions, A denotes an atomic formula and τ ranges over types of order 0 or 1 not containing o. The sequents for which proofs are constructed in hH^2 are restricted to the form $\Delta \longrightarrow G$ where Δ is a set of D-formulas and G is a G-formula. For such sequents, provability in intuitionistic logic is completely characterized by the more restricted notion of (cut-free) uniform proofs [11]. In the case of hH^2, every sequent in a uniform proof of $\Delta \longrightarrow G$ is of the form $\Delta, \mathcal{L} \longrightarrow G'$ for some G-formula G' and for some set of atoms $\mathcal{L}$. Thus, during the search for a proof of $\Delta \longrightarrow G$, the initial context Δ is *global*: changes occur only in the set of atoms on the left and the goal formula on the right.

We briefly illustrate the ease with which type assignment for the simply typed λ-calculus can be encoded in hH^2. There are two classes of objects in this domain: types and terms. For types we will consider a single base type called i and the arrow constructor for forming function types. Terms can be variables x, applications $(m\ n)$ where m and n are terms, and typed abstractions $(\lambda x{:}a.r)$ where r is a term and a is the type of x. The standard rules for assigning types to terms are given in Figure 3. Object-level untyped λ-terms and simple types can be encoded in a simply typed (meta-level) λ-calculus as follows. The simple types are built from the two constructors i and *arr* and terms are built using the two constructors *app* and *abs*. Here, the constructor *abs* takes two arguments: one for the type of the variable being abstracted and the other for the actual abstraction. Terms in the specification logic contain binding and so there is no need for an explicit constructor for variables. Thus, the (object-level) term $(\lambda f{:}i \to i.(\lambda x{:}i.(f\ x)))$ can be encoded as the meta-level term $abs\ (arr\ i\ i)\ (\lambda f.abs\ i\ (\lambda x.app\ f\ x))$.

Given this encoding of the untyped λ-calculus and simple types, the inference rules of Figure 3 can be specified by the hH^2 formulas in Figure 4 involving the binary predicate *of*. Note that this specification in hH^2 does not maintain an explicit

$$element_N\ B\ (B :: L) \triangleq \top \qquad element_{(s\ N)}\ B\ (C :: L) \triangleq element_N\ B\ L$$

$$member\ B\ L \triangleq \exists n.nat\ n \wedge element_n\ B\ L$$

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

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

context for typing assumptions but uses hypothetical judgments instead. Also, the explicit side-condition in the rule for typing abstractions is not needed since it is captured by the usual proof theory of the universal quantifier in the hH^2 logic.

Encoding specification logic provability in $\mathcal{G}$ The definitional clauses in Figure 5 encode hH^2 provability in $\mathcal{G}$. In these and other such clauses in this paper, we use the convention that capitalized variables are implicitly universally quantified at the head. This encoding of hH^2 provability derives from McDowell and Miller [10]. As described earlier, uniform proofs in hH^2 contain sequents of the form $\Delta, \mathcal{L} \longrightarrow G$ where Δ is a fixed set of D-formulas and $\mathcal{L}$ is a varying set of atomic formulas. Our encoding uses the $\mathcal{G}$ predicate *prog* to represent the D-formulas in Δ: the D formula $\forall\bar{x}.[G_1 \supset \cdots \supset G_n \supset A]$ is encoded as the clause $\forall\bar{x}.prog\ A\ (G_1 \wedge \cdots \wedge G_n) \triangleq \top$ and $\forall\bar{x}.A$ is encoded by the clause $\forall\bar{x}.prog\ A\ tt \triangleq \top$. Sequents are encoded using the atomic formula $(seq_N\ L\ G)$ where L is a list encoding the set of atomic formulas $\mathcal{L}$ and G encodes the G-formula. The argument N, written as a subscript, encodes the height of the proof tree that is needed in inductive arguments. The constructor $\langle\cdot\rangle$ is used to inject the special type of atom into formulas. To simplify notation, we write $L \Vdash G$ for $\exists n.nat\ n \wedge seq_n\ L\ G$. When L is *nil* we write simply $\Vdash G$.

Proofs of universally quantified G formulas in hH^2 are generic in nature. A natural encoding of this (object-level) quantifier in the definition of *seq* uses a (meta-level) ∇-quantifier. In the case of proving an implication, the atomic assumption is maintained in a list (the second argument of *seq*). The penultimate clause for *seq* implements backchaining over a fixed hH^2 specification (stored as *prog* atomic formulas). The matching of atomic judgments to heads of clauses is handled by the treatment of definitions in the logic $\mathcal{G}$, thus the penultimate rule for *seq* simply performs this matching and makes a recursive call on the corresponding clause body.

With this kind of an encoding, we can now formulate and prove in $\mathcal{G}$ statements about what is or is not provable in hH^2. Induction over the height of derivations may be needed in such arguments and this can be realized via natural number induction on n in $seq_n\ L\ P$. Furthermore, the *def*$\mathcal{L}$ rule encodes case analysis in the derivation of an atomic goal, leading eventually to a consideration of the different ways in which an atomic judgment may have been inferred in the specification logic. Abella is designed to hide much of the details of how the *seq* and *prog* specifications

work and to reflect instead the aggregate structure described here.

Since we have encoded the entire specification logic, we can prove general properties about it in $\mathcal{G}$ that can then be used in reasoning about particular specifications. In Abella, various such specification logic properties can be invoked either automatically or through the use of tactics. For example, the following property, which is provable in $\mathcal{G}$, states the judgment $\ell \Vdash g$ is not affected by permuting, contracting, or weakening the context of hypothetical assumptions ℓ.

$$\forall \ell_1, \ell_2, g.(\ell_1 \Vdash g) \wedge (\forall e.\mathit{member}\ e\ \ell_1 \supset \mathit{member}\ e\ \ell_2) \supset (\ell_2 \Vdash g)$$

This property can be applied to any specification judgment that uses hypothetical assumptions. Using it with the encoding of typing judgments for the simply typed λ-calculus, for example, we easily obtain that permuting, contracting, or weakening the typing context of a typing judgment does not invalidate that judgment.

Two additional properties of our specification logic which are useful and provable in $\mathcal{G}$ are called the *instantiation* and *cut* properties. The instantiation property recovers the notion of universal quantification from our representation of the specification logic $\forall$ using ∇. The exact property is

$$\forall \ell, g.(\nabla x.(\ell\ x) \Vdash (g\ x)) \supset \forall t.(\ell\ t) \Vdash (g\ t).$$

Stated another way, although ∇ quantification cannot be replaced by $\forall$ quantification in general, it can be replaced in this way when dealing with specification judgments. The cut property allows us to remove hypothetical judgments using a proof of such judgments. This property is stated as the formula

$$\forall \ell_1, \ell_2, a, g.(\ell_1 \Vdash \langle a \rangle) \wedge (a :: \ell_2 \Vdash g) \supset (\ell_1, \ell_2 \Vdash g),$$

which can be proved in $\mathcal{G}$: here, ℓ_1, ℓ_2 denotes the appending of two contexts. As a concrete example, we can again take our specification of simply typed λ-calculus and use the instantiation and cut properties to establish a type substitution property, *i.e.*, if $\Gamma_1, x : a \vdash m : b$ and $\Gamma_2 \vdash n : a$ then $\Gamma_1, \Gamma_2 \vdash m[x := n] : b$.

Encoding properties of specifications in definitions Definitions were used above to encode the specification logic and also particular specifications in $\mathcal{G}$. There is another role for definitions in Abella: they can be used also to capture implicit properties of a specification that are needed in a reasoning task. As an example, consider the encoding of type assignment. Here, the instances of $(\mathit{seq}_N\ L\ G)$ that arise all have L bound to a list of entries of the form $(\mathit{of}\ x\ t)$ where x is a nominal constant that is, moreover, different from all other such constants appearing in L. Observing these properties is critical to proving the uniqueness of type assignment. Towards this end, we may define a predicate *cntx* via the following clauses:

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

Reasoning within $\mathcal{G}$, it can now be shown that L in every $(\mathit{seq}_N\ L\ G)$ atom whose proof is considered always satisfies the property expressed by *cntx* and, further, if L satisfies such a property then the uniqueness of type assignment is guaranteed.

Induction on definitions The logic $\mathcal{G}$ supports induction only over natural numbers. Thus the definitions of *element* and *seq* in Figure 5 both make use of

$$\forall a, r[value\ (abs\ a\ r)]$$

$$\forall m, n, m'[step\ m\ m' \supset step\ (app\ m\ n)\ (app\ m'\ n)]$$
$$\forall m, n, n'[value\ m \wedge step\ n\ n' \supset step\ (app\ m\ n)\ (app\ m\ n')]$$
$$\forall a, r, m[value\ m \supset step\ (app\ (abs\ a\ r)\ m)\ (r\ m)]$$

$$\forall m[steps\ m\ m] \qquad \forall m, n, p[step\ m\ p \wedge steps\ p\ n \supset steps\ m\ n]$$

$$type\ i \qquad \forall a, b[type\ a \wedge type\ b \supset type\ (arr\ a\ b)]$$

$$\forall a, b, m, n[of\ m\ (arr\ a\ b) \wedge of\ n\ a \supset of\ (app\ m\ n)\ b]$$
$$\forall a, b, r[type\ a \wedge \forall x[of\ x\ a \supset of\ (r\ x)\ b] \supset of\ (abs\ a\ r)\ (arr\ a\ b)]$$

Fig. 6. Specification of simply-typed λ-calculus

a natural number argument to provide a target for induction. In Abella, such arguments are unnecessary since the system implicitly assigns such an additional argument to all definitions. Thus when we refer to induction over a definition we mean induction on the implicit natural number argument of that definition.

4 Example: Normalizability in the Typed λ-Calculus

In order to illustrate the strengths and weaknesses of Abella, we detail in this section a proof of normalizability for the call-by-value, simply typed λ-calculus (sometimes also called "weak normalizability"). We follow here the proof presented in [17]. Stronger results are possible for the full, simply typed λ-calculus, but the one at hand suffices to expose the interesting reasoning techniques. The proof under consideration is based on Tait's logical relations argument [21] and makes use of simultaneous substitutions.

Figure 6 contains the specification of call-by-value evaluation and of simple typing for the λ-calculus. Values are recognized by the predicate *value*. Small-step evaluation is defined by *step*, and a possibly zero length sequence of small steps is defined by *steps*. The predicate *type* recognizes well-formed types, and *of* defines the typing rules of the calculus. A noteworthy aspect of the specification of the *of* predicate is that it uses the *type* predicate to ensure that types mentioned in abstraction terms are well-formed: a fact used in later arguments.

The goal of this section is to prove weak normalizability, which we can now state formally in our meta-logic as follows:

$$\forall M, A.(\Vdash \langle of\ M\ A \rangle) \supset \exists V.(\Vdash \langle steps\ M\ V \rangle) \wedge (\Vdash \langle value\ V \rangle).$$

The rest of this section describes definitions and lemmas necessary to prove this formula. In general, almost all results in this section have simple proofs based on induction, case analysis, applying lemmas, and building results from hypotheses. For such proofs, we will omit the details except to note the inductive argument and key lemmas used. The full details of this development are available in the software distribution of Abella.

Evaluation and typing Definitions can be used in Abella to introduce useful intervening concepts. One such concept is that of halting. We say that a term

M halts if it evaluates to a value in finitely many steps and we define a predicate capturing this notion as follows:

$$\mathit{halts}\ M \triangleq \exists V.(\Vdash \langle \mathit{steps}\ M\ V\rangle) \wedge (\Vdash \langle \mathit{value}\ V\rangle).$$

An most important property about halting is that it is invariant under evaluation steps (both forwards and backwards). Using the abbreviation $F \equiv G$ for $(F \supset G) \wedge (G \supset F)$, we can state this property formally as

$$\forall M, N.(\Vdash \langle \mathit{step}\ M\ N\rangle) \supset (\mathit{halts}\ M \equiv \mathit{halts}\ N).$$

This result is immediate in the backward direction, *i.e.*, $\mathit{halts}\ N \supset \mathit{halts}\ M$. In the forward direction it requires showing that one step of evaluation is deterministic:

$$\forall M, N, P.(\Vdash \langle \mathit{step}\ M\ N\rangle) \wedge (\Vdash \langle \mathit{step}\ M\ P\rangle) \supset N = P.$$

This formula is proved by induction on the height of the derivation of either one of the judgments involving the *step* predicate.

A standard result in the λ-calculus, which we will need later, is that one step of evaluation preserves typing. This is stated formally as

$$\forall M, N, A.(\Vdash \langle \mathit{step}\ M\ N\rangle) \wedge (\Vdash \langle \mathit{of}\ M\ A\rangle) \supset (\Vdash \langle \mathit{of}\ N\ A\rangle).$$

The proof of this formula uses induction on the height of the derivation of the judgment involving the *step* predicate. An interesting case in this proof is when $\mathit{step}\ M\ N$ is $\mathit{step}\ (\mathit{app}\ (\mathit{abs}\ B\ R)\ P)\ (R\ P)$ for some B, R, and P, *i.e.*, when β-reduction is performed. Deconstructing the typing judgment

$$(\Vdash \langle \mathit{of}\ (\mathit{app}\ (\mathit{abs}\ B\ R)\ P)\ A\rangle)$$

we can deduce that $(\Vdash \langle \mathit{of}\ P\ B\rangle)$ and $((\mathit{of}\ x\ B) :: \mathit{nil} \Vdash \langle \mathit{of}\ (R\ x)\ A\rangle)$ where x is a nominal constant. Here we use the instantiation property of our specification logic to replace x with P yielding $((\mathit{of}\ P\ B) :: \mathit{nil} \Vdash \langle \mathit{of}\ (R\ P)\ A\rangle)$. Next we apply the cut property of our specification logic to deduce $(\Vdash \langle \mathit{of}\ (R\ P)\ A\rangle)$ which is our goal.

Finally, we note that the contexts which are constructed during the proof of a typing judgment always have the form $(\mathit{of}\ x_1\ a_1) :: \ldots :: (\mathit{of}\ x_n\ a_n) :: \mathit{nil}$ where the x_i's are distinct nominal constants and the a_i's are valid types. We introduce the following formal definition of *cntx* to exactly describe such contexts:

$$\mathit{cntx}\ \mathit{nil} \triangleq \top \qquad (\nabla x.\mathit{cntx}\ ((\mathit{of}\ x\ A) :: L)) \triangleq (\Vdash \langle \mathit{type}\ A\rangle) \wedge \mathit{cntx}\ L$$

Note, ∇ in the definition head ensures that the x_i's are distinct nominal constants.

The logical relation The difficulty with proving weak normalizability directly is that the halting property is not closed under application, *i.e.*, *halts* M and *halts* N does not imply $\mathit{halts}\ (\mathit{app}\ M\ N)$. Instead, we must strengthen the halting property to one which includes a notion of closure under application. We define the logical relation *reduce* by induction over the type of a term as follows:

$$\begin{aligned}
&\mathit{reduce}\ M\ i && \triangleq (\Vdash \langle \mathit{of}\ M\ i\rangle) \wedge \mathit{halts}\ M\\
&\mathit{reduce}\ M\ (\mathit{arr}\ A\ B) && \triangleq (\Vdash \langle \mathit{of}\ M\ (\mathit{arr}\ A\ B)\rangle) \wedge \mathit{halts}\ M \wedge\\
& && \quad \forall N.(\mathit{reduce}\ N\ A \supset \mathit{reduce}\ (\mathit{app}\ M\ N)\ B)
\end{aligned}$$

Note that *reduce* is defined with a negative use of itself. Such a usage is permitted

in $\mathcal{G}$ only if there is a stratification condition that ensures that there are no logical cycles in the definition. In this case, the condition to use is obvious: the second argument to *reduce* decreases in size in the recursive use.

Like *halts*, the *reduce* relation is preserved by evaluation:

$$\forall M, N, A.(\Vdash \langle step\ M\ N\rangle) \wedge (\Vdash \langle of\ M\ A\rangle) \supset (reduce\ M\ A \equiv reduce\ N\ A).$$

This formula is proved by induction on the definition of *reduce*, using the lemmas that *halts* is preserved by evaluation and *of* is preserved by evaluation.

Clearly *reduce* is closed under application and it implies the halting property, thus we strengthen our desired weak normalizability result to the following:

$$\forall M, A.(\Vdash \langle of\ M\ A\rangle) \supset reduce\ M\ A.$$

In order to prove this formula we will have to induct on the height of the proof of the judgment $(\Vdash \langle of\ M\ A\rangle)$. However, when we consider the case that M is an abstraction, we will not be able to use the inductive hypothesis on the body of M since *reduce* is defined only on closed terms, *i.e.*, those typeable in the empty context. The standard way to deal with this issue is to generalize the desired formula to say that if M, a possibly open term, has type A then each closed instantiation for all the free variables in M, say N, satisfies *reduce* N A. This requires a formal description of simultaneous substitutions that can "close" a term.

Arbitrary cascading substitutions and freshness Given $(L \Vdash \langle of\ M\ A\rangle)$, *i.e.*, an open term and its typing context, we define a process of substituting each free variable in M with a value V which satisfies the logical relation for the appropriate type. We define this *subst* relation as follows:

$$\begin{aligned} &subst\ nil\ M\ M \triangleq \top \\ &(\nabla x.\ subst\ ((of\ x\ A) :: L)\ (R\ x)\ M) \triangleq \\ &\qquad \exists V.\ reduce\ V\ A \wedge (\Vdash \langle value\ V\rangle) \wedge subst\ L\ (R\ V)\ M \end{aligned}$$

By employing ∇ in the head of the second clause, we are able to use the notion of substitution in the meta-logic to directly and succinctly encode substitution in the object language. Also note that we are, in fact, defining a process of cascading substitutions rather than simultaneous substitutions. Since the substitutions we define (using closed terms) do not affect each other, these two notions of substitution are equivalent. We will have to prove some part of this formally, of course, which in turn requires proving results about the (non)occurrences of nominal constants in our judgments. The results in this section are often assumed in informal proofs.

One consequence of defining cascading substitutions via the notion of substitution in the meta-logic is that we do not get to specify where substitutions are applied in a term. In particular, given an abstraction *abs* A R we cannot preclude the possibility that a substitution for a nominal constant in this term will affect the type A. Instead, we must show that well-formed types cannot contain free variables which can be formalized as $\forall A.\nabla x.(\Vdash \langle type\ (A\ x)\rangle) \supset \exists B.\ A = \lambda y.B$. This formula essentially states that any well-formed type which possibly depends on a nominal constant x must depend on it only in a vacuous way.

The above result about types assumes that judgments concerning *type* occur in

an empty context. Now, such judgments actually enter the picture through uses of the specification logic rule for *of* that deals with the case of abstractions. This means that we have to consider judgments involving *type* that have a context meant to be used in judgments involving the *of* predicate. To use the result we have just established, we must show that these contexts can be ignored. We formalize this as $\forall L, A.\ cntx\ L \wedge (L \Vdash \langle type\ A \rangle) \supset (\Vdash \langle type\ A \rangle)$, a formula that can be proved using induction on the proof of the judgment $(L \Vdash \langle type\ A \rangle)$. In the base case we must establish $\forall L, A.\ cntx\ L \wedge member\ (type\ A)\ L \supset \bot$, which is proved by induction on the proof of *member*.

Another necessary result is that in any provable judgment of the form $(L \Vdash \langle of\ M\ A \rangle)$, any nominal constant (denoting a free variable) in M must also occur in L, *i.e.*,

$$\forall L, R, A. \nabla x.\ cntx\ L \wedge (L \Vdash \langle of\ (R\ x)\ (A\ x) \rangle) \supset \exists M.\ R = \lambda y. M$$

The proof is by induction on the height of the derivation of the judgment involving *of*. In the base case, we need that an element of a list cannot contain any nominal constant which does not occur in the list, *i.e.*, $\forall L, E. \nabla x.\ member\ (E\ x)\ L \supset \exists F.\ E = \lambda y. F$. This formula is proved by induction on *member*.

We next show that typing judgments produce well-formed types by proving

$$\forall L, M, A.\ cntx\ L \wedge (L \Vdash \langle of\ M\ A \rangle) \supset (\Vdash \langle type\ A \rangle).$$

The induction here is on the height of the derivation of the judgment involving *of* and the base case is $\forall L, M, A.\ cntx\ L \wedge member\ (of\ M\ A)\ L \supset (\Vdash \langle type\ A \rangle)$, which is proved by a simple induction on *member*.

Given our repertoire of results about the occurrences of nominal constants in judgments, we can now prove fundamental properties of arbitrary cascading substitutions. The first property states that closed terms, those typeable in the empty context, are not affected by substitutions, *i.e.*,

$$\forall L, M, N, A.\ (\Vdash \langle of\ M\ A \rangle) \wedge subst\ L\ M\ N \supset M = N.$$

The proof here is by induction on *subst* which corresponds to induction on the length of the list L. The key step within the proof is using the lemma that any nominal constant in the judgment $(\Vdash \langle of\ M\ A \rangle)$ must also be contained in the context of that judgment. Since the context is empty in this case, there are no nominal constants in M and thus the substitutions from L do not affect it.

We must show that our cascading substitutions act compositionally on terms in the object λ-calculus. This is stated formally for application as follows:

$$\forall L, M, N, R.\ cntx\ L \wedge subst\ L\ (app\ M\ N)\ R \supset \exists M', N'.\ R = app\ M'\ N' \wedge subst\ L\ M\ M' \wedge subst\ L\ N\ N'.$$

This is proved by induction on *cntx*, which amounts to induction on the length of the list L. For abstractions we prove the following, also by induction on *cntx*:

$$\forall L, M, R, A.\ cntx\ L \wedge subst\ L\ (abs\ A\ M)\ R \wedge (\Vdash \langle type\ A \rangle) \supset \exists M'.\ R = abs\ A\ M' \wedge (\forall V.\ reduce\ V\ A \wedge (\Vdash \langle value\ V \rangle) \supset \nabla x.\ subst\ ((of\ x\ A) :: L)\ (M\ x)\ (M'\ V)).$$

Here we have the additional hypothesis of $(\Vdash \langle type\ A \rangle)$ to ensure that the substitutions created from L do not affect A. At one point in this proof we have to show that the order in which cascading substitutions are applied is irrelevant. The key to showing this is realizing that all substitutions are for closed terms. Since closed terms cannot contain any nominal constants, substitutions do not affect each other.

Finally, we must show that cascading substitutions preserve typing. Moreover, after applying a full cascading substitution for all the free variables in a term, that term should now be typeable in the empty context:

$$\forall L, M, N, A.\ cntx\ L \wedge subst\ L\ M\ N \wedge (L \Vdash \langle of\ M\ A \rangle) \supset (\Vdash \langle of\ N\ A \rangle).$$

This formula is proved by induction on *cntx* and by using the instantiation and cut properties of our specification logic.

The final result Using cascading substitutions we can now formalize the generalization of weak normalizability that we described earlier: given a (possibly open) well-typed term, every closed instantiation for it satisfies the logical relation *reduce*:

$$\forall L, M, N, A.\ cntx\ L \wedge (L \Vdash \langle of\ M\ A \rangle) \wedge subst\ L\ M\ N \supset reduce\ N\ A.$$

The proof of this formula is by induction on the height of the derivation of the typing judgment $(L \Vdash \langle of\ M\ A \rangle)$. The inductive cases are fairly straightforward using the compositional properties of cascading substitutions and various results about invariance under evaluation. In the base case, we must prove

$$\forall L, M, N, A.\ cntx\ L \wedge member\ (of\ M\ A)\ L \wedge subst\ L\ M\ N \supset reduce\ N\ A,$$

which is done by induction on *cntx*. Weak normalizability is now a simple corollary where we take L to be nil. Thus we have proved $\forall M, A.(\Vdash \langle of\ M\ A \rangle) \supset halts\ M$.

5 Assessment and Future Work

The Abella system has been tested with several prototypical examples; details are available with the system distribution. These experiments indicate considerable promise for the two-level logic based approach in reasoning about formal systems. However, the experiments have also revealed some issues with Abella at a practical level. We discuss these below and suggest work aimed at addressing them.

Base case lemmas Every lemma whose proof uses induction on a specification logic judgment with a non-empty context requires another lemma to be proved for the base case where that judgment follows because it is in the context. This creates mundane overhead. The work in these base case lemmas consists of a simple induction over the length of the context. Support for richer tactics for induction on specification judgments might lead to more user friendly behavior in such cases.

Types in specifications The specification logic is embedded as an untyped logic in $\mathcal{G}$. This is usually not an issue: specification logic judgments themselves impose type restrictions on terms. For example, the typing judgment $of\ M\ A$ holds only if M is a λ-term. However, sometimes explicit type judgments—such as the judgment *type* for recognizing well-formed simple types—are required in specifications. One

possibility that is being considered for addressing the typing issue that is of an implementation such as Abella automatically generating recognizer predicates based on type information. These predicates could then be implicitly attached to all declarations of meta-level variables.

Different specification logics Currently, Abella has built into it exactly one specification language (hH^2) and exactly one proof system for it (uniform proofs). Certain application areas might benefit from having other proof systems for intuitionistic logic available as well as other specification logics. For example, linear logic specification languages [7,9] can be used to provide declarative specifications of the operational semantics of programming languages that contain features such as references, exceptions, and concurrency. Thus, McDowell and Miller [10] presented a *seq*-like predicate for a subset of intuitionistic linear logic that they used to specify the operational semantics of a simple functional language extended with references and to then prove a subject-reduction theorem for that language. It would be natural to consider extending the specification logic in Abella to be all of intuitionistic linear logic (or, in fact, all of linear logic) since this would enhance that logic's expressiveness a great deal. Such an extension could be designed so that if a given specification did not employ the novel linear logic connectives, then the encoding of *seq* would modularly revert back to that of intuitionistic logic.

6 Related Work

Nominal logic approach The Nominal package for Isabelle/HOL automates a process of defining and proving standard results about α-equivalence classes [23]. This allows for formal reasoning over objects with binding which is close to informal reasoning. One drawback of the nominal approach is that it does not provide a notion of substitution, and thus users must define their own substitution function and prove various properties relating to it. A proof of weak normalizability for the simply typed λ-calculus has been conducted with the nominal package [16], and in this case a notion of simultaneous substitution is used. For the nominal approach, this extended notion of substitution can be defined directly since one works with α-equivalence classes and not higher-order terms as in our case. Additionally, the cost of defining and reasoning about simultaneous substitution is not a significant step up from what is already required for standard substitution in the nominal approach.

The specification language for the nominal package is functions and predicates over α-equivalence classes. This language does not have a built-in notion of hypothetical judgments which are typically useful for describing structural rules over objects with binding. For example, by encoding the simply typed λ-calculus in our specification language using hypothetical judgments for typing assumptions, we derive a type substitutivity property as consequence of general instantiation and cut properties of the logic, see Section 3. In the nominal approach, such a proof must be conducted manually.

Twelf The Twelf system [18] uses LF terms and types for a specification language [6] and the meta-logic $\mathcal{M}_2^+$ [19] for reasoning. The primary difference between the

Twelf approach and ours is that the $\mathcal{M}_2^+$ meta-logic is relatively weak in expressive power. For instance, it is restricted to Π_2 formulas (*i.e.*, $\forall\exists$ formulas) and lacks logical connectives such as conjunction, disjunction, and implication. Despite these restrictions, the meta-logic is expressive enough for most common reasoning tasks and has been very successful in practice. Another significant difference is that $\mathcal{M}_2^+$ is designed with an inherent notion of a global hypothetical context. Thus the meta-logic builds in some notion of which judgments can depend on assumptions of other judgments. This is less of a concern in our approach since each judgments has its own local context.

Due to the Π_2 restriction of the meta-logic $\mathcal{M}_2^+$, it is not possible to encode a direct proof of weak normalizability for the simply typed λ-calculus using a logical relations argument. Recently, however, an indirect proof was completed using an intermediate *assertion logic* which has enough richness to encode the proper logical relation [20]. This is a useful technique for extending the expressive power of the Twelf system, but it comes with the cost of moving from a two-level logic approach to a three-level logic approach.

Locally nameless The locally nameless representation for syntactic objects with binding is a first-order approach using de Bruijn indices for bound variables and names for free variables. This balance between two representational techniques has been used successfully in practice [1]. Our approach to representation can be seen as a meta-level version of this balance where we use (meta-level) λ-terms to represent explicitly bound variables and (meta-level) nominal constants for implicitly bound variables (*i.e.*, free variables). With this understanding, the trade-off between the first-order and meta-level approaches to bound/free variable representation is that the former works with existing theorem provers while the latter has substitution and equivariance built-in.

References

[1] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *35th ACM Symp. on Principles of Programming Languages*, pages 3–15. ACM, January 2008.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *21th Conference on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[4] Andrew Gacek. The Abella interactive theorem prover (system description). In *Fourth International Joint Conference on Automated Reasoning*, 2008. Available from http://arxiv.org/abs/0803.2305. To appear in IJCAR.

[5] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*. IEEE Computer Society Press, 2008. To appear.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[7] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

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

[9] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, September 1996.

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

[11] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

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

[13] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[14] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conference on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[16] Julien Narboux and Christian Urban. Nominal formalisations of typical SOS proofs. Available at http://dpt-info.u-strasbg.fr/~narboux/papers/SOS.pdf, 2008.

[17] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[18] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conference on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

[19] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, October 2000. CMU-CS-00-146.

[20] Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*. IEEE Computer Society Press, 2008. To appear.

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

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

[23] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In R. Nieuwenhuis, editor, *20th Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 38–53. Springer, 2005.