A two-level logic approach to reasoning about computations

Andrew Gacek · Dale Miller · Gopalan Nadathur

Abstract Relational descriptions have been used in formalizing diverse computational notions, including, for example, operational semantics, typing, and acceptance by non-deterministic machines. We therefore propose a (restricted) logical theory over relations as a language for specifying such notions. Our *specification logic* is further characterized by an ability to explicitly treat binding in object languages. Once such a logic is fixed, a natural next question is how we might prove theorems about specifications written in it. We propose to use a second logic, called a *reasoning logic*, for this purpose. A satisfactory reasoning logic should be able to completely encode the specification logic. Associated with the specification logic are various notions of binding: for quantifiers within formulas, for eigenvariables within sequents, and for abstractions within terms. To provide a natural treatment of these aspects, the reasoning logic must encode binding structures as well as their associated notions of scope, free and bound variables, and capture-avoiding substitution. Further, to support arguments about provability, the reasoning logic should possess strong mechanisms for constructing proofs by induction and co-induction. We provide these capabilities here by using a logic called $\mathcal{G}$ which represents relations over λ-terms via definitions of atomic judgments, contains inference rules for induction and co-induction, and includes a special *generic* quantifier. We show how provability in the specification logic can be transparently encoded in $\mathcal{G}$. We also describe an interactive theorem prover called Abella that implements $\mathcal{G}$ and this two-level logic approach and we present several examples that demonstrate the efficacy of Abella in reasoning about computations.

1 Introduction

We are interested in this paper in specifying computations and then reasoning about them. A range of formalisms have been used as a means for realizing the first of these objectives. For example, the execution semantics of programming languages have been describe via the λ-calculus [Reynolds, 1972, Plotkin, 1976], the π-calculus [Milner, 1992], and abstract machines [Landin, 1964]. A specification formalism that has been particularly successful and widely applicable is *operational semantics* in both its "small-step" version [Plotkin, 1981] and its "big-step" version [Kahn, 1987]. Of the many mature and flexible choices that can be made, we pick here *relational specifications* and their direct encoding as theories in restricted logics. This choice allows us to transparently encode operational semantics as well as a range of other notions including, most notably, typing. Another consequence of our choice is that our specification language will, in fact, be a specification *logic*. More specifically, it will turn out to be a simple, well understood logic that can be interpreted as a logic programming language in the style of λProlog [Nadathur and Miller, 1988].

After one has picked a language for writing specifications, there is still a choice to be made about a language for reasoning about them. The choices of these two languages are often related. If one has selected a specification language relying on, say, process calculus, then a reasoning language that exploits bisimulations and congruences would be a natural choice: see, for example, [Sangiorgi, 1994]. If one chooses abstract machines for specifications

A. Gacek and D. Miller
INRIA Saclay - Île-de-France & LIX/École Polytechnique, Palaiseau, France
E-mail: gacek at lix.polytechnique.fr, dale.miller at inria.fr

G. Nadathur
Department of Computer Science and Engineering, University of Minnesota
4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455 USA
E-mail: gopalan at cs.umn.edu

then inductive definitions are a natural choice for a reasoning language. In this paper, our reasoning language will be a *logic* that contains standard but powerful mechanisms for induction and co-induction as well as the ∇-quantifier (both in formulas and in *definitions*) [Miller and Tiu, 2005, Gacek et al., 2009]. Our choice of reasoning logic has a number of appealing aspects, chief among them being that it is powerful enough to capture within it many other reasoning techniques such as bisimulation and inductive definitions.

The approach we shall describe in this paper is thus characterized by the use of *two* logics, one for specifying computations and the other for reasoning about these logic specifications. We pick both of logics to be intuitionistic here but other choices are also sensible: for example, McDowell and Miller [2002] used a linear logic as a specification language in order to provide declarative specifications for a programming language with state. The logical symbols of these two logics will be separated in our treatment: in fact, provability in the specification logic will be an inductively defined predicate of the reasoning logic. Although we distinguish the logics in this way, the term structures that they use will be identical: in particular, the construction of terms in both logics will use the same application and abstraction operations. As a result, term equality in the reasoning logic will immediately reflect term equality in the specification logic.

In many commonly used approaches, it is problematic to treat specification-language abstraction through reasoning-logic abstraction. The reasoning logic often involves function types that contain recursive functions; this is the case, for example, in Coq and Isabelle/HOL. If function abstraction at the two levels are identified, function types in the specification language would also have to contain recursively defined functions. Since the specification language is intended to treat syntactic expressions and not general functions, this raises issues about the adequate representation of syntax: see, for example, the discussion about "exotic terms" in [Despeyroux et al., 1995]. In the setting that we shall soon unfold, we get around such problems by making function types in both the specification and the reasoning logic weak: while term equality will still be governed by the rules for $\alpha\beta\eta$-conversion, this will happen within the simply typed λ-calculus that does not include stronger principles such as recursion. To recover the lost strength, we will use inductively and co-inductively defined *predicates* for reasoning about computations. However, at the predicate level, the specification logic and the reasoning logic will be strictly separated. In summary, function types in both logics will be weak and will be used exclusively to represent syntax that may contain bindings. Following Miller [2000], we shall call this style of encoding data with bindings the λ-*tree* approach to abstract syntax.

When we develop the two-level logic approach in detail, the formulas of the specification logic will become terms of the reasoning logic. This approach to encoding an object logic within a second logic should be contrasted to the approach of provability logic (see, for example, [Smorynski, 2004]) where natural numbers are used to denote syntactic objects of an object-logic and primitive recursive functions are used to parse and manipulate those objects-cum-natural-numbers. Our encoding is more direct: *both* terms *and* formulas of the specification logic are represented by terms in the reasoning logic, with simple types being used to separate terms from formulas. Moreover, the presence of binding in the terms of the reasoning logic makes it possible to represent quantified formulas in the specification logic in an immediate and natural manner.

There are a number of advantages to the two-level logic approach to reasoning and the particular realization of it that we discuss in this paper. First, because of the term structures used in the reasoning logic, only mild encoding techniques are needed to embed the specification logic in it: for example, specification-level term equality is directly captured, quantification in the specification logic is treated by using λ-abstraction to bind the quantified variable, and the instantiation of quantifiers is realized through β-conversion. Second, since specifications are written in a logic and since such a logic typically has meta-theoretic properties (such as cut-admissibility) that can be formalized in the reasoning logic, powerful techniques become available for reasoning about descriptions presented in the specification logic. Third, as a series of examples illustrates, this two-level logic approach can result in natural, readable, and completely formal proofs of well-known theorems about computational systems. Finally, when one moves to implementing theorem provers based on this architecture, only one notion of binding, variable, term equality, substitution, and unification needs to be treated for both logics.

In the next section we describe the aspects of the reasoning logic $\mathcal{G}$ that we shall use in this paper. Section 3 presents the specification language hH^2 and shows how cut-free sequent calculus provability for it can be given an adequate encoding in $\mathcal{G}$. Section 4 describes briefly the structure of a theorem-prover called Abella that can be used to interactively construct sequent calculus proofs in $\mathcal{G}$. This description is then exploited in Section 5 to present examples of the use of the two-level logic approach. Section 6 describes related work and Section 7 concludes with an indication of some future directions.

2 The Reasoning Logic

The logic $\mathcal{G}$ [Gacek et al., 2009] is an extension of an intuitionistic and predicative subset of Church's Simple Theory of Types [Church, 1940]. 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 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. If $\mathcal{Q}$ is the abstraction operator or a quantifier, we will often use the shorthand $\mathcal{Q}x_1, \ldots, x_n.P$ for the expression $\mathcal{Q}x_1 \ldots \mathcal{Q}x_n.P$.

The usual interpretation of universally quantified formulas equates them with the set of all their instances. However, in (weak) logics meant for specifications over λ-tree syntax, an expression such as "$B(x)$ holds for all x" is often meant as a statement about the existence of a uniform argument for every instance rather than a more general assertion about the truth of some property for these instances. The ∇-quantifier [Miller and Tiu, 2005] is included in $\mathcal{G}$ to encode such *generic* judgments. Specifically, the language contains logical constants ∇_τ of type $(\tau \to o) \to o$ for each τ, not containing o, that is in a designated set of *nominal* types. As with the other quantifiers, $\nabla_\tau x.B$ abbreviates $\nabla_\tau \lambda x.B$.

Any adequate notion of derivation must associate with the ∇-quantifier at least the idea of generalizing on a unique name, but in such a way that $\nabla_\tau x.F$ is equivalent to $\nabla_\tau y.(F[y/x])$; the notation $F[t/x]$ denotes here and below the result of a capture-avoiding replacement of x by t in F. The $FO\lambda^{\Delta\nabla}$ logic [Miller and Tiu, 2005] realizes such a view within a sequent calculus presentation of intuitionistic provability by attaching a local signature to each formula in a sequent. In many reasoning situations, it is useful to strengthen the interpretation of ∇ by associating with it the ∇-*exchange rule* given by the equivalence $\nabla x.\nabla y.F \equiv \nabla y.\nabla x.F$ and the ∇-*strengthening rule* given by the equivalence $\nabla x.F \equiv F$, provided x is not free in F [Tiu, 2006]. The ∇-strengthening rule brings with it an ontological commitment to an arbitrary number of distinct objects at the types over which ∇-quantification is permitted. This is an acceptable commitment in many applications where ∇-quantification is typically used to represent object-level free variables which are themselves infinite in number. The addition of these rules renders both the length of a local signature and the order of names in it unimportant. These signatures can therefore be made implicit by distinguishing the variables bound by them as *nominal constants*. It is necessary to recognize, however, that the particular names used for such constants have significance only within a single formula and that, in this situation, the main impact is to ensure that each name refers to a distinct atomic object.

The treatment of the ∇-quantifier outlined above was introduced in the LG^ω system [Tiu, 2006] and has been adopted in $\mathcal{G}$. Specifically, an infinite collection of nominal constants is assumed for each type at which ∇-quantification is permitted. The set of all nominal constants is denoted by $\mathcal{C}$. These constants are distinct from (eigen)variables and the usual, non-nominal constants that we denote by $\mathcal{K}$. We define the *support* of a term (or formula) t, written $\text{supp}(t)$, 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:

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

Given two formulas B and B', we write $B \approx B'$ to denote the fact that there is a permutation π such that B λ-converts to $\pi.B'$. It is easy to see that $\approx$ is an equivalence relation. Following the earlier discussion, $\mathcal{G}$ is designed to preserve provability of sequents with respect to replacement of formulas under this relation.

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 $\Sigma : \Gamma \longrightarrow C$ where Γ is a set of formulas, C is a formula 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, h is an appropriately typed variable not occurring in Σ, $\bar{c}$ is a listing of the variables in $\text{supp}(B)$, and $h\ \bar{c}$ represents the application of h to these constants; raising, a technique introduced in [Miller, 1992], is used here to encode the dependency of the quantified variable on $\text{supp}(B)$. 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 rule is a well-formed term of type τ constructed from the variables in Σ and the constants in $\mathcal{K} \cup \mathcal{C}$. Finally, we note that the *id* rule gives expression to the richer notion of equality between formulas.

The notion of *substitution* plays an important role in defining the remaining rules of the logic. As usual, we identify a substitution θ as a type-preserving mapping from variables to terms such that the set $\{x \mid x\theta \neq x\}$,

$$\frac{B \approx B'}{\Sigma : \Gamma, B \longrightarrow B'} \ id \qquad\qquad \frac{\Sigma : \Gamma \longrightarrow B \quad \Sigma : B, \Delta \longrightarrow C}{\Sigma : \Gamma, \Delta \longrightarrow C} \ cut$$

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

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

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

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

the *domain* of θ, is finite. We denote the mapping of a variable x in the domain of a substitution to the term t by t/x. The usual application of a substitution $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$ to a term t requires paying attention to the scope of binders. In the presence of the λ-conversion rules, such an application, that we write as $t[\theta]$, is given precisely by the term $((\lambda x_1 \ldots \lambda x_n.t) \ t_1 \ \ldots \ t_n)$. In $\mathcal{G}$, we also have to pay attention to the fact that a substitution that is determined in the context of one formula may have to be applied to another formula; in this case, we must be careful not to confuse the scopes of nominal constants. Specifically, letting π be a permutation of nominal constants such that $\pi.c$ does not appear in the range of θ for any $c \in \mathrm{supp}(B)$, the *nominal capture avoiding application* of the substitution θ to the formula B is written as $B[\![\theta]\!]$ and is defined to be $(\pi.B)[\theta]$. This definition is ambiguous since many permutations can be chosen for π but the ambiguity is harmless since the result under all acceptable choices will be equivalent under $\approx$, the intended notion of equality for formulas.

The logic $\mathcal{G}$ supports the possibility of recursively defining atomic judgments. This allows specifications to be directly embedded in the logic. For example, list membership can be defined by the following two clauses for *member*:

$$\forall x, \ell. \ member \ x \ (x :: \ell) \triangleq \top \qquad\qquad \forall x, y, \ell. \ member \ x \ (y :: \ell) \triangleq member \ x \ \ell$$

The part of the clause to the left of $\triangleq$ is called the *head* while the part to the right is called the *body*. The intuitive reading of a single clause is that if the body is true then the head is true. Moreover, the reading of the complete set of clauses for a given predicate, such as *member*, is that the predicate holds for some arguments just in the case that the predicate with these arguments matches the head of one of the clauses and the corresponding instance of the body of that clause is true.

As seen in the example above, the head of a clause can use patterns to characterize the structure of arguments. We also allow ∇-quantification to be used in the head to constrain the structure of terms relative to nominal constants. For instance, the clause $(\nabla z.name \ z) \triangleq \top$ defines a predicate *name* which holds only on nominal constants. When a clause has both $\forall$ and ∇ quantification, the order of these quantifiers allows us to further restrict the structure of terms. For example, the clause $\forall x.(\nabla z.fresh \ z \ x) \triangleq \top$ defines a predicate *fresh* which holds only when its first argument is a nominal constant which does not occur in its second argument. This idea is particularly useful in recursive definitions such as the following definition of *cntx* which recognizes well-formed typing contexts:

$$cntx \ nil \triangleq \top \qquad\qquad \forall \alpha, \ell.(\nabla x.cntx \ (\langle x, \alpha \rangle :: \ell)) \triangleq cntx \ \ell$$

These clauses say that *cntx* L holds if and only if L is a list of pairs of the form $\langle x, \alpha \rangle$ in which x is a nominal constant that does not appear elsewhere in the list.

Formally, definitions consist of a finite set of clauses of the form $\forall \bar{x}.(\nabla \bar{z}.p \ \bar{t}) \triangleq B$ where $p \ \bar{t}$ and B are formulas, neither of which contain any nominal constants. Moreover, the free variables of $\bar{t}$ must be among $\bar{z}, \bar{x}$ and the free variables of B must be among $\bar{x}$. The logic $\mathcal{G}$ is parameterized by the set of clauses, called $\mathcal{D}$, chosen for a particular reasoning task. Given a definitional clause $\forall \bar{x}.(\nabla \bar{z}.p \ \bar{t}) \triangleq B$ and a substitution σ such that the list $\bar{z}\sigma$ contains only distinct nominal constants which do not appear in $\mathrm{supp}(\bar{x}\sigma)$ and such that the free variables of $B[\sigma]$ are a subset of the free variables of $(p \ \bar{t})[\sigma]$, we say that $(p \ \bar{t})[\sigma] \triangleq B[\sigma]$ is an *instance* of the original clause. Note that instances do not need to be ground and may contain other free variables. To treat definitions in our calculus, we add the rules *defL* and *defR* shown in Figure 2 for unfolding predicates on the left and the right of sequents using their defining clauses. The expression $\Sigma\theta$ in the *defL* rule, denoting the application of a substitution $\theta = \{r_1/x_1, \ldots, r_n/x_n\}$ to the signature Σ, is defined to be the result of removing from Σ the variables $\{x_1, \ldots, x_n\}$ and then adding every variable that is free in any term in $\{r_1, \ldots, r_n\}$. This rule also uses the nominal capture avoiding application of a substitution to a set of formulas that is defined in the obvious way: $\Gamma[\![\theta]\!] = \{B[\![\theta]\!] \mid B \in \Gamma\}$.

$$\frac{\Sigma : \Gamma \longrightarrow B}{\Sigma : \Gamma \longrightarrow p\ \bar{t}}\ def\mathcal{R},\ p\ \bar{t} \triangleq B \text{ an instance from } \mathcal{D}$$

$$\frac{\left\{\Sigma\theta : \Gamma[\![\theta]\!], B \longrightarrow C[\![\theta]\!] \mid \text{for all } \theta \text{ and all instances } (p\ \bar{t})[\![\theta]\!] \triangleq B \text{ from } \mathcal{D}\right\}}{\Sigma : \Gamma, p\ \bar{t} \longrightarrow C}\ def\mathcal{L}$$

Fig. 2 Definition rules

In the $def\mathcal{L}$ rule we consider all possible substitutions which allow an atomic formula to match the head of a clause in $\mathcal{D}$. Note that these substitutions are intended to affect the eigenvariables Σ. For example, consider applying the $def\mathcal{L}$ rule to the sequent

$$\Sigma, x, \ell : \Gamma, member\ x\ \ell \longrightarrow C$$

assuming the clauses shown earlier for *member*. Two of the upper sequents for such an application will be the following:

$$\Sigma, x, \ell' : \Gamma[\![x :: \ell'/\ell]\!], \top \longrightarrow C[\![x :: \ell'/\ell]\!]$$
$$\Sigma, x, y, \ell' : \Gamma[\![y :: \ell'/\ell]\!], member\ x\ \ell' \longrightarrow C[\![y :: \ell'/\ell]\!]$$

The first of these results from the eigenvariable ℓ being replaced by $x :: \ell'$ for some new eigenvariable ℓ' and the second corresponds to ℓ being replaced by $y :: \ell'$ where y and ℓ' are new eigenvariables. Note also that these are not the only upper sequents for the described rule: there will, in fact, be infinitely many other upper sequents, obtained by choosing more specific substitutions for the variables in Σ, x, ℓ.

The $def\mathcal{L}$ rule may have no premises. This happens if there are no substitutions under which an atom in the left of a sequent matches the head of a clause in $\mathcal{D}$, something that would be the case if, for example, *member x nil* appeared there. In this case, the rule provides an immediate proof of its conclusion. At the other extreme, there may be an infinite number of substitutions which yield relevant instances as we have just seen. Having an infinite set of premises is an obstacle to the effective application of the rule. However, the following fact about $\mathcal{G}$ helps overcome this difficulty in practice: the provability of $\Sigma : \Gamma \longrightarrow C$ implies the provability of $\Sigma\theta : \Gamma[\![\theta]\!] \longrightarrow C[\![\theta]\!]$ for any θ. Thus, the set of premises to be considered can be limited if we can identify a set of most general upper sequents from which all other upper sequents can be derived by applying a nominal capture-avoiding substitution. Looking back at the example of $\Sigma, x, \ell : \Gamma, member\ x\ \ell \longrightarrow C$, the two upper sequents that we have presented explicitly constitute a most general set of upper sequents for the application of $def\mathcal{L}$ in this case. In practice, such most general upper sequents are almost always computable and finite [Gacek et al., 2009]. We do not discuss these aspects which are important to implementations any further here, but we will use the general observations to limit consideration in particular examples to finite sets of most general upper sequents.

Identifying what constitutes a most general upper sequent for the $def\mathcal{L}$ rule may require some thought in the case of definitions with ∇-quantification in the head. Consider, for example, the derivation of the sequent $x, y, z : fresh\ x\ y \longrightarrow q\ x\ y\ z$ using the $def\mathcal{L}$ rule, assuming that *fresh* is defined by $\forall y.(\nabla z.fresh\ z\ y) \triangleq \top$ and q is some predicate. The following two sequents in which a is a nominal constant would be upper sequents in this case:

$$y, z : \top \longrightarrow q\ a\ y\ z \qquad\qquad y, z' : \top \longrightarrow q\ a\ y\ (z'\ a)$$

The second sequent here is strictly more general than the first: we can obtain the first from the second via the substitution $\{(\lambda x.z)/z'\}$ while there is no nominal capture-avoiding substitution which yields the second from the first. In fact, the second sequent constitutes a complete set of most general upper sequents for the use of $def\mathcal{L}$ in this case. Intuitively, in order to obtain such a most general sequent, the eigenvariables in the lower sequent must be raised over the nominal constants introduced by the definition that is used in conjunction with the $def\mathcal{L}$ rule. Notice also that the constraints expressed by the quantification in the head of a clause may necessitate the "pruning" of some of such raising substitutions. For example, while y may be replaced initially by $y'\ a$ in the sequent $x, y, z : fresh\ x\ y \longrightarrow q\ x\ y\ z$, the need to match the resulting atom $fresh\ x\ (y'a)$ with the instance $fresh\ a\ y$ of the head of the clause under the proviso that y cannot depend on a will result in y' being substituted for by $\lambda u.y$.

The meaning of the set of clauses for a predicate is given by any one of the possible fixed-points that can be associated with the clauses. While the $def\mathcal{L}$ and $def\mathcal{R}$ rules do not discriminate between the fixed points, $\mathcal{G}$ allows for a refinement that selects the least or the greatest fixed point, based on an inductive or co-inductive reading of the clauses for a given predicate. More precisely an *inductive* clause is denoted by $\overset{\mu}{=}$ in place of $\triangleq$ while a *co-inductive* clause is denoted by $\overset{\nu}{=}$ in place of $\triangleq$. We require that the clauses for a given predicate be uniformly annotated to be inductive, co-inductive or neither. The $def\mathcal{L}$ and $def\mathcal{R}$ rules may be used with clauses

$$\frac{\{\bar{x}_i : B_i[S/p] \longrightarrow \nabla \bar{z}_i.S\ \bar{t}_i\} \quad \Sigma : \Gamma, S\ \bar{t} \longrightarrow C}{\Sigma : \Gamma, p\ \bar{t} \longrightarrow C}\ \mathcal{IL}$$

provided p is defined by the clauses $\{\forall \bar{x}_i.(\nabla \bar{z}_i.p\ \bar{t}_i) \overset{\mu}{=} B_i\}$ and
S is a term with no nominal constants and of the same type as p.

Fig. 3 The induction rule

in any of these forms. Predicates that are inductively defined admit additionally the induction rule $\mathcal{IL}$ shown in
Figure 3. In a proof search setting, the term corresponding to the schema variable S in this rule functions like
the induction hypothesis and is accordingly called the invariant of the induction. Note that each clause results in
an additional upper sequent for this rule which requires that clause to preserve the induction hypothesis. There
is also a co-induction rule in the logic $\mathcal{G}$ though it does not have a natural presentation with the clause-based
treatment of definitions used in this paper [Gacek et al., 2009].

The interpretation of definitions as fixed-points and the possibility of reading individual clauses inductively
or co-inductively is sensible only if such clauses satisfy suitable stratification conditions. For example, a clause
such as $a \triangleq (a \supset \bot)$, in which a predicate has a negative dependency on itself should be forbidden. In this paper,
we shall rely on a simple method for ensuring stratification that is due to Tiu and Momigliano [2009]. This
method uses the idea of associating with each predicate p a natural number, $\text{lvl}(p)$, that is called its *level*. This
measure is then extended to formulas as follows: $\text{lvl}(\top) = \text{lvl}(\bot) = 0$; $\text{lvl}(p\ \bar{t}) = \text{lvl}(p)$; $\text{lvl}(B \wedge C) = \text{lvl}(B \vee C) =$
$\max(\text{lvl}(B), \text{lvl}(C))$; $\text{lvl}(\mathcal{Q}x.B) = \text{lvl}(B)$ where $\mathcal{Q}$ is $\nabla, \forall$ or $\exists$; and $\text{lvl}(B \supset C) = \max(\text{lvl}(B) + 1, \text{lvl}(C))$. In this
context, we consider a definition to be stratified if we can assign levels to predicates in such a way that for any
clause for p with body B in the definition it is the case that $\text{lvl}(B[\lambda \bar{y}.\top/p]) < \text{lvl}(p)$. The logic $\mathcal{G}$ has been shown
to be consistent under this constraint [Gacek et al., 2009].

3 The Two-level Logic Approach to Reasoning

The logic $\mathcal{G}$ has significant expressive power, being able to treat λ-tree syntax directly and to support inductive
and co-inductive reasoning. As such, it can already be used for constructing specifications of computations and
then for reasoning about them. However, we will not use it in this immediate fashion, choosing instead to embed
a specification logic into it and then using the specification logic to encode the systems that we wish to formalize.
The particular specification logic that we will use in this scheme is the intuitionistic theory of second-order
hereditary Harrop formulas that we call hH^2. This logic provides a convenient vehicle for formulating structural,
rule-based characterizations of a variety of properties such as evaluation and type assignment. Informally, one may
think of hH^2 as an extension of a simple Prolog-like logic with support for representing and manipulating λ-tree
syntax [Miller, 2000]. An especially useful feature of encodings in hH^2 is that derivations that are constructed
in hH^2 based on such encodings end up reflecting the structure of computations in the object systems.[1] The
embedding of hH^2 within $\mathcal{G}$ that we describe transparently reflects derivations in hH^2 and hence gives us the
ability to formalize a process of reasoning directly about computations. Moreover, by proving meta-theoretic
properties of hH^2 within $\mathcal{G}$, we obtain a collection of general logical principles that can be applied in arguments
about computations in any of the encoded object systems.

This section elaborates the specific two-level logic approach outlined above. Section 3.1 presents the logic
hH^2 and Section 3.2 describes an example specification in hH^2. Finally, Section 3.3 provides an embedding of
hH^2 into $\mathcal{G}$ and shows how some of the meta-theory of hH^2 can be formalized through this embedding.

3.1 The Specification Logic

Formulas in hH^2 are of two kinds. The *goal formulas* are determined by the grammar

$$G = \top \mid A \mid G \wedge G \mid A \supset G \mid \forall_\tau x.G,$$

where A denotes atomic formulas whose arguments are monomorphically typed λ-terms and τ ranges over types
that do not themselves contain the type of formulas. *Definite clauses* are formulas of the form $\forall x_1 \dots \forall x_n.(G_1 \supset
\cdots \supset G_m \supset A)$, where n and m may both be zero and where quantification is, again, over variables whose
types do not contain that of formulas. This restricted set of formulas is "second-order" in that to the left of an

---

[1] Since hH^2 is a subset of the λProlog language [Nadathur and Miller, 1988], these specifications can also be compiled
and executed, using an implementation of λProlog such as Teyjus [Nadathur and Mitchell, 1999, Gacek et al., 2008].

$$\frac{}{\Sigma : \Delta \vdash \top} \ \text{TRUE} \qquad \frac{\Sigma : \Delta \vdash G_1 \quad \Sigma : \Delta \vdash G_2}{\Sigma : \Delta \vdash G_1 \wedge G_2} \ \text{AND}$$

$$\frac{\Sigma : \Delta, A \vdash G}{\Sigma : \Delta \vdash A \supset G} \ \text{AUGMENT} \qquad \frac{\Sigma \cup \{c : \tau\} : \Delta \vdash G[c/x]}{\Sigma : \Delta \vdash \forall_\tau x.G} \ \text{GENERIC}$$

$$\frac{\Sigma : \Delta \vdash G_1[\bar{t}/\bar{x}] \quad \cdots \quad \Sigma : \Delta \vdash G_n[\bar{t}/\bar{x}]}{\Sigma : \Delta \vdash A} \ \text{BACKCHAIN}$$
$$\text{where } \forall \bar{x}.(G_1 \supset \ldots \supset G_n \supset A') \in \Delta \text{ and } A'[\bar{t}/\bar{x}] = A$$

Fig. 4 Derivation rules for the hH^2 logic

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

Fig. 5 Rules for relating a λ-term to a simple type

$$\forall m, n, a, b.(of \ m \ (arr \ a \ b) \supset of \ n \ a \supset of \ (app \ m \ n) \ b)$$
$$\forall r, a, b.(\forall x.(of \ x \ a \supset of \ (r \ x) \ b) \supset of \ (lam \ a \ r) \ (arr \ a \ b))$$

Fig. 6 Second-order hereditary Harrop formulas (hH^2) encoding simply typing

implication in a definite formula one finds goal formulas and to the left of an implication in a goal formula, one finds only atomic formulas. These definite clauses, in fact, coincide with the second-order fragment of higher-order hereditary Harrop formulas [Miller et al., 1991].

Provability in hH^2 is formalized by a sequent calculus proof system in which sequents are of the form $\Sigma : \Delta \vdash G$, where Δ is a list of definite clauses, G is a goal formula, and Σ is a set of eigenvariables. The inference rules for hH^2 are presented in Figure 4: an immediate consequence of the results in [Miller et al., 1991] is that this proof system is complete for the intuitionistic theory of hH^2. The GENERIC rule introduces an eigenvariable when read in a proof search direction, and there is an associated freshness side-condition: c must not already be in Σ. In the BACKCHAIN rule, for each term $t_i \in \bar{t}$ we enforce the type constraint that $\Sigma \vdash t_i : \tau_i$ holds where τ_i is the type of the quantified variable x_i. An important property to note about these rules is that if we use them to search for a proof of the sequent $\Sigma : \Delta \vdash G$, then all the intermediate sequents that we will encounter will have the form $\Sigma' : \Delta, \mathcal{L} \vdash G'$ for some Σ' with $\Sigma \subseteq \Sigma'$, some goal formula G', and some list of atomic formulas $\mathcal{L}$. Thus the initial context Δ is *global*: changes occur only in the list of atoms on the left and the goal formula on the right. In presenting sequents, we will elide the signature when it is inessential to the discussion.

3.2 An Example

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 5. Object-level simple types and untyped λ-terms can be encoded in a simply typed (meta-level) λ-calculus as follows. We assume the types ty and tm for representing object-level simple types and untyped λ-terms. The simple types are built from the two constructors $i : ty$ and $arr : ty \rightarrow ty \rightarrow ty$ and terms are built using the two constructors $app : tm \rightarrow tm \rightarrow tm$ and $lam : ty \rightarrow (tm \rightarrow tm) \rightarrow tm$. Here, the constructor lam takes two arguments: one for the type of the variable being abstracted and the other for the actual abstraction. Note, in particular, that the bound variable in an object-level abstraction will be encoded by an explicit, specification logic abstraction: thus, the object-level term $(\lambda f : i \rightarrow i.(\lambda x : i.(f \ x)))$ will be represented by the specification logic term $lam \ (arr \ i \ i) \ (\lambda f.lam \ i \ (\lambda x.app \ f \ x))$.

Given this encoding of the untyped λ-calculus and simple types, the inference rules of Figure 5 can be specified by the hH^2 definite clauses in Figure 6 involving the binary predicate of. Note that this specification in hH^2 does not maintain an explicit 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 freshness side-condition of the GENERIC rule in hH^2.

The properties that we prove in $\mathcal{G}$ will eventually be about specification logic judgments. To reflect such properties into related properties about the object system, we will establish two results about our encodings: that

there exists a bijection, ϕ, between expressions of the object system and their specification logic representations and that this bijection preserves the judgments of interest. These properties constitute what is referred to as the *adequacy* of an encoding. We illustrate below the structure of adequacy arguments in the context of our encoding of the simply typed λ-calculus.

We start by defining the mapping ϕ from object-level simple types to hH^2 terms of type tp and from object-level untyped λ-terms to hH^2 terms of type tm.

$$\phi(i) = i \qquad \phi(a \to b) = \text{arr } \phi(a) \ \phi(b)$$

$$\phi(x) = x \qquad \phi(m \ n) = \text{app } \phi(m) \ \phi(n) \qquad \phi(\lambda x : a.r) = \text{lam } \phi(a) \ (\lambda x.\phi(r))$$

In the first case for the mapping of terms, x is used to denote both an object-level and a corresponding specification logic variable. Note that under this mapping bound object-level variables will correspond to variables bound by λ's in the specification logic, and object-level free variables will correspond (eventually) to eigenvariables in the specification logic. The mapping ϕ is bijective so long as we only allow eigenvariables at type tm. In later arguments, we will need the fact that bound variables in both the object system and the specification logic can be renamed so that, for example, rules with freshness side-conditions can be correctly applied. It is important that such object-level and specification logic renamings are carried out in a consistent fashion. A more general form of this property is that ϕ is compositional with respect to substitution which can be stated as follows:

$$\phi(r[x := n]) = \phi(r)[\phi(n)/x]$$

Notice that we have used object-level substitution on the left and specification logic substitution on the right. This equality can be proved by induction on the structure of r.

We now want to define a mapping from object-level derivations of typing judgments to derivations in hH^2 of sequents of the form $\Delta, \mathcal{L} \vdash of \ e \ t$ where Δ is a list of the clauses from Figure 6 and $\mathcal{L}$ is a list of atomic formulas of the form $of \ x_1 \ a_1, \ldots, of \ x_k \ a_k$ where each x_i is a unique eigenvariable. Towards this end, we first define the following bijection between a list of typing assumptions Γ from the simply typed λ-calculus and a list of atomic formulas of the form described for $\mathcal{L}$.

$$\phi(x_1 : a_1, \ldots, x_k : a_k) = of \ x_1 \ \phi(a_1), \ldots, of \ x_k \ \phi(a_k)$$

Using this, we can define the mapping for the (atomic) typing derivation for variables as follows:

$$\phi \left(\frac{}{\Gamma \vdash x_i : a_i} \right) = \frac{}{\Delta, \phi(\Gamma) \vdash of \ x_i \ \phi(a_i)}$$

If the object system typing derivation to which ϕ is applied is correct, then it must be that $x_i : a_i \in \Gamma$. Thus the right-hand side is an instance of the BACKCHAIN rule on the clause $of \ x_i \ \phi(a_i)$ which is in $\phi(\Gamma)$.

Derivations in the object system that have the typing rule for applications at the end are mapped in the expected way:

$$\phi \left(\frac{\overset{\vdots}{\Gamma \vdash m : a \to b} \quad \overset{\vdots}{\Gamma \vdash n : a}}{\Gamma \vdash m \ n : b} \right) = \frac{\phi \left(\overset{\vdots}{\Gamma \vdash m : a \to b} \right) \quad \phi \left(\overset{\vdots}{\Gamma \vdash n : a} \right)}{\Delta, \phi(\Gamma) \vdash of \ \phi(m \ n) \ \phi(b)}$$

$$= \frac{\Delta, \phi(\Gamma) \vdash of \ \phi(m) \ (\text{arr } \phi(a) \ \phi(b)) \quad \Delta, \phi(\Gamma) \vdash of \ \phi(n) \ \phi(a)}{\Delta, \phi(\Gamma) \vdash of \ (\text{app } \phi(m) \ \phi(n)) \ \phi(b)}$$

This is clearly a well-formed instance of the BACKCHAIN rule using the clause for typing applications in Δ.

In mapping derivations in the object system that have the rule for typing abstractions at the end, we need to be mindful of the variable naming restriction and how this is realized in the specification logic. Suppose we want to define the following mapping:

$$\phi \left(\frac{\overset{\vdots}{\Gamma, x : a \vdash r : b}}{\Gamma \vdash (\lambda x : a.r) : a \to b} \right)$$

$$seq_{(s\ N)}\ L\ \top\ \overset{\mu}{=}\ \top \qquad\qquad\qquad\qquad nat\ z\ \overset{\mu}{=}\ \top$$

$$seq_{(s\ N)}\ L\ (B \wedge C)\ \overset{\mu}{=}\ seq_N\ L\ B \wedge seq_N\ L\ C \qquad nat\ (s\ N)\ \overset{\mu}{=}\ nat\ N$$

$$seq_{(s\ N)}\ L\ (A \supset B)\ \overset{\mu}{=}\ seq_N\ (A :: L)\ B$$

$$seq_{(s\ N)}\ L\ (\forall B)\ \overset{\mu}{=}\ \nabla x.seq_N\ L\ (B\ x) \qquad member\ B\ (B :: L)\ \overset{\mu}{=}\ \top$$

$$seq_{(s\ N)}\ L\ \langle A \rangle\ \overset{\mu}{=}\ member\ A\ L \qquad\qquad member\ B\ (C :: L)\ \overset{\mu}{=}\ member\ B\ L$$

$$seq_{(s\ N)}\ L\ \langle A \rangle\ \overset{\mu}{=}\ \exists b.prog\ A\ b \wedge seq_N\ L\ b$$

Fig. 7 Second-order hereditary Harrop logic in $\mathcal{G}$

Here we assume that x does not appear in Γ so that the naming restriction is satisfied. We map this to the following specification logic derivation:

$$\cfrac{\cfrac{\cfrac{\overset{\vdots}{\phi(:)}}{\Delta, \phi(\Gamma), of\ x\ \phi(a) \vdash of\ \phi(r)\ \phi(b)}}{\Delta, \phi(\Gamma) \vdash of\ x\ \phi(a) \supset of\ \phi(r)\ \phi(b)}\ AUGMENT}{\cfrac{\Delta, \phi(\Gamma) \vdash \forall x.(of\ x\ \phi(a) \supset of\ ((\lambda x.\phi(r))\ x)\ \phi(b))}{\Delta, \phi(\Gamma) \vdash of\ (lam\ \phi(a)\ (\lambda x.\phi(r)))\ (arr\ \phi(a)\ \phi(b))}\ GENERIC}\ BACKCHAIN$$

In the GENERIC rule we overload notation to let x be the eigenvariable we select. Since it does not appear in Γ it will not appear in $\phi(\Gamma)$, and thus the freshness side-condition on the GENERIC rule is satisfied. In fact, the naming restriction in the object logic matches up with the freshness side-condition in the specification logic exactly as needed.

The inverse of the ϕ mapping for typing judgments can be defined in the expected way, and it can be seen from this that ϕ is a bijection. Therefore our encoding of the typing relation is adequate.

3.3 Encoding Specification Logic Provability in $\mathcal{G}$

The definitional clauses in Figure 7 encode hH^2 provability in $\mathcal{G}$; this encoding is based on ideas from [McDowell and Miller, 2002]. Formulas in hH^2 are represented in this setting by terms of type *form* and we reuse the symbols $\wedge$, $\vee$, $\supset$, $\top$, and $\forall$ for constants involving this type in $\mathcal{G}$; we assume that the context will make clear which reading of these symbols is meant. The constructor $\langle \cdot \rangle$ is used to inject atomic formulas in hH^2 into specially marked expressions of type *form* in $\mathcal{G}$. As we have seen earlier, provability in hH^2 is about deriving sequents of the form $\Delta, \mathcal{L} \vdash G$, where Δ is a fixed list of definite clauses and $\mathcal{L}$ is a varying list of atomic formulas. Our encoding uses the $\mathcal{G}$ predicate *prog* to represent the definite clauses in Δ. In particular, the definite clause $\forall \bar{x}.[G_1 \supset \cdots \supset G_n \supset A]$ is encoded as the clause $\forall \bar{x}.prog\ A\ (G_1 \wedge \cdots \wedge G_n) \triangleq \top$ and particular specifications written in hH^2 will be reflected into $\mathcal{G}$ through corresponding collections of *prog* clauses. Sequents in hH^2 are represented in $\mathcal{G}$ by means of atomic formulas of the form $seq_N\ L\ G$ where L is a list encoding the atomic formulas in $\mathcal{L}$ and where G encodes the goal formula. The provability of such sequents in hH^2, given by the rules in Figure 4, leads to the clauses that define *seq* in Figure 7. The argument N that is written as a subscript in the expression $seq_N\ L\ G$ encodes (roughly) the height of the corresponding hH^2 derivation and is needed in formalizing proofs by induction on these heights. This argument has type *nt* that is endowed with two constructors: z of type *nt* and s of type $nt \rightarrow nt$.

A few remarks are appropriate pertaining to the encoding of hH^2 provability. First, note that proofs of universally quantified goal formulas in hH^2 are generic in nature. Thus, a natural way to encode the proof rule for the (specification-logic) universal quantifier is to use the ∇-quantifier, as is done in the clause defining *seq* for this case. Second, observe that in proving an implication, the atomic assumption is added, as would be expected, to the list that is the second argument of *seq*. Third, the last clause for *seq* can be seen to implement backchaining over a given hH^2 specification, stored as *prog* clauses. The matching of atomic judgments to heads of clauses is handled by the treatment of definitions in the logic $\mathcal{G}$; thus the last rule for *seq* simply performs this matching and makes a recursive call on the corresponding clause body. Finally, observe that the way the natural number (subscript) arguments are used in the *seq* clauses ensures a correct encoding of the fact that the premise sequents of a rule in hH^2 must be shorter than the derivation of the conclusion sequent.

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 heights of derivations may be needed in such arguments and this can be realized via natural number induction on N in $seq_N\ L\ P$, realized using induction over the clauses in Figure 7 defining the nat predicate. Notice also that 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.

3.3.1 Formalizing Properties of the Specification Logic

Since we have encoded the entire derivability relation of hH^2, we can prove general properties about it in $\mathcal{G}$ and then use these in reasoning about particular specifications. For example, the following formula, which is provable in $\mathcal{G}$, states that the judgment $seq_n\ \ell\ g$ is not affected by permuting, contracting, or weakening the context ℓ.

$$\forall n, \ell_1, \ell_2, g.(seq_n\ \ell_1\ g) \wedge (\forall e.member\ e\ \ell_1 \supset member\ e\ \ell_2) \supset (seq_n\ \ell_2\ g)$$

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

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

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

$$\forall n, m, \ell, a, g.(nat\ n \wedge seq_n\ \ell\ \langle a\rangle) \wedge (nat\ m \wedge seq_m\ (a :: \ell)\ g) \supset \exists p.(nat\ p \wedge seq_p\ \ell\ g),$$

which can be proved in $\mathcal{G}$. To demonstrate the usefulness of the instantiation and cut properties, we observe that using these together with our encoding of typing for the simply typed λ-calculus leads to an easy proof of the type substitution property, *i.e.*, if $\Gamma, x : a \vdash m : b$ and $\Gamma \vdash n : a$ then $\Gamma \vdash m[x := n] : b$.

3.3.2 Adequacy of the Encoding of the Specification Logic

We are eventually interested in lifting the results we prove about encodings to related results about the original object systems. In the two-level logic approach, adequacy proofs of this kind can be factored through an adequacy result for the encoding of the specification logic; in the present context, this corresponds to the adequacy of the encoding of hH^2 in $\mathcal{G}$ via the definition of seq and $prog$. One benefit of the two-level logic approach is that adequacy of the encoding of the specification logic needs to be established only once for all applications, provided this is properly parameterized by the embedding of specifications themselves via the $prog$ clauses. Thus, the important statement of adequacy for the combination of hH^2 and $\mathcal{G}$ is the following:

Theorem 1 *Let Δ be a list of closed definite clauses, $\mathcal{L}$ a list of atoms, G a goal formula, and Σ a set of eigenvariables containing at least the free variables of Δ, $\mathcal{L}$, and G. Suppose that all non-logical specification logic constants and types are represented by equivalent constants and types in $\mathcal{G}$ and let ψ denote the obvious mapping between formulas in hH^2 and terms in $\mathcal{G}$. Suppose also that specification logic $\forall$-quantification (eigenvariables) and reasoning logic ∇-quantification (nominal constants) are allowed only at inhabited types. Then $\Sigma : \Delta, \mathcal{L} \vdash G$ has a derivation in hH^2 if and only if $\exists n.nat\ n \wedge seq_n\ \psi(\mathcal{L})\ \psi(G)$ is provable in $\mathcal{G}$ with the clauses for nat, member, and seq as stated before and the clauses for prog as given by the prescribed encoding of Δ.*

The proof of this theorem is straightforward and its details are available in [Gacek, 2009b]. The only interesting point is the relevance of the condition that specification logic $\forall$-quantification and reasoning logic ∇-quantification are allowed only at inhabited types. This condition is needed because we have chosen to use a shallow encoding of the typing judgment of the specification logic. That is, rather than encoding an explicit typing judgment for specification logic terms, we have relied on the typing judgment of $\mathcal{G}$ to enforce the well-formedness of terms. Due to the lack of restrictions on the occurrences of nominal constants, the typing judgment in $\mathcal{G}$ is more permissive than the specification logic typing judgment. However, as the statement of the theorem indicates, this difference only manifests itself at uninhabited types. For inhabited types, the instantiation property of hH^2 can be used to

remove any "stray" nominal constants. A deeper encoding involving an explicit typing judgment would avoid this condition, but would also impose additional costs in terms of reasoning both about and through the encoding. In our experience, the shallow encoding has turned out to provide a good balance in practice.

The theorem above restricts the definitions of the predicates *nat*, *member*, *seq*, and *prog*, but makes no explicit reference to other predicates. Indeed, the definitions of other predicates have no affect on the adequacy of the encoding of the specification logic. Additionally, $\mathcal{G}$ may make use of additional constants and types which are unconnected to the constants and types used to represent the specification logic without affecting the adequacy of the encoding.

4 The Architecture of Abella

Abella is an interactive theorem prover for the logic $\mathcal{G}$ which incorporates the two-level logic approach to reasoning [Gacek, 2008, 2009a]. In this section we briefly describe the architecture of Abella. In particular, we illustrate how $\mathcal{G}$ and the two-level logic approach are presented to the user within this system and we introduce terminology and notation that are useful in the example applications that we consider in the next section.

4.1 Proof Construction, Tactics, and (Co)Induction

The high-level structure of Abella is similar to that of most other tactics-based theorem provers. At any time, the state of the prover is represented by a collection of subgoals, all of which need to be solved for the overall proof to succeed. The user applies a tactic to a subgoal in order to make progress towards a completed proof. If we think of a completed proof as a derivation for a sequent in $\mathcal{G}$, then the subgoals correspond to sequents whose derivations will complete the proof being sought. A tactic corresponds in this setting to a scheme for using the rules of $\mathcal{G}$ to produce new subgoals whose derivations can, in turn, be used to produce a derivation of the subgoal under consideration.

The tactics in Abella are designed to model natural proof steps. Some tactics serve to collect related proof rules under a single name. For example, Abella has a "case analysis" tactic which uses a rule such as $\vee\mathcal{L}$, $\wedge\mathcal{L}$, $\bot\mathcal{L}$, *def*$\mathcal{L}$, $\exists\mathcal{L}$, or $\nabla\mathcal{L}$, depending on the structure of the formula to which it is applied. Other tactics combine the use of many rules in tandem. For example, Abella has an "apply" tactic which takes a lemma or hypothesis of the form $\forall\bar{x}.H_1 \supset \ldots \supset H_n \supset C$ and hypotheses $H'_1,\ldots,H'_n$ and tries to find terms $\bar{t}$ such that for each $i \in \{1,\ldots,n\}$ it is the case that $H'_i \longrightarrow H_i[\bar{t}/\bar{x}]$ can be provided a proof using only the *id* rule. If successful, the tactic adds a new hypothesis $C[\bar{t}/\bar{x}]$.

Abella has treatments for induction and co-induction which simplify much of the work involved in formulating invariants and co-invariants. We will focus on the treatment of induction here: further details of the approach to induction and co-induction in Abella are available in [Gacek, 2009b]. Suppose we have the sequent

$$\Sigma : p\ \bar{t}, H_1, \ldots, H_n \longrightarrow C,$$

where p is inductively defined. The induction tactic can be applied to this sequent by designating $p\ \bar{t}$ as the the *induction formula*. The application of the tactic is based on the additional formula

$$\forall\Sigma.(p\ \bar{t})^* \supset H_1 \supset \ldots \supset H_n \supset C,$$

in which $\forall\Sigma$ denotes a list of universal quantifiers, one for each variable in Σ. This formula, which we call the *induction hypothesis* and denote by *IH*, has an occurrence in it of the induction formula that is annotated with *. The formula annotated in this way in the induction hypothesis can only be matched by another formula that has the same annotation. The induction tactic now transforms the original sequent into

$$\Sigma : \textit{IH}, (p\ \bar{t})^{@}, H_1, \ldots, H_n \longrightarrow C.$$

The atomic formula $p\ \bar{t}$ that has the annotation $^{@}$ here is treated as if the annotation is not present, with the exception that when it is unfolded using a *def*$\mathcal{L}$ rule any new atoms that are introduced that have p as their head symbol are annotated with *. These formulas that are annotated with * are treated just like the formula with the $^{@}$ annotation except that they are also eligible to be used with the induction hypothesis. Thus, viewed intuitively, the induction tactic simply generates an induction hypothesis that is usable when the induction formula is unfolded. This tactic can be seen as the special case of the use of the $\mathcal{IL}$ rule; a detailed justification is presented elsewhere [Gacek, 2009b].

4.2 Treatment of the Two-level Logic Approach to Reasoning

Abella incorporates the two-level logic approach to reasoning using the specification logic hH^2 and its encoding via *seq* and *prog*. Moreover, the actual details of the encoding are hidden from the user. As we have observed already, hH^2 is a subset of the λProlog language. Abella allows hH^2 specifications to be written in λProlog syntax, thereby permitting one to reason about computations based on the same descriptions that are used to prototype them. Following this approach also creates the feeling that one is reasoning directly about hH^2 derivations that reflect the encoded computations.

Abella uses specialized syntax to simplify the presentation of specification logic judgments. In particular, the judgment $\exists n.nat\ n \wedge seq_n\ L\ \langle A \rangle$ is presented as $\{L \vdash A\}$. Moreover, the list L is decomposed into a presentable format that matches the way hypotheses are typically written in an hH^2 judgment. For example, the judgment $\{H_1 :: H_2 :: L \vdash A\}$ is presented more suggestively as $\{L, H_2, H_1 \vdash A\}$. If the list ends in *nil* rather than a variable then we simply write $\{H_2, H_1 \vdash A\}$. If the entire list is *nil* then we elide even the turnstile, writing the judgment as $\{A\}$. Looking at the clauses in Figure 7, we see that any *seq* judgment in which the last argument is a non-atomic goal can be immediately and deterministically transformed into a collection of such judgments in which the last argument is an atomic goal. Thus the specialized $\{\cdot \vdash \cdot\}$ notation is the only representation of the specification logic that needs to be exposed to the user. For example, using the clauses from Figure 6 in Abella, case analysis on an assumption $\{of\ (lam\ A\ R)\ (arr\ A\ B)\}$ results directly in the new assumption $\{of\ c\ A \vdash of\ (R\ c)\ B\}$ where c is a nominal constant.

As we have observed in Section 3.3, hH^2 is a logic with notable meta-theoretic properties which can be formalized and established as theorems of $\mathcal{G}$. Combining such results with the apply tactic leads to an expanded collection of tactics within Abella which are geared to reasoning about hH^2 specifications. For example, given $\{L, A \vdash B\}$ and $\{L \vdash A\}$ the *cut* tactic allows one to derive $\{L \vdash B\}$. Similarly, given a hypothesis $\{L \vdash A\}$, a nominal constant v in that hypothesis, and a term t of the same type as v, the *inst* tactic allows one to derive $\{L[t/v] \vdash A[t/v]\}$. Also, a tactic is available for deriving from $\{L \vdash A\}$ the hypothesis $\{K \vdash A\}$ if the list L denotes a set that is a subset of the set denoted by the list K.

Finally, the treatment of induction described previously is extended to formulas of the form $\{L \vdash A\}$ by attaching annotations directly to such formulas. This treatment is justified by unfolding $\{L \vdash A\}$ to $\exists n.nat\ n \wedge seq_n\ L\ \langle A \rangle$, applying the $\exists \mathcal{L}$ and $\wedge \mathcal{L}$ rules, and using the induction tactic with *nat* n as the induction formula.

5 Examples

We now illustrate the two-level logic approach to reasoning through concrete examples. We start with a specification of evaluation and typing for the simply typed λ-calculus for which we prove some basic properties. We then consider extensions in two different directions. In one direction, we enrich the collection of terms to the language of PCF [Plotkin, 1977] and we demonstrate that the associated reasoning scales up smoothly. In the other direction, we retain the simple language but enhance the complexity of the properties we prove.

In the examples we present, we will omit the outermost universal quantifiers when we write specification formulas, using the convention that tokens given by capital letters denote variables that are implicitly universally quantified over the entire formula. We will also assume the availability of two special predicates: the binary infix predicate $=$ for each type that is defined by the clause $X = X \triangleq \top$ and, for each nominal type, the unary predicate *name* that is defined by the clause $(\nabla x.name\ x) \triangleq \top$. Finally, we will assume that the formula

$$\forall L, E.\nabla x.\ member\ (E\ x)\ L \supset \exists E'.\ E = \lambda y.E',$$

is derivable. This formula, which can be proved by a straightforward induction on the definition of *member*, states that if a list does not contain a nominal constant then no element of the list can contain that constant.

We will leave out many details of proofs in our presentation, restricting ourselves to indicating the general structure of the argument and to highlighting especially interesting applications of inference rules and the use of induction.

5.1 Type Preservation for the Simply Typed λ-Calculus

We recall the encoding of the simply typed λ-calculus in hH^2 that was presented in Section 3.2. We use *ty* and *tm* as the types for hH^2 terms that encode the types and terms of the (object) λ-calculus. The hH^2 constants $i : ty$ and $arr : ty \rightarrow ty \rightarrow ty$ are used to denote a base type and the arrow type; we assume for simplicity that there is only one base type in the object language. The hH^2 constants $app : tm \rightarrow tm \rightarrow tm$

$$eval \ (lam \ A \ R) \ (lam \ A \ R)$$
$$eval \ M \ (lam \ A \ R) \supset eval \ (R \ N) \ V \supset eval \ (app \ M \ N) \ V$$

$$of \ M \ (arr \ A \ B) \supset of \ N \ A \supset of \ (app \ M \ N) \ B$$
$$\forall x.(of \ x \ A \supset of \ (R \ x) \ B) \supset of \ (lam \ A \ R) \ (arr \ A \ B)$$

Fig. 8 Evaluation and typing in the simply typed λ-calculus

and $lam : ty \rightarrow (tm \rightarrow tm) \rightarrow tm$ are used to denote object-level applications and (typed) abstractions. In this context, call-by-name evaluation and (monomorphic) typing for the simply typed λ-calculus can be specified by the hH^2 formulas as shown in Figure 8.

Consider now proving that evaluation in the simply typed λ-calculus preserves typing. Stated in terms of the encoding in hH^2, this property can be expressed through the following formula in $\mathcal{G}$:

$$\forall E, V, A. \ \{eval \ E \ V\} \supset \{of \ E \ A\} \supset \{of \ V \ A\}. \tag{1}$$

We show below how a proof can be constructed in Abella of a sequent with only this formula on the right.

Using the right rules for the universal quantifier and implication, the starting goal can be reduced to the subgoal corresponding to the sequent

$$\{eval \ E \ V\}, \{of \ E \ A\} \longrightarrow \{of \ V \ A\}.$$

We can prove this sequent by induction on $\{eval \ E \ V\}$ using the rest of the sequent to generate the induction invariant. Let us abbreviate that induction hypothesis, namely, $[\forall E, V, A. \ \{eval \ E \ V\}^* \supset \{of \ E \ A\} \supset \{of \ V \ A\}]$ by IH. The resulting induction yields two sequents, one for each clause defining $eval$. The base case, namely,

$$IH, \{of \ (lam \ B \ R) \ A\} \longrightarrow \{of \ (lam \ B \ R) \ A\}$$

is trivial. The other case is given by the sequent

$$IH, \{eval \ M \ (lam \ B \ R)\}^*, \{eval \ (R \ N) \ V\}^*, \{of \ (app \ M \ N) \ A\} \longrightarrow \{of \ V \ A\}.$$

Applying case analysis to the typing judgment on the left yields the sequent

$$IH, \{eval \ M \ (lam \ B \ R)\}^*, \{eval \ (R \ N) \ V\}^*,$$
$$\{of \ M \ (arr \ C \ A)\}, \{of \ N \ C\} \longrightarrow \{of \ V \ A\}.$$

Applying the induction hypothesis to the evaluation and typing judgments on M yields the sequent

$$IH, \ldots, \{eval \ (R \ N) \ V\}^*, \{of \ N \ C\}, \{of \ (lam \ B \ R) \ (arr \ C \ A)\} \longrightarrow \{of \ V \ A\}.$$

Case analysis can be applied to the new typing judgment and this yields

$$IH, \ldots, \{eval \ (R \ N) \ V\}^*, \{of \ N \ B\}, \{of \ c \ B \vdash of \ (R \ c) \ A\} \longrightarrow \{of \ V \ A\}.$$

Notice that this analysis has forced $B = C$ and thus all instances of C have been replaced. In the last hypothesis of this sequent, c is a nominal constant so we can apply the instantiation property of hH^2 to obtain $\{of \ N \ B \vdash of \ (R \ N) \ A\}$. We can then use the cut property with the assumption $\{of \ N \ B\}$ to produce the following sequent.

$$IH, \ldots, \{eval \ (R \ N) \ V\}^*, \{of \ (R \ N) \ A\} \longrightarrow \{of \ V \ A\}.$$

Applying the induction hypothesis to the two assumptions displayed above completes this proof.

Proofs of properties such as the one above involve what is often called a "substitution lemma." In this case, assuming a conventional syntax representation, such a lemma would be stated as "if B has type α and the variable x and term t have the same type β, then $B[t/x]$ has type α." Such a lemma can be proved using an induction on the details of the construction of terms and their binding structure. Notice that in the proof above, this substitution lemma comes *for free*: it is a direct application of the cut-admissibility result for hH^2. Of course, the proof of cut-admissibility requires a detailed induction on the structure of hH^2 proofs. As this example illustrates, however, once cut-admissibility has been established, one should be able to get most substitution lemmas for free by using such meta-level properties of hH^2.

Our ultimate objective is to show the type preservation property for the simply typed λ-calculus. We obtain this result from the property stated in formula (1) by using the adequacy of our encodings. Suppose that e evaluates to v and that $\vdash e : a$ holds. Let Δ be the clauses in Figure 8. By the adequacy of these clauses, which can be proved as shown in Section 3, we know that $\Delta \vdash eval\ \phi(e)\ \phi(v)$ and $\Delta \vdash of\ \phi(e)\ \phi(a)$ must have derivations in hH^2. Then from the adequacy of the seq encoding of hH^2 into $\mathcal{G}$ we know that $\{eval\ \phi(e)\ \phi(v)\}$ and $\{of\ \phi(e)\ \phi(a)\}$ must both have proofs in $\mathcal{G}$. Using the proofs of these two formulas together with the proof of formula (1), we can construct a proof of $\{of\ \phi(v)\ \phi(a)\}$. Then by the adequacy of seq, it must be that $\Delta \vdash of\ \phi(v)\ \phi(a)$ has a derivation in hH^2. Finally by the adequacy of the clauses in Δ it must be that $\vdash v : a$ holds. Notice that one must prove adequacy for the clauses which make up a specification, but one does not need to ever re-prove the adequacy of seq. Thus, the two-level logic approach to reasoning does not introduce any recurring costs with respect to adequacy of the associated reasoning.

5.2 Type Uniqueness for the Simply Typed λ-Calculus

Proving the formula $[\forall E, A, B.\ \{of\ E\ A\} \supset \{of\ E\ B\} \supset A = B]$, that is, that types are unique for the simply typed λ-calculus, brings out another important aspect of the two-level logic approach to reasoning: the reasoning logic can be used to make explicit, and thereby to exploit in reasoning, properties of terms that arise dynamically when the specification logic is used to "carry out" computations described in it. Specifically, in this example we will use $\mathcal{G}$ to characterize the typing contexts that are constructed in hH^2 when using hypothetical judgments to assign types to abstractions.

In order to prove the theorem about uniqueness of types, we will need to generalize it to allow for the assignment of types relative to typing contexts. These typing contexts can be characterized in $\mathcal{G}$ by a variant of the $cntx$ predicate that we saw in Section 2 that is defined by the following clauses:

$$ctx\ nil \overset{\mu}{=} \top \qquad\qquad (\nabla x.\ ctx\ (of\ x\ A :: L)) \overset{\mu}{=} ctx\ L.$$

It is easy to see that if the judgment $ctx\ L$ holds, then L must be a list of elements of the form $(of\ x\ A)$ where each x is a nominal constant that does not appear later in the list. Thus, the type assignments in L must be to nominal constants and the assignment to each such constant must be unique. These properties, which are needed for proving the uniqueness of typing, are written as the following formulas in $\mathcal{G}$:

$$\forall X, A, L.\ ctx\ L \supset member\ (of\ X\ A)\ L \supset name\ X \tag{2}$$

$$\forall X, A, B, L.\ ctx\ L \supset member\ (of\ X\ A)\ L \supset member\ (of\ X\ B)\ L \supset A = B. \tag{3}$$

Both formulas can be established as lemmas in $\mathcal{G}$ by a simple induction on the structure of the ctx definition. Notice that in the second formula, the universally quantifier over X could have been replaced by the generic quantifier over X. We also note that the proof of this second formula makes use of the general lemma about list membership and nominal constants described at the beginning of this section.

The generalization of the type uniqueness theorem is now given as the following formula:

$$\forall E, A, B, L.\ ctx\ L \supset \{L \vdash of\ E\ A\} \supset \{L \vdash of\ E\ B\} \supset A = B.$$

Attempting to prove this formula yields the sequent

$$ctx\ L, \{L \vdash of\ E\ A\}, \{L \vdash of\ E\ B\} \longrightarrow A = B.$$

Applying induction on the first typing judgment with the following inductive hypothesis (again denoted by IH)

$$\forall E, A, B, L.\ ctx\ L \supset \{L \vdash of\ E\ A\}^* \supset \{L \vdash of\ E\ B\} \supset A = B.$$

results in three cases. The first case is

$$IH, ctx\ L, member\ (of\ E\ A)\ L, \{L \vdash of\ E\ B\} \longrightarrow A = B.$$

We can apply lemma (2) here to obtain

$$IH, ctx\ L, member\ (of\ E\ A)\ L, name\ E, \{L \vdash of\ E\ B\} \longrightarrow A = B.$$

Applying case analysis to the assumption $name\ E$ leads to a single premise since there is a most general upper sequent for this use of $def\mathcal{L}$. In particular, E is replaced by a nominal constant c and every other variable is raised over this constant. Thus we have the following sequent:

$$IH, ctx\ (L\ c), member\ (of\ c\ (A\ c))\ (L\ c), \{L\ c \vdash of\ c\ (B\ c)\} \longrightarrow A\ c = B\ c.$$

Now case analysis on the remaining typing assumption results in the single sequent

$$IH, ctx\ (L\ c), member\ (of\ c\ (A\ c))\ (L\ c), member\ (of\ c\ (B\ c))\ (L\ c) \longrightarrow A\ c = B\ c.$$

At this point we can apply lemma (3) to finish this case.

The second of the three original cases is the sequent

$$IH, ctx\ L, \{L \vdash of\ M\ (arr\ C\ A)\}^*, \{L \vdash of\ N\ C\}^*, \{L \vdash of\ (app\ M\ N)\ B\} \longrightarrow A = B.$$

Now we can perform case analysis on the remaining typing assumption for *app M N*. This results in two cases. The first is that *of* (*app M N*) *B* may occur in the list *L*. This case can be handled using lemma (2), *i.e.*, we can determine *name* (*app M N*) which when subjected to case analysis will result in zero cases (that is, it is recognized as a false assumption). The other case is

$$IH, ctx\ L, \{L \vdash of\ M\ (arr\ C\ A)\}^*, \dots, \{L \vdash of\ M\ (arr\ D\ B)\}, \dots \longrightarrow A = B.$$

At this point we can apply the induction hypothesis to the two typing judgments for *M* to determine that *arr C A = arr D B* and therefore $A = B$.

The remaining case in the original proof is the sequent

$$IH, ctx\ L, \{L, of\ c\ C \vdash of\ (R\ c)\ D\}^*, \{L \vdash of\ (lam\ C\ R)\ B\} \longrightarrow arr\ C\ D = B.$$

Here *c* is a nominal constant. Case analysis on the typing judgment for *lam C R* results in two cases. Again, the first one can be dismissed using lemma (2). The second one is as follows.

$$IH, ctx\ L, \{L, of\ c\ C \vdash of\ (R\ c)\ D\}^*, \{L, of\ c\ C \vdash of\ (R\ c)\ F\} \longrightarrow arr\ C\ D = arr\ C\ F.$$

Here we have opted to use the nominal constant *c* in deconstructing this second typing judgment. Any other choice is equally valid and does not affect the proof. In order to use the induction hypothesis we must be able to show that *ctx* (*of c C :: L*) holds: but this is immediate from the definition of *ctx* and the fact that *c* is a nominal constant which does not appear in *L*. Therefore we can use the induction hypothesis and determine that $D = F$, thus finishing the proof.

5.3 Extension to the Language of PCF

We now extend the specification of the simply typed λ-calculus to treat an abstract version of the programming language PCF presented by Plotkin [1977]. To do this, we replace the base type *i* : *ty* with the types for numbers *num* : *ty* and booleans *bool* : *ty*. We also enrich the set of terms by allowing the following constants.

zero : *tm*	*succ* : *tm* → *tm*	*if* : *tm* → *tm* → *tm* → *tm*
true : *tm*	*pred* : *tm* → *tm*	*rec* : *ty* → (*tm* → *tm*) → *tm*
false : *tm*	*iszero* : *tm* → *tm*	

Using these, the specification for evaluation and typing in PCF is presented in Figure 9.

We shall not repeat the proofs of type preservation and type uniqueness for PCF, but rather we will explain how these proofs differ from the ones for the simply typed λ-calculus. First, for type preservation, the statement is unchanged:

$$\forall E, V, A.\ \{eval\ E\ V\} \supset \{of\ E\ A\} \supset \{of\ V\ A\}.$$

The basic structure of this proof is the same, however, when we induct on $\{eval\ E\ V\}$ we get 13 cases instead of two, since *eval* has that many more cases now. These additional cases are either easy or similar to the cases in the earlier version of the proof. The substitution property for typing judgments is again obtained for free using the instantiation and cut properties of hH^2. The only increase in proof size is due to a widening of the central case analysis. The story for type uniqueness is the same: since typing contexts have not been changed, the definition of *ctx* is as before and the proof of the formula

$$\forall E, A, B, L.\ ctx\ L \supset \{L \vdash of\ E\ A\} \supset \{L \vdash of\ E\ B\} \supset A = B$$

proceeds as before but with additional cases as expected from the additional clauses in the specification of typing.

$$eval\ zero\ zero$$
$$eval\ true\ true$$
$$eval\ false\ false$$
$$eval\ M\ V \supset eval\ (succ\ M)\ (succ\ V)$$
$$eval\ M\ zero \supset eval\ (pred\ M)\ zero$$
$$eval\ M\ (succ\ V) \supset eval\ (pred\ M)\ V$$
$$eval\ M\ zero \supset eval\ (iszero\ M)\ true$$
$$eval\ M\ (succ\ V) \supset eval\ (iszero\ M)\ false$$
$$eval\ M\ true \supset eval\ N_1\ V \supset eval\ (if\ M\ N_1\ N_2)\ V$$
$$eval\ M\ false \supset eval\ N_2\ V \supset eval\ (if\ M\ N_1\ N_2)\ V$$
$$eval\ (lam\ A\ R)\ (lam\ A\ R)$$
$$eval\ M\ (lam\ A\ R) \supset eval\ (R\ N)\ V \supset eval\ (app\ M\ N)\ V$$
$$eval\ (R\ (rec\ A\ R))\ V \supset eval\ (rec\ A\ R)\ V$$

$$of\ zero\ num$$
$$of\ true\ bool$$
$$of\ false\ bool$$
$$of\ M\ num \supset of\ (succ\ M)\ num$$
$$of\ M\ num \supset of\ (pred\ M)\ num$$
$$of\ M\ num \supset of\ (iszero\ M)\ bool$$
$$of\ M\ bool \supset of\ N_1\ A \supset of\ N_2\ A \supset of\ (if\ M\ N_1\ N_2)\ A$$
$$of\ M\ (arr\ A\ B) \supset of\ N\ A \supset of\ (app\ M\ N)\ B$$
$$(\forall x.of\ x\ A \supset of\ (R\ x)\ B) \supset of\ (lam\ A\ R)\ (arr\ A\ B)$$
$$(\forall x.of\ x\ A \supset of\ (R\ x)\ A) \supset of\ (rec\ A\ R)\ A$$

Fig. 9 Evaluation and typing in PCF

$$term\ M \supset term\ N \supset term\ (app\ M\ N)$$
$$(\forall x.term\ x \supset term\ (R\ x)) \supset term\ (abs\ R)$$

$$path\ M\ done$$
$$path\ M\ P \supset path\ (app\ M\ N)\ (left\ P)$$
$$path\ N\ P \supset path\ (app\ M\ N)\ (right\ P)$$
$$(\forall x.\forall p.path\ x\ p \supset path\ (R\ x)\ (S\ p)) \supset path\ (abs\ R)\ (bnd\ S)$$

Fig. 10 Specification of paths through λ-terms

5.4 Comparing Paths in λ-Terms

Terms in the untyped, pure λ-calculus can be visualized as tree structures. As such, we can define paths in a term as paths that start at the root in the corresponding tree. We shall formally prove here that if every path in one λ-term is also a path in another λ-term, then the two terms are equal.

To formalize this theorem, we first need a representation of untyped λ-terms and paths in hH^2. We introduce the two types tm and pt for this purpose and we use the constructors shown below.

$$app : tm \to tm \to tm \qquad\qquad done : pt \qquad\qquad left : pt \to pt$$
$$abs : (tm \to tm) \to tm \qquad\qquad bnd : (pt \to pt) \to pt \qquad\qquad right : pt \to pt$$

Notice that since we are concerned with only pure λ-terms, we only need the two constructors app and abs for representing them.

We now introduce the predicates $term$ and $path$ defined by the specification logic formulas in Figure 10. Note that we allow partial paths using $done$. Notice also that the $\mathcal{G}$ formula

$$\forall R, S.\{path\ (abs\ R)\ (bnd\ S)\} \supset (\forall x.\forall p.\{path\ x\ p\} \supset \{path\ (R\ x)\ (S\ p)\})$$

is a kind of converse to the last clause specifying *path* and is also trivial to prove. Thus, if we have a path $(bnd\ S)$ through the term $(abs\ R)$ and a path P through term N, then the result of substituting path P into the abstraction S is a path in the term resulting from substituting N into the abstraction R. This formula is another example of a "substitution lemma for free."

We wish now to prove the theorem:

$$\forall M, N.\ \{term\ M\} \supset (\forall P.\ \{path\ M\ P\} \supset \{path\ N\ P\}) \supset M = N. \tag{4}$$

Since induction in $\mathcal{G}$ is an introduction rule for defined predicates, the assumption $\{term\ M\}$ is placed in this formula to enable induction on the structure of M. Before we prove this formula, we need to strengthen it. In particular, when M is an abstraction we need to consider how the contexts for the *term* and *path* judgments will grow. The defined predicate *ctxs* describes how these two contexts are related.

$$ctxs\ nil\ nil\ \overset{\mu}{=}\ \top \qquad\qquad (\nabla x.\nabla p.ctxs\ (term\ x :: L)\ (path\ x\ p :: K))\ \overset{\mu}{=}\ ctxs\ L\ K.$$

Along with this definition, we need the following lemmas which allow us to extract information about a term based on its membership in one of the contexts described by *ctxs*.

$$\forall X, L, K.\ ctxs\ L\ K \supset member\ (term\ X)\ L \supset name\ X \wedge \exists P.\ member\ (path\ X\ P)\ K$$

$$\forall X, P, L, K.\ ctxs\ L\ K \supset member\ (path\ X\ P)\ K \supset name\ X \wedge name\ P.$$

The proofs of both lemmas are by straightforward induction on the *member* hypotheses.

We can state the strengthened version of the theorem as the following lemma.

$$\forall L, K, M, N.\ ctxs\ L\ K \supset \{L \vdash term\ M\} \supset$$

$$(\forall P.\ \{K \vdash path\ M\ P\} \supset \{K \vdash path\ N\ P\}) \supset M = N.$$

The proof of this lemma proceeds by induction on $\{L \vdash term\ M\}$. The base case needs the following lemma, which is proved by induction on one of the *member* hypotheses and which uses the general lemma about list membership and nominal constants described in the preamble of this section.

$$\forall L, K, X_1, X_2, P.\ ctxs\ L\ K \supset member\ (path\ X_1\ P)\ K \supset$$

$$member\ (path\ X_2\ P)\ K \supset X_1 = X_2.$$

In the other cases of the proof, we need to show that the top-level constructor of M is also the top-level constructor of N. We do this by constructing a partial path through the top-level constructor of M: since paths in M are also paths in N, the top-level constructor of N must match that of M. Once we know that the top-level constructors are the same, we can use the assumption that all paths in M are paths in N to show that all paths in an immediate subterm of M are paths in the corresponding immediate subterm of N. Then by induction we can conclude that those subterms are equal.

There is one technical complication in the proof of path equivalence which comes from the inductive case concerning abstractions. Suppose $M = abs\ R$ and $N = abs\ R'$. Here we know

$$\forall P.\ \{K \vdash path\ (abs\ R)\ P\} \supset \{K \vdash path\ (abs\ R')\ P\}$$

but in order to use the inductive hypothesis we must show

$$\forall P.\ \{K, path\ x\ p \vdash path\ (R\ x)\ P\} \supset \{K, path\ x\ p \vdash path\ (R'\ x)\ P\},$$

where x and p are nominal constants. Now the problem is that when we go to prove this latter formula, the $\forall \mathcal{R}$ rule says that we must replace P by $(P'\ p\ x)$ for some new eigenvariable P'. Note that P' is raised over both p and x even though the dependency on x must be vacuous. The following lemma establishes this vacuity and is used to finish this case of the proof.

$$\forall K, M, P.\nabla x, p.\ \{K, path\ x\ p \vdash path\ (M\ x)\ (P\ p\ x)\} \supset \exists P'.\ P = \lambda z.P'$$

This lemma is proved by induction on the *path* judgment and uses the general lemma about nominal constants and list membership. Note that we single out *path x p* being the first member of the context even though new *path* assumptions may be added during induction. This is not a problem since we can always use the property of hH^2 which allows contexts to be freely rearranged. With this issue resolved, the proof of this theorem can now be completed.

$$step\ (app\ (lam\ A\ R)\ M)\ (R\ M)$$

$$step\ M\ M' \supset step\ (app\ M\ N)\ (app\ M'\ N)$$

$$step\ N\ N' \supset step\ (app\ M\ N)\ (app\ M\ N')$$

$$(\forall x.step\ (R\ x)\ (R'\ x)) \supset step\ (lam\ A\ R)\ (lam\ A\ R')$$

Fig. 11 One step β-reduction in the simply typed λ-calculus

In the theorem about paths, we have encoded the the property that all paths in m are paths in n via the formula

$$\forall P.\ \{path\ \phi(m)\ P\} \supset \{path\ \phi(n)\ P\}. \tag{5}$$

There is a question about the adequacy of this encoding even after we have established the adequacy of our representation of terms and paths and we have shown that, for all terms m and paths p, m has path p if and only if $\{path\ \phi(m)\ \phi(p)\}$ is provable in $\mathcal{G}$. To resolve this question in one direction, assume that the formula (5) is provable. Let p be any path in m so that we have a proof of $\{path\ \phi(m)\ \phi(p)\}$. Using formula (5) and this proof we can construct a proof of $\{path\ \phi(n)\ \phi(p)\}$. Thus n has path p. For the other direction, we argue that if every path in m is a path in n then we can prove formula (5). Such a proof reduces to constructing a derivation of the sequent

$$\{path\ \phi(m)\ P\} \longrightarrow \{path\ \phi(n)\ P\}. \tag{6}$$

We can construct a proof of this sequent by repeatedly unfolding $\{path\ \phi(m)\ P\}$ and the new hypotheses which result from it. This process will terminate since $\phi(m)$ is a finite term with no variables and the recursive clauses of $path$ always deconstruct their first argument. The sequents which result from this repeated case analysis will have the form $\longrightarrow \{path\ \phi(n)\ P\}$ for some instance of P such that $\longrightarrow \{path\ \phi(m)\ P\}$ is provable. By the assumption of adequacy for the $path$ predicate, we know $P = \phi(p)$ where p is a path in m. Thus p is also a path in n and thus each sequent $\longrightarrow \{path\ \phi(n)\ P\}$ is provable.

5.5 Other Examples

There are many other examples of topics that have been completely formalized within $\mathcal{G}$ and checked using the Abella prover. We list some of these examples here: complete details of the proofs can be found on the website for Abella [Gacek, 2009a].

Meta-theory of the λ-calculus We have used Abella to specify both big-step and small-step evaluation for untyped λ-terms and then to prove that they compute the same values and that they are both determinate and type-preserving. We have also encoded a proof of the Church-Rosser theorem and have also proved strong normalization for the simply typed λ-calculus. The latter theorem and proof deserve a few additional words. Strong normalization for the λ-calculus can be defined elegantly as

$$sn\ M\ \overset{\mu}{=}\ \forall N.\{step\ M\ N\} \supset sn\ N,$$

where $step$ (specified in Figure 11) relates two terms when the second is the replacement of exactly one β-redex in the first. Induction on sn corresponds to induction on the tree of possible β-reductions for a term which in this case can be used in place of induction on the longest possible length of a β-reduction. Using the predicate of defined in Figure 8, the strong normalization theorem for the simply typed λ-calculus is stated simply as

$$\forall M, A.\{of\ M\ A\} \supset sn\ M.$$

The proof of this theorem uses a logical relations style argument based on the predicate $reduce$ defined as

$$reduce\ M\ i\ \overset{\mu}{=}\ \{of\ M\ i\} \wedge sn\ M$$

$$reduce\ M\ (arr\ A\ B)\ \overset{\mu}{=}\ \{of\ M\ (arr\ A\ B)\} \wedge \forall U.\ reduce\ U\ A \supset reduce\ (app\ M\ U)\ B.$$

Abella allows such a definition although it does not satisfy the stratification condition described in Section 2. As we mention in Section 7, more flexible notions of stratification need to be identified and validated in order to justify this proof.

Meta-theory of the π-calculus We have specified the semantics of the finite π-calculus using the specification logic and formalized the notion of open bisimulation using a co-inductive definition in the reasoning logic. We have shown that open bisimulation is an equivalence relation and a congruence using this formalization. This formalization constitutes an elegant treatment of the π-calculus where all issues involving bindings, names, and substitutions are handled declaratively without explicit side-conditions.

The POPLmark challenge problems The POPLmark challenge [Aydemir et al., 2005] is a selection of problems which highlights the traditional difficulties in reasoning about systems which manipulate objects with binding. The particular tasks of the challenge involve reasoning about evaluation, typing, and subtyping for $F_{<:}$, a λ-calculus with bounded subtype polymorphism. We have solved parts 1a and 2a of this challenge using Abella, which represent the fundamental reasoning tasks involving objects with binding.

Cut-elimination We have shown that the cut rule can be eliminated from LJ while preserving the provability relation. The encoding of sequents in our specification logic used hypothetical judgments to represent LJ hypotheses and generic judgments to represent LJ universals. This allowed the cut-elimination proof to take advantage of Abella's built-in treatment of meta-properties of the specification logic.

6 Related Work

The range of applications that we have demonstrated for our reasoning logic $\mathcal{G}$ depends on its strong declarative treatment of binding as well as its treatment of fixed points (*i.e.*, induction and co-induction). In comparing our work to the many other research efforts devoted to building theorem provers that can reason about specifications of computations, it is convenient to characterize the latter approaches using these two axes of logical expressiveness. Some of these systems start with a clean and comprehensive foundation for fixed points and (co)induction, treating the notion of of binding as something that can be implemented later within such an inductive logic. Other systems start with a logically supported approach to binding and then later provide some aspects of inductive reasoning over binding structure. We use this coarse classification below to organize our comments about related efforts.

6.1 Inductive Frameworks with Treatments of Binding Added

Many systems for reasoning about computations start with established inductive logic theorem provers such as Coq [Bertot and Castéran, 2004] (based on the *Calculus of Inductive Constructions* [Coquand and Paulin, 1988]) and Isabelle/HOL [Nipkow et al., 2002], and then use those systems to build approaches to binding and substitution. We discuss three examples of this approach: the locally nameless representation, the Nominal package for Isabelle/HOL [Urban, 2008], and Hybrid [Felty and Momigliano, 2010].

The locally nameless representation of binding structure uses de Bruijn indices for bound variables and names for free variables. The central benefits of this approach are that α-equivalent terms are syntactically equal, the statements of lemmas and theorems rarely need to talk about arithmetic operations over de Bruijn indices, and capture-avoiding substitution can be defined in a straightforward and structurally recursive way. However, one must still define this substitution manually and prove lemmas about its behavior. Additionally, there is no device like ∇ for quantifying over fresh variable names. Instead, practitioners of the locally nameless approach (see, for example, Aydemir et al. [2008]) advocate an encoding of such quantification using *cofinite quantification*, *i.e.*, quantification over all names not belonging to some arbitrary, finite set. This technique, however, still requires sometimes explicitly proving that free variables can be renamed while preserving provability of a judgment.

The Nominal package for Isabelle/HOL automates the formalization of alpha-equivalence classes based on ideas from nominal logic [Pitts, 2003]. The user is then left to define and reason about a notion of capture-avoiding substitution over terms constructed with such alpha-equivalence classes. Reasoning over open terms is supported in the Nominal package via the nominal logic И-quantifier which has similarities to the ∇-quantifier. However, the И-quantifier is "over-worked" in the nominal approach since it is also used to introduce names which are bound by name abstractions. This creates some additional difficulties such as when introducing functions and predicates in the nominal approach one must prove properties which state that name swapping does not change the results of a function or the provability of a predicate—a property which is enforceable statically for definitions of predicates in $\mathcal{G}$ due to the separation between free and bound variables.

Hybrid adds support to traditional theorem provers such as Coq and Isabelle/HOL for reasoning about binding structures by translating such structure into a de Bruijn representation. The logic of the theorem prover then serves as the meta-logic in which reasoning is conducted. This approach necessarily produces more overhead

during reasoning due to the occasional need to reason about the effects of the translation, although one might expect that such reasoning can eventually be automated. Hybrid is often used in a two-level logic approach using a specification logic that is essentially the same hH^2 specification language considered in this paper. The Hybrid system, by design, lacks a reasoning logic with a device like the ∇-quantifier for reasoning about open terms and generic judgments. Recent work has suggested that such a device is not necessary for simple reasoning tasks such as type uniqueness arguments [Felty and Momigliano, 2009], although it is unclear if the Hybrid approach will scale to problems such as those proposed by the POPLmark challenge [Aydemir et al., 2005]. For these types of problems one needs to recognize as equivalent those judgments which differ only in the renaming of free variables. Such a property is built into $\mathcal{G}$ through the use of nominal constants to denote such free variables. To use such an approach in Hybrid, one will have to manually develop and prove properties about notions of variable permutations.

6.2 Binding Frameworks with Treatments of Induction Added

There are a variety of systems for reasoning about computations which take binding as a primitive notion and then attempt to define separately notions of induction over that structure. Many of these start with the LF logical framework [Harper et al., 1993], a dependently typed λ-calculus with a direct treatment of variable binding. While the LF type system can be used to describe both the structure and behavior of many computational systems, it does not include a notion of induction: inductive arguments about LF specifications are typically supported by constructing a second layer on top of LF.

Twelf [Pfenning and Schürmann, 1999], the most popular tool for reasoning about LF specifications, provides an operational semantics for LF that defines recursive relations over LF terms. Subject to some side-conditions, these relations can then be interpreted as proofs about LF specifications. Similar functional approaches have been developed starting with $\mathcal{M}_2^+$ [Schürmann, 2000], a simple meta-logic for reasoning over LF representations where proof terms are represented as recursive functions. More recent work includes the Delphin [Poswolsky and Schürmann, 2008] and Beluga [Pientka, 2008] functional languages which can be used in the same spirit as $\mathcal{M}_2^+$. New work by Licata et al. [2008] proposes a language which combines LF with recursive functions over LF so that a strict separation into levels is no longer needed. In all of these approaches, however, side-conditions for termination and coverage are required and algorithms have been devised to check for such properties. Since termination and coverage are in general undecidable, such algorithms are necessarily incomplete.

6.3 The Development of a Logic for both Bindings and Fixed Points

The logic $\mathcal{G}$ is the result of an extended effort to design a single logic that integrates induction and co-induction with the ability to reason flexibly about bindings. The λProlog language [Nadathur and Miller, 1988] provided a starting point as a specification language that allowed a completely declarative treatment of binding. In order to support reasoning about specifications written in the hH^2 subset of λProlog, McDowell and Miller [2000] developed the two-level logic approach used in this paper but with a much weaker reasoning logic called $FO\lambda^{\Delta\mathbb{IN}}$. That logic provided induction on natural numbers but did not contain ∇-quantification. As a result of this missing ingredient, reasoning about object-level bindings became unduly complicated; see, for example, the discussion on explicit eigenvariable encoding in [McDowell and Miller, 2000].

The ∇-quantifier was first introduced in [Miller and Tiu, 2005]. The logic that was first proposed did not include inference rules for induction and co-induction but these were added shortly thereafter by Tiu [2004]. The initial logics adopted a minimalistic view of the ∇-quantifier that turned out to be inadequate for many instances of inductive reasoning over binding structures. To redress this situation Tiu [2006] proposed the addition of the ∇-exchange and ∇-strengthening rules and developed the nominal constant based treatment of the ∇-quantifier used in this paper. The resulting logic still did not have the ability to concisely characterize occurrences of nominal constants in terms and was consequently awkward to use in reasoning about open terms and contexts. The missing piece was provided by the notion of nominal abstraction Gacek et al. [2009]. This final logic, $\mathcal{G}$, combines into one proof system, the two separate components for reasoning about fixed points and about binding. These components are independently constructed yet their interaction is well-behaved and quite useful.

7 Conclusions and Future Work

We have presented an intuitionistic logic, $\mathcal{G}$, in which binding is treated directly using the ∇-quantifier (both in formulas and the head of definitions) and in which least and greatest fixed points are treated directly using

inference rules for induction and co-induction. In a logic that has this kind of expressiveness, it is possible to inductively define proofs systems for specification logics, such as hH^2. This makes it possible to use a theorem-proving approach in which the specification logic is used in an intrinsic way and in which reasoning takes place through a transparent embedding of that logic in the richer reasoning logic. We have described a system called Abella that exploits this two-level logic approach and we have shown its flexibility and power through a sequence of reasoning examples. While the illustrations we have been able to consider in this paper are limited, Abella has had a number of significant theorem-proving successes that are described more completely on the web page associated with it.

Experience with the two-level logic approach to reasoning has provided us with insights into possible ways to enhance the logic $\mathcal{G}$ and the methodology built into Abella. We indicate a few such directions that we intend to pursue in the near future.

More permissive stratification conditions for definitions The current stratification condition for definitions in $\mathcal{G}$ is somewhat simplistic: that condition rules out seemingly well-behaved definitions such as that of the reducibility relation used in logical relations arguments; see Section 5.5 for details. One could imagine a more sophisticated condition which would allow definitions to be stratified based on an ordering relation over the arguments of the predicate being defined. The proof theoretic arguments needed to prove cut-elimination for a logic with such definitions seem rather delicate, particularly since we allow substitutions which may interfere with any ordering based on term structure.

Contexts are special In principle, provability in the specification logic is captured by an inductive definition of the *seq* predicate; in practice, it has been most useful for Abella to provide some special treatment of that predicate (via the $\{\cdot \vdash \cdot\}$ notation). Similarly, while contexts are, in principle, just another list structure, it seems likely that they should also have some special attributes associate to them. As some examples illustrated, the current practice requires stating a definition describing a context, proving various inversion lemmas about membership in such contexts, and then applying these lemmas at the appropriate times. Treating context as special objects should make it possible to automate several of these lemmas or to arrange things so that such lemmas are not needed but have their effects embedded into the prover.

Types-as-predicates As we have described the logic $\mathcal{G}$, there is no direct connection between predicates (on which we may apply induction) and the simple types attributed to variables. The description of the type and its constructors is not sufficient: it is necessary to define a predicate that describes the members of the type. For example, if we wish to do induction on the structure of untyped λ-terms (as in Section 5.4), we need to build the predicate *term* from the description of the type *tm*. Linking simple types to the predicates that define them is a natural enhancement to a theorem prover for $\mathcal{G}$.

Alternate specification logics and proof systems In this paper, we fixed the specification logic to be hH^2 and we fixed the proof system for hH^2 to be based on goal-directed proof search. Clearly some applications of the two-level logic approach might benefit from using a different proof system (based on, say, bottom-up proof search) or a different logic. For example, McDowell and Miller [2002] showed that switching to a linear logic specification logic made it possible to treat programming languages with references. More concretely, we have implemented a full hereditary Harrop formula specification logic in Abella and have begun experimenting with reasoning over it.

Automating proof search Abella currently relies extensively on user guidance in constructing proofs. Recent work has developed formal theorems and implementation techniques for structuring proof search in $\mathcal{G}$-like logics: see, for example, [Baelde et al., 2007, Baelde, 2008]. It would be interesting to use such results to build a greater degree of automation into Abella.

8 Acknowledgements

21

References

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.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve 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.

David Baelde. *A linear approach to the proof-theory of least and greatest fixed points*. PhD thesis, Ecole Polytechnique, December 2008.

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.

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.

Alonzo Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.

Thierry Coquand and Christine Paulin. Inductively defined types. In *Conference on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer-Verlag, 1988.

Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.

Amy Felty and Alberto Momigliano. Reasoning with hypothetical judgments and open terms in Hybrid. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 83–92, 2009.

Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax, 2010. To appear in *Journal of Automated Reasoning*.

Andrew Gacek. The Abella system and homepage. `http://abella.cs.umn.edu/`, 2009a.

Andrew Gacek. The Abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008. URL `http://arxiv.org/abs/0803.2305`.

Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009b.

Andrew Gacek, Steven Holte, Gopalan Nadathur, Xiaochu Qi, and Zach Snow. The Teyjus system – version 2, March 2008. Available from `http://teyjus.cs.umn.edu/`.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. Technical report, CoRR, August 2009. URL `http://arxiv.org/abs/0908.1390`. Extended version of paper in LICS 2008. To appear in Information & Computation.

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

Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer, March 1987.

P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 241–252. IEEE Computer Society Press, 2008.

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

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.

Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.

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

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

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.

Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

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

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.

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

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.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM, 2008.

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

Gordin Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

Gordon Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5, 1977.

Gordon Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

Adam Poswolsky and Carsten Schürmann. System description: Delphin - A functional programming language for deductive systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, volume 228, pages 113–120, 2008.

John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1): 120–153, May 1994.

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

Craig Smorynski. Modal logic and self-reference. In Dov Gabbay and Franz Guenther, editors, *Handbook of Philosophical Logic, Volume 11 (Second Edition)*, pages 1–54. Kluwer Academic, 2004.

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

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.

Alwen Tiu and Alberto Momigliano. Induction and co-induction in sequent calculus. Available from `http://arxiv.org/abs/0812.4727`, 2009.

Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.