Reasoning with Hypothetical Judgments and Open Terms in Hybrid

Amy P. Felty
School of Information Technology and Engineering
University of Ottawa, Ontario, Canada
afelty@site.uottawa.ca

Alberto Momigliano
School of Informatics, University of Edinburgh
Scotland, United Kingdom
amomigl1@inf.ed.ac.uk

Abstract

Hybrid is a system developed to specify and reason about logics, programming languages, and other formal systems expressed in higher-order abstract syntax (HOAS). An important goal of Hybrid is to exploit the advantages of HOAS within the well-understood setting of higher-order logic as implemented by systems such as Isabelle and Coq. In this paper, we add new capabilities for reasoning by induction on encodings of object-level inference rules. Elegant and succinct specifications of such inference rules can often be given using hypothetical and parametric judgments, which are represented by embedded implication and universal quantification. Induction over such judgments is well-known to be problematic. In previous work, we showed how to express this kind of judgment using a two-level approach, but reasoning by induction on such judgments was restricted to closed terms. The new capabilities we add include techniques for adding arbitrary "new" variables to contexts and inductively reasoning about open terms. Very little overhead is required, namely a small library of definitions and lemmas, yet the reasoning power of the system and the class of properties that can be proved is significantly increased. We illustrate the approach using PCF, a simple programming language that serves as the core of a variety of functional languages. We encode the typing judgment, and prove by induction on this judgment that well-typed PCF terms have unique types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal definitions and theory—semantics; F.4.1 [*Mathematical Logic*]: Lambda Calculus and Related Systems—Mechanical theorem proving, Proof theory; I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Deduction, Inference engines, logic programming, meta theory

General Terms Languages, Verification

Keywords higher-order abstract syntax, logical frameworks, name-binding, induction, interactive theorem proving

1. Introduction

Higher-order abstract syntax (HOAS) is a representation technique that allows direct and concise specifications of a wide variety of formal systems. This technique is often used within a typed *logical framework* that supports reasoning about such encodings. One of the main uses of these logical frameworks is to represent and prove properties about the semantic foundations of declarative programming languages. Using HOAS, binding constructs in the represented language (the *object logic* or OL) are encoded using the binding constructs provided by an underlying λ-calculus or function space of the logical framework (the *meta-language*). For example, consider the untyped λ-calculus as an OL. Its terms can be encoded, for instance, by introducing a type tm and two constructors: abs of type $(\underline{tm} \rightarrow tm) \rightarrow tm$, and app of type $tm \rightarrow tm \rightarrow tm$. Using such a representation allows us to delegate to the meta-language α-conversion and capture-avoiding substitution. Further, object logic substitution can be rendered as meta-level β-conversion.

In addition, in such logical frameworks, embedded implication and universal quantification are often used to represent *hypothetical* and *parametric* judgments—following [Miller and Tiu 2005], we will also call them *generic*—which allow elegant and succinct specifications of OL inference rules. For instance, if our example OL includes rules for adding types to untyped terms, the following rule for the abstraction case:

$$\frac{\begin{array}{c}(x : \tau_1)\\ \vdots \\ M : \tau_2\end{array}}{\lambda x.M : \tau_1 \Rightarrow \tau_2}$$

can be expressed using the *typeof* predicate in the following formula:

$$\begin{array}{l}\forall M : tm \rightarrow tm. \forall \tau_1, \tau_2 : tp.\\ \quad (\forall x : tm. (\underline{typeof\ x\ \tau_1}) \longrightarrow (typeof\ (M\ x)\ \tau_2))\\ \quad\quad \longrightarrow (typeof\ (abs\ M)\ (\tau_1 \Rightarrow \tau_2)).\end{array}$$

Hybrid [Momigliano et al. 2008] is a system developed to support HOAS encoding and reasoning. It is implemented in both Isabelle/HOL [Nipkow et al. 2002] and Coq [Bertot and Castéran 2004]. Implementing Hybrid as a tool within such systems allows users wishing to reason with HOAS encodings to draw on the powerful deduction capabilities enjoyed by these systems: rich principles of (co)induction and tactic-style automation, not to mention general rewriting, decision procedures, model checking, interface to automated theorem provers, code generation etc. Hybrid provides additional tool support within this setting to automate tasks specific to reasoning with HOAS.

One of the main challenges in developing Hybrid came from the presence of negative occurrences in the definitions of the types and predicates introduced in HOAS encodings of OL terms and judgments (e.g., the underlined occurrences of tm in the type of abs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.
Copyright © 2009 ACM 978-1-60558-568-0/09/09. . . $5.00

and of *typeof* in the formula expressing the typing rule above). In systems such as Coq and Isabelle/HOL, such encodings cannot be defined directly using inductive definitions of the metalanguage. Set-theoretically, these definitions do not yield monotone operators and cannot be constructed as a least fixed point [Gunter 1992, Paulson 1994]. Type-theoretically, they infringe on the *strict positivity* requirement [Paulin-Mohring 1993] used to obtain strong normalization, and thus are not inductive in any ordinary sense. To overcome the problem of negative occurrences in types such as tm, we introduced a de Bruijn representation of λ-terms that provides a definitional layer [Ambler et al. 2002]. Higher-order syntax encodings are defined on top of this layer so that they expand to de Bruijn terms. To overcome the problem of negative occurrences in predicates such as *typeof*, Hybrid adopts the two-level approach first introduced in the $FO\lambda^{\Delta\hat{I}N}$ logic [McDowell and Miller 2002], later adapted to Coq [Felty 2002], now applied within a variety of logics [Miller and Tiu 2005, Tiu 2007, Gacek et al. 2008], and implemented in the Abella system [Gacek 2008]. In a two-level system, the *specification* and (inductive) *meta-reasoning* are done within a single system but at different *levels*. An intermediate level is introduced by inductively defining a *specification logic* (SL), and OL judgments (including hypothetical and parametric judgments) are encoded in the SL.

Previous work on Hybrid involved a variety of case studies, some of them quite large. In [Felty and Momigliano 2008], for example, formal proofs of type soundness (subject reduction) for two example OLs and SLs are given, one for a small language to illustrate the methodology, and another for a more complex one, driven by a sub-structural SL, to illustrate that the approach scales well on both sides. Induction over OL judgments is required in such proofs, but in this case induction was on the evaluation judgment, whose encoding does not use hypothetical or parametric judgments. In other words, although generic judgments (e.g., typing) were crucial for these proofs, namely to provide *inversion* principles, induction over these judgments was not.

Induction over hypothetical and parametric judgments introduces new challenges, which is the central issue that we address in this paper. Statements have to be generalized to non-empty contexts, and these contexts have to be of a certain form, which must enforce the property in question. To try to motivate these challenges, consider a proof by induction over the *typeof* predicate.[1] If the case for abstraction is defined as above, the induction hypothesis takes on a similar form, roughly:

$$\forall x : tm.(P\ (typeof\ x\ \tau_1)) \longrightarrow (P\ (typeof\ (M\ x)\ \tau_2)).$$

Using this induction hypothesis in a proof requires finding an appropriate instantiation term for x. Note that the universal quantifier can be instantiated with any term, which provides the required flexibility in doing proofs using this form of HOAS. On the other hand, using the induction hypothesis for the abstraction case in an informal proof means reasoning about an *arbitrary* x such that $x : \tau_1$, and a term $M : \tau_2$ that possibly contains free occurrences of x. Thus x is a variable and M is an open term. In a variety of approaches to HOAS that allow definitions in the form of our example *typeof* clause above, M is akin to a second-order logic variable, possibly depending on a parameter x. There is no notion of free variables or open terms, so one is *required* to find some closed term as the instantiation term for x. (We remark that the induction hypothesis takes on a slightly different form in a two-level system, but the issues discussed here remain the same.) The contribution of the work presented in this paper is a methodology that keeps the advantages of this form of HOAS, but adds a very small number of definitions and lemmas that allow reasoning about "arbitrary" variables in such a way that keeps the formal proof close to the informal one, and adds only a surprisingly minimal amount of additional infrastructure. In fact, Hybrid has a built in VAR constructor to allow one to encode free variables of OLs, and a definition (newvar) that provides the capability of creating a variable which is *new*, in particular w.r.t. a context. Our approach to induction over predicates with hypothetical judgments makes essential and novel use of this built-in constructor and definition. In Hybrid's underlying language, free variables are essentially represented by natural numbers (VAR takes a natural number argument). One of the main operations involving such variables is to introduce new ones into a proof, such as x in the above example. To do so, we simply add 1 to the maximum of the free variables used so far. Reasoning about such "fresh" variables is factored into a small number of lemmas. The lemmas are relatively simple; no reasoning about substitution or α-conversion is needed as in first-order approaches. In a sense, newvar is the "poor man" version of the *freshness predicate* (_ # _) of nominal logic [Pitts 2003].

We introduce the approach by formally proving that well-typed terms in PCF [Scott 1993] have unique types (type unicity). The informal proof is a straightforward induction over the typing derivation that assigns a type to a PCF term. We use this simple example to illustrate the methodology, though the techniques are general and should scale to larger case studies. The inductive case for the abstraction operator—and this applies to other *binding* term constructors—which uses a rule similar to the one discussed earlier, is the challenging case. We express the induction hypothesis as a "context invariant," which is a property that must be preserved when adding a "fresh" variable to the context, as is required in this case. The general infrastructure we build is designed so that it is straightforward to express context invariants and prove that they are preserved when adding a "fresh" variable.

The paper starts with Section 2 recalling some basic notions of the implementation of Hybrid. Section 3 introduces the SL, a fragment of second-order minimal logic. Section 4 introduces the example OL, in particular, presenting the encodings of the syntax and typing rules of PCF, and briefly discussing the adequacy of these encodings. In Section 5, we present the formal proof of type unicity for PCF. We discuss related work in Section 6 and conclude in Section 7.

Hybrid was first developed in Isabelle/HOL [Ambler et al. 2002] and for the sake of this paper, we use a pretty-printed version of Isabelle/HOL concrete syntax. Note, however, that the proof of the main result and all the code mentioned in Section 5 were (so far) conducted only in Coq, due to some backward compatibility issues with the current release of Isabelle/HOL. In particular, a type declaration has the form $s :: [t_1, \ldots t_n] \Rightarrow t$. We stick to the usual logical symbols for connectives and quantifiers ($\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$). Free variables (upper-case) are implicitly universally quantified (from the outside) as in logic programming. The sign $\equiv$ (Isabelle meta-equality) is used for *equality by definition*, and $\bigwedge$ for Isabelle universal meta-quantification. A rule (a sequent) of the schematic form $\frac{H_1 \ldots H_n}{C}$ is represented as $[\![H_1; \ldots; H_n]\!] \Longrightarrow C$. The keyword *inductive* introduces an inductive relation in Isabelle/HOL, *datatype* introduces a new datatype, and *primrec* a primitive recursive function. We use the same notation for Coq, though in Coq all the arrows map to the same operator, and there is only one universal quantifier. In addition both *inductive* and *datatype* map to Coq's *Inductive* keyword, and *primrec* maps to *Fixpoint*.

Every theorem, lemma, and corollary is machine-checked. Source files for the code can be found at:

`hybrid.dsi.unimi.it/ppdp09` [Hybrid Group 2009].

[1] We ask the reader to indulge us, while talking about an induction principle, which is not inductive in the standard sense [Schürmann 2001].

2. An Introduction to Hybrid

At the base level, we start with an inductive definition of de Bruijn expressions:

$$\begin{array}{l} \mathit{datatype}\ \mathit{expr} = \\ \quad \mathsf{CON}\ \mathit{con} \mid \mathsf{VAR}\ \mathit{var} \mid \mathsf{BND}\ \mathit{bnd} \mid \mathit{expr}\ \ \$\ \ \mathit{expr} \mid \mathsf{ABS}\ \mathit{expr} \end{array}$$

In our setting, bnd and var are defined to be the natural numbers, and con is used to represent the constants of an OL. Thus at this level, con is a parameter to this type, and given a particular instantiation, we will later use a type abbreviation, such as $uexp = = con\ expr$.

Central to our approach is the introduction of a binding operator called lambda that (1) allows a direct expression of λ-abstraction, and (2) is *defined* in such a way that expanding its definition results in the conversion of a term to its de Bruijn representation. Hybrid does not contain any axioms requiring external justification as in the Theory of Contexts [Honsell et al. 2001].

As an example, consider the λ-calculus as an OL and the sample term $\Lambda V_1.(\Lambda V_2.V_1V_2)V_1V_3$, where we use upper case letters for variables and a capital Λ for abstraction. This term is represented in Hybrid as:

$$\mathsf{lambda}\ \lambda v_1.(((\mathsf{lambda}\ \lambda v_2.(v_1\ \$\ v_2))\ \$\ v_1)\ \$\ \mathsf{VAR}\ 3)$$

and expanding definitions results in the de Bruijn term:

$$\mathsf{ABS}\ (((\mathsf{ABS}\ (\underline{\mathsf{BND}\ 1}\ \$\ \mathsf{BND}\ 0))\ \$\ \underline{\mathsf{BND}\ 0})\ \$\ \mathsf{VAR}\ 3).$$

In the above, all the variable occurrences bound by the first ABS, which corresponds to the bound variable V_1 in the object-level term, are underlined. Note that the definition of the lambda operator must expand to a term with ABS at the head. Furthermore, we must include a definition of a function f such that (lambda e) is (ABS (f e)) where f replaces occurrences of the bound variable in e with de Bruijn index 0, taking care to increment the index as it descends through inner abstractions. We first define a function lbind of two arguments such that formally:

$$(\mathsf{lambda}\ e) == \mathsf{ABS}\ (\mathsf{lbind}\ 0\ e)$$

and (lbind i e) replaces occurrences of the bound variable in e with de Bruijn index i, where recursive calls on inner abstractions will increase the index.

We express lbind as a total function operating on all functions of type ($expr \Rightarrow expr$), even *exotic* ones, i.e., those that do not encode honest-to-goodness λ-terms. For example, we could have $e = (\lambda x.\mathsf{count}\ x)$ where (count x) counts the total number of variables and constants occurring in x. Only functions that behave uniformly on their arguments represent λ-terms, e.g., those functions that hereditarily abstract only over the constructors of the $expr$ datatype. We refer the reader to [Despeyroux et al. 1995] for an analysis of this phenomenon. We define lbind so that it maps non-uniform subterms to a default value. The subterms we target are those that do not satisfy the predicate ordinary :: $[\, expr \Rightarrow expr \,] \Rightarrow bool$, defined as follows:

$$\begin{array}{rcl} \mathsf{ordinary}\ e & == & (\exists a.\ e = (\lambda v.\ \mathsf{CON}\ a) \vee \\ & & e = (\lambda v.\ v) \vee \\ & & \exists n.\ e = (\lambda v.\ \mathsf{VAR}\ n) \vee \\ & & \exists j.\ e = (\lambda v.\ \mathsf{BND}\ j) \vee \\ & & \exists f\ g.\ e = (\lambda v.\ f\ v\ \$\ g\ v) \vee \\ & & \exists f.\ e = (\lambda v.\ \mathsf{ABS}\ (f\ v))) \end{array}$$

We do not define lbind directly, but instead define a relation lbnd :: $[\, bnd, expr \Rightarrow expr, expr \,] \Rightarrow bool$ and prove that this relation defines a function, mapping the first two arguments to the third.

$$\begin{array}{rcl} \mathit{inductive}\ \mathsf{lbnd} :: [\, bnd, expr \Rightarrow expr, expr \,] \Rightarrow bool & & \\ & \Longrightarrow & \mathsf{lbnd}\ i\ (\lambda v.\ \mathsf{CON}\ a)\ (\mathsf{CON}\ a) \\ & \Longrightarrow & \mathsf{lbnd}\ i\ (\lambda v.\ v)\ (\mathsf{BND}\ i) \\ & \Longrightarrow & \mathsf{lbnd}\ i\ (\lambda v.\ \mathsf{VAR}\ n)\ (\mathsf{VAR}\ n) \\ & \Longrightarrow & \mathsf{lbnd}\ i\ (\lambda v.\ \mathsf{BND}\ j)\ (\mathsf{BND}\ j) \\ [\![\ \mathsf{lbnd}\ i\ f\ s; & & \\ \quad \mathsf{lbnd}\ i\ g\ t\ ]\!] & \Longrightarrow & \mathsf{lbnd}\ i\ (\lambda v.\ f\ v\ \$\ g\ v)\ (s\ \$\ t) \\ \mathsf{lbnd}\ (i+1)\ f\ s & \Longrightarrow & \mathsf{lbnd}\ i\ (\lambda v.\ \mathsf{ABS}\ (f\ v))\ (\mathsf{ABS}\ s) \\ \neg(\mathsf{ordinary}\ e) & \Longrightarrow & \mathsf{lbnd}\ i\ e\ (\mathsf{BND}\ 0) \end{array}$$

We now define lbind :: $[\, bnd, expr \Rightarrow expr \,] \Rightarrow expr$ as follows, thus completing the definition of lambda:

$$\mathsf{lbind}\ i\ e == \mathit{THE}\ s.\ \mathsf{lbnd}\ i\ e\ s$$

where *THE* is Isabelle's notation for the definite description operator ι. Note that this operator is not available in Coq. The use of this operator is the main reason for the differences in the two libraries. The Coq version instead uses a definite description axiom available in Coq's classical reasoning library.

Ruling out non-uniform functions, which was mentioned before, is important for a variety of reasons. For example, it is necessary for proving that our encoding adequately represents the λ-calculus. To prove adequacy, we identify a subset of the terms of type $expr$ such that there is a bijection between this subset and the λ-terms that we are encoding. There are two aspects we must consider in defining a predicate to identify this subset. First, recall that BND i corresponds to a bound variable in the λ-calculus, and VAR i to a free variable; we refer to *bound* and *free indices* respectively. We call a bound index i *dangling* if i or less ABS labels occur between the index i and the root of the expression tree. We must rule out terms with dangling indices. Second, in the presence of the lambda operator, we may have functions of type ($expr \Rightarrow expr$) that do not behave uniformly on their arguments. We must rule out such functions. We define a predicate proper, which rules out dangling indices from terms of type $expr$, and a predicate abstr, which rules out dangling indices and exotic terms in functions of type ($expr \Rightarrow expr$).

To define proper we first define level. Expression e is said to be at *level* $l \geq 0$, if enclosing e inside l ABS nodes ensures that the resulting expression has no dangling indices.

$$\begin{array}{rcl} \mathit{inductive}\ \mathsf{level} & :: & [\, bnd, expr \,] \Rightarrow bool \\ & \Longrightarrow & \mathsf{level}\ i\ (\mathsf{CON}\ a) \\ & \Longrightarrow & \mathsf{level}\ i\ (\mathsf{VAR}\ n) \\ j < i & \Longrightarrow & \mathsf{level}\ i\ (\mathsf{BND}\ j) \\ [\![\ \mathsf{level}\ i\ s; \mathsf{level}\ i\ t\ ]\!] & \Longrightarrow & \mathsf{level}\ i\ (s\ \$\ t) \\ \mathsf{level}\ (i+1)\ s & \Longrightarrow & \mathsf{level}\ i\ (\mathsf{ABS}\ s) \end{array}$$

Then, proper :: $expr \Rightarrow bool$ is defined simply as:

$$\mathsf{proper}\ e == \mathsf{level}\ 0\ e.$$

We actually take this a step further and use the typedef mechanism of Isabelle/HOL to define a type of proper terms [Momigliano et al. 2008], eliminating the need for the proper predicate. The Coq version, which was used for the development in the rest of this paper, does not (yet) have an analogous improvement, so proper annotations will continue to appear in this paper.

To define abstr, we first define abst as follows:

$$\begin{array}{rcl} \mathit{inductive}\ \mathsf{abst} & :: & [\, bnd, expr \Rightarrow expr \,] \Rightarrow bool \\ & \Longrightarrow & \mathsf{abst}\ i\ (\lambda v.\ \mathsf{CON}\ a) \\ & \Longrightarrow & \mathsf{abst}\ i\ (\lambda v.\ v) \\ & \Longrightarrow & \mathsf{abst}\ i\ (\lambda v.\ \mathsf{VAR}\ n) \\ j < i & \Longrightarrow & \mathsf{abst}\ i\ (\lambda v.\ \mathsf{BND}\ j) \\ [\![\ \mathsf{abst}\ i\ f; \mathsf{abst}\ i\ g\ ]\!] & \Longrightarrow & \mathsf{abst}\ i\ (\lambda v.\ f\ v\ \$\ g\ v) \\ \mathsf{abst}\ (i+1)\ f & \Longrightarrow & \mathsf{abst}\ i\ (\lambda v.\ \mathsf{ABS}\ (f\ v)) \end{array}$$

Given $\mathsf{abstr} :: [\, expr \Rightarrow expr \,] \Rightarrow bool$, we set:

$$\mathsf{abstr}\ e == \mathsf{abst}\ 0\ e.$$

Equality in Isabelle/HOL is extensional, contrary to Coq. This difference affects definitions like abst and lbnd that have an argument of functional type ($expr \Rightarrow expr$). In the Coq version, we define extensional equality on type $expr$ *explicitly*. Formally, $(f =_{ext} g) == \forall x.(fx = gx)$. This equality appears in the definition of abst, for example. We first define an auxiliary predicate abst_aux defined exactly as abst above, and then define abst as:

$$\mathsf{abst}\ i\ e == \exists e'.\ e' =_{ext} e \wedge \mathsf{abst\_aux}\ i\ e'.$$

It follows directly from the inductive definition of de Bruijn expressions that the functions CON, VAR, $, and ABS are injective, with disjoint images. With the introduction of abstr, we can now also prove the following fundamental theorem:

THEOREM 1.

$$[\![\ \mathsf{abstr}\ e;\ \mathsf{abstr}\ f\ ]\!] \Longrightarrow (\mathsf{lambda}\ e = \mathsf{lambda}\ f) = (e = f)$$

which says that lambda is injective on the set of abstractions.

Hybrid also includes a definition $\mathsf{newvar} :: expr \Rightarrow var$ used to generate fresh variables.

$$\begin{array}{lcl} primrec\ \mathsf{newvar} & :: & expr \Rightarrow var \\ \mathsf{newvar}\ (\mathsf{CON}\ a) & = & 0 \\ \mathsf{newvar}\ (\mathsf{VAR}\ n) & = & n+1 \\ \mathsf{newvar}\ (\mathsf{BND}\ i) & = & 0 \\ \mathsf{newvar}\ (s\ \$\ t) & = & max\ (\mathsf{newvar}\ s)\ (\mathsf{newvar}\ t) \\ \mathsf{newvar}\ (\mathsf{ABS}\ s) & = & \mathsf{newvar}\ s \end{array}$$

In particular $(\mathsf{newvar}\ e)$ returns a variable whose index is 1 greater than the maximum value of the free variables occurring in e. Since all of our previous case studies (see [Momigliano et al. 2008]) entailed induction on closed terms or didn't require reasoning over contexts if not for substitution purposes (realized via the crucial use of cut in two-level proofs of subject reduction), we had no use for this definition, although it is central to the development of the reasoning techniques explained in this paper.

3. The Specification Logic

We use a simple SL, a sequent formulation of a fragment of second-order minimal logic with backchaining, adapted from [McDowell and Miller 2002]. The formalization presented here is taken fairly directly from [Momigliano et al. 2008]. Its syntax can be encoded directly as the datatype:

$$\begin{array}{l} datatype\ oo = \\ \quad \mathsf{tt} \mid \langle atm \rangle \mid oo\ \mathsf{and}\ oo \mid atm\ \mathsf{imp}\ oo \mid \mathsf{all}\ (uexp \Rightarrow oo) \end{array}$$

where atm is a parameter used to represent atomic predicates of the OL and $\langle\_\rangle$ coerces atoms into propositions of type oo. We use the symbol $\triangleright$ for the sequent arrow of the SL, in this case decorated with natural numbers to allow reasoning by (complete) induction on the *height* of a proof. The inference rules of the SL are represented as the inductive definition in Figure 1. For convenience we write $\Gamma \triangleright G$ if there exists an n such that $\Gamma \triangleright_n G$, and furthermore we simply write $\triangleright G$ when $\emptyset \triangleright G$. The first four clauses of the definition directly encode the introduction rules of a sequent calculus for this logic. In the last two rules, atoms are provable either by assumption or via *backchaining* over a set of Prolog-like rules, which encode the properties of the OL in question. They are encoded as an inductive definition of the predicate prog of type $[\, atm, oo \,] \Rightarrow bool$, which will be instantiated in the next section. The notation $A \longleftarrow G$ represents an instance of one of the clauses

$$\begin{array}{rcl} inductive\ \_ \rhd_\_ \_ :: [\, atm\ list, nat, oo \,] \Rightarrow bool & & \\ & \Longrightarrow & \Gamma \rhd_n \mathsf{tt} \\ [\![\ \Gamma \rhd_n G_1;\ \Gamma \rhd_n G_2\ ]\!] & \Longrightarrow & \Gamma \rhd_{n+1} (G_1\ \mathsf{and}\ G_2) \\ [\![\ (\forall x.\ \mathsf{proper}\ x \longrightarrow \Gamma \rhd_n G\ x)\ ]\!] & \Longrightarrow & \Gamma \rhd_{n+1} (\mathsf{all}\ x.\ G\ x) \\ [\![\ \{A\} \cup \Gamma \rhd_n G\ ]\!] & \Longrightarrow & \Gamma \rhd_{n+1} (A\ \mathsf{imp}\ G) \\ [\![\ A \in \Gamma\ ]\!] & \Longrightarrow & \Gamma \rhd_n \langle A \rangle \\ [\![\ A \longleftarrow G; \Gamma \rhd_n G\ ]\!] & \Longrightarrow & \Gamma \rhd_{n+1} \langle A \rangle \end{array}$$

Figure 1. Encoding of Specification Logic

of this inductive definition. The sequent calculus is parametric in those clauses and so are its meta-theoretical properties.

To reason about OLs, a small set of structural rules of the SL is proved once and for all.

THEOREM 2.

1. *Height weakening:* $[\![\ \Gamma \rhd_n G;\ n < m\ ]\!] \Longrightarrow \Gamma \rhd_m G$
2. *Context weakening:* $[\![\ \Gamma \rhd_n G;\ \Gamma \subseteq \Gamma'\ ]\!] \Longrightarrow \Gamma' \rhd_n G.$
3. *Atomic cut:* $[\![\ A, \Gamma \rhd G;\ \Gamma \rhd \langle A \rangle\ ]\!] \Longrightarrow \Gamma \rhd G.$

4. The Object Logic

The types of PCF include natural numbers, booleans, and function types. They are encoded directly as a standard datatype:

$$datatype\ tp = num \mid bl \mid tp \Rightarrow tp$$

To define the terms of PCF, we first declare the constants that belong to the enumerated datatype con:

$$\begin{array}{l} datatype\ con = cZERO \mid cTRUE \mid cFALSE \\ \quad \mid cSUCC \mid cPRED \mid cIS\_ZERO \mid cIF \\ \quad \mid cAPP \mid cABS \mid cREC \mid cTP\ tp \end{array}$$

This is the instantiation of con that we consider in the rest of the paper. Note that cTP is used to coerce an OL type to a constant in order to allow types to appear inside terms. After all, without those annotations type unicity would not hold. Recall that $uexp = = con\ expr$. We add the following definitions:

$$\begin{array}{rcl} \mathsf{0}, \mathsf{t}, \mathsf{f} & :: & uexp \\ \mathsf{suc}, \mathsf{pred}, \mathsf{is0?} & :: & uexp \Rightarrow uexp \\ \mathsf{if} & :: & [\, uexp, uexp, uexp \,] \Rightarrow uexp \\ @ & :: & [\, uexp, uexp \,] \Rightarrow uexp \\ \mathsf{abs}, \mathsf{rec} & :: & [\, tp, uexp \Rightarrow uexp \,] \Rightarrow uexp \end{array}$$

$$\begin{array}{rcl} \mathsf{0} & == & \mathsf{CON}\ cZERO \\ \mathsf{t} & == & \mathsf{CON}\ cTRUE \\ \mathsf{f} & == & \mathsf{CON}\ cFALSE \\ \mathsf{succ}\ E & == & \mathsf{CON}\ cSUCC\ \$\ E \\ \mathsf{pred}\ E & == & \mathsf{CON}\ cPRED\ \$\ E \\ \mathsf{is0?}\ E & == & \mathsf{CON}\ cIS\_ZERO\ \$\ E \\ \mathsf{if}\ E_1\ E_2\ E_3 & == & \mathsf{CON}\ cIF\ \$\ E_1\ \$\ E_2\ \$\ E_3 \\ E_1\ @\ E_2 & == & \mathsf{CON}\ cAPP\ \$\ E_1\ \$\ E_2 \end{array}$$

$$\begin{array}{rcl} \mathsf{abs}\ x^T.\ E\ x & == & \mathsf{CON}\ cABS\ \$ \\ & & \quad \mathsf{CON}\ (cTP\ T)\ \$\ (\mathsf{lambda}\ (\lambda x.\ E\ x)) \\ \mathsf{rec}\ x^T.\ E\ x & == & \mathsf{CON}\ cREC\ \$ \\ & & \quad \mathsf{CON}\ (cTP\ T)\ \$\ (\mathsf{lambda}\ (\lambda x.\ E\ x)) \end{array}$$

where abs and rec are meta-level binders and $(\mathsf{abs}\ x^T.\ E\ x)$ is an abbreviation for $(\mathsf{abs}\ T\ (\lambda x.E\ x))$, and similarly for rec.

Note again that the above are only *definitions* and by themselves would not inherit any of the properties of the constructors of a datatype. However, thanks to our package it is now possible to

prove the freeness properties of those definitions as if they were the constructors of what Isabelle/HOL would ordinarily consider an "impossible" datatype as discussed earlier. In particular, all constructors have distinct images, e.g.:

$$\mathsf{abs}\ x.\ ^T E\ x \quad \neq \quad (E_1\ @\ E_2)$$

and every *binding* constructor is *injective* on *abstractions*:

$$[\![\ \mathsf{abstr}\ E;\ \mathsf{abstr}\ E'\ ]\!] \Longrightarrow (\mathsf{rec}\ x^T.\ E\ x = \mathsf{rec}\ x^T.\ E'\ x) = (E = E')$$

Encoding an OL involves introducing a specific type for con (as was just done), instantiating the type atm, and defining the prog predicate. The atomic formulas of the OL include for this case study the PCF typing judgment and a well-formedness judgment for PCF terms. The latter is used for adequacy as discussed further below.

$$datatype\ atm = uexp : tp \mid \mathsf{isterm}\ uexp$$

The clauses for PCF typing and selected clauses for well-formedness of PCF terms are given as rules of the prog (recall the notation $\_ \longleftarrow \_$) inductive definition in Figure 2. (The omitted well-formedness clauses are similar to those that are there.) The types of 0, t, and f are axioms, and thus they only have, for the sake of uniformity, the trivial subgoal tt on the right of the arrow. The types of terms constructed from succ, pred, is0?, if, and @ depend on the types of the arguments, which are expressed as straightforward typing subgoals on the right of the arrow. The clause for abs has a similar form to the one for the untyped λ-calculus discussed in Section 1, except that the type of the bound variable is included as an argument and the encoding uses quantifiers of the SL. In addition, an abstr assumption is required for the functional argument E. The clause for rec has similar form to the clause for abs, and expresses the usual typing rule for the recursive function constructor. In Section 2, we discussed the adequacy of the encoding of λ-terms as terms of type $expr$. As discussed in [Momigliano et al. 2008, Felty and Momigliano 2008], it is also important to show that both terms and judgments of an OL are adequately encoded, and in a two-level system that the SL is adequately encoded. We refer the reader to the results discussed there, some of which can be directly applied here. The isterm predicate is an important part of adequacy for our OL. In particular, we can show that there is a bijection between closed object-level terms and terms of type $uexp$ for which the judgment $\rhd \langle \mathsf{isterm}\ \_ \rangle$ is provable. In addition, the following lemmas are an important part of showing the adequacy of the OL typing judgment.

LEMMA 3.

1. $\rhd \langle E : T \rangle \Longrightarrow \mathsf{proper}\ E$
2. $\rhd \langle E : T \rangle \Longrightarrow \rhd \langle \mathsf{isterm}\ E \rangle$

5. Formal Proof of Type Unicity

In this section, we prove that types assigned to PCF terms are unique. In particular, if $\rhd \langle E : T \rangle$ and $\rhd \langle E : T' \rangle$, then T and T' are the same type. As is quite common in formal proofs about semantics of programming languages, we need to consider a more general statement involving a non-empty context. This kind of generalization is needed particularly when the induction is on a deduction judgment in which some rules (such as the typing rules for the abs and rec operators) involve adding a new assumption to the context.[2] We start by formulating the induction hypothesis or "context invariant" used in this proof. In particular, we introduce $\mathsf{cxtInv} :: atm\ list \Rightarrow bool$, and define:

$$\begin{array}{l}\mathsf{cxtInv}\ \Gamma = \\ (e : t) \in \Gamma \Longrightarrow \\ \quad \exists v.\ e = \mathsf{VAR}\ v \wedge (\forall n\ t'.\ \Gamma \rhd_n \langle \mathsf{VAR}\ v : t' \rangle \Longrightarrow t = t')\end{array}$$

This invariant expresses that every term occurring in a typing judgment in the context is a variable, and if a variable occurs more than once associated with more than one type, then these types must all be the same. Note that this definition is just a restatement of type unicity specific to variables inside the context.

One of the key insights of our case study is that using this context invariant requires reasoning about a new variable that does not already occur in the context. We start by building up the definitions and lemmas that we need for such reasoning. First, we extend the definition of newvar to atoms and contexts.

$$\begin{array}{rcl} primrec\ \mathsf{nvA} & :: & atm \Rightarrow var \\ \mathsf{nvA}\ (e : t) & = & \mathsf{newvar}\ e \\ \mathsf{nvA}\ (\mathsf{isterm}\ e) & = & \mathsf{newvar}\ e \end{array}$$

$$\begin{array}{rcl} primrec\ \mathsf{nvC} & :: & atm\ list \Rightarrow var \\ \mathsf{nvC}\ [] & = & 0 \\ \mathsf{nvC}\ (a :: l) & = & max\ (\mathsf{nvA}\ a)\ (\mathsf{nvC}\ l) \end{array}$$

The nvC function simply folds max over a context. The lemmas below follow from these definitions.

LEMMA 4.

1. $\mathsf{nvC}\ (((\mathsf{VAR}\ v) : T) :: \Gamma) > v$
2. $((\mathsf{VAR}\ v) : T) \in \Gamma \Longrightarrow (\mathsf{nvC}\ \Gamma) > v$

The first one states that the natural number associated with a new context variable (generated by newvar) is greater than the variable occuring in the typing judgment at the head of the context. Its proof follows fairly directly from definitions. Part (2) states that a newly generated context variable has a value greater than any variables already occurring in the context. Its proof follows from (1), an induction on lists, and arithmetic on natural numbers. The main result we need in the rest of the development is a direct corollary of (2): that a new variable is distinct from any already occurring in the context, i.e., $((\mathsf{VAR}\ v) : T) \in \Gamma \Longrightarrow (\mathsf{nvC}\ \Gamma) \neq v$.

Note that the definition of nvA depends on the OL, and this dependence comes from the instantiation of atm. For a certain class of OLs, we can describe a general way to define nvA: consider all arguments $e_1, \ldots, e_n$ of type $uexp$ in atom a; define $(\mathsf{nvA}\ a)$ to be the maximum of $(\mathsf{newvar}\ e_1), \ldots, (\mathsf{newvar}\ e_n)$. This definition works for OLs where every argument of an atomic predicate is either of type $uexp$ (and involved in the calculation) or of some type independent of $uexp$ (and not involved in the calculation, e.g., tp in our example). For such OLs, the proof of the lemma corresponding to Lemma 4 should be easy to automate.

The main lemma in the proof of type unicity expresses that the context invariant is *preserved* when adding a new (fresh) variable (and its type) to the context.

LEMMA 5. $\mathsf{cxtInv}\ \Gamma \Longrightarrow \mathsf{cxtInv}\ ((\mathsf{VAR}\ (\mathsf{nvC}\ \Gamma)) : T) :: \Gamma$.

It is easy to see why this lemma holds; if the invariant holds of Γ and we add a typing assumption about a new variable, we guarantee that there is no other typing assumption about this variable already in Γ. The only way to build a deduction that assigns a type to this new variable is to use the SL's axiom rule, namely the second to

[2] Although both adequacy lemmas above seem to follow this pattern, they are in fact *non*-examples of the techniques that we are discussing here. In the first case, in the critical *abs* rule the induction is not needed as it is a simple (non-inductive) fact that $\mathsf{abstr}\ E \Longrightarrow \mathsf{proper}\ (\mathsf{abs}\ x^T.\ E\ x)$. The second lemma does require generalization, but this does not offer any new insight since the isterm predicate is just the typing judgment minus the types.

$$
\begin{array}{rcl}
\mathit{inductive}\ \_ \longleftarrow \_ & :: & [\,\mathit{atm}, \mathit{oo}\,] \Rightarrow \mathit{bool} \\
& \Longrightarrow & \mathsf{0} : \mathit{num} \longleftarrow \mathsf{tt} \\
& \Longrightarrow & \mathsf{t} : \mathit{bl} \longleftarrow \mathsf{tt} \\
& \Longrightarrow & \mathsf{f} : \mathit{bl} \longleftarrow \mathsf{tt} \\
& \Longrightarrow & (\mathsf{succ}\ E) : \mathit{num} \longleftarrow \langle E : \mathit{num}\rangle \\
& \Longrightarrow & (\mathsf{pred}\ E) : \mathit{num} \longleftarrow \langle E : \mathit{num}\rangle \\
& \Longrightarrow & (\mathsf{is0?}\ E) : \mathit{bl} \longleftarrow \langle E : \mathit{num}\rangle \\
& \Longrightarrow & (\mathsf{if}\ E_1\ E_2\ E_3) : T \longleftarrow \langle E_1 : \mathit{bl}\rangle\ \mathsf{and}\ \langle E_2 : T\rangle\ \mathsf{and}\ \langle E_3 : T\rangle \\
& \Longrightarrow & (E_1\ @\ E_2) : T \longleftarrow \langle E_1 : (T' \Rightarrow T)\rangle\ \mathsf{and}\ \langle E_2 : T'\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] & \Longrightarrow & (\mathsf{abs}\ x^T.\, E\ x) : (T \Rightarrow T') \longleftarrow \mathsf{all}\, x.\, (x : T)\ \mathsf{imp}\ \langle (E\ x) : T'\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] & \Longrightarrow & (\mathsf{rec}\ x^T.\, E\ x) : T \longleftarrow \mathsf{all}\, x.\, (x : T)\ \mathsf{imp}\ \langle (E\ x) : T\rangle \\
& & \\
& \Longrightarrow & \mathsf{isterm}\ 0 \longleftarrow \mathsf{tt} \\
& \Longrightarrow & \mathsf{isterm}\ (\mathsf{succ}\ E) \longleftarrow \langle \mathsf{isterm}\ E\rangle \\
& \Longrightarrow & \mathsf{isterm}\ (\mathsf{if}\ E_1\ E_2\ E_3) \longleftarrow \langle \mathsf{isterm}\ E_1\rangle\ \mathsf{and}\ \langle \mathsf{isterm}\ E_2\rangle\ \mathsf{and}\ \langle \mathsf{isterm}\ E_3\rangle \\
& \Longrightarrow & \mathsf{isterm}\ (E_1\ @\ E_2) \longleftarrow \langle \mathsf{isterm}\ E_1\rangle\ \mathsf{and}\ \langle \mathsf{isterm}\ E_2\rangle \\
[\![\ \mathsf{abstr}\ E\ ]\!] & \Longrightarrow & \mathsf{isterm}\ (\mathsf{abs}\ x^T.\, E\ x) \longleftarrow \mathsf{all}\, x.\, (\mathsf{isterm}\ x)\ \mathsf{imp}\ \langle \mathsf{isterm}\ (E\ x)\rangle \\
& \vdots &
\end{array}
$$

Figure 2. OL clauses encoding typing and well-formedness of PCF terms

last clause in Figure 1, and thus the new variable must have unique type T.

The generalized form of type unicity is expressed in the following theorem.

THEOREM 6.

$$[\![\ \mathsf{cxtInv}\ \Gamma;\ \Gamma \rhd_n \langle E : T\rangle;\ \Gamma \rhd_{n'} \langle E : T'\rangle\ ]\!] \Longrightarrow T = T'.$$

Proof The proof is by complete induction on n, the height of the first typing derivation. The induction hypothesis IH is:

$$\begin{array}{l}\forall m < n, n', \Gamma, E, T, T'. \\ \quad \mathsf{cxtInv}\ \Gamma \longrightarrow \Gamma \rhd_m \langle E : T\rangle \longrightarrow \Gamma \rhd_{n'} \langle E : T'\rangle \longrightarrow T = T'.\end{array}$$

A derivation of $\Gamma \rhd_n \langle E : T\rangle$ must end with the atom rules, i.e., one of the last two rules of Figure 1. Applying (standard) inversion breaks the proof into the following two cases.

$$\begin{array}{l}[\![\ IH[i+1/n];\ \mathsf{cxtInv}\ \Gamma;\ \Gamma \rhd_{i+1} \langle E : T\rangle; \\ \quad (E : T) \longleftarrow G;\ \Gamma \rhd_{n'} \langle E : T'\rangle\ ]\!] \Longrightarrow T = T' \\ [\![\ IH;\ \mathsf{cxtInv}\ \Gamma;\ (E : T) \in \Gamma; \\ \quad \Gamma \rhd_{n'} \langle E : T'\rangle\ ]\!] \Longrightarrow T = T'\end{array}$$

The second follows from the context invariant. In the first, by applying inversion to $(E : T) \longleftarrow G$, the proof is further broken down into 10 cases, one for each of the typing clauses of Figure 2.

The 8 cases for the clauses that do not involve generic judgments follow fairly directly from the context invariant and the induction hypothesis. We show the application case:

$$\begin{array}{l}[\![\ IH[i+1/n];\ \mathsf{cxtInv}\ \Gamma;\ \Gamma \rhd_{i+1} \langle (E_1 @\ E_2) : T\rangle; \\ \quad \Gamma \rhd_i \langle E_1 : (U \Rightarrow T)\rangle\ \mathsf{and}\ \langle E_2 : U\rangle; \\ \quad \Gamma \rhd_{n'} \langle (E_1 @ E_2) : T'\rangle\ ]\!] \Longrightarrow T = T'\end{array}$$

Inversion on the second to last premise, followed by inversion on the last premise gives us two subgoals:

$$\begin{array}{l}[\![\ IH[i'+2/n];\ \mathsf{cxtInv}\ \Gamma;\ \Gamma \rhd_{i'+2} \langle (E_1 @\ E_2) : T\rangle; \\ \quad \Gamma \rhd_{i'} \langle E_1 : (U \Rightarrow T)\rangle;\ \Gamma \rhd_{i'} \langle E_2 : U\rangle; \\ \quad \Gamma \rhd_j \langle E_1 : (U' \Rightarrow T')\rangle;\ \Gamma \rhd_j \langle E_2 : U'\rangle\ ]\!] \Longrightarrow T = T' \\ [\![\ IH[i'+2/n];\ \mathsf{cxtInv}\ \Gamma; \\ \quad \Gamma \rhd_{i'+2} \langle (E_1 @\ E_2) : T\rangle; \ldots; \\ \quad ((E_1 @ E_2) : T') \in \Gamma\ ]\!] \Longrightarrow T = T'\end{array}$$

The first one follows by the induction hypothesis applied to

$$\Gamma \rhd_{i'} \langle E_1 : (U \Rightarrow T)\rangle \text{ and } \Gamma \rhd_j \langle E_1 : (U' \Rightarrow T')\rangle,$$

and the second one is proved by contradiction because the last premise violates the context invariant, which states that only variables occur in Γ with their types.

The case for typing an abstraction is:

$$\begin{array}{l}[\![\ IH[i+1/n];\ \mathsf{cxtInv}\ \Gamma;\ \mathsf{abstr}\ E; \\ \quad \Gamma \rhd_{i+1} \langle (\mathsf{abs}\ x^{T_1}.\, E\ x) : (T_1 \Rightarrow T_2)\rangle; \\ \quad \Gamma \rhd_i \mathsf{all}\, x.\, (x : T_1\ \mathsf{imp}\ \langle (E\ x) : T_2\rangle); \\ \quad \Gamma \rhd_{n'} \langle (\mathsf{abs}\ x^{T_1}.\, E\ x)\ : T'\rangle\ ]\!] \Longrightarrow (T_1 \Rightarrow T_2) = T'\end{array}$$

Again applying inversion on the second to last premise, followed by inversion on the last gives us two subgoals, where the second one violates the context invariant just as in the application case. The remaining goal is:

$$\begin{array}{l}[\![\ IH[i'+2/n];\ \mathsf{cxtInv}\ \Gamma;\ \mathsf{abstr}\ E; \\ \quad \mathsf{abstr}\ E';\ (\mathsf{lambda}\ E = \mathsf{lambda}\ E'); \\ \quad \Gamma \rhd_{i'+2} \langle (\mathsf{abs}\ x^{T_1}.\, E\ x) : (T_1 \Rightarrow T_2)\rangle; \\ \quad \forall x.\ \mathsf{proper}\ x \longrightarrow \Gamma \rhd_{i'} x : T_1\ \mathsf{imp}\ \langle (E\ x) : T_2\rangle; \\ \quad \forall x.\ \mathsf{proper}\ x \longrightarrow \Gamma \rhd_j x : T_1\ \mathsf{imp}\ \langle (E'\ x) : T_3\rangle\ ]\!] \\ \Longrightarrow (T_1 \Rightarrow T_2) = (T_1 \Rightarrow T_3)\end{array}$$

Using Theorem 1, we can conclude that $E = E'$, reducing the above goal to:

$$\begin{array}{l}[\![\ IH[i' + 2/n];\ \mathsf{cxtInv}\ \Gamma;\ \mathsf{abstr}\ E; \ldots; \\ \quad \forall x.\ \mathsf{proper}\ x \longrightarrow \Gamma \rhd_{i'} x : T_1\ \mathsf{imp}\ \langle (E\ x) : T_2 \rangle; \\ \quad \forall x.\ \mathsf{proper}\ x \longrightarrow \Gamma \rhd_{j} x : T_1\ \mathsf{imp}\ \langle (E\ x) : T_3 \rangle\ ]\!] \\ \Longrightarrow (T_1 \Rightarrow T_2) = (T_1 \Rightarrow T_3)\end{array}$$

It is at this point that we introduce a new variable that does not occur in Γ. Let Y be $(\mathsf{VAR}\ (\mathsf{nvC}\ \Gamma))$. By definition variables are proper, and so we can conclude $(\mathsf{proper}\ Y)$. Instantiating x with this new variable gives us:

$$\begin{array}{l}[\![\ IH[i' + 2/n];\ \mathsf{cxtInv}\ \Gamma;\ \mathsf{abstr}\ E; \ldots; \\ \quad \Gamma \rhd_{i'} Y : T_1\ \mathsf{imp}\ \langle (E\ Y) : T_2 \rangle; \\ \quad \Gamma \rhd_{j} Y : T_1\ \mathsf{imp}\ \langle (E\ Y) : T_3 \rangle\ ]\!] \\ \Longrightarrow (T_1 \Rightarrow T_2) = (T_1 \Rightarrow T_3)\end{array}$$

Applying inversion once more on these instantiated hypotheses, the above goal reduces to:

$$\begin{array}{l}[\![\ IH[i'' + 3/n];\ \mathsf{cxtInv}\ \Gamma;\ \mathsf{abstr}\ E; \ldots; \\ \quad (Y : T_1), \Gamma \rhd_{i''} \langle (E\ Y) : T_2 \rangle; \\ \quad (Y : T_1), \Gamma \rhd_{j'} \langle (E\ Y) : T_3 \rangle\ ]\!] \\ \Longrightarrow (T_1 \Rightarrow T_2) = (T_1 \Rightarrow T_3)\end{array}$$

We can apply Lemma 5 to premise $(\mathsf{cxtInv}\ \Gamma)$ to conclude

$$\mathsf{cxtInv}\ (Y : T_1) :: \Gamma.$$

It is now possible to apply the induction hypothesis, instantiating Γ with $((Y : T_1) :: \Gamma)$ and using hypotheses:

$$(Y : T_1), \Gamma \rhd_{i''} \langle (E\ Y) : T_2 \rangle \text{ and } (Y : T_1), \Gamma \rhd_{j'} \langle (E\ Y) : T_3 \rangle$$

to conclude that $T_2 = T_3$, from which the desired result follows immediately.

The case for typing recursion is simple and doesn't involve reasoning about an extended context:

$$\begin{array}{l}[\![\ IH[i + 1/n];\ \mathsf{cxtInv}\ \Gamma;\ \mathsf{abstr}\ E; \\ \quad \Gamma \rhd_{i+1} \langle (\mathsf{rec}\ x^T.\, E\ x) : T \rangle; \\ \quad \Gamma \rhd_{i} \mathsf{all}\, x. (x : T\ \mathsf{imp}\ \langle (E\ x) : T \rangle); \\ \quad \Gamma \rhd_{n'} \langle (\mathsf{rec}\ x^T.\, E\ x)\ : T' \rangle\ ]\!] \Longrightarrow T = T'\end{array}$$

In this case, inversion on the last premise gives two subgoals. The one that corresponds to the axiom rule results in the premise $((\mathsf{rec}\ x^T.\, E\ x)\ : T') \in \Gamma$, which, similar to the other cases, contradicts the context invariant. The other one corresponds to backchaining, and a further inversion on the typing clause for rec in Figure 2 directly gives $T = T'$. This completes the proof.

The main type unicity result, expressed as the corollary below, follows immediately.

COROLLARY 7. $[\![\rhd \langle E : T \rangle;\ \rhd\ \langle E : T' \rangle\ ]\!] \Longrightarrow T = T'$.

6. Related Work

In this section, we only discuss proofs of type unicity (TU) in the literature that use full HOAS. Approaches we neglect here include, among others, those based on a nameless representation [McKinna and Pollack 1999] or based on nominal techniques [Pitts 2003]. For a fuller discussion of related work please see [Felty and Momigliano 2008]. We acknowledge that our notion of newvar has an obvious nominal flavor and we plan to study this connection in the future and possibly learn much in view of automation from the *nominal datatype package* developed on top of Isabelle/HOL [Nominal Methods Group 2009].

TU in $FO\lambda^{\Delta IN}$**.** One of the early formal TU proofs that involves representing and reasoning with HOAS was done in $FO\lambda^{\Delta IN}$ [McDowell 1997] with the Pi proof editor [Eriksson 1994]. This proof is far from satisfactory as we will see in a second, but we were able to port it fairly directly to Hybrid. It uses the following context invariant:

$$\mathsf{cxtInv1}\ \Gamma == (e : t) \in \Gamma \Longrightarrow \rhd \langle e : t' \rangle \Longrightarrow t = t'.$$

Compare the above definition to the one we used in Section 5:

$$\begin{array}{l}\mathsf{cxtInv}\ \Gamma == \\ (e : t) \in \Gamma \Longrightarrow \\ \quad \exists v.\ e = \mathsf{VAR}\ v \wedge (\forall n\ t'.\ \Gamma \rhd_n \langle \mathsf{VAR}\ v : t' \rangle \Longrightarrow t = t').\end{array}$$

The invariant in the $FO\lambda^{\Delta IN}$ proof expresses the same thing, except that it does not restrict terms appearing in Γ to be variables. In fact, the $FO\lambda^{\Delta IN}$ encoding of PCF that was used for this proof did not have object-level variables and so reasoning was restricted to closed terms. In the case for typing an abstraction, instead of introducing a new variable that does not appear in Γ to instantiate x, as we did in the proof of Theorem 6, the completely *ad hoc* closed term $(\mathsf{rec}\ x^T.\, x)$ of type T was used. Note that adding this term to Γ does preserve the invariant, i.e., $\mathsf{cxtInv1}\ ((\mathsf{rec}\ x^T.\, x) : T) :: \Gamma$ is provable. Note also that this proof does not work unless the rec operator is part of the object language. This proof technique, simply put, does not solve the issue of reasoning with open terms in any generality; its main use was to motivate the need for a more general way to handle contexts, which led to the so-called *eigenvariable representation*, introduced in $FO\lambda^{\Delta IN}$ [McDowell 1997, Miller and Tiu 2002], one of the early techniques developed for handling variable contexts in HOAS. The idea consists of considering the whole sequent as bound by the list of the current eigenvariables. It included a higher-order representation of lists of variables and machinery to manipulate them. This syntax gets very heavy very quickly and its internalization in the proof-theory of the ∇ quantifier was a major accomplishment, soon adopted in successors to $FO\lambda^{\Delta IN}$ and in the Abella system (see below).

In our earlier work on two-level reasoning [Felty 2002], the type unicity theorem from $FO\lambda^{\Delta IN}$ is mentioned and in fact proven in Coq, but no proof is given in that paper. For the sake of completeness we ported this proof to Hybrid as well; it uses a context invariant whose content is captured by the following definition:

$$\mathsf{cxtInv2}\ \Gamma == (e : t) \in \Gamma \Longrightarrow \exists E.\ e = (\mathsf{rec}\ x^t.\, (E\ x)).$$

Note that this proof builds the dependence on rec right into the context invariant. In fact, this invariant is stronger than the previous one—we can prove $(\mathsf{cxtInv2}\ \Gamma) \Longrightarrow (\mathsf{cxtInv1}\ \Gamma)$—and therefore at least as unsatisfactory.

TU in Abella. Because of the adoption of a common two-level architecture, it is enlightening to compare our proof with what one can do in the Abella system [Gacek 2008]. As a matter of fact, Abella supports two different meta-logics, LG^{ω} [Tiu 2007] and a superset, $\mathcal{G}$ [Gacek et al. 2008]. Both support (co)inductive partial definitions [Momigliano and Tiu 2003] and the ∇ quantifier, in the stronger (w.r.t. $FO\lambda^{\Delta\nabla}$ [Miller and Tiu 2005]) "nominal-ish" version. $\mathcal{G}$ extends LG^{ω} by allowing ∇'s in the *head of definitions*, a feature that simplifies context invariants needed in generic proofs. In both logics, the TU proof goes through by:

1. defining (in the fixed point sense, concrete syntax "`:=`") a notion of well-formed context;
2. proving the "regularity" of the context, i.e., showing that for every atom $e : \tau$ occurring in the context, e must only be a variable/nominal constant/fresh name;
3. proving TU for the elements of the context;

4. using the latter, as we do, as a helper lemma to establish the main result by taking care of the axiom case in the induction on the SL derivation.

Note of course that in both encodings, the SL universal quantification is mapped to ∇.

In Abella's concrete syntax, provability in the SL is denoted by brackets and the numerical information about the height of the derivation is kept symbolically, a user-interface that Hybrid should adopt. For the sake of conciseness we restrict ourselves to the abstraction/application part of PCF. In particular a well-formed context is defined in LG^{ω} as follows, where "`of`" denotes OL typability:

```
Def ctx nil.
Def ctx (of X T :: L) :=
  (forall M N, X = M @ N -> false) /\
  (forall T R, X = abs T R -> false) /\
  (forall T', member (of X T') L -> false) /\
   ctx L.
```

Hence a context is forced to be a list of distinct atoms `(of X T)` with unique types by ruling out all other possibilities.[3] Having done that, the user still needs to *prove* that these impossibilities hold for every constructor, e.g.:

```
Thm: ctx L -> member (of (M @ N) T) L -> false.
```

Clearly this does not scale too well even to a small language such as PCF. Note also that, in our proof, this is handled directly by the invariant; see for example the contradiction subcase in the application case of Theorem 6. Another arguably rather *ad hoc* fact about the non-occurrence of nominal constants with a particular kind of scope in a list is needed.

```
Thm nominal_absurd:
    nabla x, member (of x (T x)) L -> false.
```

In fact, $\mathcal{G}$ allows the user to escape from some of this boilerplate. As it is possible to define very simple yet powerful notions of being a *name* and of *freshness*; thanks to the possibility of ∇ in the head of definitions, a simple lemma proved once and for all subsumes some of the aforementioned required results in LG^{ω}:

```
Def nabla x, name x.

Def nabla x, fresh x E.

Thm: member E L -> fresh X L -> fresh X E.
```

Now a context can be defined more succinctly; the `fresh X L` condition could even be removed to show the cunning resemblance with our Lemma 5.

```
Def ctx nil.
Def ctx (of X T :: L) := fresh X L /\ ctx L.
```

Only one technical lemma is required, connected to point 2. of the above methodology:

```
Thm: ctx L -> member (of E T) L -> name E.
```

In both proofs, a final lemma states TU for the element of the well-formed context:

```
Thm: ctx L -> member (of E T1) L ->
              member (of E T2) L -> T1 = T2.
```

Finally, the statement corresponding to Theorem 6:

```
Thm: ctx L -> {L |- of E T1} ->
              {L |- of E T2} -> T1 = T2.
```

Even a superficial comparison of proof scripts shows that Abella's proofs are shorter and neater than in Hybrid. This is no surprise as Abella is a small dedicated system, tailored to HOAS encodings with a very simple, though effective, tactic language. The flip side is that everything else needs to be encoded directly, while we have the luxury to rely on Isabelle/HOL and Coq. For example instead of a call to the arithmetic tactic, the Abella user needs to encode Peano axioms as logic programs and establish a large library of the required lemmas. Note also that while $\mathcal{G}$ is based on (monomorphic) simple types, judgments in Abella are untyped, hence adequacy has to be enforced by predicates, as we do.

TU in Twelf. As well-known, in the Twelf methodology [Pfenning and Schürmann 1999] the LF type theory is used to encode OLs as judgments and to specify meta-theorems as relations (type families) among them; a logic programming-like interpretation provides an operational semantics to those relations, so that an external check for totality (incorporating termination, well-modedness, and coverage checking [Schürmann and Pfenning 2003, Pientka 2005]) verifies that the given relation is indeed a realizer for that (meta)theorem.

The encoding of the typing relation is analogous to ours (and Abella's), yet, as Twelf is an intentionally weak framework, does not need to be encapsulated in a SL layer. We recall that curly brackets denote the dependent product and that Twelf's type reconstruction allows the user to omit many arguments.

```
tp_abs : of (abs E) (T1 => T2)
          <- ({x:exp} of x T1 -> of (E x) T2).
tp_app : of (E1 @ E2) T1
          <- of E1 (T2 => T1)
          <- of E2 T2.

%block tp_var : some {T:tp}
                block {x:exp} {u:of x T}.
%worlds (tp_var) (of E T).
```

The difference between reasoning on open or closed terms emerges here with the notion of *regular worlds*: in fact every time the clause `tp_abs` is invoked, it introduces a new parameter `x:exp` and a new assumption `u: of x T` for some T. This regularity is declared with `blocks` and `worlds` and the totality checker will use this information in proofs with non-empty contexts.

Because type equality is, in this case study, simply the identity, we choose a "shallow" encoding of it as an identity type family over OL types.

```
eq : tp -> tp -> type.
refl: eq T T.
```

We then need some lemmas about equality, such as congruence and inversion w.r.t. the type constructor(s). Although these proofs are immediate, they cannot be delegated to the system, in contrast with Abella and Hybrid.

```
id_arr_cong: eq T1 S1 -> eq T2 S2 ->
             eq (T1 => T2) (S1 => S2) -> type.

id1 : id_arr_cong refl refl refl.

id_arr_inv: eq (T1 => T2) (S1 => S2) ->
            eq T1 S1 -> eq T2 S2 -> type.

id2 : id_arr_inv refl refl refl.
```

[3] Note that, in additional contrast, our invariant lets the context contain more than one occurrence of a variable as long as all are assigned the same type.

Now we can declare the higher-order type family corresponding to TU and specify every case in the proof as an inhabitant of such a family. We also add directives for totality checking:

```
tp_uniq: {E:exp} {T1:tp} {T2:tp}
          of E T1 -> of E T2 -> eq T1 T2 -> type.

%mode tp_uniq +E +P1 +P2 -R

tu_var: tp_uniq _ P P refl.

tu_abs: tp_uniq _ (tp_abs P1) (tp_abs P2) EqAbs
             <- ({x:exp} {u:of x T1}
                 tp_uniq _ (P1 x u) (P2 x u) Eq)
             <- id_arr_cong refl Eq EqAbs.

tu_app: tp_uniq _ (tp_app P1 P2) (tp_app P3 P4) Eq@
             <- tp_uniq _ P2 P4 EE2
             <- tp_uniq _ P1 P3 EE1
             <- id_arr_inv EE2 EE1 Eq@.

%worlds (tp_var) (tp_uniq _ _ _ _).
%terminates P1 (tp_uniq  _ P1  _ _).
%covers  tp_uniq +E +P1 +P2 -R.
```

Note that although OL variables have no independent representation, we still have to find an inhabitant (`tu_var`) corresponding to the case in the informal proof when the two given derivation are simply the variable axiom. By construction they must be the same and so we return the identity, i.e., `refl: eq T T`. In the `abs` case `P1` and `P2` are higher-order functions parametric in the eigenvariable `x` and hypothetical in `u:of x T1`. Congruence combines the derivation obtained by IH to obtain a derivation of `EqAbs: eq (T1 => T2)(T1 =>T3))`. The case for application is similar but uses inversion to build the required identity proof. Since the `abs` case extends the context, it requires a world declaration. Finally, the relation is proved to be a total function by termination and coverage checking.

This is extremely elegant and terse. However, world checking is far from simple as worlds have some delicate structural properties (weakening, no exchange, strengthening) [Harper and Licata 2007]. It may also be noted that worlds are in a sense extra-logical w.r.t. the LF type theory.[4] Further, as coverage checking is undecidable, the algorithm approximates it, yielding sometimes false positives, whose error messages are tricky to understand. It is difficult (and an object of current work) to compare in a meaningful way a traditional tactic-based system such as Hybrid with a proof-checker plus static analysis system such as Twelf. One has to appreciate the succinctness of Twelf encodings and the automation of totality checking. Still, an idea such as newvar is arguably easy to understand for anyone familiar with the problem of bound variable names and renaming.

7. Conclusion and Future Work

We have presented an approach to reasoning inductively on generic judgments with the Hybrid system, which provides additional support for reasoning about objects encoded using HOAS. Since our architecture is based on a very small set of theories that definitionally builds an HOAS meta-language on top of a standard proof-assistant, this allows us to do without any axiomatic assumptions, in particular freeness of HOAS constructors and extensionality properties at higher types, which in our setting are theorems. The additional support we provide for inductive reasoning with generic judgments adds a significant amount of new reasoning power with a small amount of new definitions and lemmas. Arguably, these definitions and lemmas are fairly simple. Furthermore, various forms of automated support available in such proof assistants, such as tactic-style reasoning and decision procedures, are readily available and can be augmented with support specific to reasoning about HOAS specifications.

Note that by using a well-understood logic and system, and working in a purely definitional way, we avoid the need to justify *consistency* by syntactic or semantic means. For example, we do not need to show a cut-elimination theorem for a new logic as in [Gacek et al. 2008],[5] nor prove results such as strong normalization of calculi of the $\mathcal{M}_\omega$ family or about the coverage theory behind Twelf [Harper and Licata 2007, Schürmann and Pfenning 2003]. Hence our proofs are easier to trust, as far as one trusts Isabelle/HOL and Coq.

Future work must include more extended case studies to demonstrate that the new infrastructure applies to a large class of proofs. For example, we hope to show that the techniques described here can be used fairly directly to complete a formal proof of the POPLMARK challenge [Aydemir et al. 2005]. The proof of reflexivity of subtyping for System F with subtypes involves induction on a (sub)typing relation that is similar to the one used here. Transitivity is definitely harder. One further issue is investigating whether we can approximate the extraordinarily elegant encoding offered in [Pientka 2007]. Indeed, another thread we are planning to pursue is the use of our framework to aid in gaining a better understanding and "popularization" of Twelf proofs, where Hybrid would work as as the target of a sort of "compilation" of such proofs into the well-understood higher-order logic of Isabelle/HOL.

More in-depth comparisons with nominal logic ideas such as freshness and the Gabbay-Pitts quantifier are in order. On the practical side, we envision developing a package similar in spirit to the nominal datatype package for Isabelle/HOL [Nominal Methods Group 2009]. For Hybrid, such a package would automatically supply a variety of support from a user specification of an OL, such as validity predicates like isterm, a series of theorems expressing freeness of the constructors of such a "type", namely injectivity and distinctness theorems, and automated generation of the definitions and lemmas related to newvar. To work at two levels, such a package would include a number of pre-compiled SLs (including cut-elimination proofs and other properties) as well as some lightweight tactics to help with two-level inference.

Acknowledgments

The first author's research is supported in part by the Natural Sciences and Engineering Research Council of Canada. We'd like to thanks Dan Licata for his help with Twelf error messages.

References

Simon Ambler, Roy L. Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In Carreño et al. [2002], pages 13–30.

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 Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in*

[4] We are aware that they can be justified in terms of meta-logics over LF, see the seminal [Schürmann 2000].

[5] Recall that Abella, as a system, supports more than natural number induction of the $\mathcal{G}$ logic—incidentally, nested induction is crucial to Gacek's solution of the first part of the POPLMARK challenge [Aydemir et al. 2005]. A demonstration of cut-elimination requires extending the already complex proof for the *Linc* logic [Momigliano and Tiu 2003].

Higher Order Logics, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

Victor Carreño, César Muñoz, and Sofiène Tashar, editors. *15th International Conference on Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, 2002. Springer.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *2nd International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995.

Lars-Henrik Eriksson. Pi: an interactive derivation editor for the calculus of partial inductive definitions. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 821–825. Springer, 1994.

Amy P. Felty. Two-level meta-reasoning in Coq. In Carreño et al. [2002], pages 198–213.

Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *CoRR*, abs/0811.4367, 2008.

Andrew Gacek. The Abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *23rd Annual IEEE Symposium on Logic in Computer Science*, pages 33–44. IEEE Computer Society, 2008.

Elsa L. Gunter. Why we can't have SML-style datatype declarations in HOL. In Luc J. M. Claesen and Michael J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, volume A-20, pages 561–568. North-Holland/Elsevier, 1992.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer, 2001.

Hybrid Group. Hybrid: A package for higher-order syntax in Isabelle and Coq. www.hybrid.dsi.unimi.it, 2009.

Raymond McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, 1997.

Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. Comput. Log.*, 3(1):80–136, January 2002.

James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reasoning*, 23(3–4):373–409, 1999.

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

Dale Miller and Alwen Fernanto Tiu. Encoding generic judgments. In Manindra Agrawal and Anil Seth, editors, *22nd Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2002.

Alberto Momigliano and Alwen Fernanto Tiu. Induction and co-induction in sequent calculus. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2003.

Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-level Hybrid: A system for reasoning using higher-order abstract syntax. *Electr. Notes Theor. Comput. Sci.*, 196:85–93, 2008.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

Nominal Methods Group. Nominal Isabelle. isabelle.in.tum.de/nominal/, 2009.

Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 1994.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.

Brigitte Pientka. Verifying termination and reduction properties about higher-order logic programs. *J. Autom. Reasoning*, 34(2):179–207, 2005.

Brigitte Pientka. Proof pearl: The power of higher-order encodings in the logical framework lf. In Klaus Schneider and Jens Brandt, editors, *20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2007.

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

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

Carsten Schürmann. A type-theoretic approach to induction with higher-order encodings. In Robert Nieuwenhuis and Andrei Voronkov, editors, *8th International Conference Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, pages 266–281. Springer, 2001.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In David A. Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2003.

Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121 (1-2): 411–440, 1993. doi: http://dx.doi.org/10.1016/0304-3975(93)90095-B.

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