The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

LICS'06 and IJCAR'06 Workshop

LFMTP'06
Logical Frameworks and Meta-Languages: Theory and Practice

August 16th, 2006

Proceedings

Editors:
Alberto Momigliano & Brigitte Pientka

Contents

Preface

This volume contains the papers presented at LFMTP'06, the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice. LFMTP'06 merges two previous workshop series: LFM (workshop on Logical Frameworks and Meta-languages) and MERλIN (workshop on MEchanized Reasoning about Languages with variable BIndingIN). It was organized by the editors of this volume.

There were 20 submissions of which the committee decided to accept 10. The programme also includes one invited talk by Professor Gordon Plotkin, LFCS, University of Edinburgh. An abstract of this talk is included in the Proceedings as well.

Logical frameworks and meta-languages form a common substrate for representing, implementing, and reasoning about a wide variety of deductive systems of interest in logic and computer science. Their design and implementation and their applications, for example, to proof-carrying code have been the focus of considerable research over the last decade.

The broad subject areas of LFMTP'06 are:

- The automation and implementation of the meta-theory of programming languages and related calculi, particularly work which involves variable binding and fresh name generation.
- The theoretical and practical issues concerning the encoding of variable binding and fresh name generation, especially the representation of, and reasoning about, datatypes defined from binding signatures.
- Case studies of meta-programming, and the mechanization of the meta-theory of programming languages and calculi, in particular focusing on experiences with encoding programming languages theory and instances of proof-carrying code or proof-carrying authorization.

The program committee of LFMTP'06 consisted of

- Andrew Appel, Princeton University
- Thierry Coquand, Goteborg University
- Martin Hofmann, LMU Munich
- Furio Honsell, University of Udine
- Dale Miller, Inria Futurs
- Alberto Momigliano, University of Edinburgh
- Brigitte Pientka, McGill University
- Andrew Pitts, Cambridge University.

The editors would like to thank the other committee members for their input into all stages of the organization of LFMTP'06. The papers were refereed by the program committee and by the following outside referees, whose help is also gratefully acknowledged. Special thanks are also due to Marino Miculan, he knows why.

Jeremy Avigad
James Brotherston
Chad Brown
Alberto Ciaffaglione
Roy Crole
Jamie Gabbay
Fabio Gadducci
Matthew Lakin
Marino Miculan
Ivan Scagnetto
Ulrich Schöpp
Carsten Schürmann
Alwen Tiu
Christian Urban

LFMTP'06 is held on August 16th, 2006 in association with FLOC'06, the 2006 Federated Logic Conference, Seattle, August 10 - 22, 2006. In particular, LFMTP'06 is co-sponsored by LICS and IJCAR. We are very grateful to the FLOC organizers, especially to Tom Ball, FLOC Co-chair, Gopal Gupta, FLOC Workshop Chair, Maria Paola Bonacina and Phil Scott, IJCAR and LICS Workshop Chairs respectively, who all worked very hard to make our life as workshop organizers easier.

July 21st, 2006

Alberto Momigliano
Brigitte Pientka

A Logic for Reasoning about Generic Judgments

Alwen Tiu

Australian National University and National ICT Australia

Abstract

This paper presents an extension of a proof system for encoding generic judgments, the logic $FO\lambda^{\Delta\nabla}$ of Miller and Tiu, with an induction principle. The logic $FO\lambda^{\Delta\nabla}$ is itself an extension of intuitionistic logic with fixed points and a "generic quantifier", ∇, which is used to reason about the dynamics of bindings in object systems encoded in the logic. A previous attempt to extend $FO\lambda^{\Delta\nabla}$ with an induction principle has been unsuccessful in modeling some behaviours of bindings in inductive specifications. It turns out that this problem can be solved by relaxing some restrictions on ∇, in particular by adding the axiom $B \equiv \nabla x.B$, where x is not free in B. We show that by adopting the equivariance principle, the presentation of the extended logic can be much simplified. Cut-elimination for the extended logic is stated, and some applications in reasoning about an object logic and a simply typed λ-calculus are illustrated.

Keywords: Proof theory, higher-order abstract syntax, logical frameworks.

1 Introduction

This paper aims at providing a framework for reasoning about specifications of deductive systems using *higher-order abstract syntax* [20]. Higher-order abstract syntax is a declarative approach to encoding syntax with bindings using Church's simply typed λ-calculus. The main idea is to support the notions of α-equivalence and substitutions in the object syntax by operations in λ-calculus, in particular α-conversion and β-reduction. There are at least two approaches to higher-order abstract syntax. The *functional programming* approach encodes the object syntax as a data type, where the binding constructs in the object language are mapped to functions in the functional language. In this approach, terms in the object language become values of their corresponding types in the functional language. The *proof search* approach encodes object syntax as expressions in a logic whose terms are simply typed, and functions that act on the object terms are defined via relations, i.e., logic programs. There is a subtle difference between this approach and the former; in the proof search approach, the simple types are inhabited by well-formed expressions, instead of values as in the functional approach (i.e., the abstraction type is inhabited by functions). The proof search approach is often referred to as λ-*tree syntax* [16], to distinguish it from the functional approach. This paper concerns

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

the λ-tree syntax approach.

Specifications which use λ-tree syntax are often formalized using hypothetical and generic judgments in intuitionistic logic. It is enough to restrict to the fragment of first-order intuitionistic logic whose only formulas are those of hereditary Harrop formulas, which we will refer to as the HH logic. Consider for instance the problem of defining the data type for untyped λ-terms. One first introduces the following constants:

$$app : tm \to tm \to tm \qquad abs : (tm \to tm) \to tm$$

where the type tm denotes the syntactic category of λ-terms and app and abs encode application and abstraction, respectively. The property of being a λ-term is then defined via the following theory:

$$\bigwedge M \bigwedge N(lam\ M \wedge lam\ N \Rightarrow lam\ (app\ M\ N))\ \ \&$$
$$\bigwedge M((\bigwedge x.lam\ x \Rightarrow lam\ (M\,x)) \Rightarrow lam\ (abs\ M))$$

where $\bigwedge$ is the universal quantifier and $\Rightarrow$ is implication.

Reasoning about object systems encoded in HH is reduced to reasoning about the structure of proofs in HH. McDowell and Miller formalize this kind of reasoning in the logic $FO\lambda^{\Delta\mathbb{N}}$ [10], which is an extension of first-order intuitionistic logic with fixed points and natural numbers induction. This is done by encoding the sequent calculus of HH inside $FO\lambda^{\Delta\mathbb{N}}$ and prove properties about it. We refer to HH as object logic and $FO\lambda^{\Delta\mathbb{N}}$ as meta logic. McDowell and Miller considered different styles of encodings and concluded that explicit representations of hypotheses and, more importantly, eigenvariables of the object logic are required in order to capture some statements about object logic provability in the meta logic [11]. One typical example involves the use of hypothetical and generic reasoning as follows: Suppose that the following formula is provable in HH.

$$\bigwedge x.p\,x\,s \Rightarrow \bigwedge y.p\,y\,t \Rightarrow p\,x\,t.$$

By inspection on the inference rules of HH, one observes that this is only possible if s and t are syntactically equal. This observation comes from the fact that the right introduction rule for universal quantifier, reading the rule bottom-up, introduces new constants, or eigenvariables. The quantified variables x and y will be replaced by distinct eigenvariables and hence the only matching hypothesis for $p\,x\,t$ would be $p\,x\,s$, and therefore s and t has to be equal. Let $\vdash_{HH} F$ denote the provability of the formula F in HH. Then in the meta logic, we would want to be able to prove the statement:

$$\forall s \forall t.(\vdash_{HH} \bigwedge x.p\,x\,s \Rightarrow \bigwedge y.p\,y\,t \Rightarrow p\,x\,t) \supset s = t.$$

The question is then how we would interpret the object logic eigenvariables in the meta logic. It is argued in [11] that the existing quantifiers in $FO\lambda^{\Delta\mathbb{N}}$ cannot be used to capture the behaviours of object logic eigenvariables directly. McDowell and Miller then resort to a non-logical encoding technique (in the sense that no logical connectives are used) which has some similar flavor to the use of deBruijn indices.

The use of this encoding technique, however, has a consequence that substitutions in the object logic has to be formalized explicitly.

Motivated by the above mentioned limitation of $FO\lambda^{\Delta\mathbb{N}}$, Miller and Tiu later introduced a new quantifier ∇ to $FO\lambda^{\Delta\mathbb{N}}$ which allows one to move the binders from the object logic to the meta logic. A generic judgment in the object logic, for instance $\vdash_{HH} \bigwedge x.G\,x$ is reflected in the meta logic as $\nabla x.\ \vdash_{HH} G\,x$. More generally, object logic eigenvariables are ∇-quantified at the meta level. This meta logic, called $FO\lambda^{\Delta\nabla}$ [17], allows one to perform case analyses on the provability of the object logic. Tiu later extended $FO\lambda^{\Delta\nabla}$ with induction and co-induction rules, resulting in the logic Linc [24]. However, some inductive properties about the object logic are not provable in Linc, e.g, the implication

$$\vdash_{HH} \bigwedge x.G\,x \supset \forall t.\ \vdash_{HH} G\,t \tag{1}$$

which states the extensional property of object logic universal quantification.

The inductive proof of the formula (1) would require an induction hypothesis that quantifies over object logic signatures, i.e., it is a statement of the sort

$$\text{“for all” } \vec{z},\ \forall H \nabla \vec{z}(\vdash_{HH} \bigwedge x.H\,\vec{z}\,x \supset \forall t.\ \vdash_{HH} H\,\vec{z}\,t)$$

where $\vec{z}$ is a list of object logic eigenvariables occurring in the object sequents. An obvious extension to Linc to formalize this statement would be to allow for quantification over arbitrary lists of variables which act like variable contexts to the object logic. However this is technically non-trivial and may require complicated proof theory. In this paper we follow an easier but weaker approach, which is expressive enough to allow for inductive reasoning over object specifications involving bindings. Instead of having explicit quantification over variable contexts, we require every proposition to hold in any variable context. This effectively translates to admitting the following axiom in $FO\lambda^{\Delta\nabla}$:

$$B \supset \nabla x.B, \qquad x \text{ is not free in } B, \tag{2}$$

which is not provable in $FO\lambda^{\Delta\nabla}$. Extensions to $FO\lambda^{\Delta\nabla}$ have been previously proposed in a couple of previous works [5,3]. In both works, it is suggested that adding the following axioms

$$\nabla x \nabla y.B\,x\,y \supset \nabla y \nabla x.B\,x\,y \quad \text{and} \quad B \equiv \nabla x.B, \tag{3}$$

where x is not free in B in the second scheme, to $FO\lambda^{\Delta\nabla}$ would result in a natural semantics for the extended logic. As it turns out, admitting these axioms would give a simpler proof theory too, compared to just having (2). We therefore adopt the axioms (3) in the extension of $FO\lambda^{\Delta\nabla}$ discussed in the paper. This extended logic, called LG^{ω}, is obtained by extending $FO\lambda^{\Delta\nabla}$ with natural number induction and with the axiom schemes (3). We show that inductive properties of λ-tree syntax specifications can be stated directly and in a purely logical fashion, and proved in LG^{ω}.

Relation to nominal logic

To guarantee good proof search behavior and syntactic consistency of the logic LG^{ω} (i.e., cut-elimination), the axiom schemes (3) need to be absorbed into the

rules of the proof system of LG^ω. There are at least a couple of ways of achieving this. One way is to extend the proof system of $FO\lambda^{\Delta\nabla}$ with some structural rules corresponding to the axioms (3). The other is to adopt the notion of *equivariant predicates* as in nominal logic [21], that is, provability of a predicate is invariant under permutations of *names*. We show here the second approach, which is simpler. The equivalent of the two formulations can be found in an extended version of the paper [25]. The equivariant principle is technically enforced by introducing a countably infinite set of name constants into the logic, and change the identity rule of the logic to allow equivalence under permutations of name constants:

$$\frac{\pi.B = \pi'.B'}{\Gamma, B \vdash B'}\ id$$

where π and π' are permutations on names. LG^ω is in fact very close to nominal logic, when we consider only the behaviours of logical connectives. In particular, the quantifier ∇ in LG^ω shares the same properties, in relation to other connectives of the logic, with the N quantifier in nominal logic. However, there are two important differences in our approach. First, we do not attempt to redefine α-conversion and substitutions in LG^ω in terms of permutations (or *swapping*) and the notion of *freshness* as in nominal logic. Name swapping and freshness constraints are not part of the syntax of LG^ω. These notions are present only in the meta theory of the logic. In LG^ω, for example, variables are always considered to have empty support, that is, $\pi.x = x$ for every permutation π. This is because we restrict substitutions to the "closed" ones, in the sense that no name constants can appear in the substitutions. A restricted form of open substitutions can be recovered indirectly at the meta theory of LG^ω. The fact that variables have empty support allows one to work with permutation free formulas and terms. So in LG^ω, we can prove that $p\ x\ a \supset p\ x\ b$, where a and b are names, without using explicit axioms of permutations and freshness. In nominal logic, one would prove this by using the swapping axiom $p\ x\ a \supset p\ ((a\ b).x)\ ((a\ b).b)$, where $(a\ b)$ denotes a swapping of a and b, and then show that $(a\ b).x = x$. The latter might not be valid if x is substituted by a, for example. The validity of this formula in nominal logic would therefore depend on the assumption on the support of x.

The second difference between LG^ω and nominal logic is that LG^ω allows closed terms (again, in the sense that no name constants appear in them) of type name, while in nominal logic, allowing such terms would lead to inconsistency [21]. As an example, the type tm in the encoding of λ-terms mentioned previously can be treated as a nominal type in LG^ω. This has an important consequence that we do not need to redefine the notion of substitutions for the encoded λ-terms, which is instead mapped to β-reduction in the meta language of LG^ω.

The rest of this paper is organized as follows. In Section 2 we introduce a proof system for LG^ω. Section 3 states some meta theories of LG^ω, in particular cut-elimination and a translation from LG^ω without fixed points and induction to $FO\lambda^\nabla$ with the axioms (3). Section 4 shows an encoding of HH logic in LG^ω and how some properties of the object logic can be formalized in LG^ω. Section 5 illustrates the use of HH to specify the typing judgments of λ-calculus and the evaluation relation on λ-terms. It also shows an example of reasoning about the

encoded λ-calculus, by induction on the provability of the typing judgments in the object logic HH. Section 6 discusses some related and future work. The proofs of the main results in this paper can be in an extended version of the paper [25].

2 A logic for generic judgments

We first define the core fragment of the logic LG^ω which does not have fixed point rules or induction. The starting point is the logic $FO\lambda^\nabla$ introduced in [17]. $FO\lambda^\nabla$ is an extension of a subset of Church's Simple Theory of Types in which formulas are given the type o. The core fragment of LG^ω, which we refer to as LG, shares the same set of connectives as $FO\lambda^\nabla$, namely, $\bot$, $\top$, $\wedge$, $\vee$, $\supset$, $\forall_\tau$, $\exists_\tau$ and ∇_τ. The type τ in the quantifiers is restricted to that which does not contain the type o. Hence the logic is essentially first-order. We abbreviate $(B \supset C) \wedge (C \supset B)$ as $B \equiv C$.

To enforce equivariant reasoning, we introduce a distinguished set of base types, called *nominal types*, which is denoted with $\mathcal{N}$. Nominal types are ranged over by ι. We restrict the ∇ quantifier to nominal types. For each nominal type $\iota \in \mathcal{N}$, we assume an infinite number of constants of that type. These constants are called *nominal constants*. We denote the family of nominal constants by $\mathcal{C}_\mathcal{N}$. The role of the nominal constants is to enforce the notion of equivariance: provability of formulas is invariant under permutations of nominal constants. Depending on the application, we might also assume a set of non-nominal constants, which is denoted by $\mathcal{K}$.

We assume the usual notion of capture-avoiding substitutions. Substitutions are ranged over by θ and ρ. Application of substitutions is written in a postfix notation, e.g., $t\theta$ is an application of θ to the term t. Given two substitutions θ and θ', we denote their composition by $\theta \circ \theta'$ which is defined as $t(\theta \circ \theta') = (t\theta)\theta'$. A *typing context* is a set of typed variables or constants. The judgment $\Delta \vdash t : \tau$ denotes the fact that the term t has type the simple type τ, given the typing context Δ. Its operational semantics is the usual type system for Church's simple type theory. A *signature* is a set of variables. A substitution θ respects a given signature Σ if there exists a set of typed variables Σ' such that for every $x : \tau \in \Sigma$ which is in the domain of θ, it holds that $\mathcal{K} \cup \Sigma' \vdash \theta(x) : \tau$. We denote by $\Sigma\theta$ the minimal set of variables satisfying the above condition. The substitution θ in this case is called a Σ-*substitution*. We assume that variables, free or bound, are of a different syntactic category from constants.

Definition 2.1 A permutation on $\mathcal{C}_\mathcal{N}$ is a bijection from $\mathcal{C}_\mathcal{N}$ to $\mathcal{C}_\mathcal{N}$. The permutations on $\mathcal{C}_\mathcal{N}$ are ranged over by π. Application of a permutation π to a nominal constant a is denoted with $\pi(a)$. We shall be concerned only with permutations which respect types, i.e., for every $a : \iota$, $\pi(a) : \iota$. Further, we shall also restrict to permutations which are finite, that is, the set $\{a \mid \pi(a) \neq a\}$ is finite. Application of a permutation to an arbitrary term (or formula), written $\pi.t$, is defined as follows:

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

A permutation involving only two nominal constants is called *swapping*. We use $(a\ b)$, where a and b are constants of the same type, to denote the swapping $\{a \mapsto b, b \mapsto a\}$.

The *support* of a term (or formula) t, written $supp(t)$, is the set of nominal constants appearing in it. It is clear from the above definition that if $supp(t)$ is empty, then $\pi.t = t$ for all π. The definition of Σ-substitution implies that for every θ and for every $x \in \Sigma$, $\theta(x)$ has empty support. Therefore Σ-substitutions and permutations commute, that is, $(\pi.t)\theta = \pi.(t\theta)$.

A sequent in LG^ω is an expression of the form $\Sigma; \Gamma \vdash C$ where Σ is a signature and the formulas in $\Gamma \cup \{C\}$ are in $\beta\eta$-normal form. The free variables of Γ and C are among the variables in Σ. The inference rules for the core fragment of LG^ω, i.e., the logic LG, are given in Figure 1.

In the $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$ rules, a denotes a nominal constant. In the $\exists\mathcal{L}$ and $\forall\mathcal{R}$ rules, we use *raising* [14] to encode the dependency of the quantified variable on the support of B, since we do not allow Σ-substitutions to mention any nominal constants. In the rules, the variable h has its type raised in the following way: suppose $\vec{c}$ is the list $c_1 : \iota_1, \ldots, c_n : \iota_n$ and the quantified variable x is of type τ. Then the variable h is of type: $\iota_1 \to \iota_2 \to \ldots \to \iota_n \to \tau$. This raising technique is similar to that of $FO\lambda^{\Delta\nabla}$, and is used to encode explicitly the minimal support of the quantified variable. Its use prevents one from mixing the scopes of $\forall$ (dually, $\exists$) and ∇. That is, it prevents the formula $\forall x \nabla y.p\, x\, y \equiv \nabla y \forall x.p\, x\, y$, and its dual, to be proved.

Looking at the introduction rules for $\forall$ and $\exists$, one might notice the asymmetry between the left and the right introduction rules. The left rule for $\forall$ allows instantiations with terms containing any nominal constants while the raised variable in the right introduction rule of $\forall$ takes into account only those which are in the support of the quantified formula. However, we will see that we can extend the dependency of the raised variable to an arbitrary number of fresh nominal constants not in the support without affecting the provability of the sequent (see Lemma 3.5 and Lemma 3.6).

We now extend the logic LG with a proof theoretic notion of equality and fixed points, following on works by Hallnas and Schroeder-Heister [7,22], Girard [6] and McDowell and Miller [10]. The equality rules are as follows:

$$\frac{\{\Sigma\theta; \Gamma\theta \vdash C\theta \ \mid\ (\lambda\vec{c}.t)\theta =_{\beta\eta} (\lambda\vec{c}.s)\theta\}}{\Sigma; \Gamma, s = t \vdash C}\ \mathrm{eq}\mathcal{L} \qquad \frac{}{\Sigma; \Gamma \vdash t = t}\ \mathrm{eq}\mathcal{R}$$

where $supp(s = t) = \{\vec{c}\}$ and θ is a Σ-substitution in the eq$\mathcal{L}$ rule. In the eq$\mathcal{L}$ rule, the substitution θ is a *unifier* of $\lambda\vec{c}.s$ and $\lambda\vec{c}.t$. Note that the λ-abstraction on $\vec{c}$ in eq$\mathcal{L}$ is quite redundant, since Σ-substitutions cannot mention nominal constants and it will be equally valid to say that θ is a unifier of t and s. The use of λ's in the rule is just to make it clear that the unification problem that arises in the rule is the usual higher-order unification and to conform with the formulations of equality rules in Linc [18,24].

We specify the premise of the rule as a set to mean that every element of the set is a premise. Since the terms s and t can be arbitrary higher-order terms, in general

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

$$\frac{}{\Sigma;\Gamma,\bot \vdash C}\ \bot\mathcal{L} \qquad \frac{}{\Sigma;\Gamma \vdash \top}\ \top\mathcal{R}$$

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

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

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

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

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

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

Fig. 1. The inference rules of LG

the set of their unifiers can be infinite. However, in some restricted cases, e.g., when $\lambda\vec{c}.s$ and $\lambda\vec{c}.t$ are *higher-order pattern* terms [13,19], if both terms are unifiable, then there exists a most general unifier. The applications we are considering are those which satisfy the higher-order pattern restrictions.

Definition 2.2 To each atomic formula, we associate a fixed point equation, or a *definition clause*, following the terminology of $FO\lambda^{\Delta\nabla}$. A definition clause is written $\forall\vec{x}.p\,\vec{x} \stackrel{\triangle}{=} B$ where the free variables of B are among $\vec{x}$. The predicate $p\,\vec{x}$ is called the *head* of the definition clause, and B is called the *body*. A *definition* is a set of definition clauses. We often omit the outer quantifiers when referring to a definition clause.

The introduction rules for defined atoms are as follows:

$$\frac{\Sigma;\Gamma, B[\vec{t}/\vec{x}] \vdash C}{\Sigma;\Gamma, p\,\vec{t} \vdash C}\ def\mathcal{L}, p\,\vec{x} \stackrel{\triangle}{=} B \qquad \frac{\Sigma;\Gamma \vdash B[\vec{t}/\vec{x}]}{\Sigma;\Gamma \vdash p\,\vec{t}}\ def\mathcal{R}, p\,\vec{x} \stackrel{\triangle}{=} B$$

In order to prove the cut-elimination theorem and the consistency of LG^ω, we allow only definition clauses which satisfy an *equivariance preserving* condition and a certain positivity condition, so as to guarantee the existence of fixed points.

Definition 2.3 We associate with each predicate symbol p a natural number, the *level* of p. Given a formula B, its *level* $lvl(B)$ is defined as follows:

(i) $lvl(p\,\bar{t}) = lvl(p)$
(ii) $lvl(\bot) = lvl(\top) = 0$
(iii) $lvl(B \wedge C) = lvl(B \vee C) = \max(lvl(B), lvl(C))$
(iv) $lvl(B \supset C) = \max(lvl(B) + 1, lvl(C))$
(v) $lvl(\forall x.B) = lvl(\nabla x.B) = lvl(\exists x.B) = lvl(B)$.

A definition clause $p\,\vec{x} \stackrel{\triangle}{=} B$ is stratified if $lvl(B) \leq lvl(p)$ and $supp(B) = \emptyset$. We consider only definition clauses which are stratified.

An example that violates the first restriction in Definition 2.3 is the definition $p \stackrel{\triangle}{=} p \supset \bot$. In [22], Schroeder-Heister shows that admitting this definition in a logic with contraction leads to inconsistency. To see why we need the second restriction on name constants, consider the definition $q\,x \stackrel{\triangle}{=} (x = a)$, where a is a nominal constant. Let b be a nominal constant different from a. Then $q\,b$ is both true, since it is equivariant to $q\,a$ and false, by the definition of fixed point.

In examples and applications, we often express definition clauses with patterns in the heads. Let us consider, for example, a definition clause for lists. We first introduce a type lst to denote lists of elements of type α, and the constants

$$nil : lst \qquad :: \quad : \alpha \to lst \to lst$$

which denote the empty list and a constructor to build a list from an element of type α and another list. The latter will be written in the infix notation. The definition clause for *lists* is as follows.

$$list\ L \stackrel{\triangle}{=} L = nil \vee \exists_\alpha A \exists_{lst} L'.L = (A :: L') \wedge list\ L'.$$

Using patterns, the above definition of lists can be rewritten as

$$list\ nil \stackrel{\triangle}{=} \top. \qquad list\ (A :: L) \stackrel{\triangle}{=} list\ L.$$

We shall often work directly with this patterned notation for definition clauses. For this purpose, we introduce the notion of *patterned definitions*. A *patterned definition clause* is written $\forall \vec{x}.H \stackrel{\triangle}{=} B$ where the free variables of H and B are among $\vec{x}$. The stratification of definitions in Definition 2.3 applies to patterned definitions as well. Since the patterned definition clauses are not allowed to have free occurrences of nominal constants, in matching the heads of the clauses with an atomic formula in a sequent, we need to raise the variables of the clauses to account for nominal constants that are in the support of the introduced formula. Given a patterned definition clause $\forall x_1 \ldots \forall x_n.H \stackrel{\triangle}{=} B$ its raised clause with respect to the list of constants $c_1 : \iota_1 \ldots c_n : \iota_n$ is

$$\forall h_1 \ldots \forall h_n.H[h_1\ \vec{c}/x_1, \ldots, h_n\ \vec{c}/x_n] \stackrel{\triangle}{=} B[h_1\ \vec{c}/x_1, \ldots, h_n\ \vec{c}/x_n].$$

The introduction rules for patterned definitions are

$$\frac{\{\Sigma\theta; B\theta, \Gamma\theta \vdash C\theta\}_\theta}{\Sigma; A, \Gamma \vdash C}\ def\mathcal{L} \qquad \frac{\Sigma; \Gamma \vdash B\theta}{\Sigma; \Gamma \vdash A}\ def\mathcal{R}$$

In the $def\mathcal{L}$ rule, B is the body of the raised patterned clause $\forall x_1 \ldots \forall x_n.H \stackrel{\triangle}{=} B$ and $(\lambda\vec{c}.H)\theta = (\lambda\vec{c}.A)\theta$ where $\{\vec{c}\}$ is the support of A. In the $def\mathcal{R}$ rule, we match A with the head of the clause, i.e., $\lambda\vec{c}.A = (\lambda\vec{c}.H)\theta$. These patterned rules can be derived using the non-patterned definition rules and the equality rules, as shown in [24].

Natural number induction.

We introduce a type nt to denote natural numbers, with the usual constants $z : nt$ (zero) and $s : nt \to nt$ (the successor function), and a special predicate $nat : nt \to o$. The rules for natural number induction are the same as those in $FO\lambda^{\Delta\mathbb{N}}$ [10], which are the introduction rules for the predicate nat.

$$\frac{\vdash D\,z \quad j; D\,j \vdash D\,(s\,j) \quad \Sigma; \Gamma, D\,I \vdash C}{\Sigma; \Gamma, nat\,I \vdash C}\ nat\mathcal{L}$$

$$\frac{}{\Sigma; \Gamma \vdash nat\ z}\ nat\mathcal{R} \qquad \frac{\Sigma; \Gamma \vdash nat\,I}{\Sigma; \Gamma \vdash nat\,(s\,I)}\ nat\mathcal{R}$$

The logic LG extended with the equality, definitions and induction rules is referred to as LG^{ω}.

3 The meta theory of LG^{ω}

In this section we investigate some properties of the logic LG^{ω}. We first look at the properties of the ∇ quantifier in relation to other connectives. The proof of the following proposition is straightforward by inspection on the rules of LG.

Proposition 3.1 *The following formulas are provable in LG:*

(i) $\nabla x.(Bx \wedge Cx) \equiv \nabla x.Bx \wedge \nabla x.Cx$.

(ii) $\nabla x.(Bx \supset Cx) \equiv \nabla x.Bx \supset \nabla x.Cx$.

(iii) $\nabla x.(Bx \vee Cx) \equiv \nabla x.Bx \vee \nabla x.Cx$.

(iv) $\nabla x.B \equiv B$, *provided that* x *is not free in* B.

(v) $\nabla x \nabla y.Bxy \equiv \nabla y \nabla x.Bxy$.

(vi) $\forall x.Bx \supset \nabla x.Bx$.

(vii) $\nabla x.Bx \supset \exists x.Bx$.

The formulas (i) – (iii) are provable in $FO\lambda^{\nabla}$. The proposition is true also in nominal logic with ∇ replaced by N.

The following properties concern the transformation of derivations. Provability is preserved under Σ-substitutions, permutations and a restricted form of name substitutions.

Lemma 3.2 Substitutions. *Let* Π *be a proof of* $\Sigma; \Gamma \vdash C$ *and let* θ *be a* Σ*-substitution. Then there exists a proof* Π' *of* $\Sigma\theta; \Gamma\theta \vdash C\theta$.

Lemma 3.3 Permutations. *Let* Π *be a proof of* $\Sigma; B_1, \ldots, B_n \vdash B_0$. *Then there exists a proof* Π' *of* $\Sigma; \pi_1.B_1, \ldots, \pi_n.B_n \vdash \pi_0.B_0$.

Lemma 3.4 Restricted name substitutions. *Let* Π *be a proof of*

$$\Sigma, x : \iota; B_1, \ldots, B_n \vdash B_0.$$

Then there exists a proof of Π' *of* $\Sigma; B_1[a_1/x], \ldots, B_n[a_n/x] \vdash B_0[a_0/x]$, *where* $a_i \notin supp(B_i)$ *for each* $i \in \{0, \ldots, n\}$.

The next two lemmas are crucial to the cut-elimination proof: they allow one to reintroduce the symmetry between $\forall\mathcal{L}$ and $\forall\mathcal{R}$, and dually, between $\exists\mathcal{L}$ and $\exists\mathcal{R}$ rules.

Lemma 3.5 Support extension. *Let* Π *be a proof of* $\Sigma, h; \Gamma \vdash B[h\ \vec{a}/x]$ *where* $\{\vec{a}\} = supp(B)$, $h \notin \Sigma$ *and* h *is not free in* Γ *and* B. *Let* $\vec{c}$ *be a finite list of nominal constants not in the support of* B. *Then there exists a proof* Π' *of* $\Sigma, h'; \Gamma \vdash B[h'\ \vec{a}\vec{c}/x]$.

Lemma 3.6 Support extension. *Let* Π *be a proof of* $\Sigma, h; B[h\ \vec{a}/x], \Gamma \vdash C$ *where* $\{\vec{a}\} = supp(B)$, $h \notin \Sigma$ *and* h *is not free in* Γ, B *and* C. *Let* $\vec{c}$ *be a finite list of nominal constants not in the support of* B. *Then there exists a proof* Π' *of* $\Sigma, h'; B[h'\ \vec{a}\vec{c}/x], \Gamma \vdash C$ *where* $h' \notin \Sigma$.

The main result on the meta theory of LG^ω is the cut-elimination theorem, from which the consistency of the logic follows.

Theorem 3.7 *The cut rule is admissible in* LG^ω.

Corollary 3.8 *The logic* LG^ω *is consistent, i.e., it is not the case that both* A *and* $A \supset \bot$ *are provable.*

Finally, we show that the formulation of LG is equivalent to $FO\lambda^\nabla$ extended with the axiom schemes of name permutations and weakening.

Theorem 3.9 *Let* F *be a formula which contains no occurrences of nominal constants. Then* F *is provable in* $FO\lambda^\nabla$ *extended with the axiom schemes* $B \equiv \nabla x.B$ *and* $\nabla x \nabla y.B\, x\, y \supset \nabla y \nabla x.B\, x\, y$ *if and only if* F *is provable in* LG.

4 Encoding an object logic

We now consider an encoding of the logic HH mentioned in the introduction in LG^ω. The encoding of this object logic has been done in $FO\lambda^{\Delta\mathbb{N}}$ by McDowell and Miller [11]. The formalization of the object logic properties in this section follows closely the $FO\lambda^{\Delta\mathbb{N}}$ encoding. The only major difference is that we do not need an explicit encoding of eigenvariables; eigenvariables are mapped to nominal constants in the meta logic LG^ω.

The object logic formulas are generated by the following grammar.

$$D ::= A \mid G \Rightarrow A \mid \textstyle\bigwedge_\tau x.D$$
$$G ::= A \mid tt \mid G \,\&\, G \mid A \Rightarrow G \mid \textstyle\bigwedge_\iota x.G \mid \bigvee_\tau .G$$

where A ranges over atomic (object-level) formula, $\Rightarrow$, $\&$, $\bigwedge$ and $\bigvee$ denote implication, conjunction, universal quantifier and existential quantifier, respectively. D and G represent definite clauses and goal formulas, respectively. Notice that in goal formulas, universal quantification is restricted to nominal types. The sequent rules for HH are the standard right introduction rules for the logical connectives plus

$$
\begin{array}{lll}
seq_I\ L\ tt & \triangleq \top. & seq_I\ L\ \langle A\rangle \triangleq elem\ A\ L.\\
seq_{(s\ I)}\ L\ (A\,\&\,B) & \triangleq seq_I\ L\ A \wedge seq_I\ L\ B. & \\
seq_{(s\ I)}\ L\ (A \Rightarrow B) & \triangleq seq_I\ (A :: L)\ B. & \\
seq_{(s\ I)}\ L\ (\bigwedge x.Gx) & \triangleq \nabla x.seq_I\ L\ Gx. & \\
seq_{(s\ I)}\ L\ \bigvee x.Gx & \triangleq \exists x.seq_I\ L\ Gx. & \\
seq_{(s\ I)}\ L\ \langle A\rangle & \triangleq \exists B.prog\ A\ B \wedge seq_I\ L\ B. &
\end{array}
$$

Fig. 2. Definition of an object logic.

the *backchaining* rule:

$$\frac{\Gamma, \bigwedge \vec{x}.G \supset A \longrightarrow G\theta}{\Gamma, \bigwedge \vec{x}.G \supset A \longrightarrow A'}\ bc, A\theta = A'$$

This sequent system is complete for the HH fragment of intuitionistic logic, as a consequence of *uniform provability* of intuitionistic logic [15].

In order to encode the object-logic formulas into LG^ω, we first introduce some types and constants. The object logic formulas are given the type prp, while atomic formulas are given the type atm. The formulas of HH are encoded using the following constants:

$$
\begin{array}{ccc}
\langle\,\rangle : atm \to prp & \& : prp \to prp \to prp & \bigwedge_\tau : (\iota \to prp) \to prp\\
tt : prp & \Rightarrow : atm \to prp \to prp & \bigvee_\tau : (\tau \to prp) \to prp
\end{array}
$$

We denote the encoding of an object level formula A in LG^ω with $[\![A]\!]$.

Since the set of definite clauses in the sequents does not change in the proofs in HH, we will not put them explicitly in the HH sequents in their encoding in LG^ω. Hence hypotheses of HH sequents are lists of atomic formulas. The object logic sequent is represented using the predicate $seq : nt \to atmlist \to prp \to o$ where $atmlist$ is the type for lists of atomic formulas, with the usual constructors nil and $::$. The natural number in the encoding of sequents will be used as a measure of the length of object logic proofs. Inductive properties about the provability in HH will be proved using this measure. An object sequent $\Gamma \longrightarrow A$ is represented as the atomic formula $(seq_I\ [\![\Gamma]\!]\ [\![A]\!])$ in LG^ω. We encode definite clauses using a predicate called $prog : atm \to prp \to o$. A definite clause $\bigwedge \vec{x}.G \Rightarrow A$ is encoded as the definition clause $\forall \vec{x}.prog\ A\ G \triangleq \top$. The patterned definition of the sequent rules of HH is given in Figure 2. It uses the following definition clauses.

$$
\begin{array}{lll}
list_i\ nil & \triangleq \top. & list_{(s\ i)}\ (A :: L) \triangleq list_i\ L.\\
list\ L & \triangleq \exists i.nat\ i \wedge list_i\ L. & \\
elem\ A\ (A :: L) & \triangleq \top. & elem\ A\ (B :: L) \triangleq elem\ A\ L.
\end{array}
$$

We refer to this definition together with the definition in Figure 2 as $\mathcal{D}(HH)$ and any additional definite clauses with $\mathcal{D}(prog)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{X, I_2; seq_{I_2}\ [p\ X]\ \langle p\ a\rangle \vdash \bot}{X, I_2; \nabla y.seq_{I_2}\ [p\ X]\ \langle p\ y\rangle \vdash \bot}\ \nabla\mathcal{L}}{X, I_1; seq_{I_1}\ [p\ X]\ (\bigwedge y.\langle p\ y\rangle) \vdash \bot}\ def\mathcal{L}}{X, I; seq_I\ nil\ (p\ X \Rightarrow \bigwedge y.\langle p\ y\rangle) \vdash \bot}\ def\mathcal{L}}{\vdash \forall X \forall I.(seq_I\ nil\ (p\ X \Rightarrow \bigwedge y.\langle p\ y\rangle) \supset \bot)}\ \forall\mathcal{R}; \supset \mathcal{R}$$

Fig. 3. A derivation in LG^ω.

Example:

The formula $p\ X \Rightarrow \bigwedge y.p\ y$ is not provable in the empty theory, whatever the value of X is. This fact is formalized in LG^ω as the formula

$$\forall X \forall I.(seq_I\ nil\ (p\ X \Rightarrow \bigwedge y.\langle p\ y\rangle) \supset \bot).$$

A partial derivation of this formula in LG^ω is shown in Figure 3. In the figure the notation $[p\,X]$ stands for the list $(p\,X :: nil)$. The derivation is completed by applying $def\mathcal{L}$ to the topmost sequent, resulting in two matching cases: the identity rule and the backchaining rule. Since we assume no definite clauses, this leaves us with proving the sequent: $X, I_2; elem\ (p\ X)\ (p\ a :: nil) \vdash \bot$. Applying $def\mathcal{L}$ to this sequent results in the sequent $X, I_2; elem\ (p\ X)\ nil \vdash \bot$, since $\lambda a.p\ X$ and $\lambda a.p\ a$ are not unifiable. Another application of $def\mathcal{L}$ gives us empty premise and hence the sequent is provable. □

It is straightforward to see that the structure of the HH proofs corresponds to the structure of proofs of its encoding in LG^ω; in particular, the backchaining rule in HH corresponds to the $def\mathcal{R}$ rule (for the patterned definition) in LG^ω. We now state some properties of the encoding of HH in LG^ω.

Theorem 4.1 *Let $\mathcal{D}(prog)$ be a definition corresponding to a set of definite clauses $\mathcal{P}$. Then the sequent $\mathcal{P}, \Gamma \longrightarrow G$ is derivable in HH if and only if $seq_i\ [\![\Gamma]\!]\ [\![G]\!]$ is derivable in LG^ω with the definition $\mathcal{D}(prog) \cup \mathcal{D}(HH)$ for some natural number i.*

Theorem 4.2 *The following formulas are provable in LG^ω with the definition of the object logic HH:*

(i) *Structural rules:* $\forall L \forall L' \forall G \forall i.\, nat\, i \supset list\ L \supset list\ L'$
$$(\forall A.elem\ A\ L \supset elem\ A\ L') \supset seq_i\ L\ G \supset seq_i\ L'\ G.$$

(ii) *Atomic cut:* $\forall L \forall G \forall A.list\ L \supset\ \exists i.(nat\ i \wedge seq_i\ L\ (A \Rightarrow G)) \supset$
$$\exists i.(nat\ i \wedge seq_i\ L\ \langle A\rangle) \supset \exists i.nat\ i \wedge seq_i\ L\ G.$$

(iii) *Specialization:* $\forall L \forall G \forall i.nat\ i \supset list\ L \supset seq_{(s\,i)}\ L\ (\bigwedge G) \supset \forall x.seq_i\ L\ (G\,x)$.

We conclude this section by a remark that ∇ is strictly speaking not necessary for capturing object logic provability, as Theorem 4.2 (3) shows, rather it is the use of nominal constants to model eigenvariables that allows that. The use of ∇, however, results in a more natural correspondence between the encoding of HH and its actual sequent proofs.

5 Reasoning about operational semantics

Following McDowell and Miller [11], we use the encoding of HH in LG^ω to specify and reason about the operational semantics of simply typed λ-calculus. Reasoning about more complicated languages like PCF can be done as well using a similar approach (see [11]).

We introduce a type ty to denote object-level types. The type tm denotes the object-level λ-terms and is considered a nominal type. The language of the (object-level) λ-terms is encoded using the following constants:

$$app : tm \to tm \to tm \qquad abs : ty \to (tm \to tm) \to tm$$

which denote application and abstraction, respectively. The object-level type constructor, i.e., the 'arrow', is encoded via the constant $ar : ty \to ty \to ty$. Object-level base types are ranged over by α.

The evaluation relation and the typing judgments of the simply typed calculus are given as definite clauses below.

$$\begin{aligned}
eval\ (abs\ T\ M)\ (abs\ T\ M) &\Leftarrow tt.\\
eval\ (app\ M\ N)\ V &\Leftarrow \bigvee P \bigvee T. eval\ M\ (abs\ P\ T) \,\&\, eval\ (P\ N)\ V.\\
typeof\ (abs\ T\ M)\ (ar\ T\ T') &\Leftarrow \bigwedge x. typeof\ x\ T \Rightarrow typeof\ (M x)\ T'.\\
typeof\ (app\ M\ N)\ T &\Leftarrow \bigvee T'. typeof\ M\ (ar\ T'\ T) \,\&\, typeof\ N\ T'.
\end{aligned}$$

It is straightforward to translate these clauses to *prog* clauses.

We state a couple of properties here as formulas in LG^ω. In the following theorems, we use the notation $L \triangleright G$ to denote the formula $\exists i. nat\ i \wedge seq_i\ L\ G$. If L is nil we simply write $\triangleright G$.

Theorem 5.1 Subject reduction. *The following formula is provable*

$$\forall M \forall V \forall T.\ \triangleright \langle eval\ M\ V\rangle \wedge \triangleright \langle typeof\ M\ T\rangle \supset \triangleright \langle typeof\ V\ T\rangle.$$

A proof of a similar theorem is given in [11] for the untyped λ-term in the logic $FO\lambda^{\Delta\mathbb{N}}$. This proof can be adapted straightforwardly to give a proof for the above theorem. A more interesting property is the determinacy of type assignments, provided that the typing context is well-formed, that is, each variable in the context is assigned a unique type. The well-formedness of a typing context L is specified as the formula:

$$\forall X \forall T_1 \forall T_2. elem\ (typeof\ X\ T_1)\ L \supset elem\ (typeof\ X\ T_2)\ L \supset T_1 = T_2.$$

The above formula will be denoted by $ctx\ L$.

Theorem 5.2 *The following formula is provable:*

$$\forall L \forall X \forall T_1 \forall T_2.\, list\ L \supset ctx\ L \supset L \triangleright \langle typeof\ X\ T_1\rangle \supset L \triangleright \langle typeof\ X\ T_2\rangle \supset T_1 = T_2.$$

6 Related and future work

There have been many previous related works in providing frameworks for higher-order abstract syntax, or more generally abstract syntax with bindings. A non-exhaustive list includes encodings in proof assistants like Coq [4], HOL [26], Isabelle [27], and Twelf [23], categorical frameworks [8], *the theory of contexts* [9], nominal logic [21], and proof search frameworks [11,24]. The approach taken here is similar to the latter; the novelty of our work lies in the use of equivariance principle within the usual style of higher-order abstract syntax specifications. An immediate future work will be to implement the logic LG^ω, possibly on top of an existing proof assistant, and to perform large case studies, in particular, the problem sets put out in the POPLmark Challenge [1].

In the current work we show only the treatment of natural number induction. Extensions to iterated (co-)inductive definitions can be done in a similar way as in [18,24].

Semantics of LG. There have been a couple of attempts at giving a semantics for the logic $FO\lambda^\nabla$: Cheney and Gabbay proposed an encoding into nominal logic [5,3], and Miculan and Yemane give a categorical semantics [12]. In both works, it is suggested that extending $FO\lambda^\nabla$ with the axiom schemes (3) would result in a natural semantics for ∇. The work by Miculan and Yemane seems closer to the logic LG and could very well serve as a basis for finding a categorical model for LG. There are some similarities between LG and Nominal Logic, but the treatment of substitutions and the addition of closed terms of type name in LG make it not obvious whether the support models of Nominal Logic can be used for LG. We leave the investigation of support models for LG (or a classical version of LG), such as the ones in [21,2], as a future work.

Acknowledgement

The author would like to thank James Cheney for his many helpful remarks and suggestions, in particular those related to Nominal Logic, the anonymous referees for their useful and detailed comments, and also David Baelde, Alberto Momigliano, Michael Norrish for their comments on earlier drafts of the paper.

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: the POPLMARK challenge. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

[2] J. Cheney. Completeness and Herbrand theorems for nominal logic. Preprint. Accepted for publication at the Journal of Symbolic Logic, July 2005.

[3] J. Cheney. A simpler proof theory for nominal logic. In *Proc. FOSSACS'05*, volume 3441 of *Lecture Notes in Computer Science*. Springer, 2005.

[4] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.

[5] M. J. Gabbay and J. Cheney. A sequent calculus for nominal logic. In *Proc. 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 139–148, 2004.

[6] J.-Y. Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.

[7] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, October 1991.

[8] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.

[9] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, number 2076 in LNCS, pages 963–978. Springer-Verlag, 2001.

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

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

[12] M. Miculan and K. Yemane. A unifying model of variables and names. In *Proc. FOSSACS'05*, volume 3441 of *Lecture Notes in Computer Science*, pages 170 – 186. Springer, 2005.

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

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

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

[16] D. Miller and C. Palamidessi. Foundational aspects of syntax. In P. Degano, R. Gorrieri, A. Marchetti-Spaccamela, and P. Wegner, editors, *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, volume 31. ACM, September 1999.

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

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

[19] T. Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS 1993)*, pages 64–74. IEEE, June 1993.

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

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

[22] P. Schroeder-Heister. Cut-elimination in logics with definitional reflection. In D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, volume 619 of *LNCS*, pages 146–171. Springer, 1992.

[23] C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, October 2000.

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

[25] A. Tiu. A logic for reasoning about generic judgments. Extended version, http://users.anu.edu.edu.au/~tiu/lgext.pdf, 2006.

[26] C. Urban and M. Norrish. A formal treatment of the Barendregt Variable Convention in rule inductions. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 25–32, New York, NY, USA, 2005. ACM Press.

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

Modelling Generic Judgements

Ulrich Schöpp[1]

TCS, Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstraße 67, D-80538 München, Germany

Abstract

We propose a semantics for the ∇-quantifier of Miller and Tiu. First we consider the case for classical first-order logic. In this case, the interpretation is close to standard Tarski-semantics and completeness can be shown using a standard argument. Then we put our semantics into a broader context by giving a general interpretation of ∇ in categories with binding structure. Since categories with binding structure also encompass nominal logic, we thus show that both ∇-logic and nominal logic can be modelled using the same definition of binding. As a special case of the general semantics in categories with binding structure, we recover Gabbay & Cheney's translation of FOλ^∇ into nominal logic.

Keywords: Higher-Order Abstract Syntax, First-Order Logic, Model Theory, Categorical Logic

Introduction

Miller & Tiu [18] have introduced the logic FOλ^∇ for reasoning about specifications in λ-tree syntax (a version of higher-order abstract syntax). The main new feature of FOλ^∇ is the ∇-quantifier for the treatment of object-level eigenvariables. The design of FOλ^∇ is based on a study of the proof-theory of first-order logic and the result is an elegant and simple calculus.

In addition to the proof theory for FOλ^∇, one may be interested in a model-theoretic semantics, e.g. [6,17,25]. A simple semantics is useful for explaining the logic and for understanding the meaning of formulae. It may also help in transferring technology from other, semantics-based, approaches. Nominal Logic [19], for example, solves a problem similar to that solved by FOλ^∇, but is based on a semantic approach. A clarification of the semantics of FOλ^∇ should help in studying the connection of FOλ^∇ to Nominal Logic, in order to answer questions such as whether the elegant proof theory of FOλ^∇ can be used for Nominal Logic, or whether programming in the style of Fresh O'Caml [23,22] is possible with ∇. In general, a model-theoretic explanation of the ∇-quantifier should make it amenable

[1] Email: schoepp@tcs.ifi.lmu.de

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

for inclusion in solutions that are based on the model-theory of first-order logic, an example being Brotherston's very general approach to cyclic proofs [2].

In this paper we give a simple and direct semantic interpretation of FOλ^{∇}. We focus on classical logic with its easy-to-describe Tarski semantics and show that well-known theory can be adapted to classical FOλ^{∇}. We focus on the classical case, as it is particularly simple to describe. In the second part of this paper we give a general framework for modelling FOλ^{∇}, which we hope will help in studying the relationships of models for FOλ^{∇}, such as those of Gabbay & Cheney [6] and Miculan & Yemane [17].

To explain the interpretation informally, recall from [18] that FOλ^{∇} is a first-order logic over simply-typed λ-calculus. Object-level syntax is represented by λ-tree types (HOAS). For instance, the type of λ-terms can be represented by a type Tm with two constants $app\colon Tm \times Tm \to Tm$ and $lam\colon (Tm \to Tm) \to Tm$. In addition to the usual existential and universal quantification, $\exists x\colon \tau.\, \varphi(x)$ and $\forall x\colon \tau.\, \varphi(x)$, FO$\lambda^{\nabla}$ also features the quantification $\nabla x\colon Tm.\, \varphi(x)$. In this paper we restrict the ∇-quantifier to range only over a subclass of types such as Tm, which we call λ-tree types. A similar restriction is imposed in [25]. Our restriction on ∇ is in line with restrictions imposed on the new-quantifier N of Gabbay & Pitts, which typically ranges over names only. For the semantics, we use a standard interpretation for the quantifiers $\exists$ and $\forall$. Our interpretation of $\nabla x\colon Tm.\, \varphi(x)$ may be expressed as: 'for all new *variables* $x\colon Tm$ the formula $\varphi(x)$ holds.' The quantification ranges only over variables, but not over terms of the form $app\langle x, y\rangle$ or $lam(f)$. The meaning of the formula $\nabla x\colon Tm.\, \varphi(x)$ may equivalently be expressed as: 'for some new variable $x\colon Tm$ the formula $\varphi(x)$ holds.'

To formalise this informal description of the semantics, we have to deal with a number of issues. First, object-level terms are represented using higher-order abstract syntax. To model the type Tm, and in particular the constant lam of type $(Tm \to Tm) \to Tm$, adequately, we must find a model in which the interpretation of the function space $(Tm \to Tm)$ contains just α-equivalence classes of object-level terms. We address this issue using an approach of Hofmann [12], who has shown that certain presheaves provide a canonical model in which such an interpretation of Tm is available. A second issue is that to model ∇ we want to explain quantification over variables only. The internal logic of the presheaf-category, in which the terms are interpreted, is not suitable for this purpose, since it can express only propositions that are closed under all substitutions. We address this issue by modelling the terms in a presheaf category, but taking the logic from a so-called category with binding structure. This approach of modelling terms in one place and using the logic from another place is also used by Hofmann [12]. We consider a number of different logics including Nominal Logic, in which case we recover the interpretation of FOλ^{∇} in Nominal Logic proposed by Gabbay & Cheney [6].

In Part I of this paper we spell out directly an interpretation of classical FOλ^{∇}. The aim of this part is to present the interpretation as simple as possible, without assuming knowledge of FOλ^{∇}. We show that the semantics is close to the standard Tarski semantics for classical logic and that the well-known completeness argument goes through almost unchanged. The point of Part I is to stress that just a little change to the standard Tarski-semantics is enough for modelling the ∇-quantifier.

In Part II, we generalise the development from the first part by giving models of ∇ in *categories with binding structure* [20, Chapter 10], which capture a general definition of binding. Since Nominal Logic also fits into the framework of categories with binding structure, the development in Part II will therefore show that both ∇ and nominal logic can be modelled using the very same definition of binding.

1 Part I
Classical First-Order Logic with ∇

1.1 Simply typed λ-calculus

We fix the notation for a simply typed λ-calculus.

$$\begin{array}{rl} \text{Types:} & \tau := \text{base types} \mid \lambda\text{-tree types} \mid 1 \mid \tau \times \tau \mid \tau \to \tau \\ \text{Terms:} & M := x \mid c \mid * \mid \langle M, M\rangle \mid \pi_1(M) \mid \pi_2(M) \mid \lambda x{:}\tau.\, M \mid M\ M \\ \text{Contexts:} & \Sigma := \cdot \mid \Sigma,\, x\colon \tau \end{array}$$

Terms are identified up to renaming of bound variables. We assume constants c to be defined in a signature $\mathcal{S}$, and we assume the choice of $\mathcal{S}$ and the primitive types to be such that the types and terms are countable.

In addition to the normal base types, there is a second kind of base types, which we call λ-tree types. The intended use of λ-tree types is for higher-order abstract syntax encodings. We use ι to range over λ-tree types.

Contexts are subject to the usual convention that no variable may be declared more than once. A context that contains only declarations $x\colon \iota$ of λ-tree types is called a λ-tree context. We use σ to range over λ-tree contexts.

The typing judgement is defined by the following rules.

$$\frac{(x\colon \tau) \in \Sigma}{\Sigma \vdash x\colon \tau} \qquad \frac{(c\colon \tau) \in \mathcal{S}}{\Sigma \vdash c\colon \tau} \qquad \frac{}{\Sigma \vdash *\colon 1}$$

$$\frac{\Sigma \vdash M\colon \tau \qquad \Sigma \vdash N\colon \tau'}{\Sigma \vdash \langle M, N\rangle\colon \tau \times \tau'} \qquad \frac{\Sigma \vdash M\colon \tau \times \tau'}{\Sigma \vdash \pi_1(M)\colon \tau} \qquad \frac{\Sigma \vdash M\colon \tau \times \tau'}{\Sigma \vdash \pi_2(M)\colon \tau'}$$

$$\frac{\Sigma,\, x\colon \tau \vdash M\colon \tau'}{\Sigma \vdash \lambda x{:}\tau.\, M\colon \tau \to \tau'} \qquad \frac{\Sigma \vdash M\colon \tau \to \tau' \qquad \Sigma \vdash N\colon \tau}{\Sigma \vdash M\ N\colon \tau'}$$

A *substitution* $\rho\colon \Sigma \to \Sigma'$ is a function that assigns to each variable $(x\colon \tau)$ in Σ' a term $\Sigma \vdash M\colon \tau$. We use the following notation for substitutions:

$$[M_1/x_1] \ldots [M_n/x_n]\colon \Sigma \to (x_1\colon \tau_1, \ldots, x_n\colon \tau_n)$$

We use the letters ρ and θ to range over substitutions. An *order-preserving renaming* is a substitution of the following form (note the order):

$$[y_1/x_1] \ldots [y_n/x_n]\colon (y_1\colon \tau_1, \ldots, y_n\colon \tau_n) \to (x_1\colon \tau_1, \ldots, x_n\colon \tau_n)$$

We use the letters α and β to range over order-preserving renaming.

We write $\mathbf{T}$ for the category having contexts as objects and substitutions as morphisms, and we write $\mathbf{L}$ for the sub-category of $\mathbf{T}$ consisting of λ-tree contexts and substitutions between them.

The semantic structure corresponding to simply-typed λ-calculus is that of a Cartesian closed category with finite products. An interpretation of the syntax in such a category $\mathbf{C}$ is given by a functor $\|-\|\colon \mathbf{T} \to \mathbf{C}$ that preserves finite products and exponents.

In order to model higher-order syntax adequately, we will interpret the types as presheaves on $\mathbf{L}$, i.e. as functors $\mathbf{L}^{\mathrm{op}} \to \mathbf{Sets}$. This means that the interpretation $\|\tau\|$ of a type τ is a mapping that assigns to each λ-tree context σ a set $\|\tau\|\sigma$ and that assigns to each substitution $\theta\colon \sigma \to \sigma'$ and each element $x \in \|\tau\|\sigma'$ an element $x[\theta] \in \|\tau\|\sigma$ in such a way that the equations $x[id] = x$ and $x[\theta \circ \rho] = x[\theta][\rho]$ hold. A term $\Sigma \vdash M\colon \tau$ is interpreted as a natural transformation $\|M\|\colon \|\Sigma\| \to \|\tau\|$. This means that the interpretation of a term is given by, for each σ, a function $\|M\|_\sigma\colon \|\Sigma\|\sigma \to \|\tau\|\sigma$, such that these functions behave well with respect to substitution, that is $(\|M\|_{\sigma'}(x))[\theta] = \|M\|_\sigma(x[\theta])$ holds for all $\theta\colon \sigma \to \sigma'$ and all $x \in \|\Sigma\|\sigma'$.

For a small category $\mathbf{C}$, we write $\widehat{\mathbf{C}}$ for the category of functors $\mathbf{C}^{\mathrm{op}} \to \mathbf{Sets}$.

The point of using presheaves is that λ-tree syntax can be modelled adequately, as has been shown in [12]. Consider for example the λ-tree type Tm with constants *app* and *lam*, as described in the introduction. For the interpretation $\|Tm\|$ we choose the presheaf $\|Tm\|(\sigma) = \{M \mid \sigma \vdash M\colon Tm\}$ with the canonical substitution action. Now, the Cartesian closed structure on $\widehat{\mathbf{L}}$ is such that there are isomorphisms $\|Tm \times Tm\|\sigma \cong \|Tm\|\sigma \times \|Tm\|\sigma$ and $\|Tm \to Tm\|\sigma \cong \|Tm\|(\sigma, x\colon Tm)$. In particular, the interpretation of the function type $Tm \to Tm$ at stage σ consists just of terms with an additional variable x. Therefore, we can interpret the terms *app* and *lam* by the maps $\|Tm\| \times \|Tm\| \to \|Tm\|$ and $\|Tm \to Tm\| \to \|Tm\|$ given by $\langle M, N\rangle \mapsto \mathit{app}\langle M, N\rangle$ and $M \mapsto \mathit{lam}(\lambda x{:}Tm.\, M)$ respectively.

The reader unfamiliar with the presheaf semantics may find it instructive to think of the term model (up to α-equivalence), in which all types are interpreted by $\|\tau\|\sigma = \{M \mid \sigma \vdash M\colon \tau\}$.

1.2 Classical First-order Logic with ∇

Logical formulae in context Σ can be defined using a base type o and, for each relation symbol R, a constant $R\colon \tau_1 \to \cdots \to \tau_n \to \mathrm{o}$, as well as constants for the logical connectives $\neg\colon \mathrm{o} \to \mathrm{o}$ and $\vee, \wedge\colon \mathrm{o} \times \mathrm{o} \to \mathrm{o}$ and $\forall, \exists\colon (\tau \to \mathrm{o}) \to \mathrm{o}$ and $\nabla\colon (\iota \to \mathrm{o}) \to \mathrm{o}$ etc. Although these constants can be interpreted in our semantics, it is simpler to consider logical formulae as a separate entity with the evident inductive definition, which is the view we adopt in this section.

We define a sequent calculus for a classical logic with ∇. The sequents have the form $\Sigma \mid \Gamma \longrightarrow \Delta$, in which Σ is a context and Γ and Δ are (possibly infinite) sets of formulae in a local signature $\sigma \rhd A$. A local signature σ is a λ-tree context and $\sigma \rhd A$ in context Σ presupposes that A is a well-formed formula in context Σ, σ. We identify statements $\sigma \rhd A$ up to bound renaming of variables in σ. One may think of $\sigma \rhd A$ as the formula $\nabla\sigma.\, A$. For $\cdot \rhd A$ we write just A.

The rules of the sequent calculus, which are given in Fig. 1, are a straightforward extension to classical logic of the rules in [18].

Just as in FOλ^∇, the ∇-quantifier commutes with all other logical connectives, i.e. one can prove equivalences of the form $(\nabla x\colon \iota.\, A \wedge B) \dashv\vdash (\nabla x\colon \iota.\, A) \wedge (\nabla x\colon \iota.\, B)$

General Rules

$$(\textsc{Axiom})\ \frac{}{\Sigma \mid \Gamma \longrightarrow \Delta}\ \Gamma \cap \Delta \neq \emptyset$$

$$(\textsc{Cut})\ \frac{\Sigma \mid \Gamma \longrightarrow \mathcal{B} \qquad \Sigma \mid \mathcal{B}, \Delta \longrightarrow \Phi}{\Sigma \mid \Gamma, \Delta \longrightarrow \Phi}$$

Logical Rules

$$(\bot\text{-L})\ \frac{}{\Sigma \mid \Gamma, \sigma \rhd \bot \longrightarrow \Delta} \qquad (\top\text{-R})\ \frac{}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd \top}$$

$$(\neg\text{-L})\ \frac{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A}{\Sigma \mid \Gamma, \sigma \rhd \neg A \longrightarrow \Delta} \qquad (\neg\text{-R})\ \frac{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A}{\Sigma \mid \Gamma, \sigma \rhd \neg A \longrightarrow \Delta}$$

$$(\wedge\text{-L})\ \frac{\Sigma \mid \Gamma, \sigma \rhd A, \sigma \rhd B \longrightarrow \Delta}{\Sigma \mid \Gamma, \sigma \rhd A \wedge B \longrightarrow \Delta} \qquad (\wedge\text{-R})\ \frac{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A \qquad \Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd B}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A \wedge B}$$

$$(\vee\text{-L})\ \frac{\Sigma \mid \Gamma, \sigma \rhd A \longrightarrow \Delta \qquad \Sigma \mid \Gamma, \sigma \rhd B \longrightarrow \Delta}{\Sigma \mid \Gamma, \sigma \rhd A \vee B \longrightarrow \Delta} \qquad (\vee\text{-R})\ \frac{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A, \sigma \rhd B}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A \vee B}$$

$$(\supset\text{-L})\ \frac{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A \qquad \Sigma \mid \Gamma, \sigma \rhd B \longrightarrow \Delta}{\Sigma \mid \Gamma, \sigma \rhd A \supset B \longrightarrow \Delta} \qquad (\supset\text{-R})\ \frac{\Sigma \mid \Gamma, \sigma \rhd A \longrightarrow \Delta, \sigma \rhd B}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A \supset B}$$

$$(\forall\text{-L})\ \frac{\Sigma, \sigma \vdash M\colon \tau \qquad \Sigma \mid \Gamma, \sigma \rhd A[M/x] \longrightarrow \Delta}{\Sigma \mid \Gamma, \sigma \rhd \forall x\colon \tau.\, A \longrightarrow \Delta} \qquad (\forall\text{-R})\ \frac{\Sigma, h \mid \Gamma \longrightarrow \Delta, \sigma \rhd A[h\ \sigma/x]}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd \forall x\colon \tau.\, A}$$

$$(\exists\text{-L})\ \frac{\Sigma, h \mid \Gamma, \sigma \rhd A[h\ \sigma/x] \longrightarrow \Delta}{\Sigma \mid \Gamma, \sigma \rhd \exists x\colon \tau.\, A \longrightarrow \Delta} \qquad (\exists\text{-R})\ \frac{\Sigma, \sigma \vdash M\colon \tau \qquad \Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd A[M/x]}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd \exists x\colon \tau.\, A}$$

$$(\nabla\text{-R})\ \frac{\Sigma \mid \Gamma, (\sigma, x\colon \iota) \rhd A \longrightarrow \Delta}{\Sigma \mid \Gamma, \sigma \rhd \nabla x\colon \iota.\, A \longrightarrow \Delta} \qquad (\nabla\text{-R})\ \frac{\Sigma \mid \Gamma \longrightarrow \Delta, (\sigma, x\colon \iota) \rhd A}{\Sigma \mid \Gamma \longrightarrow \Delta, \sigma \rhd \nabla x\colon \iota.\, A}$$

Fig. 1. Sequent calculus

and $(\nabla x\colon \iota.\, \exists y\colon \tau.\, A(x,y)) \dashv\vdash (\exists h\colon \iota \to \tau.\, \nabla x\colon \iota.\, A(x, h\ x))$ and likewise for all the other connectives. This property of ∇ makes it very similar to the new-quantifier N of Gabbay & Pitts [7]. The quantifier ∇ differs from N, however, in the fact that the following equivalences are *not* provable: $\nabla x\colon \iota.\, \nabla y\colon \iota. A(x,y) \dashv\vdash \nabla y\colon \iota.\, \nabla x\colon \iota. A(x,y)$ and $\forall x\colon \iota.\, A(x) \supset \nabla x\colon \iota.\, A(x)$ and $\nabla x\colon \iota.\, A(x) \supset \exists x\colon \iota.\, A(x)$. In Part II we will show that both ∇ and N are nevertheless instances of the same structure (Prop. 2.2).

We omit sequent rules for equality and definitions. Although these rules are important for practical reasoning and are non-trivial from a proof-theoretic perspective, for semantic purposes it is simpler to replace them by axiom-schemes.

1.3 Semantic Interpretation

To give a meaning to the logic, we must first interpret the underlying λ-calculus. To do this, we assume, for each base type τ, an object $\|\tau\|$ of $\widehat{\mathbf{L}}$. We extend this assignment to an interpretation of the λ-calculus by the following clauses.

$$\begin{aligned}
\|\iota\|\sigma &= \mathbf{y}(x\colon \iota) \cong \{M \mid \sigma \vdash M\colon \iota\}, \text{ where } \iota \text{ is a } \lambda\text{-tree type}\\
\|1\|\sigma &= 1\\
\|\tau \times \tau'\|\sigma &= \|\tau\|\sigma \times \|\tau'\|\sigma\\
\|\tau \to \tau'\|\sigma &= \widehat{\mathbf{L}}(\mathbf{y}(\sigma) \times \|\tau\|, \|\tau'\|)
\end{aligned}$$

Here, $\mathbf{y}(\sigma) = \mathbf{L}(-,\sigma)$ is the Yoneda-embedding. We note that, as a consequence of the Yoneda Lemma, there is an isomorphism $\|\iota \to \tau\|(\sigma) \cong \|\tau\|(\sigma, x\colon \iota)$ for all λ-tree types ι. We extend the interpretation to contexts by use of Cartesian products.

Propositions are interpreted as subsets closed under order-preserving renaming of variables. That is, a proposition A in context Σ is interpreted by, for each σ, a subset $A(\sigma) \subseteq \|\Sigma\|\sigma$, such that, for each σ, each $x \in A(\sigma)$ and each bijective variable renaming $\alpha\colon \sigma' \to \sigma$, we have $x[\alpha] \in A(\sigma')$. Note that propositions must be closed only under variable-renaming, not under all substitutions.

The interpretation of base-types, constants and relations is recorded in a structure $\mathfrak{A}$. It assigns to each base type τ an object $\|\tau\|$ in $\widehat{\mathbf{L}}$ and to each constant $c\colon \tau$ a morphism $\|1\| \to \|\tau\|$ in $\widehat{\mathbf{L}}$. It assigns to each relation symbol $R\colon \tau_1 \to \cdots \to \tau_n \to \mathrm{o}$ a proposition on $x_1\colon \tau_1, \ldots, x_n\colon \tau_n$, as described just above.

The interpretation of formulae is defined by the satisfaction relation $\sigma \Vdash_{\rho,\Sigma} A$, in which the stage σ is a λ-tree context, A is a formula in context Σ and ρ is a Σ-valuation at stage σ defined as follows: a Σ-*valuation* at stage σ is a function ρ mapping each variable $x\colon \tau$ in Σ to an element of $\|\tau\|(\sigma)$. For a term $\Sigma \vdash t\colon \tau$, we write $\rho(t)$ for the evident element of $\|\tau\|(\sigma)$.

$$
\begin{aligned}
&\sigma \Vdash_{\rho,\Sigma} R(t_1, \ldots, t_n) \text{ iff } \langle \rho(t_1), \ldots, \rho(t_n)\rangle \in \|R\|(\sigma)\\
&\sigma \Vdash_{\rho,\Sigma} \bot \text{ never}\\
&\sigma \Vdash_{\rho,\Sigma} \top \text{ always}\\
&\sigma \Vdash_{\rho,\Sigma} \neg A \text{ iff not } \sigma \Vdash_{\rho,\Sigma} A\\
&\sigma \Vdash_{\rho,\Sigma} A \wedge B \text{ iff } \sigma \Vdash_{\rho,\Sigma} A \text{ and } \sigma \Vdash_{\rho,\Sigma} B\\
&\sigma \Vdash_{\rho,\Sigma} A \vee B \text{ iff } \sigma \Vdash_{\rho,\Sigma} A \text{ or } \sigma \Vdash_{\rho,\Sigma} B\\
&\sigma \Vdash_{\rho,\Sigma} A \supset B \text{ iff } \sigma \Vdash_{\rho,\Sigma} A \text{ implies } \sigma \Vdash_{\rho,\Sigma} B\\
&\sigma \Vdash_{\rho,\Sigma} \exists x\colon \tau.\, A \text{ iff there exists } e \in \|\tau\|(\sigma) \text{ such that } \sigma \Vdash_{\rho[e/x],(\Sigma,\, x\colon \tau)} A\\
&\sigma \Vdash_{\rho,\Sigma} \forall x\colon \tau.\, A \text{ iff } \sigma \Vdash_{\rho[e/x],(\Sigma,\, x\colon \tau)} A \text{ for all } e \in \|\tau\|(\sigma)\\
&\sigma \Vdash_{\rho,\Sigma} \nabla x\colon \iota.\, A \text{ iff } \sigma,\, x\colon \iota \Vdash_{\rho[x/x],\,(\Sigma,\, x\colon \iota)} A
\end{aligned}
$$

The interpretation of formulae is extended to local signatures by:

$$\sigma \Vdash_{\rho,\Sigma} (x_1\colon \iota_1, \ldots, x_n\colon \iota_n) \rhd A \overset{\text{def}}{\iff} \sigma \Vdash_{\rho,\Sigma} \nabla x_1\colon \iota_1. \ldots \nabla x_n\colon \iota_n.\, A$$

For a set of formulae Γ, we define $\sigma \Vdash_{\rho,\Sigma} \Gamma$ as an abbreviation for $\bigwedge_{G\in\Gamma}(\sigma \Vdash_{\rho,\Sigma} G)$. A formula A is *valid in* $\mathfrak{A}$ if, for all Σ-valuations ρ at the empty stage, $\cdot \Vdash_{\rho,\Sigma} A$ holds. A sequent $\Sigma \mid \Gamma \longrightarrow \Delta$ is *valid in* $\mathfrak{A}$ if, for all Σ-valuations ρ at the empty stage, $\cdot \Vdash_{\rho,\Sigma} \Gamma$ implies the existence of a $D \in \Delta$ such that $\cdot \Vdash_{\rho,\Sigma} D$ holds. A structure $\mathfrak{A}$ is a *model* of a set of closed formulae Γ, if all $G \in \Gamma$ are valid in the interpretation relative to $\mathfrak{A}$. A formula (respectively sequent) is *valid* if it is valid in all structures $\mathfrak{A}$.

1.4 Examples

Lambda calculus and its induction principle. As an example of an induction principle that can be justified in the semantics, we consider the induction principle for the untyped λ-calculus Tm. The following induction schema is valid for all

formulae P.

$$\Sigma \mid \begin{array}{l} \forall t, t' \colon Tm.\ P(t) \wedge P(t') \supset P(app\ \langle t, t' \rangle), \\ \forall f \colon (Tm \to Tm).\ (\forall t \colon Tm.\ P(t) \supset P(f\ t)) \supset P(lam\ f) \end{array} \longrightarrow \forall t \colon Tm.\ P(t)$$

That this induction schema is indeed valid follows because the definition of validity uses valuations at the empty stage, so that the quantification in this schema is over closed terms only. It is also possible to quantify over terms with free variables in σ by introducing a local signature σ.

$$\Sigma \mid \begin{array}{l} \sigma \rhd \forall t \colon Tm.\ isVar(t) \supset P(t), \\ \sigma \rhd \forall t, t' \colon Tm.\ P(t) \wedge P(t') \supset P(app\ t\ t'), \\ \sigma \rhd \forall f \colon (Tm \to Tm).\ (\forall t \colon Tm.\ P(t) \supset P(f\ t)) \supset P(lam\ f) \end{array} \longrightarrow \sigma \rhd \forall t \colon Tm.\ P(t)$$

Here, $isVar$ is the predicate expressing that t is a variable. It is interpreted by $\|isVar\|\sigma = \{x \mid \sigma \vdash x \colon Tm,\ x \text{ is a variable}\}$. Note that the interpretation of $isVar$ is closed under order-preserving renaming but not under all substitutions.

Finally, if we let

$$\Gamma_\sigma = \begin{array}{l} \sigma \rhd \forall t \colon Tm.\ isVar(t) \supset P(t), \\ \sigma \rhd \forall t, t' \colon Tm.\ P(t) \wedge P(t') \supset P(app\ t\ t'), \\ \sigma \rhd \forall f \colon (Tm \to Tm).\ (\nabla x \colon Tm.\, P(f\ x)) \supset P(lam\ f) \end{array}$$

and let Γ be the union of all Γ_σ for all σ, then the sequent $\Sigma \mid \Gamma \longrightarrow \sigma' \rhd \forall t \colon Tm.\ P(t)$ is valid for all σ'.

Standard classical logic. The well-known Tarski-style semantics for classical logic is a special case of the above definition. Consider the case where there are no λ-tree types. By the syntactic restrictions, no ∇-quantification is allowed in this case. Furthermore, $\widehat{\mathbf{L}}$ is just the category of sets. Hence, in this case, the above definition of the satisfaction relation coincides with the well-known standard interpretation of classical logic.

1.5 Soundness

For soundness, we first show that the ∇-quantifier commutes with all the other logical connectives. With this property, the proof of soundness is a straightforward induction on derivations. The soundness proof of Miculan & Yemane [17] is similar.

Lemma 1.1 *For any $\theta \colon \Sigma \to \Sigma'$, we have $\sigma \Vdash_{(\theta \circ \rho), \Sigma'} A \iff \sigma \Vdash_{\rho, \Sigma} A[\theta]$.*

Proof. By induction on the structure of the formula A. The only interesting case is the base case, which follows because we have $(\theta \circ \rho)(t) = \rho(t[\theta])$. □

We show that ∇ commutes with the quantifiers, omitting the similar cases for the other connectives for space-reasons.

Lemma 1.2 *The following equivalences hold.*

$$\sigma \Vdash_{\rho,\Sigma} \nabla x\colon \iota.\, \exists y\colon \tau.\, A \quad\Longleftrightarrow\quad \sigma \Vdash_{\rho,\Sigma} \exists h\colon \iota \to \tau.\, \nabla x\colon \iota.\, A[h\ x/y]$$
$$\sigma \Vdash_{\rho,\Sigma} \nabla x\colon \iota.\, \forall y\colon \tau.\, A \quad\Longleftrightarrow\quad \sigma \Vdash_{\rho,\Sigma} \forall h\colon \iota \to \tau.\, \nabla x\colon \iota.\, A[h\ x/y]$$

Proof. We consider the case for the existential quantifier. The other case is dual.

$$\begin{aligned}\sigma \Vdash_{\rho,\Sigma} \nabla x\colon \iota.\, \exists y\colon \tau.\, A &\iff \sigma,\, x\colon \iota \Vdash_{\rho[x/x],\Sigma,\, x\colon \iota} \exists y\colon \tau.\, A \\ &\iff \text{exists } e \in \|\tau\|(\sigma,\, x\colon \iota) \text{ with } \sigma,\, x\colon \iota \Vdash_{\rho[x/x,e/y],\Sigma,\, x\colon \iota,\, y\colon \tau} A\end{aligned}$$

Consider the substitution $[h\ x/y]\colon (\Sigma,\, h\colon (\iota \to \tau),\, x\colon \iota) \to (\Sigma,\, x\colon \iota,\, y\colon \tau)$. By use of the isomorphism $i\colon \|\tau\|(\sigma,\, x\colon \iota) \cong \|\iota \to \tau\|\sigma$, we obtain from e the element $i(e)$ of $\|\iota \to \tau\|\sigma$. The interpretation of application in $\widehat{\mathbf{L}}$ is such that we have $[h\ x/y] \circ \rho[i(e)/h, x/x] = \rho[x/x, e/y]$. Using the above lemma, we can continue as follows.

$$\begin{aligned}&\text{exists } e \in \|\tau\|(\sigma,\, x\colon \iota) \text{ with } \sigma,\, x\colon \iota \Vdash_{\rho[x/x,e/y],\Sigma,\, x\colon \iota,\, y\colon \tau} A \\ &\iff \text{exists } e \in \|\tau\|(\sigma,\, x\colon \iota) \text{ with } \sigma,\, x\colon \iota \Vdash_{\rho[i(e)/h,x/x],\Sigma,\, h\colon (\iota\to\tau),\, x\colon \iota} A[h\ x/y] \\ &\iff \text{exists } e' \in \|\iota \to \tau\|\sigma \text{ with } \sigma,\, x\colon \iota \Vdash_{\rho[e'/h,x/x],\Sigma,\, h\colon (\iota\to\tau),\, x\colon \iota} A[h\ x/y] \\ &\iff \text{exists } e' \in \|\iota \to \tau\|\sigma \text{ with } \sigma \Vdash_{\rho[e'/h],\Sigma,\, h\colon (\iota\to\tau)} \nabla x\colon \iota.\, A[h\ x/y] \\ &\iff \sigma \Vdash_{\rho,\Sigma} \exists h\colon (\iota \to \tau).\, \nabla x\colon \iota.\, A[h\ x/y]\end{aligned}$$

□

Using this lemma, we obtain by induction on derivations:

Proposition 1.3 (Soundness) *Any derivable sequent is valid.*

1.6 Completeness

In this section we show that the well-known completeness argument for first-order logic, see e.g. [5,11], goes through almost unchanged.

Definition 1.4

(i) A *theory* T is a set of closed formulae such that $(\cdot \mid T \longrightarrow A)$ implies $A \in T$

(ii) A theory T is syntactically consistent if $\cdot \mid T \longrightarrow \bot$ is not derivable.

(iii) A theory T is a *Henkin Theory* if, for each closed formula $\exists x\colon \tau.\, A(x)$, there exists a constant $\cdot \vdash c\colon \tau$ such that $(\exists x\colon \tau.\, A(x) \supset A(c)) \in T$ holds.

Lemma 1.5 *Let T be a syntactically consistent theory for the signature $\mathcal{S}$. There exists a signature $\mathcal{S}^*$ extending $\mathcal{S}$ with countably many new constants and a theory T^* for the signature $\mathcal{S}^*$, such that the following hold:*

(i) *T^* is conservative over T;*

(ii) *T^* is a Henkin theory.*

Notice that even though we are extending the signature with closed witnesses only, by $\|\tau\|(\sigma) \cong \|\sigma \to \tau\|(\cdot)$ we can reach all stages of the presheaves. Moreover,

suppose we have $\nabla\sigma.\,\exists x\colon \tau.\, A(x)$. This is equivalent to $\exists h\colon \iota \to \tau.\, \nabla\sigma.\, A(h\ \sigma)$, so that we have a witness c making $\nabla\sigma.\, A(c\ \sigma)$ true.

Lemma 1.6 *For any syntactically consistent theory T, there exists a maximally consistent extension $T^* \supseteq T$ for which the following hold:*

(i) $A \notin T^*$ *iff* $\neg A \in T^*$;

(ii) $A \wedge B \in T^*$ *iff* $A \in T^*$ *and* $B \in T^*$;

(iii) $A \vee B \in T^*$ *iff* $A \in T^*$ *or* $B \in T^*$;

(iv) $A \supset B \in T^*$ *iff* $\{\neg A, B\} \cap T^* \neq \emptyset$;

Lemma 1.7 *If T^* is a maximally consistent extension of T and T is a Henkin theory then T^* is also a Henkin theory.*

Lemma 1.8 *Any syntactically consistent set Γ has a model.*

Proof. Let T be the theory axiomatised by Γ and extend it to a maximally consistent Henkin theory T^*. We define a model from T^*. Base types are interpreted by $\|\tau\|(\sigma) = \{M \mid \sigma \vdash M\colon \tau\}$ and the presheaf action is given by substitution. A relation $R\colon \tau_1 \to \cdots \to \tau_n \to o$ is interpreted by

$$\|R\|(\sigma) = \{\langle t_1, \ldots, t_n\rangle \mid \nabla\sigma.\, R(t_1, \ldots, t_n) \in T^*\} \tag{1}$$

Notice that this presheaf $\|R\|$ only has to be closed under variable renaming, not under all substitutions. That it is indeed closed follows because $\nabla\sigma.\, R(t_1, \ldots, t_n)$ is closed under α-conversion.

It remains to show that this definition does indeed define a model of T^*. It suffices to show the equivalence $\cdot \Vdash_{\rho,\Sigma} A \iff A[\rho] \in T^*$ for all stages σ, all contexts Σ, all formulae A in context Σ and all Σ-valuations ρ at stage σ. The assertion follows from this, since, by letting Σ be the empty context, it can be seen that each formula $A \in T^*$ is valid in the above model.

The proof of the equivalence goes by induction on the number of logical connectives other than ∇ in a formula A. Since we have the well-known equivalences of classical logic, we can restrict our attention to the connectives $\neg$, $\vee$ and $\exists$. We proceed by case-distinction on the outermost connective in A. We show the base-case and the case for the existential quantifier. The other cases are similar.

- A is $\nabla\sigma.\, R(t_1, \ldots, t_n)$. Let $\rho' = \rho[\sigma/\sigma]$. Then we have:

$$\begin{aligned} \cdot \Vdash_{\rho,\Sigma} \nabla\sigma.\, R(t_1, \ldots, t_n) &\iff \sigma \Vdash_{\rho',(\Sigma,\,\sigma)} R(t_1, \ldots, t_n) \\ &\iff \langle \rho'(t_1), \ldots, \rho'(t_n)\rangle \in \|R\|(\sigma) \\ &\iff \nabla\sigma.\, R(t_1[\rho'], \ldots, t_n[\rho']) \in T^* \\ &\iff (\nabla\sigma.\, R(t_1, \ldots, t_n))[\rho] \in T^* \end{aligned}$$

- A is $\nabla\sigma.\, B$, where B is neither of the form $\nabla x\colon \iota.\, C$ nor $R(t_1, \ldots, t_n)$. We consider the representative case where B is $\exists x\colon \tau.\, B'$. Since the formula A is provably equivalent to $\exists h.\, \nabla\sigma.\, B[h\ \sigma/x]$, this case is handled by that for $\exists$ below.
- A is $\exists x\colon \tau.\, B$. From left to right, $\cdot \Vdash_{\rho,\Sigma} \exists x\colon \tau.\, B$ gives $e \in \|\tau\|\cdot$ with $\cdot \Vdash_{\rho[e/x],(\Sigma,\, x\colon \tau)} B$. By induction hypothesis this implies $B[\rho[e/x]] \in T^*$. Using rule $\exists$-R, we de-

rive $\cdot \mid T^* \longrightarrow (B[\rho][e/x]) \supset ((\exists x\colon \tau.\, B)[\rho])$. By maximality of T^*, we get $(\exists x\colon \tau.\, B)[\rho] \in T^*$.

From right to left, suppose $(\exists x\colon \tau.\, B)[\rho] \in T^*$. Since T^* is a Henkin theory, there exists $c \in \|\tau\|\cdot$ such that $((\exists x\colon \tau.\, B)[\rho] \supset B[\rho][e/x]) \in T^*$. By modus ponens and maximality, $B[\rho][e/x] \in T^*$. By induction hypothesis, $\cdot \Vdash_{\rho[e/x],(\Sigma,\, x\colon \tau)} B$. By definition of $\Vdash$, we get the required $\cdot \Vdash_{\rho,\Sigma} \exists x\colon \tau.\, B$.

□

Using this lemma, completeness now follows by a standard argument [5,11].

Proposition 1.9 (Completeness) *Any valid sequent* $\cdot \mid \Gamma \longrightarrow A$ *is derivable.*

2 Part II
Models in Categories with Binding Structure

In Part I of this paper we have given a simple model of classical first-order logic with ∇ that is very close to classical Tarski-semantics. In the second part we generalise this result by giving a general notion of model in *categories with binding structure*. We describe these models in the language of categorical logic, which we assume the reader to be familiar with, see e.g. [14] for an introduction.

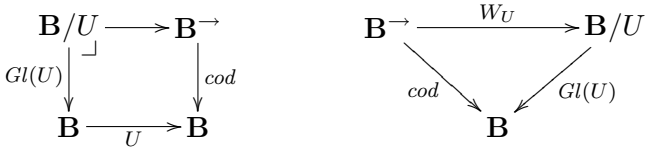
To simplify the presentation, we assume from now on that there is only a single λ-tree type Tm. In this case, the objects of $\mathbf{L}$ may be identified with finite sets. We write Tm also for the presheaf in $\mathbf{L}$ that interprets the type Tm.

2.1 Categories with Binding Structure

Categories with binding structure [20, Chapter 10] axiomatise binding in a general way. The definition of binding structure can be seen as a direct formalisation of the statement: 'Working with an α-equivalence class is the same as working with a freshly named instance'. This statement, expressing that two modes of working are equivalent, is being formalised directly as an equivalence of two categories (Def. 2.1).

Let $\mathbf{B}$ be a category with finite limits and consider its internal language, i.e. the codomain fibration on $\mathbf{B}$. The aim is to explain that constructions with ordinary judgements in the internal language of $\mathbf{B}$ are essentially the same as constructions with judgements that may make use of a fresh name. To formalise 'judgements with a fresh name' we use the glueing construction, see e.g. [24]. Given an endofunctor U on $\mathbf{B}$, consider the pullback as in the left diagram below. In this diagram, $\mathbf{B}^{\to}$ has as objects the morphisms of $\mathbf{B}$, and a morphism in $\mathbf{B}^{\to}$ from $f\colon A \to B$ to $g\colon C \to D$ is a pair $\langle u\colon A \to C, v\colon B \to D\rangle$ of $\mathbf{B}$-morphisms for which $v \circ f = g \circ u$ holds. The functor cod maps objects to their codomain and morphisms $\langle u, v\rangle$ to v. The category $\mathbf{B}/U$ has as objects the $\mathbf{B}$-morphisms of the form $f\colon A \to UB$. Its morphisms from $f\colon A \to UB$ to $g\colon C \to UD$ are pairs $\langle u\colon A \to C, v\colon B \to D\rangle$ of $\mathbf{B}$-morphisms for which $Uv \circ f = g \circ u$ holds. The functor $Gl(U)$ maps $f\colon A \to UB$ to B and $\langle u, v\rangle$ to v. Both cod and $Gl(U)$ are fibrations. There is an canonical functor $W_U\colon \mathbf{B}^{\to} \to \mathbf{B}/U$ making the triangle on the right below commute. Specifically, W_U maps an object

$f\colon A \to B$ to $Uf\colon UA \to UB$.



We use the glued fibration $Gl(- \otimes V)$ for talking about judgements with a fresh name, where $\otimes$ is a monoidal structure and V is an object of $\mathbf{B}$. The intuition is that $A \otimes B$ consists of pairs whose components do not share names, and V is an object of names. Then, the functor $W_{(-\otimes V)}$ adds a fresh name to a judgement. We write short W_V for it.

With this notation, the definition of a category with binding structure is simple:

Definition 2.1 A *category with binding structure* is a triple $(\mathbf{B}, \otimes, V)$ consisting of a category $\mathbf{B}$ with finite limits, a monoidal structure $\otimes$ on $\mathbf{B}$ and an object V of $\mathbf{B}$, such that the functor $W_{(-\otimes V)}$ is an equivalence of fibrations.

Instances of categories with binding structure are given in [20, Chapter 10]. A prime example is the category of nominal sets (also known as the Schanuel topos, or FM-Sets) [7], where $A \otimes B = \{\langle a, b\rangle\colon A \times B \mid a\#b\}$ and V is the set of atoms.

To understand the motivation for Def. 2.1, recall that in categorical logic existential quantification is modelled by a left adjoint to weakening and, dually, universal quantification is modelled by a right adjoint to weakening. The functor W_V can be thought of as a non-standard 'weakening' functor. Since it is an equivalence, there is a functor H that is both left and right adjoint to W_V. By the view of quantifiers as adjoints to weakening, H can be viewed as a non-standard quantifier that is both an existential and a universal quantifier at the same time. It can be shown that H directly generalises the new-quantifier N of Gabbay & Pitts. The defining feature of H is that it preserves all categorical constructions, e.g. $\mathrm{H}(A \Rightarrow B) \cong (\mathrm{H}A) \Rightarrow (\mathrm{H}B)$, which follows because H is part of an equivalence. This suggests a relation to ∇, which has the same defining feature, e.g. $\nabla x.\,(A \supset B) \cong (\nabla x.\,A) \supset (\nabla x.\,B)$.

In [21] and [20], the structure of categories with binding structure used as the basis of a dependent type theory.

Categories with binding structure have rich structure, see [20, Chapter 10]. The properties we use in this paper are given by the next three propositions from [20].

Proposition 2.2 *For each category with binding structure* $(\mathbf{B}, \otimes, V)$*, there exists a functor* $\mathsf{N}\colon \mathbf{Sub}(\Gamma \otimes V) \to \mathbf{Sub}(\Gamma)$ *that is left and right adjoint to* $W_V\colon \mathbf{Sub}(\Gamma) \to \mathbf{Sub}(\Gamma \otimes V)$*.*

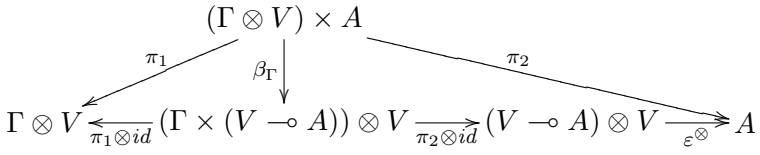
In the statement of this proposition we have used the fact that W_V, being a right adjoint, preserves monomorphisms and so restricts to a functor on subobjects.

Proposition 2.3 *In each category with binding structure* $(\mathbf{B}, \otimes, V)$*, the functor* $(-) \otimes V$ *has a right adjoint* $V \multimap (-)$*, which itself has a further right adjoint. We write* $\varepsilon^{\otimes}$ *for the co-unit of the adjunction* $(-) \otimes V \dashv V \multimap (-)$*.*

If $\mathbf{B}$ is the category of nominal sets then $(V \multimap X)$ is (isomorphic to) the abstraction

set $[\mathbb{A}]X$ of Gabbay & Pitts. The application map $(V \multimap X) \otimes V \to X$ amounts to the concretion operation $x@a$. The binding operation, mapping $a \in \mathbb{A}$ and $x \in X$ to $a.x \in [\mathbb{A}]X$ with $a\#(a.x)$ and $(a.x)@a = x$, is given by β in the next proposition.

Proposition 2.4 *In each category with binding structure $(\mathbf{B}, \otimes, V)$, there is a natural transformation $\beta\colon ((-)\otimes V)\times A \to ((-)\times(V \multimap A))\otimes V$ making the following diagram commute for all objects Γ.*



This proposition has its origin in the work of Menni [16].

2.2 Modelling ∇ in a Category with Binding Structure

As in Part I, we interpret the λ-calculus in $\widehat{\mathbf{L}}$. As outlined in the introduction, the internal logic of this category is not appropriate for modelling a logic with the ∇-quantifier. Instead, we use the internal logic of a category with binding structure $\mathbf{B}$, which we transfer to $\widehat{\mathbf{L}}$ by re-indexing along a functor $F\colon \widehat{\mathbf{L}} \to \mathbf{B}$, as in the following change-of-base situation. This idea has been used with different categories by Hofmann [12] to model the Theory of Contexts [13], see also [3].

$$\begin{array}{ccc} \mathbf{E} & \longrightarrow & \mathbf{Sub}(\mathbf{B}) \\ {\scriptstyle p}\downarrow & \lrcorner & \downarrow{\scriptstyle sub} \\ \widehat{\mathbf{L}} & \xrightarrow{F} & \mathbf{B} \end{array} \tag{2}$$

We need the subobject logic on $\mathbf{B}$ to be strong enough to provide a model for at least first-order logic. Hence, we assume *sub* to be a first-order fibration:

Definition 2.5 A fibration $q\colon \mathbf{E} \to \mathbf{B}$ is a *first-order fibration* if the following hold.

(i) Each fibre $\mathbf{E}_\Gamma$ is a preorder with finite products $(\top, \wedge)$, coproducts $(\bot, \vee)$ and exponents $(\supset)$.

(ii) For each map $u\colon \Gamma \to \Delta$, the re-indexing functor u^* has a left adjoint $\exists_u$ that satisfies the Beck-Chevalley and Frobenius conditions.

(iii) For each map $u\colon \Gamma \to \Delta$, the re-indexing functor u^* has a right adjoint $\forall_u$ that satisfies the Beck-Chevalley condition.

We refer to e.g. [14] for a definition of the Beck-Chevalley and Frobenius conditions.

To model ∇-logic in the fibration p defined in (2), we use the following structure.

Definition 2.6 A ∇-*model* consists of a category with binding structure $(\mathbf{B}, \otimes, V)$ and a functor $F\colon \widehat{\mathbf{L}} \to \mathbf{B}$ with the following additional structure.

(i) The subobject fibration on $\mathbf{B}$ is a first-order fibration.

(ii) The functor F preserves finite limits and has a right-adjoint. We use the notation $\delta_{A,B}\colon FA \times FB \to F(A \times B)$ for the natural isomorphism witnessing product-preservation of F.

(iii) There are a distinguished morphism $\eta\colon V \to F\,Tm$ and natural transformations $\imath\colon FA \otimes FB \to FA \times FB$ and $\theta\colon (V \multimap FA) \to F(Tm \Rightarrow A)$ in $\mathbf{B}$ for which the following diagram commutes.

$$\begin{array}{ccccc} (V \multimap FA) \otimes V & \xrightarrow{\theta \otimes id} & F(Tm \Rightarrow A) \otimes V & \xrightarrow{f_{Tm \Rightarrow A}} & F((Tm \Rightarrow A) \times Tm) \\ {\scriptstyle F\varepsilon^{\otimes}}\downarrow & & & & \downarrow{\scriptstyle F\varepsilon} \\ FA & = & = & = & FA \end{array}$$

Here, $f_{Tm \Rightarrow A}$ is part of the natural transformation $f\colon F(-) \otimes V \to F((-) \times Tm)$ defined by $F\Gamma \otimes V \xrightarrow{\imath} F\Gamma \times V \xrightarrow{id \times \eta} F\Gamma \times F\,Tm \xrightarrow{\delta} F(\Gamma \times Tm)$.

Using η we can map abstract variables in V into the object $F\,Tm$, which encodes the object-level syntax. The morphism θ and the diagram relate binding in the category with binding structure to higher-order abstract syntax.

Next we study the structure of the fibration p in (2). We may assume that the pullback (2) is constructed such that the fibre $\mathbf{E}_A$ over an object A in $\widehat{\mathbf{L}}$ is the partial order $\mathbf{Sub}(FA)$ of subobjects on FA in $\mathbf{B}$. Given $u\colon \Gamma \to \Delta$ in $\widehat{\mathbf{L}}$, the re-indexing functor $u^*\colon \mathbf{E}_\Delta \to \mathbf{E}_\Gamma$ for p is given by $(Fu)^*\colon \mathbf{Sub}(F\Delta) \to \mathbf{Sub}(F\Gamma)$ for *sub*. It is well-known that the structure of a first-order fibration is preserved under re-indexing along a functor F that preserves finite limits and has a right adjoint [12].

Proposition 2.7 *The functor $p\colon \mathbf{E} \to \widehat{\mathbf{L}}$ is a first-order fibration.*

We remark that this proposition can be extended to higher-order logic [12], and all the concrete models that we consider in this paper are indeed models of higher-order logic. This means that it is straightforward to extend our results to defining an interpretation of the type o and to validating higher-order logic.

Definition 2.8 For each object Γ of $\widehat{\mathbf{L}}$, define the functor $\nabla_\Gamma\colon \mathbf{E}_{\Gamma \times Tm} \to \mathbf{E}_\Gamma$ to be the functor $\forall f_\Gamma^*\colon \mathbf{Sub}(F(\Gamma \times Tm)) \to \mathbf{Sub}(F\Gamma)$.

This definition captures the quantification over some/any fresh variable of type Tm. Since $u^*\nabla_\Delta = \nabla_\Gamma(u \times id)^*$ holds for all $u\colon \Gamma \to \Delta$ in $\widehat{\mathbf{L}}$, it is justified to write just ∇.

Now we come to the central property of ∇: it commutes with existential and universal quantification. This is given by the following generalisation of Lemma 1.2.

Lemma 2.9 *For each Γ in $\widehat{\mathbf{L}}$, we have $\nabla\exists_A = \exists_{Tm \Rightarrow A}\nabla s^*$ and $\nabla\forall_A = \forall_{Tm \Rightarrow A}\nabla s^*$, where s is the map $\langle \pi_1 \times id, \varepsilon \circ (\pi_2 \times id)\rangle\colon (\Gamma \times (Tm \Rightarrow A)) \times Tm \to (\Gamma \times Tm) \times A$.*

We remark that to prove this lemma, we make essential use of the binding map β, for moving from a quantification over A to one over $(Tm \Rightarrow A)$.

With the evident translation of formulae in the structure of the first-order fibration p, we now have the following soundness result for the intuitionistic version of the sequent calculus. With Lemma 2.9, the proof is a straightforward induction.

Proposition 2.10 *For any sequent $\Sigma \mid \Gamma \longrightarrow A$ provable in the intuitionistic sequent calculus, there is a morphism $\|\Gamma\| \to \|A\|$ in $\mathbf{E}_{\|\Sigma\|}$.*

In the next section we give concrete examples of ∇-models. These examples do, in fact, all validate classical logic.

2.3 Instances of the Interpretation

Linear Species. The semantics in Part I is an explication of the interpretation in a ∇-model of linear species. The category of linear species [1] is the presheaf category $\widehat{\mathbf{C}^{\mathrm{op}}}$, where $\mathbf{C}$ is the category of finite totally ordered sets with order-preserving bijections. Along the lines of [20, Prop. 10.3.13], $\widehat{\mathbf{C}^{\mathrm{op}}}$ can be seen to be a category with binding structure. The object V is the presheaf with $V\{x\} = \{x\}$ and $V\sigma = \emptyset$ if $|\sigma| \neq 1$. The set $(A \otimes B)\sigma$ consists of pairs $\langle x \in A\sigma_1, y \in B\sigma_2\rangle$ with $\sigma = \sigma_1, \sigma_2$. Then, the set $(V \multimap A)\sigma$ is isomorphic to $A(\sigma, x)$.

To obtain a ∇-model in $\widehat{\mathbf{C}^{\mathrm{op}}}$, we take $F\colon \widehat{\mathbf{L}} \to \widehat{\mathbf{C}^{\mathrm{op}}}$ to be the canonical inclusion, arising because each presheaf in $\widehat{\mathbf{L}}$ is all the more a presheaf in $\widehat{\mathbf{C}^{\mathrm{op}}}$. For the map $\eta\colon V \to FTm$, we take the inclusion function that maps the variable $x \in V\{x\}$ to the term $x\colon Tm$ in $(FTm)\{x\} = \{t \mid x\colon Tm \vdash t\colon Tm\}$. Finally, we define the map $\theta\colon (V \multimap FA) \to F(Tm \to A)$ to be the isomorphism arising from $(V \multimap FA)\sigma \cong FA(\sigma, x)$ in $\widehat{\mathbf{C}^{\mathrm{op}}}$ and $(Tm \to A)\sigma \cong A(\sigma, x)$ in $\widehat{\mathbf{L}}$. Because the functor F is so simple, the conditions of Def. 2.6 are straightforward to verify.

A convenient way of working with the internal language of a topos is the Kripke-Joyal semantics, see e.g. [15]. When spelled out for the linear species model, the Kripke-Joyal semantics specialises exactly to the satisfaction relation $\Vdash$ from Sect. 1.3 (Theorem VI.7.1 of [15]). We should say, however, that the notion of validity in Sect. 1.3 differs from the standard notion of validity from categorical logic. Although Part I can be adapted for the standard notion of validity, we have opted for the non-standard definition, since it matches better with existing work on FOλ^{∇}.

Species. Very similar to the model in linear species is that of ordinary species of structures. In this case, $\mathbf{B}$ is the presheaf category $\widehat{\mathbf{D}^{\mathrm{op}}}$, where $\mathbf{D}$ is the category of finite sets and all bijections. This model differs from the model in linear species in that the local λ-tree contexts are unordered. As a consequence, this model validates the equivalence $\nabla x.\, \nabla y.\, B(x, y) \dashv\vdash \nabla y.\, \nabla x.\, B(x, y)$. Just as for linear species, the interpretation in $\widehat{\mathbf{D}^{\mathrm{op}}}$ can be described in the style of Part I. We expect that the completeness argument can be adapted to this case.

Nominal Sets. The prototypical instance of a category with binding structure is the category of nominal sets $\mathbf{S}$, also known as the Schanuel topos, see [20, Chapter 10]. The category $\mathbf{S}$ provides an instance of Def. 2.6 if we take $F\colon \widehat{\mathbf{L}} \to \mathbf{S}$ to be the composite of the inclusion functor $I\colon \widehat{\mathbf{L}} \to \widehat{\mathbf{I}^{\mathrm{op}}}$, where $\mathbf{I}$ is the category of finite sets and injections, with the sheafification functor $\mathbf{a}\colon \widehat{\mathbf{I}^{\mathrm{op}}} \to \mathbf{S}$ with respect to the atomic topology on $\mathbf{I}^{\mathrm{op}}$. We refer to e.g. [10,15] for more information on this situation.

We use the notation of Gabbay & Pitts [7] to describe the structure of $\mathbf{S}$. Specifically, we take V to be the object of atoms $\mathbb{A}$, and we use the freshness monoidal structure $A \otimes B = \{\langle a \in A, b \in B\rangle \mid a\#b\}$. In $\mathbf{S}$, the functor $L(X) = \mathbb{A} + (X \times X) + [\mathbb{A}]X$ has an initial algebra $[var, app, lam]\colon L(T) \to T$ that

can be used to represent untyped λ-terms. It can be shown that $F\,Tm$ is isomorphic to T, by observing that $I\,Tm$ is already a sheaf. We take $\eta\colon V \to F\,Tm$ to be the map $V = \mathbb{A} \xrightarrow{var} T \cong F\,Tm$. We have to omit the definition of θ for space reasons.

The interpretation of ∇ in $\mathbf{S}$ provides a translation of ∇-logic to Nominal Logic, since the internal logic of $\mathbf{S}$ is (a version of) Nominal Logic. The translation is similar to that described by Gabbay & Cheney in [6]. In particular, the definition of the functor ∇ in Def. 2.8 is such that $\nabla x.\,\varphi(x,y)$ is interpreted as $И n.\,\|\varphi\|(\eta(n),y)$, which agrees with the interpretation in [6]. As observed by Gabbay & Cheney, the interpretation in $\mathbf{S}$ is not complete: it validates $\forall x.\,\varphi \supset \nabla x.\,\varphi$ and $\nabla x.\,\varphi \supset \exists x.\,\varphi$.

3 Conclusion and Further Work

We have explained a simple sound and complete model for classical FOλ^{∇}, and we have worked towards identifying the essential structure of ∇ by giving an abstract model in categories with binding structure. We have shown that ∇ and $И$ can be modelled by the same concept of binding.

In further work, the semantics of intuitionistic FOλ^{∇} should be studied. A starting point in this direction is Cheney's sound and complete translation from FOλ^{∇} into intuitionistic nominal logic [4] together with Gabbay's complete model for intuitionistic nominal logic [8]. Perhaps the semantics from Part I can also be generalised directly to a Kripke-style model.

Regarding other related work, we conjecture that the model for FOλ^{∇} of Miculan & Yemane [17] fits in the general construction of Part II, but the details remain to be worked out. Finally, a-logic, proposed by Gabbay & Gabbay [9], has an informal explanation that appears to be quite similar to our explanation of FOλ^{∇}, and it would be interesting to make precise the relationship.

Acknowledgement

I thank Ian Stark, Marino Miculan, James Cheney and Lennart Beringer for discussions and the referees for their comments. In particular, Ian suggested to use presheaves over finite sets with bijections.

References

[1] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial Species and Tree-like Structures*. Cambridge University Press, 1997.

[2] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *TABLEAUX'05*, volume 3702 of *LNCS*, pages 78–92. Springer-Verlag, 2005.

[3] A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, I. Scagnetto. Consistency of the Theory of Contexts. In *Journal of Functional Programming*, 16(3):327–395, 2006

[4] J.R. Cheney. A simpler proof theory for nominal logic. In *FOSSACS 2005*, number 3441 in LNCS, pages 379–394. Springer-Verlag, 2005.

[5] D. van Dalen. *Logic and Structure*. Springer Verlag, Berlin, 1983.

[6] M.J. Gabbay and J.R. Cheney. A sequent calculus for nominal logic. In *LICS 2004*, pages 139–148. IEEE Computer Society Press, 2004.

[7] M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[8] Murdoch J. Gabbay. Fresh Logic. Pending publication, July 2003.

[9] Murdoch J. Gabbay and Michael J. Gabbay. a-logic. In *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1. College Publications, 2005.

[10] F. Gadducci, M. Miculan, and U. Montanari. About permutation algebras, (pre)sheaves and named sets. In *Higher Order and Symbolic Computation*, 2006.

[11] H. Herre. Logik. Lecture Notes, University of Leipzig, 1999.

[12] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *LICS'99*, 1999.

[13] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning about nominal algebras in HOAS. In *ICALP01*, 2001.

[14] B. Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.

[15] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.

[16] M. Menni. About Π-quantifiers. *Applied Categorical Structures*, 11(5):421–445, 2003.

[17] M. Miculan and K. Yemane. A unifying model of variables and names. In *FOSSACS'05*, volume 3441 of *LNCS*, pages 170–186. Springer-Verlag, 2005.

[18] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

[19] A.M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[20] U. Schöpp. *Names and Binding in Type Theory*. PhD thesis, University of Edinburgh, 2006.

[21] U. Schöpp and I. Stark. A dependent type theory with names and binding. In *CSL'04*, volume LNCS of *3210*, pages 235–249. Springer-Verlag, 2004.

[22] M.R. Shinwell. *The Fresh Approach: functional programming with names and binders*. PhD thesis, University of Cambridge, 2005.

[23] M.R. Shinwell and A.M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[24] P. Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999.

[25] A. Tiu. A logic for reasoning about generic judgements. In *LFMTP'06*, 2006.

Hierarchical nominal terms and their theory of rewriting

Murdoch J. Gabbay[1]

Computer Science Department
Heriot-Watt University, Riccarton
EDINBURGH EH14 4AS
GREAT BRITAIN
murdoch.gabbay@gmail.com

Abstract

Nominal rewriting introduced a novel method of specifying rewriting on syntax-with-binding. We extend this treatment of rewriting with hierarchy of variables representing increasingly 'meta-level' variables, e.g. in hierarchical nominal term rewriting the meta-level unknowns in a rewrite rule, which represent unknown terms, can be 'folded into' the syntax itself (and rewritten). To the extent that rewriting is a mathematical meta-framework for logic and computation, and nominal rewriting is a framework with native support for binders, hierarchical nominal term rewriting is a meta-to-the-omega level framework for logic and computation with binders.

Keywords: Nominal rewriting, meta-theory of logic and programming, nominal techniques.

1 Introduction

Fix $a, b, c, \ldots \in \mathbb{A}$ a set of **atoms** (or **object-level variable symbols**) for the rest of this paper. The syntax of the λ-calculus is inductively generated by the grammar

$$s ::= a \mid ss \mid \lambda a.s.$$

Consider the λ-term '$\lambda a.s$'. Here s is a *meta-level* variable ranging over terms; s is not itself a λ-term.

Mathematical writing is full of this kind of language. Nominal terms model it closely. A relevant subset of nominal terms is inductively generated by the following grammar:

$$u ::= a \mid [a]u \mid \mathsf{f}(u, \ldots, u) \mid X$$

Here X is one of a countably infinite collection of **unknowns symbols** $X, Y, Z, \ldots$. a represents object-level variable symbols, $[a]t$ represents abstraction, f is a term-former, for example λ.

[1] Thanks to Aad Mathijssen and anonymous referees for help and suggestions. We acknowledge the support of EPSRC grant number EP/C013573/1.

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

X here corresponds to s above. $\lambda[a]X$ (the term-former λ syntactically acting on the abstraction of X with a) represents $\lambda a.s$.

Instantiation of X is direct textual replacement and does not avoid capture by abstractors, so $(\lambda[a]X)[a/X]$ is equal to $\lambda[a]a$ (here $[a/X]$ means 'instantiate X to a'). This is exactly what happens when we say 'take s to be a in $\lambda a.s$'; we expect to obtain $\lambda a.a$ and *not* $\lambda a'.a$, as a *capture-avoiding* notion of substitution delivers.

Nominal terms have a well-developed meta-theory [19,5,4]. A table presents the encoding of mathematical discourse into nominal terms:

$$\text{Meta-variable } \phi \text{ or } s \longmapsto \text{Unknown } X \qquad \text{Binding} \longmapsto \text{Abstraction}$$

But we used t and u as meta-variables to range over nominal terms!

So we have not eliminated the meta-level, though we have internalised it. Does a language exist which is a fixed point of this process, in some sense? What if we iterate by allowing abstraction by unknowns $[X]t$, then internalise t as a 'stronger' unknown, and repeat this again, and again, and infinitely often? Taking the limit we obtain hierarchical nominal terms, in which infinitely many levels of meta-level discourse can be represented. What is the mathematics of this new language?

We give a theory of rewriting and a critical pairs result; there turn out to be unexpected differences with respect to nominal terms, which only have one level of atoms. We give example rewriting theories of substitution, scope and scope-extrusion, a λ-calculus, and a treatment of α-equivalence. This is arguably a comprehensive range of applications with which we lay groundwork for more advanced investigations.

2 Hierarchical nominal terms

Fix a set of **term-formers** f.

For each number $i \geq 1$ fix disjoint countably infinite **sets of atoms** $a_i, b_i, c_i, \ldots$. Say that a_i **has level** i.

The syntax of **hierarchical nominal terms** is inductively defined by

$$t ::= a_i \mid X \mid [a_i]t \mid \mathsf{f}(t_1, \ldots, t_n).$$

We may call a_i an 'atom of level i'. The intuition here is of a 'hole' which behaves like a variable towards weaker atoms, and like a constant symbol towards stronger atoms. Intuitively, weaker atoms have no access to stronger atoms; they must wait for those stronger atoms to 'become' terms; stronger atoms on the other hand have full access to weaker atoms, including to their names.

As for the rest of the syntax, $[a_i]t$ is an abstraction and $\mathsf{f}(t_1, \ldots, t_n)$ is a term-former applied to some terms. We shall see examples later; for now it suffices to mention that λ and $\forall$ are example term-formers, but also $+$ and 2, and $\lambda[a_2]c_1$, $\lambda[a_2]b_3$, $2+2$, and a_1+2, are valid hierarchical nominal terms.

Unknowns X are variable symbols representing unknown terms. They behave like atoms of level ω. We still *need* unknowns because *something* has to represent unknown terms so that we can define rewrite rules and do rewriting!

For the rest of this paper we adhere to a convention that i, j, k vary over nonzero natural numbers and a_i, b_i, c_i range *permutatively* over atoms of level i ($a_i \neq b_j$ necessarily). That is, a_i and b_i represent two *distinct* atoms of the same level. If we write a_i and c_k and $i = k$ then by our convention we still assume that a_i and c_k are distinct. Typically it will be the case that $k \leq i < j$, though not always; we shall always be clear about what we assume, when we assume it.

Call a pair of an atom and a term $a_i\#t$ a **freshness assertion**. The intuition is 'a_i does not occur in t'. For example we expect $a_2\#a_1$ to hold, because a_1 is 'far too weak and puny' to ever have a hole as big as a_2. We do not expect $a_1\#a_2$ to hold, necessarily.

Inductively define a notion of entailment on freshness assertions as follows:

$$\frac{k \leq i}{a_i\#c_k}\,(\#\mathbf{diff}) \qquad \frac{a_i\#t_1 \ldots a_i\#t_n}{a_i\#\mathsf{f}(t_1,\ldots,t_n)}\,(\#\mathsf{f})$$

$$\frac{}{a_i\#[a_i]t}\,(\#\mathbf{abs}{=}) \qquad \frac{\begin{array}{c}[a_i\#b_j]\\ \vdots \\ a_i\#t\end{array}}{a_i\#[b_j]t}\,(\#\mathbf{abs}{<})\ (i<j) \qquad \frac{a_i\#t}{a_i\#[c_k]t}\,(\#\mathbf{abs}{\geq})\ (i \geq k)$$

- (#**diff**): This implements our intuition that a strong atom 'looks like' an unknown to weaker atoms but not conversely (since it may contain them, but not conversely). Between atoms of the same level, # encodes distinctness.
- (#**abs**=), (#**abs**<), and (#**abs**≥): a_i is abstracted in $[a_i]t$. a_i is abstracted in $[b_j]t$ when it is abstracted in t — almost! In (#**abs**<) $[a_i\#b_j]$ denotes **discharge** in the natural deduction sense [2]; in sequent style (#**abs**<) would be

$$\frac{\Phi, a_i\#b_j \vdash a_i\#t}{\Phi \vdash a_i\#[b_j]t}$$

  This is a surprising twist not present in normal nominal terms and their freshness [19,5]. We want to be able to derive $a_i\#[b_j]b_j$ always but if $j > i$ this is not derivable using (#**abs**<) without the extra freshness assumption, because if $j > i$ it is not the case in general that $a_i\#b_j$ is derivable using the other rules. This issue does not arise when proving $a_i\#[c_k]u$ for $k \leq i$, because then we can deduce $a_i\#c_k$ with (#**diff**). In particular this issue *cannot* arise if there is only one level, as is the case for nominal terms. [2]
- (#f): An atom is fresh for $\mathsf{f}(t_1,\ldots,t_n)$ when it is fresh for all the t_1 up to t_n.
- There is no rule for deriving $a_i\#X$; the only way to know this, is to assume it beforehand. In this sense X is like an atom of level ω.

Call $a_i\#b_j$ for $j > i$ or $a_i\#X$ **primitive freshness assertions**. Call a possibly infinite set Δ of freshness assertions a **freshness context**. Call Δ **primitive** when all the assertions it contains are primitive.

[2] This insight derives partly from work with Giulio Manzonetto during his visit to Université Paris VII in 2004, while I supported by LIX and the École Polytechnique.

We say $a\#t$ is **entailed** by Δ and write $\Delta \vdash a\#t$, when $a\#t$ can be derived from Δ using these rules. If Δ is empty write $\Delta \vdash a\#t$ just as $\vdash a\#t$. If Δ' is another freshness context write $\Delta \vdash \Delta'$ when $\Delta \vdash a\#t$ for every $a\#t \in \Delta'$.

We now develop the primitive notion of substitution for unknowns, and in the next section we treat unification and finally rewriting.

A **substitution** σ is a map from unknowns to hierarchical nominal terms. We extend the action of substitutions to all hierarchical nominal terms by

$$a_i\sigma = a_i \quad X\sigma = \sigma(X) \quad ([a_i]t)\sigma = [a_i](t\sigma) \quad \mathsf{f}(t_1,\ldots,t_n)\sigma = \mathsf{f}(t_1\sigma,\ldots,t_n\sigma).$$

Extend the substitution action point-wise to things mentioning terms, such as sets of terms and freshness assertions, and sets thereof. For example, if Δ is a freshness context then $\Delta\sigma = \{a_i\#(t\sigma) \mid a_i\#t \in \Delta\}$.

The rules above are highly syntax-directed and have a computational content by which we can calculate for each $a\#t$ a minimal (in a suitable sense) set of assumptions necessary to entail it:

$$b_j\#a_i, \Delta \Longrightarrow \Delta \quad (j > i) \qquad\qquad a_i\#b_i, \Delta \Longrightarrow \Delta$$

$$a_i\#\mathsf{f}(t_1,\ldots,t_n), \Delta \Longrightarrow a_i\#t_1,\ldots,a_i\#t_n,\Delta \qquad a_i\#[a_i]t, \Delta \Longrightarrow \Delta$$

$$\frac{a_i\#t, \Delta \Longrightarrow \Delta' \cup S}{a_i\#[b_j]t, \Delta \Longrightarrow \Delta'} \quad (i{<}j,\ S{\subseteq}\{a_i\#b_j\}) \qquad a_i\#[c_k]t, \Delta \Longrightarrow a_i\#t, \Delta \quad (k{\leq}i)$$

Here we omit singleton set brackets, e.g. writing $a_i\#b_j$ for $\{a_i\#b_j\}$. On the left of the arrow $\Longrightarrow$ comma indicates disjoint set union. On the right of the arrow comma indicates possibly non-disjoint set union. If S and T are sets then $S \setminus T$ is the set of elements in S and not in T.

The following results are easy to prove:

Lemma 2.1 • *If Δ is finite and $\Delta \Longrightarrow \Delta'$ then Δ' is finite.*

• *$\Longrightarrow$ is terminating as a rewrite relation on finite freshness contexts.*

• *$\Longrightarrow$ is confluent on finite freshness contexts (and infinite ones too).*

Proof. The first part is easy.

For the second part assign a numerical measure $|t|$ to terms by: $|a_i| = 1$, $\mathsf{f}(t_1,\ldots,t_n) = \Sigma_{1\leq i\leq n}|t_i| + 1$, $|[a_i]t| = |t| + 1$. Extend the measure to Δ by $|\Delta|$ is a function on numbers $n > 0$ given by $|\Delta|(n)$ is the number of freshness assertions $a\#t$ such that $|t| = n$. For a suitable ordering on such functions (essentially a lexicographic ordering) it is very easy to show that $\Longrightarrow$ makes the measure strictly decrease.

For the third part, we must show that if $\Delta \Longrightarrow \Delta_1$ and $\Delta \Longrightarrow \Delta_2$, then there is some Δ' such that $\Delta_1 \Longrightarrow \Delta'$ and $\Delta_2 \Longrightarrow \Delta'$. We can prove this by considering all possible cases for both reductions; this is long but absolutely routine. □

Write $\langle\Delta\rangle_{nf}$ for the unique $\Longrightarrow$ normal form of Δ. It is not hard to check that $\langle\Delta\rangle_{nf}$ is of the form $\Delta' \cup \Delta''$ where Δ' is a primitive freshness context and

Δ'' contains only problems of the form $a_i\#a_i$. If Δ'' is empty call Δ **consistent**, otherwise call Δ **inconsistent**. Intuitively, Δ is 'satisfiable' if and only if it is consistent. Obviously, Δ is consistent if and only if $\langle\Delta\rangle_{nf}$ is consistent.

Lemma 2.2 *Suppose that Δ is a primitive freshness context and suppose that $\{a_i\#t\}$ is consistent. Then $\Delta \vdash a_i\#t$ if and only if $\Delta \vdash \langle a_i\#t\rangle_{nf}$.*

Proof. By an easy induction on the derivation of $\Delta' \vdash a_i\#t$. □

Lemma 2.3 *If Δ and $\Delta\sigma$ are consistent and $\Delta\vdash a_i\#t$ then $\langle\Delta\sigma\rangle_{nf} \vdash a_i\#(t\sigma)$.*

Proof. By induction on the derivation of $\Delta \vdash a_i\#t$. We consider two cases:

- Suppose our derivation of $\Delta \vdash a_i\#[b_j]u$ concludes with (#**abs**<). Then we have a derivation of $\Delta, a_i\#b_j \vdash a_i\#u$. Note that $a_i\#(b_j\sigma)$ equals $a_i\#b_j$. By inductive hypothesis $\langle\Delta\sigma\rangle_{nf}, a_i\#b_j \vdash a_i\#u\sigma$ is derivable. The result follows.
- Suppose $\Delta \vdash a_i\#X$ holds because $a_i\#X \in \Delta$. By Lemma 2.2 we can deduce that $\langle a_i\#\sigma(X)\rangle_{nf} \vdash a_i\#\sigma(X)$ and by some easy calculations the result follows.

□

3 Unification

An **equality assertion** is a pair of terms $t = u$. We say that '$t = u$ **holds**' when t and u are syntactically identical, we may abbreviate this just to '$t = u$', and we may shorten '$t = u$ does not hold' to $t \neq u$.

A **unification problem** is a set of freshness or equality assertions Φ. We define a **noninstantiating** reduction relation on these unification problems as follows:

$$a_i\#t \Longrightarrow \langle a_i\#t\rangle_{nf} \qquad a_i = a_i, \Phi \Longrightarrow \Phi \qquad X = X, \Phi \Longrightarrow \Phi$$
$$[a_i]t = [a_i]u, \Phi \Longrightarrow t = u, \Phi$$
$$\mathsf{f}(t_1, \ldots, t_n) = \mathsf{f}(u_1, \ldots, u_n), \Phi \Longrightarrow t_1 = u_1, \ldots, t_n = u_n, \Phi$$

Here we omit singleton set brackets, e.g. writing $t = u$ for $\{t = u\}$. On the left of the arrow $\Longrightarrow$, comma indicates disjoint set union. On the right of the arrow comma indicates possibly non-disjoint set union.

Lemma 3.1 *The noninstantiating reductions on unification problems are terminating and confluent.*

We may extend the reduction relation with **instantiating rules** as follows:

$$X = u, \Phi \overset{X\mapsto u}{\Longrightarrow} \Phi[X\mapsto u] \qquad t = X, \Phi \overset{X\mapsto t}{\Longrightarrow} \Phi[X\mapsto t]$$

Here we extend the substitution action point-wise to the terms in the freshness or equality assertions in Φ.

Call the following equality assertions **reduced**:

- $a_i = b_j$.
- $X = t$ and X occurs in t.
- $\mathsf{f}(t_1, \ldots, t_m) = \mathsf{g}(u_1, \ldots, u_n)$ (where f and g are different term-formers).

- $a_i = \mathsf{g}(u_1, \ldots, u_n)$, or $a_i = [b_j]u$, or $a_i = [a_i]u$, or $\mathsf{f}(t_1, \ldots, t_n) = [b_j]u$, or symmetric versions such as $[a_i]t = a_i$.

We may call reduced equality assertions **inconsistent**.

A **solution** to a unification problem Φ is a pair (Γ, σ) of a consistent hierarchical nominal freshness context Γ and a substitution σ such that

- For every $t = u \in \Phi$ it is the case that $t\sigma = u\sigma$.
- For every $a_i \# t \in \Phi$ it is the case that $\Gamma \vdash a_i \# t\sigma$.
- For every X it is the case that X does not occur in $X\sigma$ (or equivalently, $X\sigma\sigma = X\sigma$).

Lemma 3.2 *If* $\Phi \Longrightarrow \Phi'$ *then* (Γ, σ) *solves* Φ *if and only if* (Γ, σ) *solves* Φ'.

Lemma 3.3 *If* $\Phi \overset{X \mapsto t}{\Longrightarrow} \Phi'$ *then* (Γ, σ) *solves* Φ *if and only if* (Γ, σ) *solves* Φ'.

Define a partial ordering on solutions to a hierarchical nominal unification problem by: $(\Gamma', \sigma') < (\Gamma, \sigma)$ when for some σ'' it is the case that $\Gamma' \vdash \Gamma\sigma''$ and $X\sigma' = X\sigma\sigma''$ for all X.

Say a solution to a problem is **principal** when it is a least element in the instantiation ordering amongst solutions to the problem.

Theorem 3.4 $\langle \Phi \rangle_{nf}$ *solves* Φ *and is principal.*

Proof. By a standard proof-method similar to that used to prove Lemma 36 in [5]; the hierarchy causes no difficulties since we are only acting on unknowns. □

4 Hierarchical nominal rewrite rules

To 'do' rewriting we need to be able to address some position within a term (at which to do the rewrite!).

4.1 Positions and rewriting

Say a term has a **position** when it mentions a distinguished unknown, we usually write it -, precisely once (which identifies the position in the term at which that unknown occurs). Let L, C, P vary over terms with a position. Write $C[s]$ for $C[\text{-} \mapsto s]$ and write [-] when the term *is* its (unique) unknown. Since C is only of interest inasmuch as - may be substituted for a term, we tend to silently assume - is fresh, and we may say 'C is a position' when we mean 'C is a term *with* a distinguished position'.

For example, $[a_1](a_1, \text{-})$ is position and $(\text{-}, \text{-})$ is not.

We can now get down to defining rewriting and proving some of its properties. A **hierarchical nominal rewrite rule** is a triple

$$\nabla \vdash l \longrightarrow r$$

where ∇ is a primitive freshness context (primitive freshness contexts are necessarily consistent) and l and r are terms, such that r and ∇ mention only unknowns in l.

If $(\mathbf{R}) = \nabla \vdash l \longrightarrow r$ and $\Delta \vdash t$ is a hierarchical nominal term-in-context, write $\Delta \vdash t \xrightarrow{(\mathbf{R})} u$ and say '$\Delta \vdash t$ rewrites with $(\mathbf{R})$ to u' when

- There is a position C and substitution σ such that
- $\Delta \vdash \nabla\sigma$ and
- $C[l\sigma] = t$, and $C[r\sigma] = u$.

Write $\longrightarrow^*$ for the reflexive transitive closure of $\longrightarrow$. So $\Delta \vdash t \longrightarrow^* u$ holds when $t = u$ or when there is some sequence of $\longrightarrow$-reductions from t to u. If Δ is irrelevant or known we may write $\Delta \vdash t \longrightarrow^* u$ as just $t \longrightarrow^* u$.

Call a possibly infinite set of hierarchical nominal rewrite rules a **hierarchical nominal (term) rewrite system**.

Call a hierarchical nominal rewrite system **confluent** when if $\Delta \vdash t \longrightarrow^* u$ and $\Delta \vdash t \longrightarrow^* u'$, then v exists such that $\Delta \vdash u \longrightarrow^* v$ and $\Delta \vdash u' \longrightarrow^* v$.

Confluence is an important property because it ensures uniqueness of normal forms, a form of determinism. Local confluence is a weaker property, it is defined as 'joinability of peaks'. More precisely:

Call a pair of rewrites of the form $\Delta \vdash t \longrightarrow u_1$ and $\Delta \vdash t \longrightarrow u_2$ a **peak**. Call a hierarchical nominal rewrite system **locally confluent** when if $\Delta \vdash t \longrightarrow u_1$ and $\Delta \vdash t \longrightarrow u_2$, then a v exists such that $\Delta \vdash u_1 \longrightarrow^* v$ and $\Delta \vdash u_2 \longrightarrow^* v$. We may call such a peak **joinable**.

Suppose:

(i) $R_i = \nabla_i \vdash l_i \to r_i$ for $i = 1, 2$ are two rules mentioning disjoint unknowns,

(ii) $l_1 = L[l_1']$ such that $\nabla_1, \nabla_2, l_1' = l_2$ has a principal solution (Γ, θ), so that $l_1'\theta = l_2\theta$ and $\Gamma \vdash \nabla_i\theta$ for $i = 1, 2$.

Then call the pair of terms-in-context

$$\Gamma \vdash (r_1\theta, L\theta[r_2\theta])$$

a **critical pair**. If $L = [\text{-}]$ and R_1, R_2 are copies of the same rule, or if l_1' is an unknown, then we call the critical pair **trivial**[3].

Call a rewrite rule $R = \nabla \vdash l \longrightarrow r$ **uniform** when if $\Delta \vdash t \xrightarrow{R} u$ then $\Delta, \langle a_i\#t\rangle_{nf} \vdash a_i\#u$ for any a_i such that $\langle a_i\#t\rangle_{nf}$ is consistent.

Checking uniformity looks hard. In fact it is not:

Lemma 4.1 *$R = \nabla \vdash l \longrightarrow r$ is uniform if and only if $\nabla, \langle a_i\#l\rangle_{nf} \vdash a_i\#r$ for all a occurring in the syntax of R, and for one atom a which does not.*

Proof. See [5]. □

Uniformity ensures freshness properties are not destroyed by rewriting:

Lemma 4.2 *If R is uniform and $\Delta \vdash t \xrightarrow{R} u$ and $\Delta \vdash a_i\#t$, then $\Delta \vdash a_i\#u$.*

Proof. Suppose $\Delta \vdash a_i\#t$. By uniformity $\Delta, \langle a_i\#t\rangle_{nf} \vdash a_i\#u$. By elementary properties of natural deduction style proofs, $\Delta \vdash a_i\#u$. □

Theorem 4.3 *In a* uniform *rewrite system, peaks which are instances of trivial critical pairs are joinable.*

[3] We assume that unknowns in rules may be renamed. This is standard both in first-order and nominal rewriting [5].

Proof. Suppose two rules $R_i = \nabla_i \vdash l_i \to r_i$ for $i = 1, 2$ have a critical pair

$$\Gamma \vdash (r_1\theta, L\theta[r_2\theta])$$

Then $l_1 = L[l_1']$, and (Γ, θ) is such that $l_1'\theta = l_2\theta$, and $\Gamma \vdash \nabla_1\theta, \nabla_2\theta$. Recall also that we call the critical pair trivial when $L = [\text{-}]$ and R_1, R_2 are copies of the same rule, or l_1' is a unknown.

If R_1 and R_2 are identical, then their rewrites are identical If R_1 and R_2 differ and l_1' is a unknown, then the only way we might not be able to apply R_1 in $L\theta[r_2\theta]$ or its instances, is if some freshness condition on l_1' in ∇_1 is unsatisfiable after R_2, which was satisfiable before R_2. For uniform rules Lemma 4.2 guarantees that this cannot happen. □

5 Rewrites for substitution

Our idea when designing hierarchical nominal rewriting is that it should be able to represent meta-levels and instantiation. We used atoms to represent variable symbols. Our first task is therefore to use the framework of rewriting to give some framework by which atoms may be instantiated to terms.

Introduce a binary term-former sub and sugar $\mathsf{sub}([a]u, t)$ to $u[a{\mapsto}t]$. Rewrites for sub are:

(suba)	$a_i[a_i{\mapsto}X] \longrightarrow X$	
(sub#)	$a_i\#Z \vdash Z[a_i{\mapsto}X] \longrightarrow Z$	
(subaa)	$Z[a_i{\mapsto}a_i] \longrightarrow Z$	
(subf)	$\mathsf{f}(Z_1, \ldots, Z_n)[a_i{\mapsto}X] \longrightarrow \mathsf{f}(Z_1[a_i{\mapsto}X], \ldots, Z_n[a_i{\mapsto}X])$	
(subabs>)	$([c_k]Z)[a_i{\mapsto}X] \longrightarrow [c_k](Z[a_i{\mapsto}X])$	$(i > k)$
(subabs≤)	$b_j\#X \vdash ([b_j]Z)[a_i{\mapsto}X] \longrightarrow [b_j](Z[a_i{\mapsto}X])$	$(i \leq j)$

These are axiom-schemes for all i and j and every n, and for every term-former f (if we like). We could avoid this by enriching syntax of rewrite rules but it does not seem worth the trouble. We always assume at least an axiom **(subsub)**.

Even without term-formers aside from sub, these rules have very interesting structure. The following rewrites are derivable, where here $j > i$ and $k \leq i$:

$$\begin{aligned} Z[a_i{\mapsto}X][b_j{\mapsto}Y] &\longrightarrow^* Z[b_j{\mapsto}Y][a_i{\mapsto}X[b_j{\mapsto}Y]] \\ a_i\#Y \vdash Z[a_i{\mapsto}X][c_k{\mapsto}Y] &\longrightarrow^* Z[c_k{\mapsto}Y][a_i{\mapsto}X[c_k{\mapsto}Y]] \end{aligned} \tag{1}$$

Rewrites for the first case are:

$$\begin{aligned} Z[a_i{\mapsto}X][b_j{\mapsto}Y] &\overset{\textbf{(subf)}}{\longrightarrow^*} \mathsf{sub}(([a_i]Z)[b_j{\mapsto}Y], X[b_j{\mapsto}Y]) \\ &\overset{\textbf{(subabs>)}}{\longrightarrow^*} \mathsf{sub}([a_i](Z[b_j{\mapsto}Y]), X[b_j{\mapsto}Y]) = Z[b_j{\mapsto}Y][a_i{\mapsto}X[b_j{\mapsto}Y]]. \end{aligned}$$

The second case is similar, but we have to use **(subabs≤)** and to do that we must prove $a_i\#Y$.

Thus *strong* substitution distributes over *weak* substitution without avoiding capture whereas *weak* substitution distributes over *strong* substitution but only subject to a capture-avoidance condition $b_j\#X$. Thus $a_2[a_1{\mapsto}2][a_2{\mapsto}a_1] \longrightarrow^* 2$ arguably models *exactly* what we mean when we say 'let t be a in t with a replaced by 2' (where 2 is some term-former; any term would do as well).

Since rewriting is more general than a particular calculus or logic, this example is meant to exhibit capture-avoiding substitution, and non-capture-avoiding instantiation, as two sides of a single unified theory of sub.

Recall that *uniform* rewrite rules satisfy Theorem 4.3.

Theorem 5.1 *The rewrite rules for* sub *are all uniform. Nontrivial critical pairs may be joined. The rules above are locally confluent.*

Proof. By Lemma 4.1 we need only check a finite number of properties such as $\langle a_i\#a_i[a_i{\mapsto}X]\rangle_{nf} \vdash a_i\#X$. They are all routine. It is detailed but routine to check the nontrivial critical pairs. The third part follows by standard reasoning using Theorem 4.3. □

We believe that our rewrite system is confluent but proving this is nontrivial even in the two-level case. The problem is (1) above, which is non-directed and makes terms syntactically larger. These problems *have* been investigated and overcome (see [7] and see the brief discussion in Section 7) but investigating them here is outside the scope of this paper.

6 Scope extrusion of И

Introduce a term-former И. Sugar $И[a_i]t$ to $Иa_i.t$. Read this as 'generate a fresh name a_i in t'.

Our framework can express scope-extrusion rules consistent with this intuition, similar to the behaviour of the π-calculus restriction operator ν [13]. Assume the term-formers and rewrites of substitution above. Introduce rewrites:

$$\begin{array}{ll} (И\#) & b_j\#Z \vdash Иb_j.Z \longrightarrow Z \\ (И\mathbf{sub}) & b_j\#Y \vdash И(([b_j]Z)[a_i{\mapsto}Y]) \longrightarrow И([b_j](Z[a_i{\mapsto}Y])) \qquad (j > i) \end{array}$$

The effect of (И**sub**) is handled by (**subabs**>) when $j \leq i$, so the following rewrite is *always* valid:

$$b_j\#Y \vdash И(([b_j]Z)[a_i{\mapsto}Y]) \longrightarrow И([b_j](Z[a_i{\mapsto}Y]))$$

This beautifully implements that the abstracted atom *really is* private in the scope of И. There is no rewrite

$$(И\mathbf{FALSE}) \qquad b_j\#Z \vdash Z[a_i{\mapsto}Иb_j.Y] \longrightarrow Иb_j.(Z[a_i{\mapsto}Y])$$

because substitution might copy $Иb_j.Y$ and each copy should have a private copy of the fresh atom. For example assume a term-former f and consider scope-extrusion

rewrite rules

$$b_j\#Y \vdash \mathsf{f}(И b_j.X, Y) \longrightarrow И b_j.\mathsf{f}(X,Y) \quad b_j\#X \vdash \mathsf{f}(X, И b_j.Y) \longrightarrow И b_j.\mathsf{f}(X,Y).$$

Then in a context with $b'_j\#Z$ we may first reduce

$$b'_j\#Z \vdash (\mathsf{f}(a_i,a_i))[a_i{\mapsto}И b_j.Z] \overset{(\mathbf{subf}),(\mathbf{suba})}{\longrightarrow^*} \mathsf{f}(И b_j.Z, И b_j.Z) \overset{b_j\#И b_j.Z}{\longrightarrow} И b_j.\mathsf{f}(Z, И b_j.Z) \overset{b'_j\#Z}{\longrightarrow} И b_j.И b'_j.\mathsf{f}(Z,(b'_j\ b_j)Z).$$

In the presence of (**ИFALSE**) there is a second reduction path:

$$b'_j\#Z \vdash (\mathsf{f}(a_i,a_i))[a_i{\mapsto}И b_j.Z] \overset{(\mathbf{ИFALSE})}{\longrightarrow} И b_j.(\mathsf{f}(a_i,a_i)[a_i{\mapsto}Z]) \overset{(\mathbf{subf}),(\mathbf{suba})}{\longrightarrow^*} И b_j.(Z,Z)$$

This is not desired behaviour so we rule out (**ИFALSE**).

Theorem 6.1 (И#) *and* (**Иsub**) *are uniform (and so we can apply tools such as Theorem 4.3 to rewrite systems making use of* И*).*

Proof. We must show that:

$$\begin{array}{c} b_j\#Z,\ \langle a_i\#И b_j.Z\rangle_{nf} \vdash a_i\#Z \qquad b_j\#Z,\ \langle b_j\#И b_j.Z\rangle_{nf} \vdash b_j\#Z \\ b_j\#Z,\ \langle c_k\#[a_i]И b_j.Z\rangle_{nf} \vdash c_k\#И b_j.[a_i]Z \\ b_j\#Z,\ \langle a_i\#[a_i]И b_j.Z\rangle_{nf} \vdash a_i\#И b_j.[a_i]Z \\ b_j\#Z,\ \langle b_j\#[a_i]И b_j.Z\rangle_{nf} \vdash b_j\#И b_j.[a_i]Z. \end{array}$$

We consider a few cases, they are very easy:

- $\langle a_i\#И b_j.Z\rangle_{nf} = \{a_i\#Z\}$. The result follows.
- $b_j\#Z \in \{b_j\#Z\}$. The result follows.
- $\langle c_j\#[a_i]И b_j.Z\rangle_{nf} = c_j\#Z$. The result follows using the derivation rules for freshness assertions.
- $\vdash a_i\#И b_j.[a_i]Z$ and $\vdash b_j\#И b_j.[a_i]Z$ are easy to derive using the derivation rules for freshness assertions. The result follows.

□

7 A hierarchical λ-calculus

Assume term-formers sub, И, λ and app. Sugar $\mathsf{app}(t,u)$ to tu. Sugar $\mathsf{sub}([a]u,t)$ to $u[a{\mapsto}t]$. Rewrites of a hierarchical λ-calculus are givven by the rewrites for sub

and И, along with rewrites:

$$(\beta)\quad (\lambda a_i.Z)X \longrightarrow Z[a_i{\mapsto}X] \qquad (\sigma\#)\quad a_i\#Z \vdash Z[a_i{\mapsto}X] \longrightarrow Z$$
$$(\sigma\mathbf{a})\quad a_i[a_i{\mapsto}X] \longrightarrow X$$
$$(\sigma\mathbf{p})\quad (a_iZ_1\dots Z_n)[b_j{\mapsto}Y] \longrightarrow (a_i[b_j{\mapsto}Y])\dots(Z_n[b_j{\mapsto}Y])$$
$$(\sigma\mathbf{p}')\quad (a_iZ_1\dots Z_n)[a_i{\mapsto}X] \longrightarrow (a_i[a_i{\mapsto}X])\dots(Z_n[a_i{\mapsto}X])$$
$$(\sigma\sigma)\quad Z[a_i{\mapsto}X][b_j{\mapsto}Y] \longrightarrow Z[b_j{\mapsto}Y][a_i{\mapsto}X[b_j{\mapsto}Y]] \quad (j{>}i)$$
$$(\sigma\lambda)\quad a_i\#X \vdash (\lambda a_i.Z)[c_k{\mapsto}X] \longrightarrow \lambda a_i.(Z[c_k{\mapsto}X]) \quad (k{\leq}i)$$
$$(\sigma\lambda')\quad (\lambda a_i.Z)[b_j{\mapsto}Y] \longrightarrow \lambda a_i.(Z[b_j{\mapsto}Y]) \quad (j{>}i)$$
$$(\sigma\mathbf{tr})\quad Z[a_i{\mapsto}a_i] \longrightarrow Z \qquad (И\mathbf{p})\quad n_j\#Y \vdash (Иn_j.X)Y \longrightarrow Иn_j.(XY)$$
$$(И\lambda)\quad \lambda a_i.Иn_j.Z \longrightarrow Иn_j.\lambda a_i.Z \qquad (И\#)\quad n_j\#Z \vdash Иn_j.Z \longrightarrow Z$$
$$(И\sigma)\quad n_j\#X \vdash (Иn_j.Z)[a_i{\mapsto}X] \longrightarrow Иn_j.(Z[a_i{\mapsto}X])$$

This system is discussed in detail elsewhere [6], though we since simplified the presentation. Note the weaker treatment of substitution compared to Section 5, e.g. $(\sigma\mathbf{p})$ and $(\sigma\mathbf{p}')$ and a closely connected *lack* of a rule $(\sigma\sigma')$ corresponding to (1). This is what is needed to avoid (1) while retaining confluence.

Theorem 7.1 *Rewrites in the system above are confluent.*

Proof. For local confluence it suffices by Theorem 4.3 along with some standard further calculations, to check that *nontrivial* critical pairs may be joined. This is detailed work but essentially routine. For confluence we use Theorem 7.2 below taking $\mathcal{R}_1$ to be the system with just (β), and $\mathcal{R}_2$ to be the system with all the other rules. □

Call two hierarchical nominal term rewrite systems $\mathcal{R}_1$ and $\mathcal{R}_2$ when there is no nontrivial critical pair between a pair of rules one in $\mathcal{R}_1$ and the other in $\mathcal{R}_2$.

Theorem 7.2 *If $\mathcal{R}_1$ and $\mathcal{R}_2$ are left-linear, confluent, and orthogonal, then their union is confluent.*

The proof is identical to one the literature [18, Thm 5.10.5].

Call a term which does not mention unknowns and which mentions only atoms of level 1 a **value**. We briefly indicate how to translate the untyped λ-calculus to values: a translates to a_1 (assume some arbitrary injection of untyped λ-calculus variable symbols to atoms of level 1), tu translates to $t'u'$ if t and u translate to t' and u' respectively, and $\lambda a.t$ translates to $Иa_i.([a_i]t')$ if t translates to t'. This translation *is* correct in a natural sense and preserves strong normalisation.

8 α-equivalence

We can use substitution to recover α-equivalence:

$$(\alpha)\qquad b_i\#X \vdash ([a_i]X) \longrightarrow [b_i](X[a_i{\mapsto}b_i])$$
$$(\alpha')\qquad b_i\#Z \vdash Z[a_i{\mapsto}b_i][b_i{\mapsto}Y] \longrightarrow Z[a_i{\mapsto}Y]$$

Lemma 8.1 *In the presence of* (α) *and the rules for* sub except *for* **(subaa)**, *the rewrite* $Z[a_i{\mapsto}a_i] \longrightarrow Z$ *is valid.*

Proof. We use (α) (recall that $Z[a_i{\mapsto}a_i] = \mathsf{sub}([a_i]Z, a_i)$), (α'), and **(sub#)**. □

The rewrite $[a_i]a_i \overset{(\alpha),\textbf{(suba)}}{\longrightarrow^*} [b_i]b_i$ is valid.

This is all very well for terms not mentioning unknowns or atoms that are too strong, but if we have $[a_i]X$ or $[a_i]b_j$ for $j > i$, and want to rename a_i to some b_i such that $b_i\#X$ or $b_i\#b_j$. Where do we find this guaranteed-fresh b_i?

Say Δ has **sufficient freshnesses** when for every finite set S of atoms and/or unknowns, and for every level i, there is an atom $a_i \notin S$ such that $a_i\#b_j \in \Delta$ for every $b_j \in S$, and $a_i\#X \in \Delta$ for every X in Δ. It is not hard to prove the existence of contexts with sufficient freshnesses by an inductive construction. If Δ has sufficient freshnesses then it is infinite.

This achieves the effect of dynamic creation of names, since any syntax we rewrite is finite and we obtain the desired effect of 'always having a fresh atom', which we can always rename within its scope using α-equivalence.

In short, if the freshness context is sufficiently rich then (some fragment) of α-equivalence becomes accessible. The downside is of course that extra rules may mean extra critical pairs if we want to use Theorem 4.3.

So assuming sufficient freshnesses, how do these axioms behave?

Write $(a_i\ b_i)_{c_i} \cdot t$ for the term $\mathsf{И}c_i.t[a_i{\mapsto}c_i][b_i{\mapsto}a_i][c_i{\mapsto}b_i]$.

Lemma 8.2 *Suppose we are rewriting in a context with sufficient freshnesses and suppose* $c_i\#X$ *and* $d_i\#X$. *Then the following rewrites are valid:*

$$(a_i\ b_i)_{c_i} \cdot X \longrightarrow^* (a_i\ b_i)_{d_i} \cdot X \qquad (a_i\ b_i)_{c_i} \cdot X \longrightarrow^* (b_i\ a_i)_{c_i} \cdot X$$

$$(a_i\ b_i)_{c_i} \cdot (a_i\ b_i)_{c_i} \cdot X \longrightarrow^* X.$$

Proof. We just sketch the first reduction, the others are no harder:

$$\begin{aligned}(a_i\ b_i)_{c_i} \cdot X &\overset{(\alpha)}{\longrightarrow} \mathsf{И}d_i.(X[a_i{\mapsto}c_i][b_i{\mapsto}a_i][c_i{\mapsto}a_i][c_i{\mapsto}d_i]) \\ &\overset{(\alpha)}{\longrightarrow} \mathsf{И}d_i.(X[a_i{\mapsto}c_i][b_i{\mapsto}a_i][c_i{\mapsto}d_i][d_i{\mapsto}a_i][c_i{\mapsto}d_i]) \\ &\longrightarrow^* \mathsf{И}d_i.(X[a_i{\mapsto}d_i][b_i{\mapsto}a_i][d_i{\mapsto}a_i])\end{aligned}$$

□

In view of the lemma above we may write just $(a_i\ b_i) \cdot t$ for $(a_i\ b_i)_{c_i} \cdot t$. It is now not hard to prove that:

$$(a_i\ b_i) \cdot (c_i\ d_i) \cdot X \longrightarrow^* (c_i\ d_i) \cdot (a_i\ b_i) \cdot X$$

$$(a_i\ b_i) \cdot (a_i\ d_i) \cdot X \longrightarrow^* (a_i\ d_i) \cdot (d_i\ b_i) \cdot X \qquad (a_i\ a_i) \cdot X \longrightarrow^* X.$$

This suffices to verify that we have implemented a **permutation action** in terms of atom-for-atom substitution. Furthermore:

$$(a_i\ b_i)\cdot \mathsf{f}(t_1,\ldots,t_n) \longrightarrow^* \mathsf{f}((a_i\ b_i)\cdot t_1,\ldots,(a_i\ b_i)\cdot t_n)$$
$$(a_i\ b_i)\cdot a_i \longrightarrow^* b_i \qquad a_i\#X, b_i\#X \vdash (a_i\ b_i)\cdot X \longrightarrow^* X$$
$$(a_i\ b_i)\cdot [a_i]X \longrightarrow^* [b_i](a_i\ b_i)\cdot X \qquad (a_i\ b_i)\cdot [b_i]X \longrightarrow^* [a_i](a_i\ b_i)\cdot X$$
$$(a_i\ b_i)\cdot [c_i]X \longrightarrow^* [c_i](a_i\ b_i)\cdot X.$$

These are characteristic properties of the permutation action on nominal terms. More research on this is needed; in particular a detailed examination of how these axioms make atoms of different levels interact, might give useful information about an appropriate built-in permutation action on hierarchical nominal terms.

9 Conclusions and future work

Many systems formalise aspects of meta-level logic and programming. Examples are first- and higher-order logic [2,20], rewriting [18,12], logical frameworks [1,10,15], and many more including of course nominal-based systems [19,5,4,17,8]. This work is distinct from other investigations in several ways:

We investigated a *hierarchy* of levels modelling increasingly 'meta' treatments of an object-level theory (in the framework of rewriting). Our hierarchy of atoms can accurately capture our intuitions about how meta-level variables should be instantiated. For example the rewrite $a_2[a_1{\mapsto}2][a_2{\mapsto}a_1] \longrightarrow^* 2$ discussed in Section 5 is supposed to model what we mean when we say 'let t be a in t with a replaced by 2'.

As with all nominal-based systems we implement abstraction and freshness as an explicit and *logical* property of terms (e.g. recall the logical derivation rules for freshness from Section 2). But there are twists; the hierarchy demanded changes in derivation rules for freshness assertions, notably (#**abs**<). We also *omit* α-equivalence as primitive, which simplified the framework at the cost of having a weaker theory of equality of terms.

Recall that we have given rewrites for substitution, π-calculus style restriction [13], α-equivalence, and a computationally powerful λ-calculus inspired from previous work [6] for which we exploited the meta-level results about hierarchical nominal term rewrite systems presented here to indicate a particularly concise method of proof for confluence. We have sketched some indication of how the infinite hierarchy can give 'meta-levels within the rewrite system'.

A previous formalisation of the meta-level with an infinite hierarchy is the Russelian type hierarchy [16]. Since its inception a century ago this has sired higher-order logic [20], the polymorphic λ-calculus [9], dependent type systems [14], and higher-order rewriting [12] to name a few. We can see these systems as having a 'hierarchy of meta-levels' in the sense that objects of functional type 'talk about' the types they are functions on. Yet this forces a *particular* notion of meta-level because substitution is capture-avoiding, a syntactic identity (though see [3,11] which discuss an explicit substitution; it is capture-avoiding though) and freshness is not

directly expressed. Our hierarchy of atoms with freshness contexts gives a us a different slant; it remains to relate nominal techniques to higher-order techniques in general, and in the specific case that we have a hierarchy of atoms.

Other future work includes a more profound analysis of the NEW calculus of contexts, a λ-calculus based on the same ideas [6], restoring permutations as built-in, and further analyses of substitution and α-equivalence. We also see logics based on hierarchical nominal terms which can internalise aspects of the meta-level using the hierarchy to avoid inconsistencies. We anticipate many interesting insights into the mathematical content of naming unknowns and the meta-level.

References

[1] Arnon Avron, Furio A. Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.

[2] J. Barwise. An introduction to first-order logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 5–46. North Holland, 1977.

[3] Roel Bloo. *Preservation of Termination for Explicit Substitution*. PhD thesis, Eindhoven University of Technology, Eindhoven, 1997.

[4] James Cheney and Christian Urban. Alpha-prolog: A logic programming language with names, binding and alpha-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283. Springer-Verlag, September 2004.

[5] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. Pending publication, Information and Computation, 2005.

[6] Murdoch J. Gabbay. A new calculus of contexts. In *Proc. 7th Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'2005)*. ACM, 2005.

[7] Murdoch J. Gabbay and Aad Mathijssen. Capture-avoiding substitution as a nominal algebra. Submitted ICTAC'06.

[8] Murdoch J. Gabbay and Aad Mathijssen. Nominal algebra. Available online.

[9] J. Roger Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, July 1997.

[10] Huet, Kahn, and Paulin-Mohring. The COQ tutorial, v7.2. LogiCal Project.

[11] Pierre Lescanne. From lambda-sigma to lambda-upsilon a journey through calculi of explicit substitutions. In *POPL '94: Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–69. ACM Press, 1994.

[12] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, September 1992.

[14] B. Nordstrom, K. Petersson, and J. M. Smith. *Programming in Martin-Lof's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford, 1990. Also online at `http://www.cs.chalmers.se/Cs/Research/Logic/book/`.

[15] Larry Paulson. *The Isabelle reference manual*. Cambridge University Computer Laboratory, February 2001.

[16] B. Russell and A. Whitehead. *Principia Mathematica*. Cambidge University Press, 1910, 1912, 1913. 3 vols.

[17] M.R. Shinwell and A.M. Pitts. Fresh objective caml user manual. Technical Report UCAM-CL-TR-621, University of Cambridge, February 2005.

[18] Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[19] C. Urban, A. M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

[20] Johan van Benthem. Higher-order logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd Edition*, volume 1, pages 189–244. Kluwer, 2001.

A Head-to-Head Comparison of de Bruijn Indices and Names

Stefan Berghofer[1] and Christian Urban[2]

Institut für Informatik
Technische Universität München
Boltzmannstraße 3, 85748 Garching, Germany

Abstract

Often debates about pros and cons of various techniques for formalising lambda-calculi rely on subjective arguments, such as de Bruijn indices are hard to read for humans or nominal approaches come close to the style of reasoning employed in informal proofs. In this paper we will compare four formalisations based on de Bruijn indices and on names from the nominal logic work, thus providing some hard facts about the pros and cons of these two formalisation techniques. We conclude that the relative merits of the different approaches, as usual, depend on what task one has at hand and which goals one pursues with a formalisation.

Keywords: Proof assistants, lambda-calculi, de Bruijn indices, nominal logic work, Isabelle/HOL.

1 Introduction

When formalising lambda-calculi in a theorem prover, variable-binding and the associated notion of alpha-equivalence can cause some difficult problems. To mitigate these problems several formalisation techniques have been introduced. However, discussions about the merits of these formalisation techniques seem to be governed mainly by personal preference than by facts (see [1]). In this paper, we will study four examples and compare two formalisation techniques—de Bruijn indices [6] and names from nominal logic work [10,15]—in order to shed more light on their respective strengths and weaknesses.

In terms of ease and convenience the standard to which techniques for formalising lambda-calculi have to measure up is, in our opinion, the vast corpus of informal proofs in the existing literature. Even if one can find several works about lambda-calculi containing faulty reasoning, on the whole the informal reasoning on "paper" seems to be quite robust, in particular issues arising from binders and alpha-equivalence seem to cause little problems and introduce almost no overhead. (The point of formalising lambda-calculi is to achieve 100% correctness, to provide easy maintenance of proofs and to allow for proofs

[1] Email: berghofe@in.tum.de
[2] Email: urbanc@in.tum.de

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

about languages where a human reasoner is overwhelmed by the sheer number of cases and subtleties to be considered [3].)

When engineering a formal proof in a theorem prover, blindly applying automatic proof tools often leads to a dead end. Usually more successful is the strategy to start with a rough sketch containing a proof idea, and then to try to translate this idea into actual proof steps in the theorem prover. This style of formalising proofs is very much encouraged by the Isar-language of Isabelle [16]. In case of the substitution lemma in the lambda-calculus

> **Substitution Lemma:** If $x \not\equiv y$ and $x \notin FV(L)$, then
>
> $$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

one might start with the following informal proof given by Barendregt [4]:

> **Proof:** By induction on the structure of M.
>
> **Case 1:** M is a variable.
>
> Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.
>
> Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin FV(L)$ implies $L[x := \ldots] \equiv L$.
>
> Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.
>
> **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L. Then by induction hypothesis
>
> $$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L]) \\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$
>
> **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. □

In order to translate this informal proof to proof steps in a theorem prover, one has to decide how to encode lambda-terms and how to define the substitution operation. A naïve choice would be to represent the lambda-terms as the datatype

(1) **datatype** $lam = Var\ name \mid App\ lam\ lam \mid Lam\ name\ lam$

where the type $name$ can, for example, be strings or natural numbers. Since the term-constructor Lam has a concrete name, one has to prove the substitution lemma modulo an explicit notion of alpha-equivalence, that is one has to prove

$$M[x := N][y := L] \approx_\alpha M[y := L][x := N[y := L]] \ .$$

For the substitution operation one might follow Church [5] and define

$$(Var\ y)[x := N] \stackrel{\text{def}}{=} \begin{cases} N & \text{if } x \equiv y \\ Var\ y & \text{otherwise} \end{cases}$$

$$(App\ M_1\ M_2)[x := N] \stackrel{\text{def}}{=} App\ (M_1[x := N])\ (M_2[x := N]) \tag{2}$$

$$(Lam\ x\ M_1)[x := N] \stackrel{\text{def}}{=} Lam\ x\ M_1$$

$$(Lam\ y\ M_1)[x := N] \stackrel{\text{def}}{=} Lam\ z\ (M_1[y := z][x := N])$$

where in the last clause it is assumed that $y \not\equiv x$, and if $x \notin FV(M_1)$ or $y \notin FV(N)$ then $z \equiv y$, otherwise z is the first variable in the sequence $v_0, v_1, v_2, \ldots$ not in M_1 or N.

Unfortunately, with these naïve choices the translation of the informal proof into actual reasoning steps is a nightmare: Already the simple property stating that $L[x := \ldots] \approx_\alpha L$ provided $x \notin FV(L)$ is a tour de force. In nearly all reasoning steps involving Lam one needs the property

$$\text{if } M \approx_\alpha M' \text{ and } N \approx_\alpha N' \text{ then } M[x := N] \approx_\alpha M'[x := N']$$

in order to manually massage the lambda-terms to a suitable form. The "rough sketches" Curry gives for this property extend over 10 pages [5, Pages 94–104]. As can be easily imagined, implementing these sketches results in a rather unpleasant experience with theorem provers—nothing of the sort that makes formalising proofs "addictive in a videogame kind of way" [8, Page 53]. One reason for the difficulties is the fact that Curry's substitution operation is not equivariant—that means is not independent under renamings [10].

The main point of de Bruijn indices and names from the nominal logic work is to allow for more clever methods of representing binders and to substantially reduce the amount of effort needed to formalise proofs. In Section 2 we illustrate this in the context of the substitution lemma. Section 3 contains a brief sketch of the formalisations for the narrowing and transitivity proof of subtyping from the POPLmark-Challenge [3]. Section 4 draws some conclusions.

2 The Substitution Lemma Formalised

2.1 *Version using de Bruijn Indices*

De Bruijn indices are sometimes labelled as a *hack* [3] since they are a very useful implementation technique, but are often dismissed as being unfit for consumption by a human reader. Yet six out of the eleven solutions currently submitted for the theorem proving part of the POPLmark-Challenge are based on some form of de Bruijn indices. This indicates that de Bruijn indices are quite respectable amongst theorem proving experts. In this section, for the benefit of casual users of theorem provers, we want to study in minutiae detail a formalisation of the substitution lemma using this formalisation technique.

We assume the reader is familiar with the de Bruijn notation of lambda-terms using for example the datatype:

$$\textbf{datatype}\ \ dB\ =\ Var\ nat\ \mid\ App\ dB\ dB\ \mid\ Lam\ dB$$

[3] personal communication with N. G. de Bruijn

One central notion when working with de Bruijn indices is the *lifting* operation, written $\uparrow_k^n$ where n is an offset by which the indices greater or equal than k are incremented; k is the upper bound of indices that are regarded as *locally bound*. This operation can be defined as:

$$\uparrow_k^n (Var\ i) \quad \stackrel{\text{def}}{=} \begin{cases} Var\ i & \text{if } i < k \\ Var\ (i+n) & \text{otherwise} \end{cases}$$

$$\uparrow_k^n (App\ M_1\ M_2) \stackrel{\text{def}}{=} App\ (\uparrow_k^n M_1)\ (\uparrow_k^n M_2)$$

$$\uparrow_k^n (Lam\ M_1) \quad \stackrel{\text{def}}{=} Lam\ (\uparrow_{k+1}^n M_1)$$

The substitution of a term N for a variable with index k, written as $[k := N]$, can then be defined as follows:

$$(Var\ i)[k := N] \stackrel{\text{def}}{=} \begin{cases} Var\ i & \text{if } i < k \\ \uparrow_0^k N & \text{if } i = k \\ Var\ (i-1) & \text{if } i > k \end{cases}$$

$$(App\ M_1\ M_2)[k := N] \stackrel{\text{def}}{=} App\ (M_1[k := N])\ (M_2[k := N])$$

$$(Lam\ M)[k := N] \stackrel{\text{def}}{=} Lam\ (M[k+1 := N])$$

Since the type dB is a completely standard datatype, both definitions can be implemented by primitive recursion. The substitution lemma then takes the following form:

> **Substitution Lemma with de Bruijn Indices:** For all indices i, j, with $i \leq j$ we have that
>
> $$M[i := N][j := L] = M[j+1 := L][i := N[j-i := L]]\ .$$

Note that one proves an equation, rather than an alpha-equivalence. Because equational reasoning is usually much better supported by theorem provers or is even a basic notion in their logics, the de Bruijn indices version avoids the manual massaging of terms with respect to alpha-equivalence needed in the version with concrete names. This fact alone already relieves one of much work when formalising this lemma. Notice also that the condition $i \leq j$ is necessary, otherwise the equation does not hold in general.

Like the informal proof by Barendregt, the formalised proof proceeds by induction on the structure of M. Unlike the informal proof, however, the induction hypothesis needs to be strengthened to quantify over *all* indices i and j. This strengthening is necessary in the de Bruijn version in order to get the Lam-case through. With this strengthening the Lam and App case are completely routine. The *non*-routine case in the de Bruijn version is the Var-case where we have to show that

$$(Var\ n)[i := N][j := L] = (Var\ n)[j+1 := L][i := N[j-i := L]] \tag{3}$$

holds for an arbitrary n. Like in the informal proof, we need to distinguish cases so that we can apply the definition of substitution. There are several ways to order the cases; below we have given the cases as they are suggested by the definition of substitution (namely $n < i$, $n = i$ and $n > i$):

- Case $n < i$: We know by the assumption $i \leq j$ that also $n < j$ and $n < j+1$. Therefore both sides of (3) are equal to $Var\ n$.

- Case $n = i$: The left-hand side of (3) is therefore equal to $(\uparrow_0^i N)[j := L]$ and because we know by the assumption $i \leq j$ that $n < j + 1$, the right-hand side is equal to $\uparrow_0^i (N[j - i := L])$. Now we have to show that both terms are equal. For this we prove first the lemma

  (4) $$\forall i, j.\ \text{if } i \leq j \text{ and } j \leq i + m \text{ then } \uparrow_j^n (\uparrow_i^m N) = \uparrow_i^{m+n} N$$

  which can be proved by induction on N. (The quantification over i and j is necessary in order to get the Lam-case through.) This lemma helps to prove the next lemma

  (5) $$\forall k, j.\ \text{if } k \leq j \text{ then } \uparrow_k^i (N[j := L]) = (\uparrow_k^i N)[j + i := L]$$

  which too can be proved by induction on N. (Again the quantification is crucial to get the induction through.) We can now instantiate this lemma with $k \mapsto 0$ and $j \mapsto j - i$, which makes the precondition trivially true and thus we obtain the equation

  $$\uparrow_0^i (N[j - i := L]) = (\uparrow_0^i N)[j - i + i := L]\ .$$

  The term $(\uparrow_0^i N)[j - i + i := L]$ is equal to $(\uparrow_0^i N)[j := L]$, as we had to show. However this last step is surprisingly *not* immediate: it depends on the assumption that $i \leq j$. This is because in theorem provers like Isabelle/HOL and Coq subtraction over natural numbers is defined so that $0 - n = 0$ and consequently the equation $j - i + i = j$ does not hold in general!

- Case $n > i$: Since the right-hand side of (3) equals $(Var(n-1))[j := L]$, we distinguish further three subcases (namely $n - 1 < j$, $n - 1 = j$ and $n - 1 > j$):

  - Subcase $n - 1 < j$: We therefore know also that $n < j + 1$ and thus both sides of (3) are equal to $Var\ (n - 1)$.
  - Subcase $n - 1 = j$: Taking into account that $n > i$ implies $0 < n$, we have also $n = j + 1$ (remember that because of the "quirk" with subtraction, this is not obvious). Hence we can calculate that the left-hand side of (3) equals $\uparrow_0^j L$ and the right-hand side equals $(\uparrow_0^{j+1} L)[i := N[j - i := L]]$. To show that these terms are equal we need the lemma

    (6) $$\forall k, i.\ \text{if } k \leq i \text{ and } i < k + (j + 1) \text{ then } (\uparrow_k^{j+1} L)[i := P] = \uparrow_k^j L$$

    proved by induction on L. Instantiating this lemma with $k \mapsto 0$, $i \mapsto i$ and using the assumption $i \leq j$, we can infer that the preconditions of this lemma hold and thus can conclude that $(\uparrow_0^{j+1} L)[i := N[j - i := L]] = \uparrow_0^j L$.
  - Subcase $n - 1 > j$: We therefore also know that $n > j + 1$. These inequalities in turn imply that both sides of (3) are equal to $Var\ (n - 2)$.

This concludes the proof of the substitution lemma. □

In this formalisation considerable ingenuity is needed when inventing the lemmas (4), (5) and (6). Also they are quite "brittle"—in the sense that they seem to go through just in the form stated. To find them can be a daunting task for an inexperienced user of theorem provers (they are only in little part inspired by the facts needed in the main proof). In practice however they seem to cause few problems, because they "carry over" from language to language, and hence one does not need to "invent the wheel" again for a new language. Theorem proving experts just copy these lemmas from existing formalisations. Indeed when submitting his solution of the POPLmark-Challenge, the first author only minimally adapted to System $F_{<:}$ the proofs Nipkow [9] gave in Isabelle/HOL for the lambda-calculus. Nipkow in turn got his collection of lemmas from Rasmussen [12] who worked

with Isabelle/ZF. Nipkow wrote [9, Page 57]:

> *" Initially I tried to find and prove these lemmas from scratch but soon decided to steal them from Rasmussen's ZF proofs instead, which has obvious advantages:*
>
> *– I did not have to find this collection of non-obvious lemmas myself... "*

Rasmussen seems to have gotten his lemmas from a formalisation by Huet [7] in Coq.

In light of the subtleties and quirks in the proof based on de Bruijn indices, it might be surprising that one does not end up with a proof script of more than 100 lines of code. In fact the formalised proof by Nipkow consists of only a few lines—similar numbers for the lemmas corresponding to (4), (5) and (6). The reason is that one can "optimise" proof scripts by employing automatic proof tools. Such proof tools can make case distinctions and apply definitions without manual interference. However such optimisations are done *after* one has a formal proof like the one described above. As we mentioned earlier, just blindly attacking a problem with automatic proof tools leads to dead ends, except in the most trivial proofs, and the substitution lemma is already too complicated. This is not surprising considering how much ingenuity one needs to invent the lemmas (4), (5) and (6). However, once one knows how the proof proceeds, one can guide the automatic proof tools by providing explicitly the lemmas that lead to a proof. In case of the de Bruijn indices version of the substitution lemma, however, this kind of post-processing is not without pitfalls. For example it helps if the lemma is stated the other way around, namely as

$$M[j+1 := L][i := N[j-i := L]] = M[i := N][j := L]$$

otherwise the simplifier can easily loop. As we shall see next, the proof based on names is much more robust in this respect.

2.2 *Version using the Nominal Datatype Package*

The nominal datatype package [13,15] eases the reasoning with "named" alpha-equivalent lambda-terms; one can define them by

(7) **nominal_datatype** $lam \;=\; Var\ name \mid App\ lam\ lam \mid Lam\ «name»lam$

where $name$ is a type representing atoms [10]—in informal proofs atoms are usually referred to as variables; $«\ldots»$ indicates that a name is bound in Lam. This definition allows one to write lambda-terms as $Lam\ a\ (Var\ a)$. Unlike the naïve representation mentioned in the Introduction, however, the nominal datatype lam stands for alpha-equivalence classes, that means one has equations such as

$$Lam\ x\ (Var\ x) = Lam\ y\ (Var\ y)\ .$$

When formalising the substitution lemma, this will allow us to reap the benefits of equational reasoning. However, it raises a small obstacle for the definition of the substitution operation. Using the infrastructure of the nominal datatype package one can define this

operation as

$$(Var\ y)[x := N] \stackrel{\text{def}}{=} \begin{cases} N & \text{if } x \equiv y \\ Var\ y & \text{otherwise} \end{cases}$$

$$(App\ M_1\ M_2)[x := N] \stackrel{\text{def}}{=} App\ (M_1[x := N])\ (M_2[x := N])$$

$$(Lam\ y\ M_1)[x := N] \stackrel{\text{def}}{=} Lam\ y\ (M_1[x := N]) \qquad \text{provided } y \# (x, N)$$

where the side-constraint $y \# (x, N)$ means that $y \neq x$ and y not free in N. However to ensure that one has indeed defined a function, one needs to verify some properties of the clauses by which substitution is defined (see [11,13] for the details). This requires some small proofs that have no counterpart in the informal proof and in the formalisation based on de Bruijn indices. This need of verifying some properties arises whenever a function is defined by recursion over the structure of alpha-equated lambda-terms.

With the definition of the nominal datatype lam comes the following *strong* structural induction principle [14,15]:

$$\frac{\begin{array}{l}\forall c\, x.\ P\ (Var\ x)\ c \\ \forall c\, M_1\, M_2.\ (\forall d.\ P\ M_1\ d)\ \wedge\ (\forall d.\ P\ M_2\ d)\ \Rightarrow P\ (App\ M_1\ M_2)\ c \\ \forall c\, z\, M.\ z \# c\ \wedge\ (\forall d.\ P\ M\ d)\ \Rightarrow\ P\ (Lam\ z\ M)\ c\end{array}}{P\ M\ c}$$

This induction principle states that if one wants to establish a property P for all lambda-terms M, then, as expected, one has to prove it for the constructors Var, App and Lam. It is called *strong* induction principle because it has Barendregt's variable convention already built in. Barendregt assumes in his informal proof that in the lambda-case the binder z is not equal to x and y, and is not free in N and L. Using the strong induction principle, we will be able to mimic the variable convention by instantiating c, we call this the *context* of the induction, with $c \mapsto (x, y, N, L)$.[4] When it then comes to establishing the Lam-case, we can assume that the binder z is fresh for (x, y, N, L), that means is not equal to x and y, and is not free in N and L. As a result, the induction in the substitution lemma will go through smoothly, just like in Barendregt's informal proof. If the nominal datatype package had *not* provided such strong induction principles, reasoning would be quite inconvenient: one would have to rename binders so that, for example, substitutions can be moved under lambdas.

Despite the excellent notes from Barendregt conveying very well the proof idea, for the formalisation of the substitution lemma we need to supply some details that are left out in his notes. For example in Case 1.2 the details are left out for how to prove the property of

(8) $$x \# L \text{ implies that } L[x := P] = L\ .$$

[4] An aspect we do not dwell on here is the fact that the induction context must always be finitely supported, i.e. mentions only finitely many free names, see [10,15].

where $x \# L$ stands for $x \notin FV(L)$. This fact can be proved by an induction over L using the strong induction principle. For this we make the following instantiations:

$$\begin{aligned} P &\mapsto \lambda L.\lambda(x, P).\ x \# L \ \Rightarrow\ L[x := P] = L \\ M &\mapsto L \\ c &\mapsto (x, P) \end{aligned}$$

As a result, the variable and application case are completely routine. In the lambda-case we have to show that $x \# (Lam\ z\ L_1)$ implies $(Lam\ z\ L_1)[x := P] = (Lam\ z\ L_1)$ with the assumption that $z \# (x, P)$ and the induction hypothesis

$$\forall x, P.\ x \# L_1\ \Rightarrow L_1[x := P] = L_1\ .$$

From the assumption that z is not equal to x and not free in P, we can infer from $x \# (Lam\ z\ L_1)$ that $x \# L_1$ holds and by applying the definition of substitution that $(Lam\ z\ L_1)[x := P] = Lam\ z\ (L_1[x := P])$ holds. Now we just need to apply the induction hypothesis and are done.

Although not obvious from first glance, also in Case 2, in the last step of the calculation where the substitution is pulled back from under the binder λz, there are some details missing from Barendregt's informal proof. In order to get from $Lam\ z\ (M_1[y := L][x := N[y := L]])$ to $(Lam\ z\ M_1)[y := L][x := N[y := L]]$, we need the property that:

(9) $$\text{if } z \# N \text{ and } z \# L \text{ then } z \# (N[y := L])\ .$$

where the preconditions are given by his use of the variable convention. This property, too, can be easily proved by strong induction over the structure of N. In this induction we instantiate the induction context with $c \mapsto (z, y, L)$, because then we can in the Lam-case, say instantiated as $(Lam\ x\ N_1)$, move the substitution under the binder x and also infer from the assumption $z \# (Lam\ x\ N_1)$ that z is also fresh for N_1 (this reasoning step depends on $z \neq x$). Consequently we can apply the induction hypothesis and infer that $z \# (N_1[y := L])$ holds. Again since $z \neq x$, also $z \# (Lam\ x\ N_1[y := L])$ holds and we are done.

The formalisation of the substitution lemma

Substitution Lemma with Names: If $x \neq y$ and $x \# L$ then

$$M[x := N][y := L] = M[y := L][x := N[y := L]]\ .$$

now follows almost to the word Barendregt's informal proof. The variable-case, say with the instantiation $(Var\ z)$, proceeds by a case-analysis with $z = x$, $z \neq x \wedge z = y$ and $z \neq x \wedge z \neq y$. The calculations involved are routine using in the second case the property in (8). The application case does not need any special attention. The lambda-case, too, is relatively easy: by instantiating the induction context with $c \mapsto (x, y, N, L)$, the strong induction principle allows us to assume that the binder is not equal to x and y, and is not free in $N$and L. Consequently we can reason like Barendregt:

$$\begin{aligned} (Lam\ z\ M_1)[x := N][y := L] &= Lam\ z\ (M_1[x := N][y := L]) \\ &= Lam\ z\ (M_1[y := L][x := N[y := L]]) \\ &= (Lam\ z\ M_1)[y := L][x := N[y := L]] \end{aligned}$$

where, as mentioned earlier, in the last equation we make use of the property in (9).

The resulting formalised proof is quite simple: one only has to manually set up the induction and supply the properties (8) and (9) to the automatic proving tools for which it is a straightforward task to complete the proof (similar for the two side lemmas). We take this as an indicator that the formalised proof using names is "simpler" than the one based on de Bruijn indices.

3 Transitivity and Narrowing for Subtyping

Another proof where we can compare names and de Bruijn indices is the transitivity and narrowing proof for the subtyping relation described in the POPLmark-Challenge. This proof is quite tricky involving a simultaneous outer induction over a type and two inner inductions on the definition of the subtyping relation. The "rough notes" from which we can start the formalisations are given in [3] by the authors of this challenge.

3.1 Version using the Nominal Datatype Package

Using the nominal datatype package the types can be defined as

$$\textbf{nominal\_datatype}\ \ ty\ =\ Tvar\ name\ \mid\ Top\ \mid\ Fun\ ty\ ty\ \mid\ All\ ty\ \langle\!\langle name\rangle\!\rangle ty$$

with typing contexts being lists of pairs consisting of a name and a type. A type T is *well-formed* w.r.t. a typing context Γ, written $\Gamma \vdash T$, provided $(supp\ T) \subseteq (dom\ \Gamma)$—that means all free names of T, i.e. its *support* [10], must be included in the domain of the typing context Γ. A *valid* typing context, written $valid\ \Gamma$, is defined inductively by:

$$\frac{}{valid\ []} \qquad \frac{valid\ \Gamma \quad X\ \#\ (dom\ \Gamma) \quad \Gamma \vdash T}{valid\ ((X,T)::\Gamma)}$$

The subtyping relation, written $\Gamma \vdash S <: Q$, can then be inductively defined as follows:

$$\frac{valid\ \Gamma \quad \Gamma \vdash S}{\Gamma \vdash S <: Top}\ Top \qquad \frac{valid\ \Gamma \quad X \in (dom\ \Gamma)}{\Gamma \vdash Tvar\ X <: Tvar\ X}\ Refl$$

$$\frac{(X,S) \in \Gamma \quad \Gamma \vdash S <: T}{\Gamma \vdash Tvar\ X <: T}\ Trans \qquad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash Fun\ S_1\ S_2 <: Fun\ T_1\ T_2}\ Fun$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad X\ \#\ \Gamma \quad (X,T_1)::\Gamma \vdash S_2 <: T_2}{\Gamma \vdash All\ S_1\ X\ S_2 <: All\ T_1\ X\ T_2}\ All$$

These definitions are quite close to the "rough notes" from the POPLmark-Challenge; the only difference is that we had to ensure validity of the typing contexts in the leaves and to explicitly require that the binder X is fresh for Γ in the All-rule. The transitivity and narrowing lemma can then be stated as

Transitivity and Narrowing with Names: For all Γ, S, T, Δ, X, P, M, N:

- $\Gamma \vdash S <: Q$ and $\Gamma \vdash Q <: T$ implies $\Gamma \vdash S <: T$, and
- $\Delta@(X,Q)@\Gamma \vdash M <: N$ and $\Gamma \vdash P <: Q$
  implies $\Delta@(X,P)@\Gamma \vdash M <: N$.

About the proof of this lemma the POPLmark-paper states:

> *" The two parts are proved simultaneously, by induction on the size of Q. The argument for part (2) assumes that part (1) has been established already for the Q in question; part (1) uses part (2) only for strictly smaller Q."*

The main point we want to make here is that the formal proof using names proceeds exactly as stated, while as we shall see later this is *not* the case for the de Bruijn indices version. The main inconvenience with the named approach is, however, that the proof then proceeds by two inner inductions on the definition of the subtyping relation and in order to follow the reasoning on "paper" one has to provide *manually* a strong version of the induction principle for subtyping. This strong induction principle has the form (showing only the premise for the All-inference rule):

$$\frac{\begin{array}{l}\ldots\\ \forall \Gamma\, X\, S_1\, S_2\, T_1\, T_2\, c.\;\; X \,\#\, (c, \Gamma, T_1, S_1) \;\wedge\; \Gamma \vdash T_1 <: S_1 \;\wedge\\ \qquad\qquad (\forall d.\; P\, \Gamma\, T_1\, S_1\, d) \;\wedge\; \Gamma \vdash S_2 <: T_2 \;\wedge\; (\forall d.\; P\, \Gamma\, S_2\, T_2\, d)\\ \qquad\qquad \Rightarrow\; P\, \Gamma\, (All\; S_1\; X\; S_2)\, (All\; T_1\; X\; T_2)\, c\end{array}}{\Gamma \vdash S <: T \Rightarrow P\, \Gamma\, S\, T\, c}$$

where we can assume that $X \,\#\, (c, \Gamma, S_1, T_1)$. These freshness condition are crucial to get the induction through without the need of renaming binders. Unlike the strong structural induction principle that comes with a nominal datatype definition for "free", establishing the strong induction principle for subtyping is quite a task—something one does not want to burden up to the users of the nominal package. But so far, unfortunately, it is entirely burdened onto them. (This might change however in future versions of the nominal datatype package.)

3.2 Version using de Bruijn Indices

Two out of the three solution currently submitted that solve *all* theorem proving parts of the POPLmark-Challenge use de Bruijn indices.[5] The solution of the first author defines types as:

$$\textbf{datatype}\;\; dbT \;=\; Tvar\; nat \;\mid\; Top \;\mid Fun\; dbT\; dbT \;\mid All\; dbT\; dbT$$

with the lifting operation given by:

$$\begin{array}{lcl}\uparrow^n_k\, (Tvar\; i) & \stackrel{\text{def}}{=} & \begin{cases} Tvar\; i & \text{if } i < k \\ Tvar\; (i + n) & \text{otherwise}\end{cases}\\ \uparrow^n_k\, Top & \stackrel{\text{def}}{=} & Top\\ \uparrow^n_k\, (Fun\; S\; T) & \stackrel{\text{def}}{=} & Fun\; (\uparrow^n_k S)(\uparrow^n_k T)\\ \uparrow^n_k\, (All\; S\; T) & \stackrel{\text{def}}{=} & All\; (\uparrow^n_k S)\; (\uparrow^n_{k+1} T)\end{array}$$

Note that the lifting operation preserves the size of a dbT-type. This often allows one to establish facts involving lifting using inductions over the size, if an induction over the structure is not strong enough.

[5] The third uses higher-order abstract syntax in Twelf.

Typing contexts are lists of types and the predicate for valid contexts is defined like in the named variant, except that we do not need freshness constraints when working with de Bruijn indices. One way for defining when a type is well-formed is by using the function

$$\begin{array}{lcl} \mathit{frees}\ j\ (\mathit{Tvar}\ i) & \stackrel{\text{def}}{=} & \begin{cases} \emptyset & \text{if } i < j \\ \{i - j\} & \text{otherwise} \end{cases} \\ \mathit{frees}\ j\ (\mathit{Top}) & \stackrel{\text{def}}{=} & \emptyset \\ \mathit{frees}\ j\ (\mathit{Fun}\ S\ T) & \stackrel{\text{def}}{=} & (\mathit{frees}\ j\ S) \cup (\mathit{frees}\ j\ T) \\ \mathit{frees}\ j\ (\mathit{All}\ S\ T) & \stackrel{\text{def}}{=} & (\mathit{frees}\ j\ S) \cup (\mathit{frees}\ (j+1)\ T) \end{array}$$

and then define the well-formedness judgement $\Gamma \vdash T$ as the proposition $(\forall i \in (\mathit{frees}\ 0\ T).\ i < |\Gamma|)$ where $|\Gamma|$ stands for the length of the list Γ. The look-up function for typing context is written $\Gamma(i)$ and returns the type on the ith place in the list Γ. The inductive definition of the subtyping relation with de Bruijn indices takes then the following form:

$$\frac{\mathit{valid}\ \Gamma \quad \Gamma \vdash S}{\Gamma \vdash S <: \mathit{Top}}\ \mathit{Top} \qquad \frac{\mathit{valid}\ \Gamma \quad \Gamma \vdash \mathit{Tvar}\ i}{\Gamma \vdash \mathit{Tvar}\ i <: \mathit{Tvar}\ i}\ \mathit{Tvar}$$

$$\frac{\Gamma(i) = S \quad \Gamma \vdash (\uparrow_0^{i+1} S) <: T}{\Gamma \vdash \mathit{Tvar}\ i <: T}\ \mathit{Trans} \qquad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \mathit{Fun}\ S_1\ S_2 <: \mathit{Fun}\ T_1\ T_2}\ \mathit{Fun}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad T_1 :: \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \mathit{All}\ S_1\ S_2 <: \mathit{All}\ T_1\ T_2}\ \mathit{All}$$

Whether these definitions require much ingenuity w.r.t. the informal rules given in the POPLmark-paper is a matter of taste, but an undebatable fact is that the proof for the transitivity and narrowing lemma formulated with de Bruijn indices as follows

Transitivity and Narrowing with de Bruijn Indices: For all Γ, S, T, Δ P, M, N:

- $\Gamma \vdash S <: Q$ and $\Gamma \vdash Q <: T$ implies $\Gamma \vdash S <: T$, and
- $\Delta @ Q @ \Gamma \vdash M <: N$ and $\Gamma \vdash P <: Q$
  implies $\Delta @ P @ \Gamma \vdash M <: N$.

does *not* proceed as stated in the informal proof of the POPLmark-Challenge. Once one has set up the (outer) simultaneous induction over the size of Q, the inner induction for transitivity needs to be strengthened to apply not just for Q, but also for *all* types that have the same size as Q. That means the inner induction does *not* establish the property

$$\forall \Gamma\ S\ T.\ \Gamma \vdash S <: Q\ \wedge\ \Gamma \vdash Q <: T\ \Rightarrow \Gamma \vdash S <: T$$

rather the strengthened property

$$\begin{array}{l} \forall Q'\ \Gamma\ S\ T.\ \ (\mathit{size}\ Q) = (\mathit{size}\ Q')\ \wedge\ \Gamma \vdash S <: Q'\ \wedge\ \Gamma \vdash Q' <: T \\ \qquad\qquad \Rightarrow \Gamma \vdash S <: T \end{array}$$

This strengthened property is needed in the narrowing part of the lemma where in the Trans-case one needs transitivity not for Q, but for a lifted version of Q, where however the lifted version has the same size as Q. The interesting details in this case are as follows:

the statement to be proved is

$$\forall \Delta\, \Gamma\, M\, N\, P.\ \Delta@Q@\Gamma \vdash M <: N \Rightarrow \Gamma \vdash P <: Q \Rightarrow \Delta@P@\Gamma \vdash M <: N$$

and its proof proceeds by an (inner) induction over the left-most subtyping relation. With the induction infrastructure [17] of Isabelle, we can implement this induction as stated above, without having to introduce "seemingly pointless equalities" [6] that handle syntactic constraints, such as the typing-context being of the form $\Delta@Q@\Gamma$. By induction hypothesis we know that $\Delta@P@\Gamma \vdash (\uparrow_0^{i+1} S) <: T$ and $(\Delta@Q@\Gamma)(i) = S$, and we must show that $\Delta@P@\Gamma \vdash Tvar\ i <: T$ holds. The non-straightforward subcase is where $i = |\Delta|$, because then $(\Delta@P@\Gamma)(i) = P$ and we can infer that S equals Q. We have $\Gamma \vdash P <: Q$ by assumption and hence $\Delta@P@\Gamma \vdash (\uparrow_0^{i+1} P) <: (\uparrow_0^{i+1} Q)$ by weakening. Since $S = Q$ we can now use the transitivity property to infer that $\Delta@P@\Gamma \vdash (\uparrow_0^{i+1} P) <: T$. As can be seen, one needs transitivity for $(\uparrow_0^{i+1} Q)$ rather than for Q as stipulated in the informal proof. We then can conclude by applying the $Trans$-inference rule.

4 Conclusion

We have studied formalisations based on de Bruijn indices and on names from the nominal logic work. The former approach is already well-tested featuring in many formalisations, while the latter is still under heavy development in the nominal datatype package. Extrapolating an amazing amount from the submissions to the POPLmark-Challenge, it seems that all problems occurring in programming meta-theory can, in principle, be solved by theorem proving experts using de Bruijn indices. Further, the reasoning infrastructure needed for de Bruijn indices (mainly arithmetic over natural numbers) has been part of theorem provers, for example Coq and Isabelle/HOL, for a long time. In contrast, the nominal datatype package has been implemented in Isabelle/HOL, only. Except some preliminary work reported in [2], there is little work about replicating our results in non-HOL-based theorem provers.

Another advantage of de Bruijn indices is that they do not introduce any classical reasoning into the formalisation process. In contrast, the nominal datatype package employs in several places classical reasoning principles. It is currently unknown whether a constructive variant of the nominal datatype package that offers the same convenience is attainable. Connected with the aspect of constructivity is the infrastructure to extract programs from proofs, which exists in Isabelle for the proofs with de Bruijn indices, but does not exist at all for proofs using the nominal datatype package.

The biggest disadvantage we see with using the nominal datatype package is the amount of infrastructure that needs to be implemented. So far, this package supports only single binders (although iteration is possible and they can occur anywhere in a term-constructor). One can imagine situations where this is not general enough or requires some unpleasant encodings. Unfortunately, if more general binding structures need to be supported, a considerable body of code must be adapted.

One big advantage of the nominal datatype package, we feel, is the relatively small "gap" between an informal proof on "paper" and an actual proof in a theorem prover. An important point we would like to highlight with this paper is that in the context of theorem proving the fact about de Bruijn indices being hard to read for humans is not the worst aspect: the biggest source of grief for us is the substantial amount of ingenuity needed

[6] See solutions of the POPLmark-challenge by Chlipala and by Stump in Coq.

to translate informal proofs to versions using de Bruijn indices. Since we are also the kind of theorem prover users who copied from existing formalisations when doing our own formalisations with de Bruijn indices, we were quite surprised how much reasoning is involved, if one unravels all the steps needed for the substitution lemma. This is an important aspect if one is in the business of educating students about formal proofs in the lambda-calculus: it is not difficult to imagine that a student will give up with great disgust, if one tries to explain the subtleties of de Bruijn indices in the substitution lemma. We hope therefore that the nominal datatype package will make broad inroads in this area. The slickness with which difficult proofs involving Barendregt's variable convention can be formalised using the nominal datatype package is something we cannot live without anymore.

The conclusion we draw from the comparisons is that the decision about favouring de Bruijn indices or names from the nominal logic work very much depends on what task one has at hand. It would be quite desirable to know how the other main formalisation technique—higher order abstract syntax—fares. But alas, we are not (yet) experts in Twelf, where this technique has been extensively employed.

Acknowledgement

The first author received funding via the BMBF project Verisoft. The second author is supported by an Emmy-Noether fellowship from the German Research Council.

References

[1] *POPLmark maling list*, http://lists.seas.upenn.edu/pipermail/poplmark/.

[2] Aydemir, B., A. Bohannon and S. Weirich, *Nominal Reasoning Techniques in Coq (Work in Progress)*, in: *Proc. of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, To appear in Electronic Notes in Theoretical Computer Science, 2006, pp. 68–75.

[3] Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich and S. Zdancewic, *Mechanized Metatheory for the Masses: The PoplMark Challenge*, in: *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS **3603**, 2005, pp. 50–65.

[4] Barendregt, H., "The Lambda Calculus: Its Syntax and Semantics," Studies in Logic and the Foundations of Mathematics **103**, North-Holland, 1981.

[5] Curry, H. B. and R. Feys, "Combinatory Logic Vol. I," Studies in Logic and the Foundations of Mathematics, North-Holland, 1958.

[6] de Bruijn, N. G., *Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*, Indagationes Math. **34** (1972), pp. 381–392.

[7] Huet, G., *Residual Theory in Lambda-Calculus: A Formal Development*, Journal of Functional Programming **4** (1994), pp. 371–394.

[8] Leroy, X., *Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant*, in: *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2006), pp. 42–54.

[9] Nipkow, T., *More Church-Rosser Proofs (in Isabelle/HOL)*, Journal of Automated Reasoning **26** (2001), pp. 51–66.

[10] Pitts, A. M., *Nominal Logic, A First Order Theory of Names and Binding*, Information and Computation **186** (2003), pp. 165–193.

[11] Pitts, A. M., *Alpha-Structural Recursion and Induction (Extended Abstract)*, in: *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS **3603**, 2005, pp. 17–34.

[12] Rasmussen, O., *The Church-Rosser Theorem in Isabelle: A Proof Porting Experiment*, Technical Report 364, Cambridge University (1995).

[13] Urban, C. and S. Berghofer, *A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL*, in: *Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, LNAI **4130**, 2006, pp. 498–512.

[14] Urban, C. and M. Norrish, *A Formal Treatment of the Barendregt Variable Convention in Rule Inductions*, in: *Proc. of the 3rd International ACM Workshop on Mechanized Reasoning about Languages with Variable Binding and Names (MERLIN)*, 2005, pp. 25–32.

[15] Urban, C. and C. Tasson, *Nominal Techniques in Isabelle/HOL*, in: *Proc. of the 20th International Conference on Automated Deduction (CADE)*, LNCS **3632**, 2005, pp. 38–53.

[16] Wenzel, M., *Isar — A Generic Interpretative Approach to Readable Formal Proof Documents*, in: *Proc. of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, number 1690 in LNCS, 1999, pp. 167–184.

[17] Wenzel, M., *Structured Induction Proofs in Isabelle/Isar*, in: *Proc. of the 5th International Conference on Mathematical Knowledge Management (MKM)*, LNAI **4108**, 2006, p. ??

Nominal Reasoning Techniques in Coq (Work in Progress)

Brian Aydemir Aaron Bohannon Stephanie Weirich

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA

Abstract

We explore an axiomatized nominal approach to variable binding in Coq, using an untyped lambda-calculus as our test case. In our nominal approach, alpha-equality of lambda terms coincides with Coq's built-in equality. Our axiomatization includes a nominal induction principle and functions for calculating free variables and substitution. These axioms are collected in a module signature and proved sound using locally nameless terms as the underlying representation. Our experience so far suggests that it is feasible to work from such axiomatized theories in Coq and that the nominal style of variable binding corresponds closely with paper proofs. We are currently working on proving the soundness of a primitive recursion combinator and developing a method of generating these axioms and their proof of soundness from a grammar describing the syntax of terms and binding.

Keywords: Coq, nominal reasoning techniques, variable binding.

1 Introduction

We present here work on implementing within the Coq proof assistant [2] a "nominal" approach to formalizing syntax with variable binding. This approach is characterized by a close correspondence between common practice on paper and reasoning within Coq. For example:

(i) All occurrences of object-level variables of a given sort (binding, bound, and free) are represented uniformly using *atoms*, an infinite set of objects with decidable equality.

(ii) Alpha-equivalence of object-level terms is represented by Coq's built-in equality, not a separately defined equivalence relation.

Both of these points reflect common practice with pencil and paper formalizations. More generally, our nominal approach is designed to eliminate the need to reason

[1] Email: baydemir@cis.upenn.edu
[2] Email: bohannon@cis.upenn.edu
[3] Email: sweirich@cis.upenn.edu

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

about any terms that do not actually appear in paper proofs, e.g., pre-terms, shifted terms, and exotic terms.

Our ultimate goal is to provide a system that takes as input a specification of a language and produces as output a Coq signature providing the term constructors for this language and axioms about their behavior, including a natural induction principle. The system should also generate a module implementing the signature, thereby proving the signature's soundness. The signature will not define object-level terms as an inductive datatype in Coq. Nevertheless, we believe that the axioms in the signature can be made easily usable by generating a specialized library of tactics and lemmas. Our framework also includes a library containing concepts, such as atoms and swapping (introduced in Section 2), common to all languages.

The primary contributions of this paper are to demonstrate that a nominal approach to variable binding is indeed possible in Coq and to highlight the issues that arise when implementing such an approach in a dependently typed type theory. While we do not yet have the system described above, we have assessed the theoretical and practical viability of this approach in the particular instance of an untyped lambda calculus, while bearing in mind the issues that arise in more complex languages. We feel that our experience with this specific case will allow us to build a complete system as described above.

The rest of this paper is structured as follows. We first describe the foundational components of our approach in Section 2 and the design and implementation of our signature for an untyped lambda calculus in Section 3. We then give some empirical observations about using this signature in Section 4. We discuss related work in Section 5 and conclude in Section 6 with an overview of our ongoing work.

2 Foundations for nominal signatures

As in previous work on nominal approaches for variable binding [7,8,9], we base our work on atoms, swapping, and support. Since swapping and support cannot be defined parametrically for all types, we use an encoding of Haskell-like type classes to quantify over all types for which these notions are defined. We discuss each of these components in this section.

Our development makes extensive use of dependently typed records to capture types which possess certain properties and operations. In the case of atom swapping, the ability to abstract over such records is critical. In other cases, it simply makes our code more flexible. For example, we describe a type for finite sets with extensional equality using the record type `ExtFset`, part of which is shown below.

```
Record ExtFset (T : Set) : Type := mkExtFset {
  extFset : Set;  In : T -> extFset -> Prop;  ...  }
```

The record type is parameterized by `T`, the type of elements carried by the sets. The actual type of finite sets over `T` is given by the field `extFset`, and `In` is a set-membership predicate. The names of these fields are constants (i.e., record field selectors) whose full types are

```
extFset : ∀ T : Set, ExtFset T -> Set
In : ∀ (T : Set) (R : ExtFset T), T -> extFset T R -> Prop .
```

```
Record AtomT : Type := mkAtom {
  atom : Set;   asetR : ExtFset atom;   aset := extFset asetR;
  atom_eqdec : ∀ a b : atom, {a = b} + {a <> b};
  atom_infinite : ∀ F : aset, { b : atom | b ∉ F }  }.
```

Fig. 1. Atoms.

```
Record SwapT (A : AtomT) (X : Set) : Set := mkSwap {
  swap : (A * A) -> X -> X;
  swap_same : ∀ a x, swap (a, a) x = x;
  swap_invol : ∀ a b x, swap (a, b) (swap (a, b) x) = x;
  swap_distrib : ∀ a b c d x,
    swap (a, b) (swap (c, d) x) =
    swap (swapa A (a, b) c, swapa A (a, b) d) (swap (a, b) x) }.
```

Fig. 2. Swapping.

We use Coq's implicit arguments mechanism to infer the arguments `T` and `R` when possible, and we write `x ∉ F` for `not (In x F)` when this can be done.

In general, our use of records implements a dictionary-passing semantics for type classes. Each record type defines a type class, and fields of the record type are fields of the type class. To quantify over only those types which are members of a given type class, we quantify over its dictionary. We do not use modules for this purpose because we cannot quantify over all modules implementing a given signature.

We also use a record type to capture the essential qualities of the variable names in our object languages, namely that there are an infinite number of names and that equality on names is decidable. We call objects with these properties *atoms*; records of type `AtomT`, shown in Figure 1, consist of a type and proofs that the type is a collection of atoms. The field `atom` is the type of the atoms, `aset` is the type of finite sets of atoms, and `atom_eqdec` asserts that equality on the atoms is decidable. The function `atom_infinite`, when supplied any finite set `F` of atoms, produces an atom `b` paired with a proof that `b` is not in `F`. Note that this function requires that the type `atom` be infinite and implements "choosing a fresh atom," an operation whose details are typically left unspecified on paper. With Coq's implicit coercions, for any `A : AtomT`, we may write `A` wherever `atom A` is required. Specifically, whenever `A` occurs in a location where a term of type `Set` is required, Coq implicitly inserts an application of `atom`.

Having characterized atoms, we need to construct a definition for swapping a pair of atoms in arbitrary expressions. Atom swapping is a central concept in nominal approaches for two reasons. First, it is easy to define an appropriate method of swapping atoms on almost any type, including function types and types with nominal binding. Second, it gives us a means to generically specify which names are fresh for any such type. The important properties of atom swapping for any type `X` are specified by the record `SwapT` in Figure 2. The property `swap_same` asserts that swapping an atom with itself must always leave the expression unchanged. The next property states that swapping must be an involution. The final property allows nested swaps to be reordered.

In theory, the user may use any definition of swapping for a given type that

```
Parameters (tmvar : AtomT) (tm : Set).
Parameter var : tmvar -> tm.
Parameter app : tm -> tm -> tm.
Parameter lam : tmvar -> tm -> tm.
Axiom tm_induction : ∀ (P : tm -> Prop) (F : aset tmvar),
  (∀ x : tmvar, P (var x)) ->
  (∀ t : tm, P t -> ∀ u : tm, P u -> P (t @ u)) ->
  (∀ x : tmvar, x ∉ F -> ∀ t : tm, P t -> P (λ x . t)) ->
  (∀ t : tm, P t).
```

Fig. 3. Term constructors and induction principle.

satisfies the properties in `SwapT`, but in practice there is usually a natural one defined by the structure of the type. The simplest form of swapping is the swap of atoms `a` and `b` of type `atom A` applied to the atom `c`, also of type `atom A`, denoted by `swapa A (a, b) c`. We provide the constructor `mkAtomSwap` that uses the `swapa` function to construct the `SwapT` record. For types where no atoms (of the sort being swapped) appear (e.g., the type `nat`), the only reasonable definition of applying a swap is to leave the object unchanged.

Defining how to apply a swap to an expression with a function type is not too difficult, either. Our definition follows Pitts [8] and satisfies the properties in the `SwapT` record (if we allow ourselves an axiom of functional extensionality):

```
Variables (A : AtomT).
Variables (X : Set) (XS : SwapT A X) (Y : Set) (YS : SwapT A Y).
Definition func_swap (a b : A) (f : X -> Y) :=
    fun x => swap YS (a, b) (f (swap XS (a, b) x)).
```

Our framework for atom swapping allows users to define swapping on any *non-dependent* type that lives in the sort `Set`. It is currently unclear whether there is a good way to specify what it means to swap over a dependent type.

3 Signature for an untyped lambda calculus

In this section, we describe the main components of our signature for terms of the untyped lambda calculus. First, our signature includes a declaration of a type for terms, which live in the sort `Set`, and introduction and elimination forms for this type, as shown in Figure 3. Using Coq's notation mechanism, we write `t @ u` for `app t u` and `λ x . t` for `lam x t`. We would like the type `tm` to resemble an inductive type, so our introduction and elimination forms for it are similar to those of types defined by Coq's `Inductive` keyword.

For a natively defined inductive type `X`, Coq generates the definition of a term `X_rect` (using the language constructs `fix` and `match`), which serves as a recursion combinator that can produce results with a dependent type. When specialized to the sort `Prop`, the type of this combinator serves as an induction principle. However, it is not clear how to perform swapping on terms with dependent types, so we cannot axiomatize such a powerful recursion operator in this signature. Instead we axiomatize an independent induction principle. Importantly, this induction principle allows us to reason only about fresh names for the bound variable in the `lam`

```
Parameter fvar : tm -> aset tmvar.
Axiom fvar_lam : ∀ (x : tmvar) (s : tm),
  fvar (lam x s) = remove x (fvar s)
Parameter subst : tm -> tmvar -> tm -> tm
Axiom subst_lam : ∀ (x y : tmvar) (s t : tm),
  x <> y -> x ∉ (fvar t) ->
    (λ x . s) [y := t] = λ x . (s [y := t]).
```

Fig. 4. Free variables and substitution on terms.

```
Axiom swap_var : ∀ (x y z : tmvar),
  (x, y) • (var z) = var ((x, y) ∘ z).
Axiom swap_app : ∀ (x y : tmvar) (t u : tm),
  (x, y) • (t @ u) = ((x, y) • t) @ ((x, y) • u).
Axiom swap_lam : ∀ (x y z : tmvar) (t : tm),
  (x, y) • (λ z . t) = λ ((x, y) ∘ z) . ((x, y) • t).
Axiom eq_lam : ∀ (x y : tmvar) (t : tm),
  y ∉ fvar t -> λ x . t = λ y . ((x, y) • t).
Axiom injection_lam : ∀ (x x' : tmvar) (t t' : tm),
  λ x . t = λ x' . t' ->
    (x = x' ∧ t = t') ∨ (x ∉ fvar t' ∧ t = (x, x') • t').
```

Fig. 5. Axioms for swapping and equality.

case, by taking a finite set of names from which the bound variable is guaranteed to be distinct (recall that `aset tmvar` is the type of *finite* sets of `tmvar`s).

Our signature does not yet include a recursion combinator—we are currently working to provide such an operator (see Section 6). However, for lambda calculus terms, the main use of a recursion combinator is for the definitions of substitution and free variable functions. Therefore, our signature axiomatizes these operations—the axioms for the `lam` cases are shown in Figure 4. Even with a recursion operator, it may make sense to include these operations in a generated signature. Again, we use Coq's notation to write `s [y := t]` for `subst s y t`. Note that `subst_lam` is the only axiom defining the behavior of `subst` on `lam`-abstractions, yet `subst` must be a total function by virtue of its type. Therefore we also axiomatize alpha-equivalence for lambda terms (see Figure 5). Given a `lam`-abstraction, we can use always `eq_lam` to rename the bound variable so that `subst_lam` applies, as on paper.

Axiomatizing equivalence requires a canonical notion of swapping on lambda-terms. Thus, our signature includes the following:

```
Definition tvS := mkAtomSwap tmvar.
Parameter tmS : SwapT tmvar tm.
```

The first line constructs a default definition of swapping for `tmvar` atoms. The second asserts the existence of a definition of swapping on terms. We use Coq's notation mechanism to write `(x, y) ∘ z` for `swap tvS (x, y) z`, which applies a swap of the variable names `x` and `y` to the variable `z`, and `(x, y) • t` for `swap tmS (x, y) t`, which applies the swap to the term `t`. The result of applying a swap to a term is given by three axioms—one for each constructor—and is also shown in Figure 5. Note that this definition simply applies the swap to the

arguments of the constructor, even in the `lam` case.

We have implemented a module with this signature (thereby proving the soundness of our axioms) using a locally nameless [5] implementation of lambda terms where free variables are named and bound variables are encoded using de Bruijn indices. We define `tm` to be the type of locally nameless terms paired with well-formedness proofs indicating that all indices refer to bound variables, and we use an axiom of proof irrelevance to equate well-formedness proofs when comparing terms for equality. Thus, our induction principle allows one to prove properties about all well-formed terms without having to explicitly prove anything about indices. When proving that this principle holds, we assume its premises, in particular that

```
∀ x : tmvar, x ∉ F -> ∀ t : tm, P t -> P (λ x . t) ,
```

and then show that `P` holds for all `x` by induction on the *size* of `x`. The interesting case is when `x` is a locally nameless lambda abstraction, where we need to use the above premise to show that `P x` holds. In the abstraction's body, we instantiate the bound index to a sufficiently fresh name `y`, resulting in a term `t` such that `x = λ y . t`. Since `P t` holds by the induction hypothesis, the above premise implies that `P (λ y . t)` holds. Structural induction on `x` would fail here since `t` is not a subterm of `x`. The remainder of the signature is straightforward to implement.

4 Experience using the signature

The statements of theorems in the nominal style are about as close to those on paper as one could hope for. For example, the following two theorems can be proved from our signature by nominal induction on `M`.

```
Theorem subst_not_fv  : ∀ x M N, x ∉ (fvar M) -> M [x := N] = M.
Theorem subst_comm : ∀ x y M N L, x <> y -> x ∉ (fvar L) ->
  M [x := N] [y := L] = M [y := L] [x := N [y := L]].
```

Proof by induction using the `tm_induction` principle is not significantly different from proofs that would use the `induction` tactic on a standard inductive type. The reasoning in inductive proofs is very similar to that done on paper, too, but does require that we be precise in the lambda case about the set of variables from which the name of the binder must be distinct. Conservatively, we often assert that the bound variable is distinct from all free names appearing in any expression in our context. Using such assumptions requires a little more detail and care than is seen in paper proofs, but seems consistent with the general overhead of mechanization. Furthermore, we hope to automate this process.

Another critical issue that we have attempted to assess is whether it is practical to work from axiomatized equalities in Coq. For instance, since the behavior of `fvar` is axiomatized rather than defined concretely, tactics such as `simpl` cannot unfold its definition. Additionally, since alpha-equivalent terms are not convertible under our signature, there may be cases when it is necessary to use `eq_lam` to rewrite a term in order to apply a given lemma or hypothesis. We have, however, found Coq's `autorewrite` tactic to be quite powerful, allowing common simplifications to be performed automatically, even in cases where the rewrites have preconditions, and convertibility was not an issue in the proofs of the theorems above. Coq's

tactic language has even allowed us to easily perform more complex combinations of simplification and case analysis. There is some room for improvement, but we have found no serious obstacles to working in this style.

5 Related work

Our work is inspired by a nominal datatype package for Isabelle/HOL [1,9]. However, in addition to the common goal of providing automated tools for reasoning about datatypes with binding, we seek to explore the issues that arise when using nominal techniques in a dependently-typed type theory and to make explicit the "signature" required to provide an effective and practically usable formalization of syntax with binding. As in the Isabelle/HOL package, and unlike in nominal logic [7], wherever we require equivariance (the invariance of a relation under swapping) or finite support, we state that requirement explicitly rather than making a global assumption.

As our signature is an axiomatization of lambda-terms and related functions, it is very similar in spirit to Gordon and Melham's axiomatization [3]. It is not clear whether a direct translation of Gordon and Melham's iteration operator could be used to derive a natural induction principle in Coq, even if the development were augmented with axioms from higher-order logic. Additionally, in the `lam` case of their iteration operator, instead of quantifying over the name of the bound variable, they quantify over functions from names to terms. In other words, they provide a "nominal" introduction form for the type of terms, and a weak-HOAS elimination form. Taking into account Norrish's experience using Gordon and Melham's axioms [6], our approach avoids making a direct connection between meta- and object-level binders in favor of a pure nominal approach.

We are not the first to use a "locally nameless" approach to representing syntax with binding. McBride and McKinna [5] give a brief history of the technique, and Leroy used it in his solution [4] to the PoplMark challenge. Our use of this approach, in addition to an axiom of proof irrelevance, is crucial in making Coq's built-in equality coincide with alpha-equality on object-level terms.

6 Ongoing and future work

Our ongoing work includes implementing a combinator for defining functions on terms by primitive recursion, developing a tool to generate signatures and implementations from user-provided grammar specifications, and investigating swapping on dependent types. We discuss below our progress on the recursion combinator.

Taking the work of Pitts [8] as inspiration, we begin by defining what it means for a finite set of atoms to support an object. Intuitively, an object is supported by a set of atoms when the set includes the free names of the object. Freshness then generalizes the idea of when a name is free for an object. Precise definitions are given in Figure 6. Note that the sets that support an object change depending on the definition of swapping used, and hence so do the atoms that may be considered fresh for an object.

Based on our initial attempts to define a recursion combinator, we expect that

```
Variables (A : AtomT) (X : Set) (S : SwapT A X).
Definition supports (F : aset A) (x : X) : Prop :=
  ∀ a b : A, a ∉ F -> b ∉ F -> swap S (a, b) x = x.
Definition fresh (b : A) (x : X) : Prop :=
  ∃ F : aset A, supports F x ∧ b ∉ F.
Parameter tm_rec : ∀ (R : Set) (PR : SwapT tmvar R),
  ∀ f_var : tmvar -> R,
  ∀ f_app : tm -> R -> tm -> R -> R,
  ∀ f_lam : tmvar -> tm -> R -> R,
  ∀ F : aset tmvar, (supports ... F (f_var, f_app, f_lam)) ->
  (∃ b : tmvar, (b ∉ F ∧ ∀ x y, fresh PR b (f_lam b x y))) ->
  (tm -> R).
Axiom tm_rec_lam : ∀ R PR F f_var f_app f_lam supp fcb,
  let f := (tm_rec R PR F f_var f_app f_lam supp fcb) in
    ∀ b t, b ∉ F -> f (lam b t) = f_lam b t (f t).
```

Fig. 6. A recursion operator and related definitions. Ellipses indicate an omitted dictionary argument.

`tm_rec`, shown in Figure 6, can be implemented and `tm_rec_lam` can be proved sound. Except for the side condition `b ∉ F`, the axiom `tm_rec_lam` takes the usual form for a function defined by primitive recursion. The arguments

`∀ F : aset tmvar, (supports ... F (f_var, f_app, f_lam))` and
`∃ b : tmvar, (b ∉ F ∧ ∀ x y, fresh PR b (f_lam b x y))`

to `tm_rec` follow Pitts. The `supports` proposition concisely captures Norrish's requirements on his recursion operator that the functions `f_var`, `f_app`, and `f_lam` "respect permutation" and "not create too many fresh names" [6]. Finally, whereas `tm_rec` can be used to define only non-dependently typed functions, we plan on investigating a combinator for defining dependently typed functions.

References

[1] Berghofer, S. and C. Urban, *Nominal datatype package for Isabelle/HOL*, http://isabelle.in.tum.de/nominal/.

[2] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions," Springer-Verlag, 2004.

[3] Gordon, A. and T. Melham, *Five axioms of alpha-conversion*, in: J. von Wright, J. Grundy and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, LNCS **1125** (1996), pp. 173–190.

[4] Leroy, X., *A locally nameless solution to the POPLmark challenge*, http://cristal.inria.fr/~xleroy/POPLmark/locally-nameless/.

[5] McBride, C. and J. McKinna, *Functional pearl: I am not a number—I am a free variable*, in: *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (2004), pp. 1–9.

[6] Norrish, M., *Recursive function definition for types with binders*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004*, LNCS **3223** (2004), pp. 241–256.

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

[8] Pitts, A. M., *Alpha-structural recursion and induction (extended abstract)*, in: J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005*, LNCS **3603** (2005), pp. 17–34.

[9] Urban, C. and C. Tasson, *Nominal techniques in Isabelle/HOL*, in: R. Nieuwenhuis, editor, *Automated Deduction — CADE-20: 20th International Conference on Automated Deduction*, LNAI **3632** (2005), pp. 38–53.

Practical Reflection for Sequent Logics

Jason Hickey and Aleksey Nogin and Xin Yu and Alexei Kopylov

Department of Computer Science, 256-80
California Institute of Technology
Pasadena, CA 91125
Email: {jyh,nogin,xiny,kopylov}@cs.caltech.edu

Abstract

It is well-known that adding reflective reasoning can tremendously increase the power of a proof assistant. In order for this theoretical increase of power to become accessible to users in practice, the proof assistant needs to provide a great deal of infrastructure to support reflective reasoning. In this paper we explore the problem of creating a practical implementation of such a support layer.

Our implementation takes a specification of a logical theory (which is identical to how it would be specified if we were simply going to reason within this logical theory, instead of reflecting it) and automatically generates the necessary definitions, lemmas, and proofs that are needed to enable the reflected meta-reasoning in the provided theory.

One of the key features of our approach is that the *structure* of a logic is preserved when it is reflected. In particular, all variables, including meta-variables, are preserved in the reflected representation. This also allows the preservation of proof automation—there is a structure-preserving one-to-one map from proof steps in the original logic to proof step in the reflected logic.

To enable reasoning about terms with sequent context variables, we develop a principle for context induction, called *teleportation*.

This work is fully implemented in the MetaPRL theorem prover.

Keywords: Reflection, Higher-Order Abstract Syntax, Meta-Theory, Type Theory, MetaPRL, NuPRL, Languages with Bindings, Mechanized Reasoning.

1 Introduction

By reflection, we mean the ability to use one logic to reason about another, or the ability to use a logic to reason about itself. At its core, a reflection system has two parts. There is a *representation* function, written $\ulcorner t \urcorner$, that defines the representation or "quotation" of a logical formula t. Then, there is a *provability* operator, written $\Box q$, which is a predicate specifying that q is a quotation of a provable formula.

An implementation of a reflection system needs to have two corresponding parts: a specific representation function, and a *mechanized* reflective reasoning (including a definition of $\Box \cdot$ and some degree of reasoning automation)?

The issue of representation is central, and far from trivial. For example, while it is conceptually easy to define a representation function using a Gödel numbering [10], such schemes are impractical as the *structure* of a reflected term (a number)

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

is so different from the original formula. Any plan to re-use mechanized reasoning methods on reflected terms would be extremely difficult.

The challenge is an instance of a general canonical problem—that of using mechanized reasoning to reason about meta-properties of systems, languages, or logics. Our goal is to develop a canonical solution that can be used for meta-reasoning *in general*. In our approach, we use reflection to implement a framework where meta-reasoning is higher-order. For example, one can develop theorems of the form, "Any system that has meta-property P also has meta-property Q," or "Every meta-property of system A is also a meta-property of system B."

However, mechanized reflection is not easy. The general issue is that, if one wants to talk about provability, then it seems necessary to formalize or emulate the theorem prover and its meta-logic. This naïve approach is not only difficult, but it would also require reimplementing the theorem prover within itself. Following Barzilay [4], we aim at reusing the theorem prover instead of reimplementing it.

We present an approach to practical reflection as part of a logical framework, where the representation function $\ulcorner\cdot\urcorner$ is defined over a logic, as well as the formulas, inferences, and theorems that it contains. That is, to develop an account of system $\mathcal{L}$ and its meta-properties, one first defines the system $\mathcal{L}$ as a primitive logic, using the exact same syntax and definition mechanism that are used in not-reflective case. Then, to develop an account of the meta-properties of $\mathcal{L}$, the logic is (automatically) reflected *en masse* to $\ulcorner\mathcal{L}\urcorner$, where each theorem $\mathcal{T}$ in $\mathcal{L}$ is reflected as $\Box_{\mathcal{L}}\ulcorner\mathcal{T}\urcorner$ in $\ulcorner\mathcal{L}\urcorner$, and any proof of $\mathcal{T}$ is reflected to form a proof of $\Box_{\mathcal{L}}\ulcorner\mathcal{T}\urcorner$. In our system, it is not necessary to prepare for reflection. One may develop a theory in the usual way, calling upon reflection if/when it is necessary to perform meta-reasoning.

Of course, this would still not be practical if reasoning in the reflected logic is difficult. The fundamental reason that our approach is practical is that the representation function preserves structure *exactly* in this sense: all variables, including both object and meta-variables, are preserved by the representation. One might call this meta-higher-order abstract syntax. In particular, since we are working with logics that use sequents to express their judgments, the representation function preserves sequent context variables. To do so, we develop a weak induction principle for sequent contexts, called *teleportation*.

The benefit of preserving the term structure is that mechanized reasoning works transparently. That is, there is a one-to-one correspondence from proof steps in the original logic $\mathcal{L}$ to proof steps in $\ulcorner\mathcal{L}\urcorner$. In fact the translation is direct and mechanical, which means that proof automation in the original logic $\mathcal{L}$ also applies in the reflected logic $\ulcorner\mathcal{L}\urcorner$.

This work is implemented in the MetaPRL logical framework [14, 17], and is available at `http://www.metaprl.org/`. The following is a summary of the contributions.

- A representation function $\ulcorner e\urcorner$ that preserves the structure of formula e, specifically preserving object and meta-variables, and all binding structure.
- A one-to-one map from proofs in $\mathcal{L}$ to proofs in the reflected logic $\ulcorner\mathcal{L}\urcorner$.
- A new induction principle, called teleportation, for induction on sequent contexts.
- A practical implementation in the MetaPRL system.

t	$::= x$	object (first-order) variables
	$\mid z[t_1; \cdots ; t_n]$	second-order meta-variables
	$\mid \Gamma \vdash t$	sequents
	$\mid op\{b_1; \cdots ; b_n\}$	concrete terms
b	$::= x_1, \ldots, x_n.t$	bound terms
Γ	$::= h_1; \cdots ; h_n$	sequent contexts
h	$::= X[t_1; \cdots ; t_n]$	context meta-variables [1]
	$\mid x : t$	hypothesis bindings and terms
$\mathcal{L}$	$::= R_1; R_2; \cdots ; R_n$	a logic
R	$::= t_1 \longrightarrow \cdots \longrightarrow t_n$	an inference rule (t_i are closed w.r.t. object variables)

Fig. 1. Syntax of formulas and logics

The organization of the paper is as follows. In Section 2 we develop the syntax and language of logics. This then allows the formal definition of the representation function in Section 3, as well as the definition of provability $\Box\, t$ in Section 4. In order to work with sequent context variables, we develop the teleportation induction principle in Section 5. The final step in Section 6 is to develop methods for proof induction in reflected logics. We present related work in Section 7, and we conclude with a discussion of our approach to reflection in Section 8.

2 Terminology

We assume we are working in a meta-language with sequents, second-order meta-variables, and terms, as shown in Fig. 1. A term t is a formula containing variables, concrete terms, or sequents. A concrete term $op\{b_1; \cdots ; b_n\}$ has a name op, and some subterms $b_1, \ldots, b_n$ that have possible binding occurrences of variables. For example, a term for representing the sum $i + j$ might be defined as $\texttt{add}\{.i; .j\}$ (normally we will omit the leading . if there are no binders, writing it as $\texttt{add}\{i; j\}$). A lambda-abstraction $\lambda x.t$ would include a binding occurrence $\texttt{lambda}\{x.t\}$. Note that here the primitive binding construct is the bound term b, and λ-binders are a defined term. An alternate choice would be to use a single primitive λ binder (for example, as is done in LF [11]).

A sequent $\Gamma \vdash t$ includes a sequent context Γ, which is a sequence of dependent hypotheses $h_1; \cdots ; h_m$, where each hypothesis is a binding $x : t$ or a context variable $X[t_1; \cdots ; t_n]$ (x and X bind to the right). Note that sequents can be arbitrarily nested inside other terms and are not necessarily associated with judgments.

Second-order meta-variables $z[t_1; \cdots ; t_n]$ and context variables $X[t_1; \cdots ; t_n]$ include zero-or more term arguments $t_1, \ldots, t_n$. These meta-variables represent closed *substitution functions*, and are implicitly universally quantified for each rule in which they appear [19]. For example, a second-order variable $z[]$ represents all closed terms (we will normally omit empty bracket, writing simply z). The second-order variable $z[x]$ represents all terms with zero-or-more occurrences of the variable x (that is, any term where x is the only free variable).

[1] Strictly speaking, context variables are *bindings* and meta-variables have context arguments in addition to term argument. This does not affect the presentation until we get to context induction (Section 5, and we omit context arguments for now.

Terms		
$\ulcorner t \urcorner$:	$\ulcorner x \urcorner$	$\equiv x$
	$\ulcorner z[t_1; \cdots ; t_n] \urcorner$	$\equiv z[\ulcorner t_1 \urcorner; \cdots ; \ulcorner t_n \urcorner]$
	$\ulcorner \Gamma \vdash t \urcorner$	$\equiv \ulcorner \Gamma \urcorner \ulcorner \vdash \urcorner \ulcorner t \urcorner$
	$\ulcorner op\{b_1; \cdots ; b_n\} \urcorner$	$\equiv \ulcorner op \urcorner \{\ulcorner b_1 \urcorner; \cdots ; \ulcorner b_n \urcorner\}$
$\ulcorner b \urcorner$:	$\ulcorner x_1, \ldots, x_n.t \urcorner$	$\equiv \lambda_b x_1. \ldots \lambda_b x_n. \ulcorner t \urcorner$
Sequent contexts		
$\ulcorner \Gamma \urcorner$:	$\ulcorner h_1; \cdots ; h_n \urcorner$	$\equiv \ulcorner h_1 \urcorner; \cdots ; \ulcorner h_n \urcorner$
$\ulcorner h \urcorner$	$\ulcorner x : t \urcorner$	$\equiv x : \ulcorner t \urcorner$
	$\ulcorner X[t_1; \cdots ; t_n] \urcorner$	$\equiv X[\ulcorner t_1 \urcorner; \cdots ; \ulcorner t_n \urcorner]$
Rules and logics		
$\ulcorner \mathcal{L} \urcorner$:	$\ulcorner R_1; \cdots ; R_n \urcorner$	$\equiv \ulcorner R_1 \urcorner; \cdots ; \ulcorner R_n \urcorner$
$\ulcorner R \urcorner$:	$\ulcorner t_1 \longrightarrow \cdots \longrightarrow t_n \urcorner$	$\equiv (Z \vdash \Box_{\mathcal{L}} \ulcorner t_1 \urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box_{\mathcal{L}} \ulcorner t_n \urcorner)$

Fig. 2. The definition of the representation function

To illustrate, consider the "substitution lemma" that is valid in many logics. In textbook notation, it might be written as follows, where $t_1[x \leftarrow s]$ represents the substitution of s for x in t_1.

$$\frac{\Gamma, x : t_3, \Delta \vdash t_1 \in t_2 \quad \Gamma, \Delta \vdash s \in t_3}{\Gamma, \Delta \vdash t_1[x \leftarrow s] \in t_2}$$

In our more concrete notation, s, t_1, t_2, t_3 are all represented with second-order variables, and Γ, Δ with context variables. Substitutions are defined using the term arguments; rules are defined using the meta-implication $\cdot \longrightarrow \cdot$, and we consider all meta-variables to be universally quantified in a rule. The concrete version is written as follows (where we use $s \in t$ as a pretty form for a term $\texttt{member}\{s; t\}$, and z_i are second-order meta-variables).

$$\begin{array}{l} (X; x : z_3; Y \vdash z_1[x] \in z_2) \longrightarrow \\ (X; Y \vdash z_0 \in z_3) \longrightarrow \\ (X; Y \vdash z_1[z_0] \in z_2) \end{array} \tag{2.I}$$

In the final sequent, the term $z_1[z_0]$ specifies substitution of z_0 for x in z_1.

Note how the term arguments are used to specify binding precisely—the variable x is allowed to occur free in z_1, but in no other term. The reason we adopt this second-order notation is for this precision. All rule schemata representable with substitution notation are also representable as second-order schemata, but not vice-versa.

For the final part, a logic $\mathcal{L}$ is an ordered sequence of rules. Each rule may be an axiom, or it may be derived from the previous rules in the logic.

3 Representation of reflected terms

We will assume that we are working in the context of a logical framework, so there are at least three logics in consideration—$\mathcal{L}$: the object logic, $\mathbb{M}$: the meta-logic in

which reasoning about the object logic is to be performed; and $\mathbb{F}$: the meta-meta-logic, or framework logic, in which the meta-logic $\mathbb{M}$ is defined. The first step in the reflection process is to define a *representation* of formulas, judgments, rules and theorems of $\mathcal{L}$ in terms of formulas, propositions, and sentences in $\mathbb{M}$.

The representation function $\ulcorner\cdot\urcorner$ produces a quoted form of its argument. As we have mentioned previously, to preserve a one-to-one correspondence between proofs in an original logic $\mathcal{L}$ and its reflected logic $\ulcorner\mathcal{L}\urcorner$, it is important that $\ulcorner\cdot\urcorner$ preserve the structure of the term, including variables, meta-variables, and binding structure. Note that the representation function itself is not a part of the language of the logical framework; it is only a symbol of the "on-paper meta-meta-language" that we use for describing our implementation. Only for operators, $\ulcorner op\urcorner$ refers to some concrete way of reflecting the operator op within the system itself [21].

The representation function is shown in Fig. 2. The parts of interest are the quotations for concrete terms, sequents, and inference rules. The quoted representation of a concrete term, $\ulcorner op\{b_1;\cdots;b_n\}\urcorner$, produces a new term with a quoted name $\ulcorner op\urcorner$, and the quotation is carried out recursively on the subterms $\ulcorner b_1\urcorner;\cdots;\ulcorner b_n\urcorner$.[2] The quotation of a sequent, $\ulcorner\Gamma\vdash t\urcorner$, is similar: the "turnstile operator" is quoted, and the parts are quoted recursively.

The quotation of bound terms introduces a binder, written $\lambda_b x.t$, that represents each binding in quoted form.[2] Note that the binding variable itself is unchanged; the variable is preserved as a binding, but each binding is explicitly coded as a λ_b.

Finally, the quotation of an inference rule, $\ulcorner t_1\longrightarrow\cdots\longrightarrow t_n\urcorner$ becomes a judgment about provability $(Z\vdash\Box_{\mathcal{L}}\ulcorner t_1\urcorner)\longrightarrow\cdots\longrightarrow(Z\vdash\Box_{\mathcal{L}}\ulcorner t_n\urcorner)$. The context variable Z is fresh, and each sequent $Z\vdash\Box_{\mathcal{L}}\ulcorner t_i\urcorner$ is a judgment in the *meta-logic* about provability.

Informally, the reflected rule states that if each premise $t_1,\ldots,t_{n-1}$ is provable in logic $\mathcal{L}$, then so is t_n. A key goal is that the reflected rule $\ulcorner R\urcorner$ must be automatically derivable from the definition of $\mathcal{L}$. For clarity, when reasoning about a single logic we will normally omit the subscript $\Box_{\mathcal{L}}$ and just write $\Box$.

The choice of meta-logic is somewhat arbitrary. For our purposes, we have chosen to use computational type theory (CTT), which is a variant of Martin-Löf intuitionistic type theory as implemented in the MetaPRL logical framework [16]. In other words, our meta-logic $\mathbb{M}$is CTT and our framework logic $\mathbb{F}$ is the one provided by MetaPRL. Note that in CTT, the reflected rules $\ulcorner R\urcorner$ are sometimes required to include additional well-formedness constraints on the typing of the meta-variables.

Returning to our example, the quoted form of the substitution lemma (2.I) is as follows, where we write $s\ \ulcorner\in\urcorner\ t$ for $\ulcorner member\urcorner\{s;t\}$.

$$\begin{array}{l} Z\vdash\Box(X;x\colon z_3;Y\ \ulcorner\vdash\urcorner\ z_1[x]\ \ulcorner\in\urcorner\ z_2)\longrightarrow\\ Z\vdash\Box(X;Y\ \ulcorner\vdash\urcorner\ z_0\ \ulcorner\in\urcorner\ z_3)\longrightarrow\\ Z\vdash\Box(X;Y\ \ulcorner\vdash\urcorner\ z_1[z_0]\ \ulcorner\in\urcorner\ z_2)\end{array}\qquad(3.\mathrm{I})$$

The operators have been quoted (in this case $\ulcorner\vdash\urcorner$ and $\ulcorner\in\urcorner$), and the theorem is now a judgment about provability stated in the meta-logic as $Z\vdash\Box\cdots$. Only

[2] Further discussion on quotations of names and concrete terms can be found in [21]. The encoding we use is an essential foundation for this work, however the specific encoding details have little effect on the presentation here.

the operator names have been changed, otherwise the structure, including variables and binding, has not changed.

For an example with binding, consider the rule for universal-introduction, shown below with the translated version. In this case, the binder x is translated to a meta-binder with λ_b.

$$\left[\begin{array}{l} X; x\colon z_1 \vdash z_2[x] \longrightarrow \\ X \vdash \forall x\colon z_1. z_2[x] \end{array}\right] \quad = \quad \left(\begin{array}{l} Z \vdash \Box(X; x\colon z_1 \ulcorner\vdash\urcorner z_2[x]) \longrightarrow \\ Z \vdash \Box(X \ulcorner\vdash\urcorner \ulcorner\forall\urcorner\{z_1; \lambda_b x. z_2[x]\}) \end{array}\right)$$

3.1 Proof reflection and automation

One important consequence of structure-preservation is that *proofs* can be reflected as well. Consider a proof in the original logic $\mathcal{L}$ of some theorem $t_1 \longrightarrow \cdots \longrightarrow t_n$. In a foundational prover, the proof is expressed as a tree of inferences that can be linearized to a finite sequence of rule applications $R_1, R_2, \ldots, R_n$.

Since the structure of each inference is preserved, there is a corresponding proof in the reflected logic $\ulcorner\mathcal{L}\urcorner$ of the reflected theorem $(Z \vdash \Box\ulcorner t_1\urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box\ulcorner t_n\urcorner)$. In fact, the proof is a one-to-one map of the original theorem, using reflected justifications in place of the original. That is, the reflected proof is $\ulcorner R_1\urcorner, \ulcorner R_2\urcorner, \ldots, \ulcorner R_n\urcorner$.

While this might seem quite straightforward, the important property here is that the prover internals do not need to be reflected. It is not necessary to formalize the inference mechanics of the theorem prover, because the original mechanism works without change in the reflected theory.

Proof automation is similar. Again, in a foundational prover,[3] each run of a heuristic or decision procedure is justified by a sequence of inferences $R_1, R_2, \ldots$. The existing automation may be used for reasoning in the reflected logic, *provided* that rule selection for reflected proofs uses the reflected rules rather than the original ones.

3.2 Syntax and reasoning

Reflected rules have an important property—the quoted terms are syntactical expressions, and they can be manipulated. There are constructors and destructors for quoted terms, and more importantly there is an inductively-defined type that contains all quoted terms. The specific details of the encoding have been published previously [21]. For our current purposes it simply matters that there *is* a type, so that meta-properties can be expressed.

For example, one may wish to prove a formal cut-elimination property. Using the type `Context` for sequent contexts, and the type `BTerm` for quoted terms, a cut-elimination theorem can be written as the following predicate.

$$\forall X\colon \texttt{Context}. \forall a, b\colon \texttt{BTerm}. \Box(X \ulcorner\vdash\urcorner a) \Rightarrow \Box(X, a \ulcorner\vdash\urcorner b) \Rightarrow \Box(X \ulcorner\vdash\urcorner b)$$

In addition, we have yet to define the provability predicate $\Box\, t$, where it will again be necessary to give a type to the quoted term t. Provability is the topic of the next section.

[3] It isn't clear to us whether a similar mechanism might work for non-foundational provers (those with "trusted" decision procedures).

4 Defining provability

So far, we have postponed the treatment of the provability predicate $\Box_{\mathcal{L}}\, t$, which specifies that the quoted formula t is provable in logic $\mathcal{L}$. To define provability properly, we take the following steps.

- First, for each rule $R \in \mathcal{L}$, we define a proof *checking* predicate that specifies whether a proof step is a valid application of rule R.
- Next, we define the (legal) *derivations* to be the proof trees where each proof step in the tree is validated by some rule $R \in \mathcal{L}$.
- A formula t is *provable* in logic $\mathcal{L}$ if, and only if, there is a derivation with root t.

The usual properties hold: proof checking is decidable, provability is not decidable in general.

4.1 Proof checking

A logic $\mathcal{L}$ is an ordered list of inference rules $R_1, \ldots, R_n$. A proof is a tree of inferences, and it is legal only if each proof step corresponds to an inference using some rule R_i. A *proof step* is a node in the proof tree that corresponds to a concrete inference $t_1 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t_n$. We call the terms $t_1, \ldots, t_{n-1}$ the *premises*, and the term t_n the *goal*.

In general, a rule R defines a *schema*, where each second-order meta-variable stands for a term, and each context meta-variable stands for a context. A concrete proof step is a valid inference of a rule R *iff* for each second-order meta-variable in R there is an actual term, and for each context-meta variable in R there is an actual context, such that the concrete inference is an instance of the rule.

Let us state this more formally. The *arity* of a meta-variable is the number of arguments, so a variable $z[t_1; \cdots; t_n]$ has arity n. Let $\texttt{BTerm}\{i\}$ be the type of quoted terms of arity i, corresponding to the space of substitution functions $\texttt{BTerm}^i \to \texttt{BTerm}$. Similarly, let $\texttt{Context}\{i\}$ be the type of contexts of arity i (the contexts correspond to lists of quoted terms).

Consider a rule R with free context variables $\{X_1^{i_1}, \ldots, X_m^{i_m}\}$ and free second-order variables $\{z_1^{j_1}, \ldots, z_n^{j_n}\}$, where the superscripts i_k and j_k indicate the arities of the variables.[4] Then a concrete inference r is a valid instance of rule R *iff* the following holds.

$$\begin{array}{l}\exists X_1^{i_1} : \texttt{Context}\{i_1\}, \ldots, X_m^{i_m} : \texttt{Context}\{i_m\}. \\ \exists z_1^{j_1} : \texttt{BTerm}\{j_1\}, \ldots, z_n^{j_n} : \texttt{BTerm}\{j_n\}. r = R \in \texttt{ProofStep}\end{array} \tag{4.I}$$

That is, the concrete inference r is equal to an instance of rule R. The type `ProofStep` is the type of proof steps $\texttt{BTerm list} \times \texttt{BTerm}$ containing the pairs (*premises*, *goal*).

For the purposes of proof checking, the existential witnesses are assembled into a proof witness term, and passed as explicit arguments to the checker. A proof witness is defined to be an element of the `Witness` type, which in turn is defined as

[4] In a setting where context variables are treated as binders, the variable arities are expressions that depend on the lengths $|X_k|$.

`Context list` $\times$ `BTerm list`. Returning to the example of the substitution lemma (3.I), the corresponding proof checker is defined as follows, where r is the concrete proof step to be checked.

$$\begin{array}{l}\texttt{checks}(\texttt{subst\_lemma}, r, \langle [X; Y], [z_1; z_2; z_3; z_0]\rangle) \equiv \\ r = \left(\begin{array}{c} [(X; x\colon z_3; Y \ulcorner\vdash\urcorner z_1[x] \ulcorner\in\urcorner z_2); (X; Y \ulcorner\vdash\urcorner z_0 \ulcorner\in\urcorner z_3)], \\ (X; Y \ulcorner\vdash\urcorner z_1[z_0] \ulcorner\in\urcorner z_2) \end{array} \right) \in \texttt{ProofStep} \end{array} \tag{4.II}$$

In general, the "rule checker" predicate $\texttt{checks}\{R; r; w\}$ takes three arguments, where R is a rule, $r \in \texttt{ProofStep}$ is a concrete inference, and $w \in \texttt{Witness}$ is the witness for the rule instantiation. Given a logic $\mathcal{L}$ with rules $R_1, \ldots, R_n$, a proof step is valid *iff* it is an instance of one of the rules in the logic.

$$\texttt{checks}\{r; w\} \equiv \exists R \in \{R_1, \ldots, R_n\}.\texttt{checks}\{R; r; w\}$$

Since proof step equality is decidable, and each logic has a finite number of rules, the $\texttt{checks}\{r; w\}$ predicate is decidable as well.

4.2 Derivations

Now that we have defined proof step checking, the next part is to define the valid derivations, or proof trees. The type D of all derivations is defined inductively in the usual way.

$$\begin{array}{rcl} D_0 & \equiv & \texttt{void} \\ D_{i+1} & \equiv & \Sigma \mathit{premises}\colon D_i\ \texttt{list}.\Sigma \mathit{goal\_term}\colon \texttt{BTerm}\{0\}.\Sigma w\colon \texttt{Witness}. \\ & & \quad \texttt{checks}\{(\texttt{goal}\{\mathit{premises}\}, \mathit{goal\_term}); w\} \\ D & \equiv & \bigcup_{i \in \mathbb{N}} D_i \end{array} \tag{4.III}$$

In this definition, the term $\texttt{goal}\{[d_1; \cdots; d_n]\}$ is the list of goal terms for derivations $d_1, \ldots, d_n$.

This definition also allows us to prove an induction principle, which will form the basis for proof induction.

$$\begin{array}{l} \forall P.(\forall \mathit{premises}\colon D\ \texttt{list}.\forall g\colon \texttt{BTerm}\{0\}.\forall w\colon \texttt{Witness}. \\ \quad \texttt{checks}\{(\texttt{goal}\{\mathit{premises}\}, g); w\} \\ \quad\quad \Rightarrow (\forall p \in \mathit{premises}.P[p]) \Rightarrow P[(\mathit{premises}, g, w)]) \\ \quad \Rightarrow (\forall d\colon D.P[d]) \end{array}$$

At this point, the definition of the provability predicate $\Box\, t$ is straightforward. A quoted term t is provable *iff* there is a derivation where t is the goal term.

$$\Box\, t \equiv \exists d\colon D.(\texttt{goal}\{d\} = t \in \texttt{BTerm}\{0\})$$

5 Sequent context induction

At this point, we now have a representation function where rules are reflected into statements of provability, and in addition we have a proof-checking predicate for establishing proof correctness. The next step is to prove that the reflected rules

are valid using the definition of provability. For example, consider the substitution lemma example. From the proof-checking predicate (4.II), we must prove the reflected rule (3.I).

However, there is a substantial gap between the two forms. We have glossed over the fact that the proof-checking predicates are defined using standard existential quantifiers (4.I and 4.III). For a quantifier of the form $\exists X : \texttt{Context}\{i\}. \cdots$ the variable X is a *first-order* variable in the meta-logic $\mathbb{M}_{\mathrm{CTT}}$. In contrast, the reflected rules preserve meta-variables, and are expressed using context and second-order meta-variables (variables of the framework logic $\mathbb{F}_{\mathsf{MetaPRL}}$).

Second-order variables can be modeled with functions on `BTerm`, so the object quantifiers are expressive enough to represent second-order quantification. The question remains, how does one derive a formula involving context variables from a similar formula that does not? In general, sequent context variables are bindings, sequent contexts are not terms, and they cannot be modeled directly in the object logic.

Since the framework meta-logic we are using (the $\mathbb{F}_{\mathsf{MetaPRL}}$ meta-logic) does not include context quantifiers, one option is to add them and use them in the proof-checking predicate. However, this is undesirable in part because the framework's meta-logic would become extremely expressive and powerful, but also because the extension is perilous and difficult to get right.

Instead, we extend the framework's meta-logic with a weak theory of sequent context induction that we call *teleportation*. The central logical property is that contexts are finite and inductively defined. Note that this represents a strengthening of the meta-logic by effectively including Peano arithmetic.

5.1 Teleportation

The concept behind teleportation is deceptively simple. Since contexts are inductively defined, contexts can be "migrated," one hypothesis at a time, from one point in a rule to another. Scoping must be preserved, including *context variable* scoping, but beyond that the migration locations are unconstrained.

To formalize this more precisely, we introduce the notion of teleportation contexts, written $R[\![\Gamma]\!]$, which represents a term or a rule with exactly one occurrence of the context Γ. We will use the symbol ϵ to denote the empty context. These definitions are for presentation purposes; they are not part of the meta-logic. Teleportation is specified using a pair of nested teleportation contexts, which we will write as $F[\![\cdot; G[\![\cdot]\!]]\!]$. Here $F[\![\Gamma; G[\![\Delta]\!]]\!]$ must be a rule that has exactly one occurrence of each of the Γ, Δ and G; in addition G must be in scope of Γ.

The simplest teleportation rule hoists the context from G to F.

$$\frac{\begin{array}{ll}\text{(base)} & \forall X. \quad F[\![\epsilon; G[\![X]\!]]\!] \\ \text{(step)} & \forall X, Y, z. \; F[\![X; G[\![x : z; Y[x]]\!]]\!] \longrightarrow F[\![X; x : z; G[\![Y[x]]\!]]\!]\end{array}}{\forall X. \quad F[\![X; G[\![\epsilon]\!]]\!]}$$

For clarity, we have written explicit universal quantifiers for the meta-variables to emphasize that meta-variables are quantified for each clause/rule. Again, these do not exist explicitly in the meta-logic, and we will omit them in the remaining

rules. As usual, it is assumed that the schema language of the teleportation contexts would alpha-rename the bound variables as needed to avoid capture.

For generality, it is frequently useful to transform the hypotheses during migration. In the following rule f is an arbitrary function.

$$\frac{\begin{array}{l}(\text{base})\ F[\![\epsilon; G[\![X]\!]]\!] \\ (\text{step})\ F[\![X; G[\![x\colon f(z); Y[x]]\!]]\!] \longrightarrow F[\![X; x\colon z; G[\![Y[x]]\!]]\!]\end{array}}{F[\![X; G[\![\epsilon]\!]]\!]}$$

There is a corresponding reverse-hoisting rule.

$$\frac{\begin{array}{l}(\text{base})\ F[\![X; G[\![\epsilon]\!]]\!] \\ (\text{step})\ F[\![X; x\colon f(z); G[\![Y[x]]\!]]\!] \longrightarrow F[\![X; G[\![x\colon z; Y[x]]\!]]\!]\end{array}}{F[\![\epsilon; G[\![X]\!]]\!]}$$

We add the teleportation rules as new primitive rules in our framework logic $\mathbb{F}_{\text{MetaPRL}}$. The conservativity theorem for sequent schema [19], which states that the language of framework meta-variables is a conservative extension of the meta-theory, can be extended to include teleportation rules. The central observation here is that for any particular finite concrete context Γ, any proof using the teleportation rules can be transformed into a proof without teleportation by posing a finite sequence of lemmas, one for each of the intermediate steps.

5.2 A simple example

For a fairly natural example, consider the problem of context exchange. That is, we are given an exchange rule for hypotheses, and we wish to derive a rule for exchanging contexts.

$$\frac{X; y\colon z_2; x\colon z_1; Y[x; y] \vdash z_3[x; y]}{X; x\colon z_1; y\colon z_2; Y[x; y] \vdash z_3[x; y]} \Longrightarrow \frac{X; Z_2; Z_1; Y \vdash z}{X; Z_1; Z_2; Y \vdash z}$$

The proof in this case can be posed as a nested induction. To begin, we propose to migrate Z_2 left, where the $\bullet$ denotes the target: $X; \bullet; Z_1; Z_2; Y \vdash z$. The base case follows by assumption, and the step case presents us with the following subproblem.

$$(X; Z_3; x\colon z'; Z_1; Z_2; Y \vdash z) \longrightarrow (X; Z_3; Z_1; x\colon z'; Z_2; Y \vdash z).$$

The proof is concluded by migrating Z_1 past the hypothesis $x\colon z'$.

5.3 Computation on sequent terms

The sequent induction scheme also introduces a sequent induction combinator for computation over a sequent context. We introduce two new terms to the meta-logic. The $\texttt{sequent\_ind}\{x, y.step[x; y]; s\}$ performs computation over a sequent term s. The reduction rules for sequent computation are as follows.

$$\texttt{sequent\_ind}\{x, y.step[x; y]; (\vdash t)\} \rightarrow t$$

$$\texttt{sequent\_ind}\{x, y.step[x; y]; (z\colon t_1; X[z] \vdash t_2[z])\} \rightarrow$$

$$step[t_1; \lambda z.\texttt{sequent\_ind}\{x, y.step[x; y]; (X[z] \vdash t_2[z])\}]$$

To illustrate, suppose we wish to develop a "vector" universal quantifier. That is, a sequent with the following definition, given that the logic has a "scalar" quantifier $\forall x\colon t_1.t_2[x]$.

$$x_1\colon t_1;\cdots;x_n\colon t_n \vdash_\forall t_{n+1} \quad \equiv \quad \forall x_1\colon t_1,\ldots,x_n\colon t_n.t_{n+1}$$

The definition is implemented in terms of sequent induction.

$$\Gamma \vdash_\forall t \quad \equiv \quad \texttt{sequent\_ind}\{x,y.\forall z\colon x.(yz);(\Gamma \vdash t)\}$$

We get the following reductions.

$$\begin{array}{rcl} \vdash_\forall z & \rightarrow & z \\ x\colon z_1; X[x] \vdash_\forall z_2[x] & \rightarrow & \forall x\colon z_1.(X[x] \vdash_\forall z_2[x]) \end{array}$$

The simple introduction rule can be derived directly.

$$\frac{Z; x\colon z_1 \vdash (X[x] \vdash_\forall z_2[x])}{Z \vdash (x\colon z_1; X[x] \vdash_\forall z_2[x])}\ \text{vall-intro-single}$$

A general introduction rule is also derivable using the teleportation rules.

$$\frac{Z; X \vdash z}{Z \vdash (X \vdash_\forall z)}\ \text{vall-intro}$$

Using similar methods, it is possible to define a logic of vector operators, quantifiers, and a vector lambda calculus.

Note that in these rules, the variable X is a context variable, and the rules are valid for any instance of X.

5.4 Sequent induction and reflection

With this new tool in hand, let us return to the topic of reflection, where the issue was that we need to derive proofs of the reflected rules (with context variables) from the proof-checking predicates (no context variables).

At this point, the plan is conceptually easy. There are two parts. First, we develop a canonical representation of concrete sequents *without context variables*. For the second part, we define a (formal) function that computes the canonical representation from the non-canonical form that *includes context variables*.

The first part is an issue of coding, where the goal is to define a representation that preserves the structure of concrete sequents. We choose the following representation, where $\ulcorner\lambda_H\urcorner x\colon t_1.t_2$ is a quoted term that represents a hypothesis, its binding, and the rest of the sequent; and $\ulcorner\texttt{concl}\urcorner\{t\}$ represents the conclusion of the sequent. The proof-checking predicates operate directly on quoted terms with this representation.

$$x_1\colon t_1;\cdots;x_n\colon t_n\ulcorner\vdash\urcorner t_{n+1} \quad \equiv \quad \ulcorner\lambda_H\urcorner x_1\colon t_1.\ldots.\ulcorner\lambda_H\urcorner x_n\colon t_n.\ulcorner\texttt{concl}\urcorner\{t_{n+1}\}$$

For the second part, we define a function using `sequent_ind` that computes the canonical representation from its non-canonical form. This function, written $\vdash_B$, is defined as follows.

$$X \vdash_B t \quad \equiv \quad \texttt{sequent\_ind}\{x,y.\ulcorner\lambda_H\urcorner z\colon x.(y\ z);(X \vdash \ulcorner\texttt{concl}\urcorner\{t\})\}$$

The original reflected form of a rule $R = (\Gamma_1 \vdash t_1) \longrightarrow \cdots \longrightarrow (\Gamma_n \vdash t_n)$ is $\ulcorner R \urcorner = Z \vdash \Box(\ulcorner \Gamma_1 \urcorner \ulcorner \vdash \urcorner \ulcorner t_1 \urcorner) \longrightarrow \cdots \longrightarrow Z \vdash \Box(\ulcorner \Gamma_n \urcorner \ulcorner \vdash \urcorner \ulcorner t_n \urcorner)$. Using the non-canonical forms, the new representation is as follows.

$$\ulcorner R \urcorner \quad = \quad (Z \vdash \Box(\ulcorner \Gamma_1 \urcorner \vdash_B \ulcorner t_1 \urcorner)) \longrightarrow \cdots \longrightarrow (Z \vdash \Box(\ulcorner \Gamma_n \urcorner \vdash_B \ulcorner t_n \urcorner))$$

The right-hand-side is now proved by reducing the $\vdash_B$ sequents to canonical form, then proving that the reduced form passes the proof–checking predicate for all instances of the meta-variables. Note that contexts and context variables are not terms, and so it remains impossible to quantify over them directly. However, the reduced form of a non-canonical $\vdash_B$ sequent with context variables does contain sequent subterms with context variables. With teleportation it is possible to show that these embedded terms are well–defined.

These correspondence between a reflected rule and its proof-checking predicate is very close. In our implementation, the reflected rule and the proof checking definitions are created mechanically, and the proof is completely automated.

6 Reflection and induction

So far, we have presented a structure-preserving representation function, a mechanism for formalizing reflected logics, and a procedure for deriving reflected provability rules. This system is already powerful enough to express and prove meta-properties over reflected systems. However, it remains impractical. There is a crucial piece missing—induction on the provability predicate.

What exactly is the induction principle for provability? Suppose we wish to prove a theorem of the form $\Box\, x \Rightarrow P[x]$, where x is a variable, and P is a predicate on quoted terms. Since x is provable, that means there is a derivation with root x, and we can apply induction on the length of the derivation.

Now, for illustration, assume the logic $\mathcal{L}$ contains three rules, $\mathcal{L} = t_{11}$, $t_{21} \longrightarrow t_{22}$, $t_{31} \longrightarrow t_{32} \longrightarrow t_{33}$. Then the induction form has the following shape.

$$\frac{\begin{array}{l}\text{(rule sketch)}\\ \Gamma; \Box\, t_{11} \vdash P[t_{11}]\\ \Gamma; \Box\, t_{21}; \Box\, t_{22}; P[t_{21}] \vdash P[t_{22}]\\ \Gamma; \Box\, t_{31}; \Box\, t_{32}; \Box\, t_{33}; P[t_{31}]; P[t_{32}] \vdash P[t_{33}]\end{array}}{\Gamma; \Box\, x \vdash P[x]}$$

However, this rule is not quite right. The issue is that the terms t_{ij} will in general contain meta-variables, and the meta-variables must be separately universally quantified for each induction case. As we explained in Section 5, explicit quantification of meta-variables is not expressible in our meta-logic.

However, here it is acceptable to use object-quantifiers. There is no appreciable effect on proof automation as long as the first-order form is compatible with the automatically–generated reflected rules. The correct form of the rule explicitly quantifies over the meta-variables, re-using the mechanism for generating the proof-checking rules. For the current example, we introduce explicit quantifiers. In this case we write $t_{ij}[\boldsymbol{X}]$ to represent a term that may contain any of the variable $\boldsymbol{X}$

but is otherwise free of context variables.

$$\frac{\Gamma; \boldsymbol{X} : \texttt{Context}; \Box\, t_{11}[\boldsymbol{X}] \vdash P[t_{11}[\boldsymbol{X}]] \quad \Gamma; \boldsymbol{X} : \texttt{Context}; \Box\, t_{21}[\boldsymbol{X}]; \Box\, t_{22}[\boldsymbol{X}]; P[t_{21}[\boldsymbol{X}]] \vdash P[t_{22}[\boldsymbol{X}]] \quad \Gamma; \boldsymbol{X} : \texttt{Context}; \Box\, t_{31}[\boldsymbol{X}]; \Box\, t_{32}[\boldsymbol{X}]; \Box\, t_{33}[\boldsymbol{X}]; P[t_{31}[\boldsymbol{X}]]; P[t_{32}[\boldsymbol{X}]] \vdash P[t_{33}[\boldsymbol{X}]]}{\Gamma; \Box\, x \vdash P[x]}$$

In our implementation, we generate a variant of this rule that allows for induction over terms, not just variables. This is done by introducing a "shared" term u that establishes a connection provable term t and the predicate P. The actual theorem has the form $\Gamma; u\colon t_1; \Box\, t_2[u] \vdash P[t_3[u]]$, where u is the shared part. The new form is derivable from the previous case for provability on variables, and we omit it here. In fact, the size of the rule is one of the main drawbacks. In practice, even for fairly small logics $\mathcal{L}$, the statement of the elimination rule is already several pages long, and it is difficult to use the rule interactively. We are expecting to address this in future work.

This mechanism establishes the principle of proof induction. The principle of structural induction is reducible to proof induction by specifying the syntax of a language as a logic of type-checking.

For every object logic, the corresponding induction principle is not only automatically formulated by our system, but is also automatically derived. Since the proof induction principle implies soundness, this means that while we do not prove the soundness of our formalization *in general*, for each particular object logic, it will be established automatically.

7 Related work

This work build upon a very large number of related efforts. In fact, the number of such efforts is so big that we are unable to give an adequate overview in this limited space. Harrison [12] has written an excellent survey and critique of a broad range of approaches to reflection. We give another broad survey in a previous paper [21].

Our approach to representing the syntax with bindings has some similarities to the HOAS implemented in Coq by Despeyroux and Hirschowitz [6] and to the modal λ-calculus [9, 7, 8].

In 1931 Gödel used reflection to prove his famous incompleteness theorem [10]. A modern version of the Gödel's approach was used by Aitken *et.al.* [3, 1, 2, 5] to implement reflection in the NuPRL theorem prover. A large part of this effort was essentially a reimplementation of the core of the NuPRL prover inside NuPRL's logical theory.

A number of approaches to logical reflection were explored in the Coq proof assistant. Rueß [23] has implemented a computation reflection mechanism. Hendriks [13] formalized natural deduction for first-order logic in the proof assistant Coq, using de Bruijn indices for variable binding. O'Connor [22] constructively proved the Gödel–Rosser incompleteness theorem using the natural numbers to encode formulas and proofs.

8 Conclusion

The goal of this work is to develop a *practical* theory of logical reflection. We claim that doing so requires preserving the structure of a theory when it is reflected, including variables, meta–variables, and bindings. We presented a structure-preserving representation, building on previous work with the representation of logical terms [21]. Besides, we developed a new account of sequent context induction, called *teleportation*, to allow reasoning and computation over terms that include sequent context variables. This led to a formalization of proofs, proof–checkers, and derivations, together with automated generation of reflected rules and induction forms in the reflected theory.

In some ways, the result seems startlingly simple. When a logic is reflected, its presentation changes only slightly, and the existing reasoning methods and proof procedures continue to work. The difference is, of course, that reasoning about meta-properties of the logic becomes possible.

It was important to us that the development of the theory of reflection be accompanied by its implementation. This makes it more useful of course, but an additional reason is that the theory of reflection is rife with paradoxes, and it is easy to fall into false thinking. While we have tried to simplify the account in this paper, the actual formalization was demanding. In particular, the formalization of context induction required several man-months of effort, mainly due to the need to develop a logical infrastructure for reasoning about terms containing context variables.

We are currently using reflection to develop an account of $F_{<:}$ type theory, which has acted both as a challenge and a guide [15]. For work in the near future, we are considering alternate ways to pose the proof induction principle. Induction is, by nature, not modular. However, we believe that significant practical advances can be made through improved automation and hierarchical decomposition.

We believe that our results may be generalized to other provers and frameworks. The non-standard properties of the logical framework that we rely upon are the following. 1) Programs may be expressed without first giving them a type; in addition, programs may have more than one type. 2) Computation defines a congruence; any two programs that are computationally (beta) equivalent can be interchanged in any formal context. 3) For reasoning about sequents, the teleportation principle is needed. 4) A *function image* type [20].

References

[1] Aitken, W. and R. L. Constable, *Reflecting on NuPRL : Lessons 1–4*, Technical report, Cornell University, Computer Science Department, Ithaca, NY (1992).

[2] Aitken, W., R. L. Constable and J. Underwood, *Metalogical Frameworks II: Using reflected decision procedures*, Journal of Automated Reasoning **22** (1993), pp. 171–221.

[3] Allen, S. F., R. L. Constable, D. J. Howe and W. Aitken, *The semantics of reflected proof*, in: *Proceedings of the 5^{th} Symposium on Logic in Computer Science* (1990), pp. 95–197.

[4] Barzilay, E., "Implementing Reflection in NuPRL," Ph.D. thesis, Cornell University (2006).

[5] Constable, R. L., *Using reflection to explain and enhance type theory*, in: H. Schwichtenberg, editor, *Proof and Computation*, NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, NATO Series F **139**, Springer, Berlin, 1994 pp. 65–100.

[6] Despeyroux, J. and A. Hirschowitz, *Higher-order abstract syntax with induction in Coq*, in: *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, Lecture Notes in Computer Science **822** (1994), pp. 159–173, also appears as INRIA research report RR-2292.

[7] Despeyroux, J. and P. Leleu, *A modal lambda calculus with iteration and case constructs*, in: T. Altenkirch, W. Naraschewski and B. Reus, editors, *Types for Proofs and Programs: International Workshop, TYPES '98, Kloster Irsee, Germany, March 1998*, Lecture Notes in Computer Science **1657**, 1999, pp. 47–61.
URL http://www.springerlink.com/link.asp?id=984f76cm6b6qv0a4

[8] Despeyroux, J. and P. Leleu, *Recursion over objects of functional type*, Mathematical Structures in Computer Science **11** (2001), pp. 555–572.
URL http://citeseer.ist.psu.edu/despeyroux00recursion.html

[9] Despeyroux, J., F. Pfenning and C. Schürmann, *Primitive recursion for higher–order abstract syntax*, in: Hindley [18], pp. 147–163, an extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[10] Gödel, K., *Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I*, Monatshefte für Mathematik und Physik **38** (1931), pp. 173–198, english version in [24].

[11] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery **40** (1993), pp. 143–184, a revised and expanded version of the 1987 paper.

[12] Harrison, J., *Metatheory and reflection in theorem proving: A survey and critique*, Technical Report CRC-53, SRI International, Cambridge Computer Science Research Centre, Millers Yard, Cambridge, UK (1995).
URL http://www.cl.cam.ac.uk/users/jrh/papers/reflect.html

[13] Hendriks, D., *Proof reflection in Coq*, Journal of Automated Reasoning **29** (2002), pp. 277–307.

[14] Hickey, J., A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty and X. Yu, *MetaPRL — A modular logical environment*, in: D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, Lecture Notes in Computer Science **2758** (2003), pp. 287–303.
URL http://nogin.org/papers/metaprl.html

[15] Hickey, J., A. Nogin, X. Yu and A. Kopylov, *Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection*, Accepted to the International Conference on Functional Programming (ICFP) (2006).

[16] Hickey, J. J., B. Aydemir, Y. Bryukhov, A. Kopylov, A. Nogin and X. Yu, *A listing of MetaPRL theories*.
URL http://metaprl.org/theories.pdf

[17] Hickey, J. J., A. Nogin, A. Kopylov et al., *MetaPRL home page*.
URL http://metaprl.org/

[18] Hindley, R., editor, "Proceedings of the International Conference on Typed Lambda Calculus and its Applications (TLCA'97)," Lecture Notes in Computer Science **1210**, Springer-Verlag, Nancy, France, 1997.

[19] Nogin, A. and J. Hickey, *Sequent schema for derived rules*, in: V. A. Carreño, C. A. Muñoz and S. Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, Lecture Notes in Computer Science **2410** (2002), pp. 281–297.
URL http://nogin.org/papers/derived_rules.html

[20] Nogin, A. and A. Kopylov, *Formalizing type operations using the "Image" type constructor*, Accepted to Workshop on Logic, Language, Information and Computation (WoLLIC) (2006).

[21] Nogin, A., A. Kopylov, X. Yu and J. Hickey, *A computational approach to reflective meta-reasoning about languages with bindings*, in: *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (2005), pp. 2–12, an extended version is available as California Institute of Technology technical report CaltechCSTR:2005.003.

[22] OConnor, R., *Essential incompleteness of arithmetic verified by Coq*, in: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, Lecture Notes in Computer Science **3603**, 2005, pp. 245–260.

[23] Rueß, H., *Computational reflection in the calculus of constructions and its application to theorem proving*, in: Hindley [18].

[24] van Heijenoort, J., editor, "From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931," Harvard University Press, Cambridge, MA, 1967.

An Algebraic Framework for Logics and Type Theories

(Invited Talk)

Gordon Plotkin[1]

LFCS, School of Informatics, University of Edinburgh, U.K.

Abstract

It seems a natural principle to seek logical frameworks that are as simple and as weak as possible while still enabling direct natural representations of a wide variety of formalisms. We present an algebraic framework, which can be considered an extension of multi-sorted equational logic with abstraction and dependent types, but with no ability to form compound types (= sorts) and thereby no λ-abstraction. Abstraction enables the representation of binding mechanisms; dependent types enables, for example, the representation of the proofs of a formula; and it seems that no other representational mechanisms are necessary. We present the system as a combination of two simpler subsystems: one with only parameterisation and one with only dependent sorts. The closest related previous work is the PAL$^+$ system of Z. Luo which also supplies mechanisms for parameterisation and dependency and avoids lambda abstraction; our system is, perhaps, somewhat simpler.

[1] Email: gdp@inf.ed.ac.uk.

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

A List-machine Benchmark for Mechanized Metatheory
(Extended Abstract)

Andrew W. Appel

Princeton University and INRIA Rocquencourt

Xavier Leroy

INRIA Rocquencourt

Abstract

We propose a benchmark to compare theorem-proving systems on their ability to express proofs of compiler correctness. In contrast to the first POPLmark, we emphasize the connection of proofs to compiler implementations, and we point out that much can be done without binders or alpha-conversion. We propose specific criteria for evaluating the utility of mechanized metatheory systems; we have constructed solutions in both Coq and Twelf metatheory, and we draw conclusions about those two systems in particular.

Keywords: Theorem proving, proof assistants, program proof, compiler verification, typed machine language, metatheory, Coq, Twelf.

1 How to evaluate mechanized metatheories

The POPLmark challenge [3] aims to compare the usability of several automated proof assistants for mechanizing the kind of programming-language proofs that might be done by the author of a POPL paper, with benchmark problems "chosen to exercise many aspects of programming languages that are known to be difficult to formalize." The first POPLmark examples are all in the theory of $F_{<:}$ and emphasize the theory of binders (e.g., alpha-conversion).

Practitioners of machine-checked proof about real compilers have interests that are similar but not identical. We want to formally relate machine-checked proofs to actual implementations, not particularly to LaTeX documents. Furthermore, perhaps it is the wrong approach to "exercise aspects ... that are known to be difficult to formalize." Binders and $\alpha\beta$-conversion are certainly useful, but they

[1] Email: appel@princeton.edu

[2] Email: Xavier.Leroy@inria.fr

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

are not essential for proving real things about real compilers, as demonstrated in several substantial compiler-verification projects [9,10,13,6,7,8]. If machine-checked proof is to be useful in providing guarantees about real systems, let us play to its strengths, not to its weaknesses.

Therefore we have designed a down-to-earth example of machine-checked metatheory, closer to the semantics of typed assembly languages. It is entirely first-order, without binders or the need for alpha conversion. We specify the Structured Operational Semantics (SOS) of a simple pointer machine (cons, car, cdr, branch-if-nil) and we present a simple type system with constructors for list-of-τ and nonempty-list-of-τ. The benchmark explicitly covers the relationship of proofs about a *type system* to proofs about an executable *type checker*.

The challenge is to represent the type system, prove soundness of the type system, represent the type-checking algorithm, and prove that the algorithm correctly implements the type system. We have implemented the benchmark both in Coq and in Twelf metatheory, and we draw conclusions about the usability of these two systems.

We lack space to present here, but discuss in the full paper[1],

- how the needs of implementors (of provably correct compilers and provably sound typecheckers) differ from the needs of POPL authors addressed by the first POPLmark;
- a specification of the entire problem down to details such as the recommended ASCII names for predicates and inference rules;
- details of our Coq and Twelf solutions;
- more details about which subtasks were easy or difficult in Coq and Twelf;
- how easy it is to learn Twelf and Coq given the available documentation.

As well as a benchmark, the list machine is a useful exercise for students learning Coq or Twelf; we present the outlines of our solutions (with proofs deleted) on the Web [2].

2 Specification of the problem

Machine syntax. Machine values A are cons cells and nil.

$$a : A ::= \text{nil} \mid \text{cons}(a_1, a_2)$$

The instructions of the machine are as follows:

$$\begin{array}{lll}
\iota : I ::= & & \\
& \textbf{jump } l & \text{(jump to label } l\text{)} \\
\mid & \textbf{branch-if-nil } v\ l & \text{(if } v = nil \text{ go to } l\text{)} \\
\mid & \textbf{fetch-field } v\ 0\ v' & \text{(fetch the head of } v \text{ into } v'\text{)} \\
\mid & \textbf{fetch-field } v\ 1\ v' & \text{(fetch the tail of } v \text{ into } v'\text{)} \\
\mid & \textbf{cons } v_0\ v_1\ v' & \text{(make a cons cell in } v'\text{)} \\
\mid & \textbf{halt} & \text{(stop executing)} \\
\mid & \iota_0\ ;\ \iota_1 & \text{(sequential composition)}
\end{array}$$

In the syntax above, the metavariables v_i range over variables; the variables themselves $\mathbf{v}_i$ are enumerated by the natural numbers. Similarly, metavariables l_i range

over program labels $\mathbf{L}_i$.

A program is a sequence of instruction blocks, each preceded by a label.

$$p : P ::= \quad \mathbf{L}_n : \iota;\ p \quad | \quad \mathbf{end}$$

Operational semantics. Machine states are pairs (r, ι) of the current instruction ι and a store r associating values to variables. We write $r(v) = a$ to mean that a is the value of variable v in r, and $r[v := a] = r'$ to mean that updating r with the binding $[v := a]$ yields a unique store r'. The semantics of the machine is defined by the small-step relation $(r, \iota) \overset{p}{\mapsto} (r', \iota')$ defined by the rules below, and the Kleene closure of this relation, $(r, \iota) \overset{p}{\mapsto}{}^* (r', \iota')$.

$$\frac{}{(r,\ (\iota_1; \iota_2); \iota_3) \overset{p}{\mapsto} (r,\ \iota_1; (\iota_2; \iota_3))}$$

$$\frac{r(v) = \mathrm{cons}(a_0, a_1) \quad r[v' := a_0] = r'}{(r, (\mathbf{fetch\text{-}field}\ v\ 0\ v'; \iota)) \overset{p}{\mapsto} (r', \iota)} \qquad \frac{r(v) = \mathrm{cons}(a_0, a_1) \quad r[v' := a_1] = r'}{(r, (\mathbf{fetch\text{-}field}\ v\ 1\ v'; \iota)) \overset{p}{\mapsto} (r', \iota)}$$

$$\frac{r(v_0) = a_0 \quad r(v_1) = a_1 \quad r[v' := \mathrm{cons}(a_0, a_1)] = r'}{(r, (\mathbf{cons}\ v_0\ v_1\ v'; \iota)) \overset{p}{\mapsto} (r', \iota)}$$

$$\frac{r(v) = \mathrm{cons}(a_0, a_1)}{(r, (\mathbf{branch\text{-}if\text{-}nil}\ v\ l; \iota)) \overset{p}{\mapsto} (r, \iota)} \qquad \frac{r(v) = \mathrm{nil} \quad p(l) = \iota'}{(r, (\mathbf{branch\text{-}if\text{-}nil}\ v\ l; \iota)) \overset{p}{\mapsto} (r, \iota')}$$

$$\frac{p(l) = \iota'}{(r, \mathbf{jump}\ l) \overset{p}{\mapsto} (r, \iota')}$$

A program p runs, that is, $p \Downarrow$, if it executes from an initial state to a final state. A state is an initial state if variable $\mathbf{v}_0 = \mathrm{nil}$ and the current instruction is the one at $\mathbf{L}_0$. A state is a final state if the current instruction is **halt**.

$$\frac{\{\}[\mathbf{v}_0 := \mathrm{nil}] = r \quad p(\mathbf{L}_0) = \iota \quad (r, \iota) \overset{p}{\mapsto}{}^* (r', \mathbf{halt})}{p \Downarrow}$$

It is useful for a benchmark for machine-verified proof to include explicit ASCII names for each constructor and rule. Our full specification [1] does that.

A type system. We will assign to each live variable at each program point a list type. To guarantee safety of certain operations, we provide refinements of the list type for nonempty lists and for empty lists. In particular, the **fetch-field** operations demand that their list argument has nonempty list type, and the **branch-if-nil** operation refines the type of its argument to empty or nonempty list, depending on whether the branch is taken.

$$\begin{array}{lll} \tau : T ::= & & \\ \quad \mathrm{nil} & & \text{(singleton type containing nil)} \\ \quad | \ \mathrm{list}\,\tau & & \text{(list whose elements have type } \tau\text{)} \\ \quad | \ \mathrm{listcons}\,\tau & & \text{(non-nil list of } \tau\text{)} \end{array}$$

An environment Γ is an type assignment of types to a set of variables. We define the obvious subtyping $\tau \subset \tau'$ among the various refinements of the list type, using a common set of first-order syntactic rules, easily expressible in most mechanized metatheories. We extend subtyping widthwise and depthwise to environments.

We define the least common supertype $\tau_1 \sqcap \tau_2 = \tau_3$ of two types τ_1 and τ_2 as the smallest τ_3 such that $\tau_1 \subset \tau_3$ and $\tau_1 \subset \tau_2$.

In the operational semantics, a program is a sequence of labeled basic blocks. In our type system, a *program-typing*, ranged over by Π, associates to each program label a variable-typing environment. We write $\Pi(l) = \Gamma$ to indicate that Γ represents the types of the variables on entry to the block labeled l.

Instruction typing. Individual instructions are typed by a judgment $\Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma'$. The intuition is that, under program-typing Π, the Hoare triple $\Gamma\{\iota\}\Gamma'$ relates precondition Γ to postcondition Γ'.

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi \vdash_{\text{instr}} \Gamma'\{\iota_2\}\Gamma''}{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1; \iota_2\}\Gamma''}$$

$$\frac{\Gamma(v) = \text{list}\,\tau \quad \Pi(l) = \Gamma_1 \quad \Gamma[v := \text{nil}] = \Gamma' \quad \Gamma' \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}(v : \text{listcons}\,\tau,\ \Gamma')}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau \quad \Pi(l) = \Gamma_1 \quad \Gamma[v := \text{nil}] = \Gamma' \quad \Gamma' \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma}$$

$$\frac{\Gamma(v) = \text{nil} \quad \Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau \quad \Gamma[v' := \tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 0\ v'\}\Gamma'} \qquad \frac{\Gamma(v) = \text{listcons}\,\tau \quad \Gamma[v' := \text{list}\,\tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 1\ v'\}\Gamma'}$$

$$\frac{\Gamma(v_0) = \tau_0 \quad \Gamma(v_1) = \tau_1 \quad (\text{list}\,\tau_0) \sqcap \tau_1 = \text{list}\,\tau \quad \Gamma[v := \text{listcons}\,\tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons}\ v_0\ v_1\ v\}\Gamma'}$$

Block typing. A *block* is an instruction that does not (statically) continue with another instruction, because it ends with a jump.

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi; \Gamma' \vdash_{\text{block}} \iota_2}{\Pi; \Gamma \vdash_{\text{block}} \iota_1; \iota_2} \qquad \frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi; \Gamma \vdash_{\text{block}} \textbf{jump}\ l}$$

Program typing. We write $\models_{\text{prog}} p : \Pi$ and say that a program p has program-typing Π if for each labeled block $l : \iota$ in p, the block ι has the precondition $\Pi(l) = \Gamma$ given in Π, that is, $\Pi; \Gamma \vdash_{\text{block}} \iota$. Moreover, we demand that $\Pi(\mathbf{L}_0) = \mathbf{v}_0 : \text{nil},\ \{\,\}$ and that every label l declared in Π is defined in p.

Type system vs. type checker. We have presented some relations defined by derivation rules and some defined informally. This is a bit sloppy, especially where a derivation rule refers to an informally defined relation; any solution to the benchmark must formalize this. We will use the notation $\models_{\text{prog}} p : \Pi$ to mean that program p has type Π in the (not necessarily algorithmic) type system, and the notation $\vdash_{\text{prog}} p : \Pi$ to mean that $p : \Pi$ is derived in some algorithmic type-checker. The full paper [1] outlines two such algorithmic type-checkers. One is written in pseudo-code and corresponds to a type-checker implemented in imperative or functional style. The other refines the derivation rules given above to make them fully syntax-directed and therefore amenable to an implementation as a logic program.

Sample program. The following list-machine program has three basic blocks. Variable $\mathbf{v}_0$ is initialized to nil as prescribed by the operational semantics. Block 0 initializes $\mathbf{v}_1$ to the list $\text{cons}(\text{nil}, \text{cons}(\text{nil}, \text{nil}))$ and jumps to block 1. Block 1 is a loop that, while $\mathbf{v}_1$ is not nil, fetches the tail of $\mathbf{v}_1$ and continues. The last instruction of block 1 is actually dead code (never reached). Block 2 is the loop

exit, and halts.

$p_{\text{sample}} =$
$\mathbf{L}_0$: **cons** $\mathbf{v}_0$ $\mathbf{v}_0$ $\mathbf{v}_1$; **cons** $\mathbf{v}_0$ $\mathbf{v}_1$ $\mathbf{v}_1$; **cons** $\mathbf{v}_0$ $\mathbf{v}_1$ $\mathbf{v}_1$; **jump** $\mathbf{L}_1$;
$\mathbf{L}_1$: **branch-if-nil** $\mathbf{v}_1$ $\mathbf{L}_2$; **fetch-field** 1 $\mathbf{v}_1$ $\mathbf{v}_1$; **branch-if-nil** $\mathbf{v}_0$ $\mathbf{L}_1$; **jump** $\mathbf{L}_2$;
$\mathbf{L}_2$: **halt**;
end

The program is well-typed with

$$\Pi_{\text{sample}} \;=\; \mathbf{L}_0 : (\mathbf{v}_0 : \text{nil},\ \{\}),\ \mathbf{L}_1 : (\mathbf{v}_0 : \text{nil},\ \mathbf{v}_1 : \text{list nil},\ \{\}),\ \mathbf{L}_2 : \{\},\ \{\}$$

3 Mechanization tasks

Implementing the "list-machine" benchmark in a mechanized metatheory (MM) comprises the following tasks:

1. Represent the operational semantics in the MM.
2. Derive the fact that $p_{\text{sample}} \Downarrow$. *The MM should conveniently simulate execution of small examples, so the user can debug the SOS and get an intuitive feel for its expressiveness.*

Soundness of a type system.

3. Represent the type system in the MM (define enough notation to represent the formula $\models_{\text{prog}} p : \Pi$ and inference rules from which type-soundness could be proved).
4. Represent in the MM an algorithm for least-common-supertype, that is, the computation $\tau_1 \sqcap \tau_2 = \tau_3$ producing τ_3 from inputs τ_1 and τ_2.
5. Using the type system, derive the fact that $\models_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$. *The MM should conveniently simulate type-checking of small examples, so the user can debug the type system and get a feel for its expressiveness.*
6. Represent the statement of the defining properties of least common supertypes, e.g., $\tau_1 \sqcap \tau_2 = \tau_3 \;\Rightarrow\; \tau_1 \subset \tau_3$.
7. Prove that the $\sqcap$ algorithm enjoys these properties.
8. Represent the statement of a soundness theorem for the type system. The informal statement of soundness is, "a well-typed program will not get stuck." A program state is not stuck if it *steps or halts*:

$$\frac{\models_{\text{prog}} p : \Pi \quad \text{initial}(p, r, \iota) \quad (r, \iota) \overset{p}{\mapsto}{}^{*} (r', \iota')}{(\exists r'', \iota''.\, (r', \iota') \overset{p}{\mapsto} (r'', \iota'')) \vee \iota' = \mathbf{halt}}\ \text{soundness}$$

9. Prove the soundness theorem. The full paper [1] outlines the principal lemmas of this proof, which is a standard argument by *type preservation* and *progress*.

Efficient type-checking algorithm.

10. Represent an asymptotically efficient type-checking algorithm $\vdash_{\text{prog}} p : \Pi$ in the MM. By efficient we mean that an N-instruction program with M live variables should type-check in $O(N \log M)$ time.
11. Using the type-checking algorithm, calculate $\vdash_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$. *The MM should simulate execution of algorithms on small inputs.*
12. Prove that the type-checking algorithm terminates on any program.

13. Demonstrate the type-checker on large-scale examples with good performance. Typically this will be done through an automatic translation to Prolog or ML which is then compiled by an optimizing compiler.
14. Prove that $\vdash_{\text{prog}} p : \Pi$ implies $\models_{\text{prog}} p : \Pi$. *That is, the type-checker soundly implements the type system.*

Writing the paper.

15. Use an automatic tool to generate readable LaTeX formulas for the SOS rules, the typing rules, and the statements of (not the proofs of) the least-common-supertype lemmas and soundness theorems. *Klein and Nipkow [6] demonstrate this very nicely in the Isabelle/HOL formalization of a Java subset compiler.*

4 A proof in Twelf metatheory

The Twelf system[12] is an implementation of the Edinburgh Logical Framework (LF). One can represent the operators of a logic as type constructors in LF, and proofs in that logic as terms in LF, and one can do proof-checking by type-checking the terms (considering them as derivations).

In Twelf one can prove *theorems* (proofs *in* a logic), or *metatheorems* (proofs *about* a logic). Either approach could be used for our benchmark. Our solution uses the usual approach in Twelf, which is metatheoretic.

In this case the logics in question are our operational semantics and our type system, and the metatheorem to be proved is type soundness: that is, if one can combine the inference rules of the type system to produce a derivation of type-checking, then it must be possible to combine the inference rules of the SOS to produce (only) non-stuck derivations of execution.

This approach is aggressively syntactic. Instead of saying that p is a mapping from labels to instructions, we give syntactic constructions that (we claim) represent such a mapping. One consequence of this style is that our $\models_{\text{prog}} p : \Pi$ is not just a semantic relation, but a syntactically derivable one expressed as Horn clauses. By carefully structuring the Horn clauses that define our relations so that we can identify "input" and "output" arguments, we can ensure that the logic-programming interpretation of our clauses is actually an algorithm. This input-output organization can be specified and mechanically checked in Twelf via `%mode` declarations. Our type system is then directly executable in Twelf.

Each clause in Twelf is named. When Twelf traces out, via Prolog-style backtracking, one or more derivations of a result by the successful application of clauses, it builds as well a derivation tree for each derivation.

In LF, one can compute as well on the derivation trees themselves. Suppose we write another Prolog program (set of clauses) that takes as input a derivation tree for type-checking, and produces as output a derivation tree for safe (non-stuck) execution. If this program is *total* (that is, terminates successfully on any input) then we have constructively proved that any well typed program is safe.

To reason about this meta-program, we use (machine-checked) `%mode` declarations to explain what are the inputs and outputs of the derivation transformer. We also use (machine-checked) `%total` declarations to ensure that our meta-program

has covered all the cases that may arise, and that our meta-program does not infinite-loop. We give an example of such a proof in section 6, items 6 and 7.

Twelf has an amazing economy of features. One does not have to learn a module system—because there is none—one just uses naming conventions on all one's identifiers. One does not have to learn how to use large libraries of lemmas and tactics, because there are no libraries of lemmas and tactics: but such libraries would not be so useful, because Twelf has few abstraction features, and no polymorphism. All proofs are done with the simple mechanism of proving the totality of metaprograms. There's a calculated gamble here: In return for the benefit of proving everything in one simple style, and rarely having to translate between abstractions, one trades away many things: there are some theorems that this notation cannot even express (because the quantifiers are nested too deep, for example); and there are some things that are provable but in a contrived way (expressing semantic properties only with inductive syntactic constructors), as illustrated below.

Our Twelf proof starts by defining inductively the notion of equalities and inequalities on natural numbers, labels, variables, type structure, and term structures. We give syntactic characterizations of well-formed environments (i.e., that do not map the same variable twice).

Sometimes it is tricky to make a properly inductive syntactic definition of a semantic property. For example, consider environment subtyping, semantically $\Gamma_1 \subset_{\text{env}} \Gamma_2 \;\equiv\; \forall v.\; v \in \text{dom}\,\Gamma_2 \;\Rightarrow\; (v \in \text{dom}\,\Gamma_1 \;\wedge\; \Gamma_1(v) \subset \Gamma_2(v))$.

An "obvious" "inductive" definition uses the syntactic rules,

$$\frac{}{\Gamma \subset \{\,\}}\; a_1 \qquad \frac{\Gamma_1(v) = \tau' \quad \tau' \subset \tau \quad \Gamma_1 \subset_{\text{env}} \Gamma_2}{\Gamma_1 \;\subset_{\text{env}}\; v : \tau,\; \Gamma_2}\; a_2$$

The induction is (supposedly) over the size of the term to the right of the $\subset_{\text{env}}$ symbol. However, this definition is not sufficiently inductive for useful properties (transitivity, reflexivity) to be provable—at least, we were not able to prove them. The problem appears to be that Γ_1 does not decrease in rule a_2.

The following definition is properly inductive—we use Γ' instead of Γ_1 in the premise of rule b_2. Proving transitivity and reflexivity from this definition is easy; the difficulty is to avoid wasting time with the pseudo-inductive definition above.

$$\frac{}{\Gamma \subset_{\text{env}} \{\,\}}\; b_1 \qquad \frac{\Gamma_1 \doteq (v : \tau',\; \Gamma') \quad \tau' \subset \tau \quad \Gamma' \subset_{\text{env}} \Gamma_2}{\Gamma_1 \;\subset_{\text{env}}\; v : \tau,\; \Gamma_2}\; b_2$$

5 A proof in Coq

The Coq system [5,4] is a proof assistant based on the Calculus of Inductive Constructions. This logic is a variant of type theory, following the "propositions-as-types, proofs-as-terms" paradigm, enriched with built-in support for inductive and coinductive definitions of predicates and data types.

From a user's perspective, Coq offers a rich specification language to define problems and state theorems about them. This language includes (1) constructive logic with all the usual connectives and quantifiers; (2) inductive definitions via inference rules and axioms (as in Twelf's meta-logic); (3) a pure functional programming language with pattern-matching and structural recursion (in the style of ML or

Haskell).

For the list-machine benchmark, we used a combination of all three specification styles, following common practice in research papers on type systems. The inference rules for operational semantics and the type systems are transcribed directly as inductive definitions. Operations over stores, environments and program-typing, as well as least common supertypes and the type-checking algorithm are presented as functions. Finally, subtyping between environments $\Gamma \subset \Gamma'$ is defined by the propositional formula

$$\forall v, \forall t',\ \Gamma'(v) = t' \Rightarrow \exists t,\ \Gamma(v) = t \wedge t \subset t'$$

Unlike Twelf's meta-theory, the logic of Coq provides rich forms of polymorphism. This enabled us to factor out the treatment of stores, environments, and program-typing by reusing an efficient, polymorphic implementation of finite maps as radix-2 search trees developed earlier by Leroy as part of the Compcert project [8].

6 Comparison of mechanized proofs

	Task	Twelf	Coq	
1.	Operational Semantics	126	98	lines
2.	Derive $p \Downarrow$	1	8	
3.	Type system $\models_{\text{prog}} p : \Pi$	167	130	
4.	$\sqcap$ algorithm	*	*	
5.	Derive $\models_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$	1	no	
6.	State properties of $\sqcap$	12	13	
7.	Prove properties of $\sqcap$	114	21	
8.	State soundness theorem	29	15	
9.	Prove soundness of $\models_{\text{prog}} p : Pi$	2060	315	
10.	Efficient algorithm	22	145	
11.	Derive $\vdash_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$	1	1	
12.	Prove termination of $\vdash_{\text{prog}} p : \Pi$	18	0	
13.	Scalable type-checker	yes	yes	
14.	Prove soundness of $\vdash_{\text{prog}} p : Pi$	347	141	
15.	Generate LaTeX	no	no	

We have implemented those tasks that are implementable in both the Twelf (metatheory) and Coq systems. The number of lines of code required is summarized in the table above. Total parsing and proof-checking time[3] was 0.558 seconds real time for Twelf, 2.622 seconds for Coq.

1. Operational semantics. Both Twelf and Coq make it easy and natural to represent inductive definitions of the kind found in SOS. In Coq one also has the choice of representing operations over mappings (e.g., lookup and update in stores) either as relations (defined by inductive predicates) or as functions (defined by recursion and pattern-matching).

2. Derive $p \Downarrow$. Twelf makes it very easy to interpret inductive definitions as logic programs. Therefore this task was trivial in Twelf. Coq does not provide a general mechanism to execute inductive definitions. However, the rules for the

[3] Dell Precision 360, Linux, 2.8 GHz Pentium 4, 1GB RAM, 512kB cache.

operational semantics were simple enough that (after some experimentation) we could use the proof search facilities of Coq (the `eauto` tactic) as a poor man's logic program interpreter. A more general method to execute inductive definitions in Coq, which we implemented also, is to define an execution function (61 lines), prove its correctness with respect to the inductive definition (35 lines), then execute the function. (Evaluation of functional programs is supported natively by Coq.)

3. Represent the type system. Easy and natural in both Twelf and Coq (with, as before, the choice in Coq of using the functional presentation of operations over mappings).

4. Least-upper-bound algorithm. Because the "type system" represented in Twelf is most straightforwardly done as a constructive algorithm, this was already done as part of task 3 in our Twelf representation. In Coq, while the type system itself is not algorithmic, we chose to specify the least-upper bound operation as a function from pairs of types to types. Therefore, the algorithm to compute least-upper bounds was already done as part of task 3 in the Coq development as well.

5. Derive an example of type-checking. Trivial to do in Twelf, by running the type system as a logic program. Not directly possible in Coq because the specification of the type system is not algorithmic: it uses universal quantification over all variables to specify environment subtyping.

6. State properties of least-upper-bound. Entirely straightforward in Coq. For example, here are the Coq statements of these properties:

```
Lemma lub_comm:          forall t1 t2, lub t2 t1 = lub t1 t2.
Lemma lub_subtype_left:  forall t1 t2, subtype t1 (lub t1 t2).
Lemma lub_subtype_right: forall t1 t2, subtype t2 (lub t1 t2).
Lemma lub_least:         forall t1 t3, subtype t1 t3 ->
           forall t2, subtype t2 t3 -> subtype (lub t1 t2) t3.
```

The correspondence with the mathematical statements of these properties is obvious.

In Twelf, stating the properties of least-upper-bound must be done in a way that seems artificial at first, but once learned is reasonably natural. The lemma

$$\frac{\tau_1 \sqcap \tau_2 = \tau_3}{\tau_1 \subset \tau_3}\ \text{lub-subtype-left}$$

is represented as a logic-programming predicate,

```
lub-subtype-left: lub T1 T2 T3 -> subtype T1 T3 -> type.
```

which transforms a derivation of `lub T1 T2 T3` into a derivation of `subtype T1 T3`. The "proof" will consist of logic-programming clauses over this predicate. To be a "proof" of the property we want, we will have to demonstrate (to the satisfaction of the metatheory, which checks our claims) that our clauses have the following properties:

`%mode lub-subtype-left +P1 -P2.` The *modes* of a logic program specify which arguments are to be considered inputs (`+`) and which are outputs (`-`). Formally, given any ground term (i.e., containing no logic variables) `P1` whose type is `lub T1 T2 T3`, our clauses (if they terminate) must produce outputs `P2` of type `subtype T1 T3` that are also ground terms.

`%total P1 (lub-subtype-left P1 P2).` We ask the metatheorem to check our claim that no execution of lub-subtype-left can infinite-loop: it must either fail or produce a derivation of `subtype T1 T3`; *and* we check the claim that the execution never fails (that all cases are covered). The use of `P1` in two places in our `%total` declaration is (in some sense) mixing the thing to be proved with part of the proof: we indicate that the induction should be done over argument 1 of lub-subtype-left, not argument 2.

7. Prove properties of least-upper-bound. In Twelf this is done by writing logic-programming clauses that satisfy all the requirements listed above. For example, the following 9 clauses will do it:

```
-: lub-subtype-left lub-refl subtype-refl.
-: lub-subtype-left lub-1 subtype-refl.
-: lub-subtype-left (lub-2 P1) (subtype-list P2) <-
                lub-subtype-left P1 P2.
-: lub-subtype-left (lub-2b P1) (subtype-listcons P3) <-
                lub-subtype-left P1 P3.
-: lub-subtype-left (lub-3 P1) (subtype-list P2) <-
                lub-subtype-left P1 P2.
-: lub-subtype-left lub-4 subtype-nil.
-: lub-subtype-left lub-5 subtype-nil.
-: lub-subtype-left lub-6 (subtype-listcons subtype-refl).
-: lub-subtype-left (lub-7 P1) (subtype-listmixed P2) <-
                lub-subtype-left P1 P2.
```

These are not clauses of a type-checker, they are clauses *about* a type-checker, and serve only to "prove" the `%mode` and `%total` declarations.

In Coq, the proofs are done interactively by constructing proof scripts. For example, the proof of `lub_subtype_left` is:

```
induction t1; destruct t2; simpl; auto; rewrite IHt1; auto.
```

which corresponds to doing an induction on the structure of the first type `t1`, then a case analysis on the second type `t2`, then some equational reasoning.

There are 6 separate steps to the Coq proof, each takes just two or three tokens to write, and each takes some thought from the user. On the other hand, each of the 9 clauses of the Twelf proof, ranging in size from 6 to 16 tokens, also takes some thought. The time or effort required to build a proof is not necessarily proportional to the token count, but we report what measures we have.

8. State soundness theorem for the type system. In Coq, the statement is just ordinary mathematics. In Twelf, this is done, as above, by writing a logical predicate that relates a derivation of type-checking to a derivation of runs-or-halts, and then making the appropriate `%mode` and `%total` claims for the Twelf system to check.

9. Prove soundness of the type system. Writing such a logic program in Twelf takes more than 2000 lines; our full paper [1] explains this proof in more detail. The Coq proof of soundness is about 7 times shorter (300 lines). There are several reasons for Coq's superiority over Twelf here. The first is Coq's proof

automation facilities, which were very effective for many of the intermediate proofs: once we indicated manually the structure of the inductions, Coq's proof search tactics were often able to derive automatically the conclusion from the hypotheses. A second reason is the use of non-algorithmic specifications, especially for environment subtyping, which are simpler to reason about. The last reason is the ability to reuse basic properties over mappings, such as the so-called "good variables" properties, instead of proving them over and over again.

Twelf lacks the ability to create and re-use abstract data types, so many clauses of the program and proof must be copied and edited. Twelf has some proof automation—the `%total` declaration calculates the structural induction automatically, and (if it fails) prints a report detailing the missing cases—but it does not automate the case analysis. [4]

10. Asymptotically efficient algorithm. In Twelf, the most straightforward representation of the type system, when run as an algorithm, takes quadratic time. This is because the rules for looking up labels in global environments Π involve a search of the length of Π for each lookup. In any Prolog system that permits the efficient dynamic assertion of new clauses, one can do lookup in constant time (the Prolog system uses hashing internally). Twelf supports dynamic clauses, so we can write a nice linear-time "type-checker" as a new logic program, reusing many of the Horn clauses that constitute the "type system."

In Coq, the type-checker is defined as a function from program typing and programs to booleans. Our solution uses intermediate functions for checking environment subtyping and for type-checking instructions and blocks. These functions return option types to signal typing errors, which are propagated in a monadic style. To avoid an n^2 algorithm, we represent environments and program typing as finite maps implemented by radix-2 search trees. Therefore, the typing algorithm has $O(n \log n)$ complexity.

11. Simulate the new algorithm. This is a trivial matter both in Twelf and in Coq. In Twelf, once again, we perform a one-line query in the logic-program interpreter. In Coq, we simply request the evaluation of a function application (of the type-checker to the sample program and program typing), which is also one line.

12. Prove termination of the type-checker. Twelf has substantial automated support for doing proofs of termination of logic programs (such as the type-checker) where the induction is entirely structural. This task was very easy in Twelf.

In Coq, this task was even easier: all functions definable in Coq are guaranteed to terminate (in particular, all recursions must be either structural or well-founded by Noetherian induction), so there was nothing to prove for this task.

13. Industrial-strength type-checker. Coq has a facility to automatically generate Caml programs from functions expressed in Coq. Automatic extraction of Caml code from the Coq functional specification of the type-checker produces code that is close to what a Caml programmer would write by hand if confined to the

[4] Supplying the case analysis automatically will be the job of the Twelf metatheorem prover. Unfortunately, it appears that the metatheorem prover does not work; the Twelf manual says, "The theorem proving component of Twelf is in an even more experimental stage and currently under active development" [11] and every version of the manual since 1998 contains this identical sentence. One doubts whether the last two words are accurate.

purely functional subset of the language.

Similarly, Twelf programs (such as our type-checker) that don't use higher-order abstract syntax can be automatically translated to Prolog, and those that use HOAS can be automatically translated to lambda-Prolog. There are many efficient Prolog compilers in the world, and there is one efficient lambda-Prolog compiler.

14. Prove soundness of type-checker. Straightforward (though a bit tedious) both in Twelf and in Coq. Again, Coq's proof automation facilities result in a significantly shorter proof (3 times shorter than the Twelf proof).

15. Generate LaTeX. Although both Coq and Twelf have facilities for generating LaTeX, neither has a facility that is sufficiently useful for the purposes of this benchmark.

7 Conclusion

Proofs of semantic properties of operational specifications can be aggressively "semantic," meaning that they avoid *all* proof-theoretic induction over syntax; denotational-semantic approaches and logical-relations models have this flavor. We have not discussed these approaches in this paper, but they can be successfully mechanized in Coq, in Isabelle/HOL, or in an object logic embedded in Twelf; however, it does not seem natural to mechanize semantic proofs in Twelf metatheory.

Or the proofs can be aggressively "syntactic," meaning that *only* proof-theoretic induction is used, and we avoid any attribution of "meaning" to the operators; the Wright-Felleisen notation [14] encourages this approach. Coq and Isabelle support this style, among others; Twelf metatheory supports *only* this pure proof-theoretic style. The advantages to using a pure style are that the metatheory itself can be much smaller and simpler—making it easier to learn and easier to reason about. Indeed, Twelf is a much simpler and smaller system than Coq.

Between these two extremes, it is possible to reason using a mix of semantic and syntactic reasoning. Authors who believe they are writing in a purely Wright-Felleisen style are often reasoning semantically about such things as environments and mappings. The Coq system supports the mixed style (or either of the two extremes) reasonably well. Therefore, it may be the case that specifications expressed in Coq are closer to what one would write in a research paper. Coq proofs can be substantially shorter than Twelf proofs, especially when experienced experts are manipulating the language of tactics. Therefore Coq may be a language of choice for those who do not want to commit in advance to a purely proof-theoretic style.

However, our benchmark does not exercise one of the main strengths of the Twelf system, the higher-order abstract syntax and related proof mechanisms. For syntactic theories that use binders and $\alpha\beta\eta$-conversion, the comparison might come out differently.

References

[1] Appel, A. W. and X. Leroy, *A list-machine benchmark for mechanized metatheory*, Research report 5914, INRIA (2006).

[2] Appel, A. W. and X. Leroy, *List-machine exercise* (2006), http://www.cs.princeton.edu/~appel/listmachine/ or http://gallium.inria.fr/~xleroy/listmachine/.

[3] Aydemir, B. E., A. Bohannon, N. Foster, B. Pierce, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, M. Fairbairn and P. Sewell, *The POPLmark challenge* (2005), http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark/.

[4] Bertot, Y. and P. Castran, "Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions," EATCS Texts in Theoretical Computer Science, Springer-Verlag, 2004.

[5] *The Coq proof assistant* (1984–2006), software and documentation available from http://coq.inria.fr/.

[6] Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine and compiler*, ACM Transactions on Programming Languages and Systems (To appear).

[7] Leinenbach, D., W. Paul and E. Petrova, *Towards the formal verification of a C0 compiler*, in: *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)* (2005), pp. 2–11.

[8] Leroy, X., *Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant*, in: *POPL'06: 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2006), pp. 42–54.

[9] Moore, J. S., *A mechanically verified language implementation*, Journal of Automated Reasoning **5** (1989), pp. 461–492.

[10] Moore, J. S., "Piton: a mechanically verified assembly-language," Kluwer, 1996.

[11] Pfenning, F. and C. Schuermann, *Twelf user's guide, version 1.4* (2002), http://www.cs.cmu.edu/~twelf/guide-1-4.

[12] Pfenning, F. and C. Schürmann, *System description: Twelf — a meta-logical framework for deductive systems*, in: *The 16th International Conference on Automated Deduction* (1999).

[13] Strecker, M., *Formal verification of a Java compiler in Isabelle*, in: *Proc. Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science **2392** (2002), pp. 63–77.

[14] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.

A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F

Kevin Donnelly[1,2] and Hongwei Xi[1,3]

Computer Science Department, Boston University
Boston, USA

Abstract

We formalize in the logical framework ATS/LF a proof based on Tait's method that establishes the simply-typed lambda-calculus being strongly normalizing. In this formalization, we employ higher-order abstract syntax to encode lambda-terms and an inductive datatype to encode the reducibility predicate in Tait's method. The resulting proof is particularly simple and clean when compared to previously formalized ones. Also, we mention briefly how a proof based on Girard's method can be formalized in a similar fashion that establishes System F being strongly normalizing.

Keywords: Logical frameworks, Normalization, Tait's method, Logical relations, Reducibility candidates, HOAS, ATS/LF

1 Introduction

ATS/LF [4] is a logical framework rooted in the Applied Type System [15] and is a pure total fragment of the programming language ATS. It uses a restricted form of dependent types in which types may only be indexed by terms drawn from limited domains in which equality is decidable (and can also be effectively reasoned about). ATS/LF supports the use of higher-order abstract syntax (HOAS) [9] to encode object languages. The use of HOAS, in which object variables are identified with metavariables and β-reduction models substitution, leads to particularly simple and elegant encodings. The combination of a limited type-index language and a powerful proof language, as found in ATS/LF, allows for inductive proofs of metatheorems over full higher-order abstract syntax to be directly encoded as total recursive functions. The use of inductive datatypes with negative occurrences allows for the encoding of the reducibility predicate.

[1] The work is partly funded by NSF grant CCR-0229480

[2] Email: kevind@cs.bu.edu

[3] Email: hwxi@cs.bu.edu

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In this paper, we formalize a proof of strong normalization of the simply typed lambda-calculus (STLC) using Tait's method, closely following the one in [7]. On one hand, we use HOAS to encode lambda-terms, obviating the need for explicitly manipulating substitution on such terms. On the other hand, we use first-order abstract syntax (FOAS) to encode typing derivations in STLC, which conveniently supports inductive reasoning on typing derivations.

To our knowledge this is the first formalized (or mechanized) proof of strong normalization using Tait's method for an object language defined with HOAS. When compared to other formalized proofs of strong normalization in the literature, the brevity of our formalized proof and its closeness to the concise and elegant proof in [7] yield some concrete evidence in support of the effectiveness of the representation of STLC in ATS/LF. To further strengthen this claim, we also discuss the extension to the case of System F, formalizing a proof of strong normalization of System F based on Girard's notion of reducibility candidates [6]. We expect that the techniques developed here can also allow for the formalization of other proofs by logical relations while still being able to take advantage of HOAS.

2 ATS/LF

ATS/LF is split into two main parts: the language of types and type indices (called the *statics*), and the language of proofs (called the *dynamics*). The statics is basically simply-typed lambda-calculus with constants (but no recursion), and terms in the statics are referred to as *static terms* and types in the statics are referred to as *sorts*. There are three important built-in base sorts:

- prop : A sort for static terms which represent types of proofs.
- int : A sort for static integer terms. There are constants for each integer $(\ldots, \mathit{-1}, \mathit{0}, \mathit{1}, \ldots : \mathit{int})$ and for addition $(+ : (\mathit{int}, \mathit{int}) \rightarrow \mathit{int})$ and subtraction $(- : (\mathit{int}, \mathit{int}) \rightarrow \mathit{int})$.
- bool : A sort for static boolean conditions. There are constants for truth values $(\mathit{true}, \mathit{false} : \mathit{bool})$ and equality and inequality on integers $(=, < : (\mathit{int}, \mathit{int}) \rightarrow \mathit{bool})$.

Static constants may take multiple arguments. Equality in the statics is basically β-conversion plus Presburger arithmetic, and it is decided by converting to $\beta\eta$ long normal form and then using a decision procedure for integer (in)equalities (after mapping boolean terms to integer terms).

The dynamics is a dependently typed language with well-founded recursion, exhaustive case-analysis and inductive datatypes. Termination is checked using a programmer-supplied metric, which is a tuple of static terms representing natural numbers and decreasing in each recursive call according to the standard lexicographic ordering. Please see [13] for more details on this style of termination checking. Case coverage is checked by requiring that any unlisted cases introduce assumptions that allow *false* to be proven [14]. In the concrete syntax, a proof (function) declaration looks like:

Syntax:

terms	$t \in tm ::= x \mid \lambda x.t \mid t_1\ t_2 \mid \mathsf{c}$
types	$\tau \in tp ::= \mathsf{B} \mid \tau_1 \to \tau_2$
contexts	$\Gamma \in ctx ::= \cdot \mid \Gamma, x : \tau$

Fig. 1. Syntax for Simply-typed λ-calculus

```
prfun proofName {x1:stx1, ..., xn:stxn} .<m1, ..., mk>.
      (p1:T1, ..., pl:Tl) : [y1:sty1, ..., ym:stym] T = ...
```

This declaration is for a total recursive function called *proofName* (`prfun` is a keyword for introducing proof functions) with the type:

$$\forall x_1 : stx_1, \ldots, \forall x_n : stx_n.(T_1, \ldots, T_l) \to \exists y_1 : sty_1, \ldots, \exists y_m : sty_m.T$$

This type signature consists of four parts. First, there are n static parameters x_i of sorts stx_i, enclosed in curly braces (think of these as universally quantified). Second, there is a metric, enclosed in `.<` and `>.`, which is a k-tuple of static terms representing natural numbers and may contain $x_1, \ldots, x_n$. Third, there are l dynamic parameters p_i with types T_i that may contain $x_1, \ldots, x_n$. Fourth, there is the return type which consists of m existentially quantified static variables y_i of sorts sty_i and a type T which may contain $x_1, \ldots, x_n, y_1, \ldots, y_m$. In the case where the declared function *proofName* is not recursive, we may also use the keyword `prfn` and give no metric. Please see [4,5] for some examples of proofs formed in ATS/LF.

3 Encoding the Object Language

3.1 Syntax

The object language for which we prove strong normalization is STLC with a constant c and a base type B. The syntax of the language is shown in Figure 1. We will encode the syntax in the statics using HOAS. In order to do so we declare a static sort for each syntactic category. We begin with a sort, *tm*, with constructors for each term constructor of the object language:

$$TMlam : (tm \to tm) \to tm \qquad TMapp : (tm, tm) \to tm \qquad TMcst : tm$$

Object variables are encoded as metavariables. The constant *TMcst* is only used in the formalization as a placeholder when recursing under lambda binders. Object functions are represented by functions in the statics, and this allows us to model substitution in the object language with application in the metalanguage. The terms of the object language are encoded in the statics with the function $\ulcorner \cdot \urcorner$ defined by:

$$\ulcorner x \urcorner = x \qquad \ulcorner \mathsf{c} \urcorner = TMcst$$
$$\ulcorner \lambda x.t \urcorner = TMlam(\lambda x.\ulcorner t \urcorner) \qquad \ulcorner t_1\ t_2 \urcorner = TMapp(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner)$$

This is a compositional bijection between terms of the object language with up to n free variables and static terms of sort *tm* with up to n free variables.

To encode types we declare a sort *tp*, with constructors for each type constructor of the object language:

$$TPbas : tp \qquad TPfun : (tp, tp) \to tp$$

In some encodings with HOAS, there is no explicit representation of contexts in the representation of typing judgments, but instead the context of the metalanguage is

Reduction: $t_1 \longrightarrow t_2$

$$\frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'}\ (REDlam) \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ (REDapp1)$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}\ (REDapp2) \qquad \frac{}{(\lambda x.t_1)\ t_2 \longrightarrow t_1[t_2/x]}\ (REDapp3)$$

Fig. 2. Reduction rules for λ-calculus

utilized. Such higher-order representations of the typing judgment, as often used in Twelf [10], benefit from inheriting substitution on typing from the metalanguage, and so do not need a typing substitution lemma. On the other hand, the use of explicit contexts allows for a first-order representation of typing derivations. This, along with the separation between statics and dynamics, allows us to prove metatheorems directly, using total recursive functions, while still taking advantage of HOAS for object syntax. The inconvenience of having to prove substitution on typing derivations is minor, and not pervasive as issues involving binders in the syntax are. In fact, we do not ever need to make use of substitution on typing derivations in the proof of strong normalization. Contexts, of sort *ctx*, are represented by lists of pairs of a *tm* and a *tp*:

$$CTXnil : ctx \qquad CTXcons : (tm, tp, ctx) \rightarrow ctx$$

We may sometimes abbreviate *CTXcons(t,T,G)* as $(t, T) :: G$. Really this sort represents explicitly typed substitutions. A term of sort *ctx* only represents a well-formed context if its *tm* subterms are all distinct metavariables. We will return to this issue when we encode typing derivations.

3.2 Reduction

The rules for small-step reduction for pure λ-calculus are shown in Figure 2. Reduction, $t \longrightarrow t'$, is encoded as a datatype with type constructor $RED : (tm, tm, int) \rightarrow prop$ (where the third index measures the size of the derivation) and one term constructor to encode each rule in Figure 2. The most interesting rules are *REDlam* and *REDapp3* which correspond to the dynamic term constructors:

$$\begin{aligned} REDlam &: \forall f : tm \rightarrow tm. \forall f' : tm \rightarrow tm. \forall n : nat. \\ &\quad (\forall x : tm.\ RED(f\ x, f'\ x, n)) \rightarrow RED(TMlam\ f, TMlam\ f', n+1) \\ REDapp3 &: \forall f : tm \rightarrow tm. \forall t : tm.\ RED(TMapp(TMlam\ f, t), f\ t, 0) \end{aligned}$$

Since the rules themselves are first order, adequacy follows from the fact that the higher-order syntax in the type indices correspond to the right terms. The most interesting rule is *REDlam*: from the quantification in the argument of the constructor $(\forall x : tm.\ RED(f\ x, f'\ x, n))$ and the fact that application in the statics models substitution, we can see that $f\ x$ and $f'\ x$ represent lambda-terms with x being free and that *TMlam* f and *TMlam* f' represent these same terms with x bound by a lambda.

3.3 Type Assignment

The rules for typing judgments are shown in Figure 3. We begin by defining the context lookup relation $(x : \tau) \in \Gamma$. For this we use a datatype with type constructor $INCTX : (tm, tp, ctx, int) \rightarrow prop$, where $INCTX(t, T, G, n)$ means that (t, T) is at

Type formation: $\vdash \tau\ type$

$$\frac{}{\vdash \mathsf{B}\ type}\ (TPbas) \qquad \frac{\vdash \tau_1\ type \quad \vdash \tau_2\ type}{\vdash \tau_1 \to \tau_2\ type}\ (TPfun)$$

Typing: $\Gamma \vdash t : \tau$

$$\frac{(x:\tau) \in \Gamma \quad \vdash \tau\ type}{\Gamma \vdash x : \tau}\ (DERvar)$$

$$\frac{\Gamma, x:\tau_1 \vdash t : \tau_2 \quad \vdash \tau_1\ type}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}\ (DERlam) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ (DERapp)$$

Fig. 3. Typing rules for Simply-typed λ-calculus

the n^{th} index in G (abbreviated as $(t,T) \in_n G$), and two term constructors which correspond to the rules:

$$\frac{}{(t,T) \in_0 ((t,T) :: G)}\ (INCTXone) \qquad \frac{(t,T) \in_n G}{(t,T) \in_{n+1} ((t',T') :: G)}\ (INCTXshi)$$

Note that if $INCTX(t,T,G,n)$ is inhabited, its member is unique and isomorphic to n (since it is a non-branching tree of depth n).

We encode the judgment $\vdash \tau\ type$ with a datatype, where the type constructor is $TP : (tp, int) \to prop$ and the term constructors represent the following rules (where we write $\vdash_n T\ type$ for $TP(T,n)$):

$$\frac{}{\vdash_0 TPbas\ type}\ (TPbas) \qquad \frac{\vdash_{n_1} T_1\ type \quad \vdash_{n_2} T_2\ type}{\vdash_{n_1+n_2+1} TPfun(T_1,T_2)\ type}\ (TPfun)$$

While the constructors of this type have the same names as terms of sort tp, there is no ambiguity because dynamic terms are strictly separated from static terms. The type $TP(T,n)$ contains a single element which is isomorphic to T if the size of T is n. The size index is used to provide a metric to support induction on the structure of types. For convenience, we define $TP0(T) \equiv \exists n : nat.\ TP(T,n)$ (which we abbreviate as $\vdash T\ type$).

The encoding of the typing judgment $\Gamma \vdash t : \tau$ is a dependent datatype, $DER : (ctx, tm, tp, int) \to prop$, where the last index is a measure of the size of the typing derivation. The constructors correspond to the inference rules in Figure 4 (where $G \vdash_n t : T$ abbreviates $DER(G,t,T,n)$). The typing rule for variables is encoded by the term constructor:

$$DERvar : \forall G : ctx. \forall t : tm. \forall T : tp. \forall n : nat.\ (INCTX(t,T,G,n), TP0\ T) \to DER(G,t,T,0)$$

The context is represented as a list, so the variable lookup identifies the index in the list that corresponds to the given variable. The typing rule for lambda-abstraction is encoded by the following constructor:

$$\begin{aligned} DERlam : \forall G : ctx. \forall f : tm \to tm. \forall T_1 : tp. \forall T_2 : tp. \forall n : nat. \forall l : nat.\\ (TP0\ T_1, \forall x.\ DER(CTXcons(x,T_1,G), f\ x, T_2, n)) \to\\ DER(G, TMlam\ f, TPfun(T_1,T_2), n+1) \end{aligned}$$

Note that the quantification over x in the second argument of this constructor $(\forall x. DER(CTXcons(x,T_1,G), f\ x, T_2, n))$ guarantees that x is a metavariable not occurring in G and thus $CTXcons(x,T_1,G)$ is a well-formed context if G is. The

Encoded Typing: $G \vdash_n t : T$

$$\frac{(t,T) \in_n G \quad \vdash T \ type}{G \vdash_0 t : T} \ (DERvar)$$

$$\frac{\vdash T_1 \ type \quad (\forall x.\ (x,T_1) :: G \vdash_n f\ x : T_2)}{G \vdash_{n+1} TMlam\ f : TPfun(T_1,T_2)} \ (DERlam)$$

$$\frac{G \vdash_{n_1} t_1 : TPfun(T_1,T_2) \quad G \vdash_{n_2} t_2 : T_1}{G \vdash_{n_1+n_2+1} TMapp(t_1,t_2) : T_2} \ (DERapp)$$

Fig. 4. Encoded Typing Rules

typing rule for application is encoded by the following constructor:

$$\begin{aligned} DERapp : \forall G : ctx.\forall t_1 : tm.\forall t_2 : tm.\forall T_1 : tp.\forall T_2 : tp.\forall n_1 : nat.\forall n_2 : nat.\\ (DER(G,t_1,TPfun(T_1,T_2),n_1), DER(G,t_2,T_1,n_2)) \rightarrow \\ DER(G, TMapp(t_1,t_2), T_2, n_1+n_2+1) \end{aligned}$$

For convenience we also define $DER0(G,t,T) \equiv \exists n : nat.\ DER(G,t,T,n)$. This representation for typing derivations is quite interesting. The dynamic terms inhabiting the datatype $DER0(G,t,T)$ are isomorphic to simply-typed lambda-terms of Church-style in which variables are represented as de Bruijn indices. The context G is a typed substitution, which we can decompose into a substitution $\Theta = \langle t_1, \ldots, t_m \rangle$ (which maps the ith variable to t_i for $1 \leq i \leq m$) and a context $\Gamma = \langle T_1, \ldots, T_m \rangle$. The datatype $DER0(G,t,T)$ really represents a hypothetical judgment saying that if we have derivations of $\vdash t_i : T_i$ (for $1 \leq i \leq m$) then we can form a derivation of $\vdash t : T$. As long as Θ is a list of distinct meta-variables (say $\langle x_1, ..., x_m \rangle$), this is an adequate encoding of the usual typing judgment $x_1 : T_1, ..., x_m : T_m \vdash t : T$. We can guarantee that a context is well-formed in this way when it is empty or when it appears in a derivation that is a sub-derivation of one with an empty context. We are able to prove strong normalization for terms typed in the empty context and, since reduction under lambda is allowed, this implies strong normalization for terms containing free variables as well.

4 Strong Normalization Proof

In this section, we formalize a proof of strong normalization of STLC based on Tait's method [12]. The formalized proof is nearly identical to the one in [7], with the only exception that we use the constant `c` in some places where the proof in [7] uses a variable. The cause for this exception directly results from HOAS being chosen for representing lambda-terms (and thus making it difficult to manipulate object variables). The proofs for the final few lemmas and strong normalization theorem are given in Appendix A and the entire proof can be found on-line:

`http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/STLC-SN-hoas.dats`

Definition 4.1 (Strong Normalization) A term t is strongly normalizing with bound n, written $\mathsf{SN}_n(t)$, if for all t' such that $t \longrightarrow t'$ we have $\mathsf{SN}_{n'}(t')$ for some natural number $n' < n$ (i.e. all reduction sequences starting from t have length at most n). A term t is strongly normalizing, written $\mathsf{SN0}(t)$, if there is some n such that $\mathsf{SN}_n(t)$.

$\mathsf{SN}_n(t)$ is encoded using a dependent datatype with type constructor $SN : (tm, int) \rightarrow prop$ and one term constructor of the same name:

$$SN : \forall t : tm. \forall n : nat. (\forall t' : tm. RED0(t, t') \rightarrow \exists n' < n.\ SN(t', n')) \rightarrow SN(t, n)$$

We encode $\mathsf{SN0}(t)$ by defining $SN0(t) \equiv \exists n : nat.\ SN(t, n)$. Strong normalization is closed under forward and backward reduction.

Lemma 4.2 *If* $\mathsf{SN}_n(t)$ *and* $t \longrightarrow t'$ *then* $\mathsf{SN}_{n'}(t')$ *for some* $n' < n$.

Proof. *This follows directly from the definition of* $\mathsf{SN}_n(t)$. □

The ATS/LF proof for this lemma is given as follows:

```
prfn forwardSN {t:tm, t':tm, n:nat}
  (sn: SN(t, n), red: RED0(t, t')) : [n':nat | n' < n] SN(t', n') =
  let prval SN (fsn) = sn in fsn red end
```

The keyword `prval` here is similar to the keyword `val` in ML.

Lemma 4.3 *If for all* t', $t \longrightarrow t'$ *implies* $\mathsf{SN0}(t')$, *then* $\mathsf{SN0}(t)$.

Proof. *For any* t *there are a finite number of* t' *such that* $t \longrightarrow t'$. *For each of these* t' *we have* $\mathsf{SN}_{n'}(t')$ *for some* n'. *If we take* n *to be one plus the maximum of these* n' *(which exists because there are only finitely many) then we have* $\mathsf{SN}_n(t)$ *so* $\mathsf{SN0}(t)$. □

This is an obvious consequence of the definition of *SN0* and the fact that each term has a finite number of different reducts, and formalizing it in ATS/LF is entirely uninspiring (as the argument is purely set-theoretic). So we use the keyword `dynprf` to introduce it as an unproven lemma:

```
dynprf backwardSN : {t:tm} ({t':tm} RED0 (t, t') -> SN0 t') -> SN0 t
```

This is the only unproven lemma in the entire formalization.

Attempting to directly prove strong normalization of well-typed terms by induction on typing derivations does not work because the induction hypothesis is not strong enough to handle application terms. In order to make the proof go through, we strengthen the induction hypothesis using the notion of *reducibility*, introduced by Tait [12].

Definition 4.4 (Reducibility) A lambda-term t is reducible at a type τ, written $\mathsf{R}_\tau(t)$, if:

(i) τ is a base type (that is, B in our case) and $\mathsf{SN0}(t)$, or

(ii) τ is $\tau_1 \rightarrow \tau_2$ and for all t', $\mathsf{R}_{\tau_1}(t')$ implies $\mathsf{R}_{\tau_2}(t\ t')$.

It should be emphasized that $\mathsf{R}_\tau(t)$ does not necessarily imply that t can be assigned the type τ. As a matter of fact, we have $\mathsf{R}_\mathsf{B}(\omega)$ for $\omega = \lambda x.xx$ according to the definition. Also, it is clear that we cannot have $\mathsf{R}_{\mathsf{B}\rightarrow\mathsf{B}}(\omega)$ as it would otherwise imply $\mathsf{R}_\mathsf{B}(\omega\omega)$, which is a contradiction since $\omega\omega$ is not normalizing.

The definition in ATS/LF uses a dependent datatype with type constructor $R : (tm, tp) \rightarrow prop$ and two term constructors:

$$\begin{aligned} &Rbas : \forall t : tm.\ SN0\ t \rightarrow R(t, TPbas) \\ &Rfun : \forall t : tm. \forall T_1 : tp. \forall T_2 : tp. \\ &\qquad (\forall t_1 : tm. R(t_1, T_1) \rightarrow R(TMapp(t, t_1), T_2)) \rightarrow R(t, TPfun(T_1, T_2)) \end{aligned}$$

This is not a positive datatype because there is a negative occurrence of R in the function case. However, this definition is still well-founded because the tp index is structurally decreasing in all recursive occurrences (both positive and negative). This allows us to view the datatype as being built up inductively in levels stratified by the tp index. In particular, this means that when we are building the level corresponding to $TPfun(T_1, T_2)$, the levels corresponding to T_1 and T_2 are already complete and thus the set of functions from level T_1 to level T_2 (which are the possible arguments of $Rfun$) is also complete.

We begin by proving some important properties of the reducibility predicate. We first define neutral terms as follows.

Definition 4.5 (Neutrality) A term is neutral if it is either the constant c or an application of the form $t\ t'$.

This is defined in ATS/LF as a dependent datatype with type constructor $NEU : tm \to prop$ and term constructors:

$$NEUcst : NEU(TMcst) \qquad NEUapp : \forall t : tm. \forall t' : tm.\ NEU(TMapp(t, t'))$$

We can now state and prove four important properties of reducibility, which are given the names CR 1-4 in [7]:

CR 1: If $\mathsf{R}_\tau(t)$ then $\mathsf{SN0}(t)$,

CR 2: If $\mathsf{R}_\tau(t)$ and $t \longrightarrow t'$ then $\mathsf{R}_\tau(t')$,

CR 3: If t is neutral and for all t', $t \longrightarrow t'$ implies $\mathsf{R}_\tau(t')$, then $\mathsf{R}_\tau(t)$, and

CR 4: $\mathsf{R}_\tau(\mathsf{c})$ for any τ, which is a special case of CR 3.

We first prove CR 2 on its own, and then prove CR 1, 3 and 4 simultaneously.

Lemma 4.6 (CR 2) ***Proof.*** *By induction on τ:*

case: *$\tau = \mathsf{B}$, so we have $\mathsf{SN0}(t)$. By closure of strong normalization under forward reduction (Lemma 4.2) we have $\mathsf{SN0}(t')$, so $\mathsf{R}_\mathsf{B}(t')$.*

case: *$\tau = \tau_1 \to \tau_2$, so for all t_1, $\mathsf{R}_{\tau_1}(t_1)$ implies $\mathsf{R}_{\tau_2}(t\ t_1)$. Fix any t_1 such that $\mathsf{R}_{\tau_1}(t_1)$, then we have $\mathsf{R}_{\tau_2}(t\ t_1)$ and since $t\ t_1 \longrightarrow t'\ t_1$, by induction hypothesis, we have $\mathsf{R}_{\tau_2}(t'\ t_1)$. Therefore $\mathsf{R}_{\tau_1 \to \tau_2}(t')$.*

□

The proof is encoded in ATS/LF as follows:

```
prfun cr2 {t:tm, t':tm, T:tp, n:nat} .<n>.
  (tp: TP (T, n), r: R(t, T), rd : RED0(t, t')): R(t', T) =
  case* r of // [case*] indicates exhaustive pattern matching
    | Rbas (sn) => Rbas (forwardSN (sn, rd))
    | Rfun{_, T1, _} (fr) => let
        prval TPfun (_, tp2) = tp
      in
        Rfun(lam {t1:tm} (r:R(t1,T1)) => cr2(tp2, fr r, REDapp1 rd))
      end
```

This proof function is a fairly straightforward encoding of the argument, taking the extra argument of type $TP(T, n)$ to provide a termination metric. The proof has a slightly unusual feature: the $Rfun$ case binds the static argument T_1 in order to be able to provide the type for the lambda-bound variable r.

Lemma 4.7 (CR 1, 3, 4) ***Proof.*** *We prove CR 1, CR 3, CR 4, in that order,*

by induction on τ. The argument for CR 3 makes use of a nested induction, and CR 4 follows directly from CR 3 at each level.

case: $\tau = \mathsf{B}$. *Reducibility at base types is just strong normalization.*
CR 1: *Direct from the definition of* $\mathsf{R}_{\mathsf{B}}(\cdot)$.
CR 3: *By Lemma 4.3.*

case: $\tau = \tau_1 \to \tau_2$.
CR 1: *Let t be a term with $\mathsf{R}_{\tau_1\to\tau_2}(t)$. By CR 4 induction hypothesis, $\mathsf{R}_{\tau_1}(\mathsf{c})$, therefore $\mathsf{R}_{\tau_2}(t\ \mathsf{c})$. By CR 1 induction hypothesis $t\ \mathsf{c}$ is SN and any reduction of t induces a reduction of $t\ \mathsf{c}$, so t is SN.*
CR 3: *Let t be neutral such that for all t' with $t \longrightarrow t'$ we have $\mathsf{R}_{\tau_1\to\tau_2}(t')$. Let t_1 be a term such that $\mathsf{R}_{\tau_1}(t_1)$, we need to show $\mathsf{R}_{\tau_2}(t\ t_1)$. By CR 1 induction hypothesis we know $\mathsf{SN}_n(t_1)$ for some n and we continue by nested induction on n. $t\ t_1$ is neutral, so if we show that all terms that it reduces to are reducible, then we can use CR 1 induction hypothesis to conclude $\mathsf{R}_{\tau_2}(t\ t_1)$. Suppose $t\ t_1 \longrightarrow t_2$:*
case: *$t_2 = t'\ t_1$, with $t \longrightarrow t'$. We know $\mathsf{R}_{\tau_1\to\tau_2}(t')$ and $\mathsf{R}_{\tau_1}(t_1)$, so we have $\mathsf{R}_{\tau_2}(t'\ t_1)$.*
case: *$t_2 = t\ t_1'$ with $t_1 \longrightarrow t_1'$. By CR 2 induction hypothesis $\mathsf{R}_{\tau_1}(t_1')$, and by Lemma 4.2, $\mathsf{SN}_{n'}(t_1')$ for some $n' < n$, so by induction $\mathsf{R}_{\tau_2}(t\ t_1')$.*
These are the only possibilities because t is neutral.

□

The full ATS/LF proof of this is omitted for brevity; it consists of 4 mutually recursive proof functions:

$$
\begin{aligned}
&\mathit{cr1} : \forall t : \mathit{tm}.\forall T : \mathit{tp}.\forall n : \mathit{nat}.\ (\mathit{TP}(T,n), R(t,T)) \to \mathit{SN0}(t)\\
&\mathit{cr3} : \forall t : \mathit{tm}.\forall T : \mathit{tp}.\forall n : \mathit{nat}.\ (\mathit{NEU}(t), \mathit{TP}(T,n), \forall t'.\ \mathit{RED0}(t,t') \to R(t',T)) \to R(t,T)\\
&\mathit{cr3a} : \forall t : \mathit{tm}.\forall t_1 : \mathit{tm}.\forall T_1 : \mathit{tp}.\forall T_2 : \mathit{tp}.\forall m : \mathit{nat}.\forall n_1 : \mathit{nat}.\forall n_2 : \mathit{nat}.\\
&\qquad (\mathit{TP}(T_1,n_1), \mathit{TP}(T_2,n_2), \mathit{NEU}(t), R(t_1,T_1), \mathit{SN}(t_1,m),\\
&\qquad\ \forall t'.\ \mathit{RED0}(t,t') \to R(t', \mathit{TPfun}(T_1,T_2))) \to R(\mathit{TMapp}(t,t1), T2)\\
&\mathit{cr4} : \forall T : \mathit{tp}.\forall n : \mathit{nat}.\ \mathit{TP}(T,n) \to R(\mathit{TMcst}, T)
\end{aligned}
$$

Each of these functions takes arguments of the form $\mathit{TP}(T,n)$ in order to provide a metric that corresponds to structural recursion on T. The auxiliary lemma *cr3a* performs the inner induction on the length of the strong normalization bound of t_1, which is provided by its argument of type $\mathit{SN}(t_1,m)$.

Lemma 4.8 *If for all reducible t at type τ_1, $\mathsf{R}_{\tau_2}(t_1[t/x])$, then $\mathsf{R}_{\tau_1\to\tau_2}(\lambda x.t_1)$.*

Proof. *Assume $\mathsf{R}_{\tau_1}(t)$. By CR 1, we know there is n_1 such that $\mathsf{SN}_{n_1}(t_1[\mathsf{c}/x])$ (and therefore $\mathsf{SN}_{n_1}(t_1)$) and n_2 such that $\mathsf{SN}_{n_2}(t)$. We now proceed by induction on $n_1 + n_2$ to prove that $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t)$. We will show that $(\lambda x.t_1)\ t \longrightarrow t'$ implies $\mathsf{R}_{\tau_2}(t')$ for every t'. There are three possibilities.*

- *$(\lambda x.t_1)\ t$ reduces to $t_1[t/x]$, which is reducible by the hypothesis of the lemma.*
- *$(\lambda x.t_1)\ t$ reduces to $(\lambda x.t_1)\ t'$ with $t \longrightarrow t'$. By CR 2, $\mathsf{R}_{\tau_1}(t')$ and by Lemma 4.2 there is $n' < n$ with $\mathsf{SN}_{n'}(t')$, and thus we have $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t')$ by induction.*
- *$(\lambda x.t_1)\ t$ reduces to $(\lambda x.t_1')\ t$ with $t_1 \longrightarrow t_1'$. By CR 2, $t_1'[t/x]$ is reducible for any reducible t and the strong normalization bound of $(\lambda x.t_1')$ is less than $(\lambda x.t_1)$. So $(\lambda x.t_1')\ t$ is reducible by induction.*

Note that $(\lambda x.t_1)\ t$ *is neutral. By CR 3, we have* $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t)$. *Since* $\mathsf{R}_{\tau_2}((\lambda x.t_1)\ t)$ *holds for every* t *satisfying* $\mathsf{R}_{\tau_1}(t)$, *we have* $\mathsf{R}_{\tau_1\to\tau_2}(\lambda x.t_1)$ *by definition.* □

The formalization of this proof in ATS/LF is a total recursive function with the type:

$$\begin{aligned} \mathit{absSound} : \forall f : tm \to tm.\forall T_1 : tp.\forall T_2 : tp.\\ (\mathit{TP0}(T_1), \mathit{TP0}(T_2), \forall t : tm.R(t, T_1) \to R(f\ t, T_2)) \to\\ R(\mathit{TMlam}\ f, \mathit{TPfun}(T_1, T_2)) \end{aligned}$$

The proof closely follows the informal one given above, taking additional arguments of types $\mathit{TP0}(T_1)$ and $\mathit{TP0}(T_2)$, which are needed in calls to *cr2* and *cr3*. It also makes a call to the proof function *reduceFun* to perform the inner induction on the sum of the normalization bounds ($n_1 + n_2$ in the informal proof).

Now we can prove the main reducibility lemma which states that, given a term t, with a typing $\Gamma \vdash t : T$ and a substitution Θ such that for $x \in \mathrm{dom}(\Gamma)$, $\Theta(x)$ is reducible at type $\Gamma(x)$, then $t[\Theta]$, the result of applying Θ to t, is reducible at type T.

Lemma 4.9 *Let* t *be a term with* $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$. *If* $t_1, \ldots, t_n$ *are terms such that* $\mathsf{R}_{\tau_i}(t_i)$ *(for* $1 \le i \le n$*) then* $\mathsf{R}_\tau(t[t_1/x_1, \ldots, t_n/x_n])$.

Proof. *By induction on the derivation of* $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$. *We write* $t[\underline{t}/\underline{x}]$ *for* $t[t_1/x_1, \ldots, t_n/x_n]$.

$t = x_i$**:** *Then* $t[\underline{t}/\underline{x}] = t_i$ *and* $\tau = \tau_i$ *and by hypothesis* $\mathsf{R}_{\tau_i}(t_i)$.

$t = t'\ t''$**:** *Then, by induction hypothesis,* $\mathsf{R}_{\tau'\to\tau}(t'[\underline{t}/\underline{x}])$ *and* $\mathsf{R}_{\tau'}(t''[\underline{t}/\underline{x}])$. *By the definition of* $\mathsf{R}_\tau((t'[\underline{t}/\underline{x}])\ (t''[\underline{t}/\underline{x}]))$ *and* $(t'[\underline{t}/\underline{x}])\ (t''[\underline{t}/\underline{x}]) = (t'\ t'')[\underline{t}/\underline{x}]$.

$t = \lambda x.t'$**:** *(assume* x *is fresh with respect to* $x_1, \ldots, x_n$ *and* $t_1, \ldots, t_n$*) Then* τ *is of the form* $\tau'' \to \tau'$. *Fix* t'' *such that* $\mathsf{R}_{\tau''}(t'')$. *By induction hypothesis,* $\mathsf{R}_{\tau'}(t'[\underline{t}/\underline{x}, t''/x])$. *By Lemma 4.8,* $\mathsf{R}_{\tau''\to\tau'}(\lambda x.t'[\underline{t}/\underline{x}])$, *and by the freshness of* x, $(\lambda x.t'[\underline{t}/\underline{x}]) = (\lambda x.t')[\underline{t}/\underline{x}]$.

□

When we prove this lemma in ATS/LF, the higher-order encoding buys us quite a bit over a first-order encoding. Because of HOAS, we do not have to think about freshness of variables nor do we have to explicitly prove that the substitution commutes with the lambda binding when handling the lambda case. Lemma 4.9 is encoded in ATS/LF as a total function, which we omit for brevity:

$$\mathit{reduceLemma} : \forall G : ctx.\forall t : tm.\forall T : tp.\forall n : nat.\ (\mathit{DER}(G, t, T, n), \mathit{RS0}(G)) \to R(t, T)$$

Note that $\mathit{RS0}(G)$ is a datatype that associates with each (t_i, T_i) in G, a proof of the reducibility predicate $R(t_i, T_i)$. Also note that we take advantage of the representation of contexts as typed substitutions to state the lemma. It is now a simple matter to prove strong normalization for closed terms using Lemma 4.9 and CR 1.

$$\mathit{normalize} : \forall t : tm.\forall T : tp.\ \mathit{DER0}(\mathit{CTXnil}, t, T) \to \mathit{SN0}(t)$$

It is easy to see that this implies strong normalization for open terms as well, because any reduction on a term with free variables corresponds to a reduction in the closed term formed by abstracting these variables.

5 Strong Normalization for System F

We have also formalized a proof of strong normalization for (the Curry-style version of) System F, which can be found on-line:

`http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/F-SN-hoas.dats`

The terms and reduction rules for the language are the same as for STLC. The types of System F are given by:

$$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau$$

The types are encoded with a first-order representation using de Bruijn indices:

$$\mathit{TPvar} : \mathit{int} \rightarrow \mathit{tp} \qquad \mathit{TPfun} : (\mathit{tp}, \mathit{tp}) \rightarrow \mathit{tp} \qquad \mathit{TPall} : \mathit{tp} \rightarrow \mathit{tp}$$

This representation means that we have to spend a great deal of effort proving lemmas about renumbering and substitution. However, we do not know if it is possible to prove strong normalization using a higher-order representation for types.

We extend the type well-formedness judgment $\vdash \tau\ \mathit{type}$ to include a context: $\Delta \vdash \tau\ \mathit{type}$, and list the new rules as follows:

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \mathit{type}}\ (\mathit{TPvar}) \quad \frac{\Delta \vdash \tau_1\ \mathit{type} \quad \Delta \vdash \tau_2\ \mathit{type}}{\Delta \vdash \tau_1 \rightarrow \tau_2\ \mathit{type}}\ (\mathit{TPfun}) \quad \frac{\Delta, \alpha \vdash \tau\ \mathit{type}}{\Delta \vdash \forall\alpha.\ \tau\ \mathit{type}}\ (\mathit{TPall})$$

Typing judgments are extended to include the extra context and there are also two additional typing rules for handing type abstraction and application:

$$\frac{\Delta, \alpha; \Gamma \vdash t : \tau}{\Delta; \Gamma \vdash t : \forall\alpha.\tau}\ (\mathit{DERtabs}) \qquad \frac{\Delta; \Gamma \vdash t : \forall\alpha.\tau \quad \Delta \vdash \tau_1\ \mathit{type}}{\Delta; \Gamma \vdash t : \tau[\tau_1/\alpha]}\ (\mathit{DERtapp})$$

where *DERtabs* has the side condition that α is not free in Γ.

The approach of directly defining reducibility does not work for System F because we cannot make the argument that the datatype representing reducibility is inductive on the *tp* index. For this reason we need to generalize to reducibility candidates which are all the predicates satisfying CR 1, CR 2 and CR 3. We encode predicates as static terms of sort $\mathit{tm} \rightarrow \mathit{prop}$ (we define $\mathit{rc} \equiv \mathit{tm} \rightarrow \mathit{prop}$ for convenience) and we define propositions:

$$\begin{array}{lcl} \mathit{CR1}(R) & \equiv & \forall t : \mathit{tm}.\ R(t) \rightarrow \mathit{SN0}(t) \\ \mathit{CR2}(R) & \equiv & \forall t : \mathit{tm}. \forall t' : \mathit{tm}.\ (R(t), \mathit{RED0}(t, t')) \rightarrow R(t') \\ \mathit{CR3}(R) & \equiv & \forall t : \mathit{tm}.\ (\mathit{NEU}(t), \forall t' : \mathit{tm}. \mathit{RED0}(t, t') \rightarrow R(t')) \rightarrow R(t) \\ \mathit{RC}(R) & \equiv & (\mathit{CR1}(R), \mathit{CR2}(R), \mathit{CR3}(R)) \end{array}$$

Strong normalization (*SN0*) is defined just as before. It is straightforward to show that *SN0* meets the three conditions:

$$\mathit{sn\_is\_rc} : \mathit{RC}(\mathit{SN0})$$

As a consequence of *CR3*, any reducibility candidate holds for the constant:

$$\mathit{cr\_cst} : \forall R : \mathit{rc}.\ \mathit{RC}(R) \rightarrow R(\mathit{TMcst})$$

The crux of the reducibility candidates is to define interpretations for types as reducibility candidates and to show that whenever a term t can be given a type τ, it is in the reducibility candidate that interprets τ. The fact that a term is strongly normalizing if it is in a reducibility candidate gives us the final result.

In order to interpret types as candidates, we define the arrow and universal quantification constructors for reducibility candidates:

$$RCFUN0(R_1, R_2)(t) \equiv \forall t_1 : tm.\ R_1(t_1) \to R_2(TMapp(t, t_1))$$
$$RCALL0(RF)(t) \equiv \forall R : rc.\ RC(R) \to (RF(R))(t)$$

And we prove that these constructors preserve candidates:

$$rcfun\_is\_rc : \forall R_1 : rc. \forall R_2 : rc.\ (RC(R_1), RC(R_2)) \to RC(RCFUN0(R_1, R_2))$$
$$rcall\_is\_rc : \forall RF : rc \to rc.\ (\forall R : rc. RC(R) \to RC(RF(R))) \to RC(RCALL0(RF))$$

It is an important property that the typing rule for lambda is sound with respect to the arrow on candidates:

$$abs\_lemma : \forall R_1 : rc. \forall R_2 : rc. \forall f : tm \to tm.$$
$$(RC(R_1), RC(R_2), \forall t : tm.\ R_1(t) \to R_2(f\ t)) \to RCFUN0(R_1, R_2)(TMlam f)$$

To provide a context for parameters in reducibility candidates, we define the sort *rcs* for lists of reducibility candidates:

$$RCSnil : rcs \qquad RCScons : (rc, rcs) \to rcs$$

In order to lookup parameters in the list we use a datatype (similar to *INCTX*) with type constructor $RCSI : (rcs, rc, int) \to prop$ and term constructors:

$$RCSIone : \forall R : rc. \forall C : rcs.\ RCSI(RCScons(R, C), R, 0)$$
$$RCSIshi : \forall R : rc. \forall R' : rc. \forall C : rcs. \forall n : nat.$$
$$RCSI(C, R, n) \to RCSI(RCScons(R', C), R, n + 1)$$

We actually use *rcs* to represent Δ in typing derivations, which have type constructor $DER : (rcs, ctx, tm, tp, int) \to prop$. Only the length of the *rcs* term matters in derivations (the actual predicates in the list are not reflected in the dynamic representation), and derivations with an empty Γ and any Δ are adequately encoded. The use of *rcs* in *DER* (rather than simply a natural number bound on the indices) makes some of the lemmas easier to state.

Next, we define the interpretation of types as reducibility candidates with parameters. For this, we use a dependent datatype with type constructor $TPI : (rcs, tp, rc, int) \to prop$, and term constructors:

$$TPIvar : \forall C : rcs. \forall T : tp. \forall R : rc. \forall n : nat.\ RCSI(C, R, n) \to TPI(C, TPvar\ n, R, 0)$$
$$TPIfun : \forall C : rcs. \forall T_1 : tp. \forall T_2 : tp. \forall R_1 : rc. \forall R_2 : rc. \forall n_1 : nat. \forall n_2 : nat.$$
$$(TPI(C, T_1, R_1, n_1), TPI(C, T_2, R_2, n_2)) \to$$
$$TPI(C, TPfun(T_1, T_2), RCFUN0(R_1, R_2), n_1 + n_2 + 1)$$
$$TPIall : \forall C : rcs. \forall T : tp. \forall RF : rc \to rc. \forall n : nat.$$
$$(\forall R : rc. TPI(RCScons(R, C), T, RF(R), n)) \to$$
$$TPI(C, TPall(T), RCALL0(RF), n + 1)$$

For convenience we define $TPI0(C, T, R) \equiv \exists n : nat. TPI(C, T, R, n)$. In order to prove that the interpretation of a type is a reducibility candidate if all the free variables are interpreted by reducibility candidates, we introduce a datatype $RCS : (rcs, int) \to prop$ such that $RCS(C, n)$ is a sequence of proofs of $RC(R)$ for each R in C. We can then prove the desired lemma:

$$tpi\_is\_rc : \forall C : rcs. \forall T : tp. \forall R : rc. \forall n : nat.\ (RCS0\ C, TPI(C, T, R, n)) \to RC(R)$$

where $RCS0(C) \equiv \exists n : nat. RCS(C, n)$.

The last major lemma we need is a substitution lemma on interpretations of types, which we omit for brevity. In order to state the main lemma, we need to define an environment mapping terms to proofs showing that the terms in the appropriate candidates. For this we use the datatype $ETA : (rcs, ctx, int) \rightarrow prop$ where $ETA(C, G, m)$ is a sequence of pairs of $(TPI0(C, T, R), R(t))$ for each (t, T) in G. The main lemma is:

$$\begin{aligned} der\_rc\_lemma : \forall G : ctx.\forall t : tm.\forall T : tp.\forall n : nat.\forall C : rcs.\forall m : nat.\\ (DER(C, G, t, T, n), ETA(C, G, m), RCS0\ C) \rightarrow \\ \exists R : rc.\ (TPI0(C, T, R), R(t)) \end{aligned}$$

The proof of this lemma is quite involved, mostly due to manipulations of de Bruijn indices. The final theorem is then easy to prove:

$$der\_sn : \forall t : tm.\forall T : tp.DER0(RCSnil, CTXnil, t, T) \rightarrow SN0(t)$$

This simply means that every well-typed expression in System F is strongly normalizing.

6 Related Work

There have been several formalizations of proofs of normalization for STLC in the past. Abel [1] encodes a proof of *weak* normalization for STLC in Twelf. As in our proof, the object language is represented using HOAS. However, normalization is proved using an inductive characterization of the weakly normalizing terms, following Joachimski and Matthes [8], rather than Tait's method of reducibility predicates. Sarnat and Schürmann [11] have recently given a proof of weak normalization directly in Twelf using a logical relation. They encode minimal first-order logic which is then used in the definition of the logical relation. It is not clear whether their technique would allow a similar encoding of strong normalization. Berger, Berghofer, Letouzy and Schwichtenberg [3] give proofs of strong normalization for STLC using Tait's method in three systems: Isabelle/HOL, Coq, and Minilog. They also analyze the programs that can be extracted from the formal proofs. However, the formalizations described all make use of first-order representations (using either de Bruijn indices or names for variables) rather than HOAS and also start from a large number of unproven axioms (eleven).

Strong normalization for System F has previously been formalized by Altenkirch [2] using the Lego system. His formalization uses the de Bruijn encoding for both terms and types, and because of this, is significantly longer and more complicated than our proof. Even though our formalization contains full proof terms, rather than tactic-based scripts, it is shorter by about a factor of two.

7 Conclusion

We have presented formalizations of proofs of strong normalization for STLC and System F which use HOAS and Tait's and Girard's methods (respectively). The unique features of ATS/LF (in particular the separation between statics and dynamics) allow for the encoding of powerful logical relations arguments over the simple and elegant language encodings enabled by HOAS. In these proofs we found that HOAS made it much easier to deal with the mundane details of naming and sub-

stitution, which often take the majority of the effort in first-order encoding. [4] As a result, we are able to define the syntax and semantics of STLC and prove strong normalization as described, all in less than 300 lines of commented ATS/LF code! For System F, the proof is likewise short, under 900 lines.

References

[1] Abel, A., *Weak normalization for the simply-typed lambda-calculus in Twelf*, in: *Logical Frameworks and Metalanguages (LFM 04)*, IJCAR, Cork, Ireland, 2004.

[2] Altenkirch, T., *A Formalization of the Strong Normalization Proof for System F in LEGO*, in: M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications* (1993), pp. 13–28.

[3] Berger, U., S. Berghofer, P. Letouzey and H. Schwichtenberg, *Program extraction from normalization proofs*, Studia Logica (2005), special issue, to appear.

[4] Chen, C. and H. Xi, *Combining Programming with Theorem Proving*, in: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, 2005, pp. 66–77.

[5] Donnelly, K. and H. Xi, *Combining higher-order abstract syntax with first-order abstract syntax in ATS*, in: *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (2005), pp. 58–63.

[6] Girard, J.-Y., *Une Extension de l'Interprétation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types*, in: J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics **63** (1971), pp. 63–92.

[7] Girard, J.-Y., Y. Lafont and P. Taylor, "Proofs and Types," Cambridge Tracts in Theoretical Computer Science **7**, Cambridge University Press, Cambridge, England, 1989, xi+176 pp.

[8] Joachimski, F. and R. Matthes, *Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T*, Arch. Math. Log. **42** (2003), pp. 59–87.

[9] Pfenning, F. and C. Elliott, *Higher-order abstract syntax*, in: *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, Atlanta, Georgia, 1988, pp. 199–208.

[10] Pfenning, F. and C. Schürmann, *System description: Twelf - a meta-logical framework for deductive systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)* (1999), pp. 202–206.

[11] Sarnat, J. and C. Schürmann, *On the Representation of Logical Relations*, Yale University Technical Report, YaleU/DCS/TR1362 (2006).

[12] Tait, W. W., *Intensional Interpretations of Functionals of Finite Type I*, Journal of Symbolic Logic **32** (1967), pp. 198–212.

[13] Xi, H., *Dependent Types for Program Termination Verification*, Journal of Higher-Order and Symbolic Computation **15** (2002), pp. 91–132.

[14] Xi, H., *Dependently Typed Pattern Matching*, Journal of Universal Computer Science **9** (2003), pp. 851–872.

[15] Xi, H., *Applied Type System* (2005), available at: `http://www.cs.bu.edu/~hwxi/ATS`.

[4] Actually, we have also formalized a strong normalization proof of STLC that uses FOAS to represent lambda-terms:

`http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/STLC-SN-foas.dats`

There are several unproven lemmas in this formalization, which can certainly be finished but require some effort on handling substitution that is uninspiring and tedious.

A ATS/LF proof of final lemmas and theorem

```
...

// application reducibility lemma
prfun reduceFun
  {f:tm->tm, t:tm, T1:tp, T2:tp, n1:nat, n2:nat} .<n1+n2>.
  (tp1: TP0 T1, tp2: TP0 T2,
   sn1: SN(TMlam f, n1), sn2:SN(t, n2), r1:R(t, T1),
   fr2: {t:tm} R(t, T1) -> R(f t, T2)): R(TMapp(TMlam f, t), T2) = let

   prval r1' = fr2 r1
   prfn fr {t':tm} (red:RED0(TMapp(TMlam f, t), t')) : R(t', T2) = case* red of
     | REDapp1(red') =>
       let
          prval REDlam {f, f',_} fred' = red'
          prfn fr2' {t:tm} (r: R(t, T1)): R(f' t, T2) =
            cr2(tp2, fr2 r, fred'{t})
       in
          reduceFun(tp1, tp2, forwardSN(sn1, red'), sn2, r1, fr2')
       end
     | REDapp2(red') =>
       reduceFun(tp1, tp2, sn1, forwardSN(sn2, red'), cr2(tp1, r1, red'), fr2)
     | REDapp3() => r1'
in
   cr3(NEUapp, tp2, fr)
end

// the abstraction rule is sound with respect to redicible terms
prfn absSound {f:tm->tm, T1:tp, T2:tp}
  (tp1: TP0 T1, tp2: TP0 T2,
   frr : {t:tm} R(t, T1) -> R(f t, T2)) : R(TMlam f, TPfun(T1, T2)) =
  let
     prfn fr {t:tm} (rt: R(t, T1)) : R(TMapp(TMlam f, t), T2) =
       let
          prval snt = cr1(tp1, rt)
          prval snf = lamSN(cr1 (tp2, frr {TMcst} (cr4 tp1)))
       in
          reduceFun (tp1, tp2, snf, snt, rt, frr)
       end
  in
     Rfun(fr)
  end

// pick specified reducibility predicate from the sequence
prfun rGet {t:tm, T:tp, G:ctx, n:nat} .<n>.
  (i:INCTX(t,T,G,n),rs: RS0(G)) : R(t,T) = case* i of
  | INCTXone() => (case* rs of RScons(r,_) => r)
  | INCTXshi i => (case* rs of RScons(_,rs) => rGet(i, rs))

// The assigned type can be extracted from a derivation
prfun der2tp {G:ctx, t:tm, T:tp, n:nat} .<n>. (der: DER(G,t,T,n)): TP0 T =
  case* der of
    | DERvar (_, tp) => tp
    | DERlam (tp1, derf) => let prval tp2 = der2tp derf in TPfun (tp1,tp2) end
    | DERapp (der1, der2) => let prval TPfun (_, tp2) = der2tp der1 in tp2 end

// main lemma
prfun reduceLemma {G:ctx, t:tm, T:tp, n:nat} .<n>.
  (der: DER(G,t,T,n), rs: RS0 G): R (t, T) =
  case* der of
```

```
    | DERvar (i,_) => rGet (i, rs)
    | DERlam {_,f,T1,T2,_} (_, derf) =>
      let
         prval TPfun{T1, T2, s1, s2} (tp1, tp2) = der2tp der
         prfn gr {t:tm} (r: R(t,T1)): R(f t, T2) = let
           prval rs' = RScons (r, rs)
           prval r' = reduceLemma (derf{t}, rs')
         in
           r'
         end
         prfn fr {t:tm} (r: R(t,T1))
           : R(TMapp(TMlam f, t), T2) = let
           prval lamf_red = absSound(tp1, tp2, gr)
           prval Rfun(red_imp) = lamf_red
         in
           red_imp r
         end
      in
         Rfun fr
      end
    | DERapp (der1, der2) =>
      let
         prval r1 = reduceLemma(der1, rs)
         prval Rfun fr = r1
         prval r2 = reduceLemma(der2, rs)
      in
         fr r2
      end

// all typable terms are reducible
prfn reduce {t:tm, T:tp} (der: DER0 (CTXnil,t,T)): R (t,T) =
  reduceLemma(der, RSnil())

// the final theorem
prfn normalize {t:tm, T:tp} (der: DER0 (CTXnil,t,T)): SN0 t =
  cr1(der2tp der, reduce der)
```

Encoding Functional Relations in Scunak

Chad E. Brown[1]

Universität des Saarlandes
Saarbrücken, Germany

Abstract

We describe how a set-theoretic foundation for mathematics can be encoded in the new system Scunak. We then discuss an encoding of the construction of functions as functional relations in untyped set theory. Using the dependent type theory of Scunak, we can define object level application and lambda abstraction operators (in the spirit of higher-order abstract syntax) mediating between functions in the (meta-level) type theory and (object-level) functional relations. The encoding has also been exported to Automath and Twelf.

Keywords: Set Theory, Dependent Type Theory, Proof Irrelevance, Formal Mathematics

1 Introduction

Untyped set theory is often considered a foundation for mathematics because most of the usual mathematical objects of interest can be constructed as sets. For instance, certain sets can be considered pairs, and certain sets of pairs can be considered functions. In textbooks, this construction is described informally, as carrying out such a construction in standard first-order formulations of set theory is tedious. In this paper, we will describe how such a construction can be carried out in a natural, but fully formal, manner by encoding the construction in a dependent type theory. (Of course, such constructions have been formalized before in other systems [7,3,6].)

The construction can be carried out using the type theories implemented in Twelf [8] or Automath [9]. However, we will show how the encoding becomes easier and arguably more natural using the system Scunak. We then export the signature to Twelf and Automath.

There are essentially two reasons why the encoding is natural in Scunak. First, Scunak includes "class types." Second, the concrete syntax for types and terms includes some syntactic sugar for set theory notation.

Type-theoretically, class types are particular instances of Σ-types for pairs of the form $\langle x, p\rangle$ where x is an object and p is a proof of a property of the object.

[1] Email: cebrown@ags.uni-sb.de

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Scunak also includes proof irrelevance, so that the Σ-types behave in some ways as subset types rather than types of pairs. The reason for calling these "class" types is set theoretic. Assuming all mathematical objects are sets (a common assumption in axiomatic set theory), predicates correspond to classes. For each predicate ϕ, the class type for ϕ in Scunak is essentially

$$\{\langle x, p\rangle | x \text{ is an object and } p \text{ is a proof of } \phi(x)\}$$

This set corresponds to the class $\{x|\phi(x)\}$ if there is at most one proof of $\phi(x)$ for each x (i.e., if one has proof irrelevance). Without proof irrelevance, such Σ-types do not correspond to classes since elements in the class may have several representatives in the Σ-type. While class types play an important role in the construction described in this paper, proof irrelevance can be avoided. Consequently, we will for the most part avoid discussing proof irrelevance.

We refer to three systems throughout the paper: Scunak, Twelf, and Automath. Each of these refers to an implemented system which includes, at least, a type checker for some type theory. Twelf [8] includes a checker for the LF type theory [5] (as well as various other important features). Simply referring to "Automath" is ambiguous, since there have been a number of type theories in the Automath family which have been implemented More than once [4]. When we refer to "Automath" as a type theory, we are referring to AUT-68. When we refer to "Automath" as a system, we are referring to Freek Wiedijk's C implementation of a checker for the AUT-68 and AUT-QE type theories [9].

The new system we discuss in this paper is Scunak [2,1]. Scunak includes a type checker for what we will call the "Scunak type theory." Within this type theory, one can specify foundations for mathematics by giving a signature. We will demonstrate this in the paper by describing an axiomatic set theory and a construction of functions as functional relations. Of course, Scunak includes a concrete syntax (the PAM syntax) for specifying types and terms. The PAM syntax provides syntactic sugar for set theoretic constructions. For instance, notation such as `{x:A|x::B}` can be used where $\{x \in A | x \in B\}$ is intended. The parser expands this into a term in the Scunak type theory.

2 Syntax

We begin by briefly describing the Scunak type theory. We use $x, y, z, x^1, \ldots$ to denote variables and $c, d, c^1, \ldots$ to denote constants. For terms, we take untyped λ-terms with constants and pairing. The basic types are as follows:

- `obj` is the type of all mathematical objects. In set theory, objects are sets.
- `prop` is the type of all propositions.
- `pf` P is the type of all proofs of the proposition P.
- `class` ϕ, where ϕ is a property, is the type of pairs $\langle M, N\rangle$ where M is an object and N is a proof that M satisfies the property ϕ.

For types, we take the dependent types generated starting from these basic types. In other words, we have:

Terms	$M, N, P, \phi, \ldots$	$:= x \mid c \mid (\lambda x.M) \mid (M\ N) \mid \langle M, N\rangle \mid \pi_1(M) \mid \pi_2(M)$
Types	$A, B, C, \ldots$	$:= \texttt{obj} \mid \texttt{prop} \mid (\texttt{pf}\ P) \mid (\texttt{class}\ \phi) \mid (\Pi x : A.B)$

We use $A \to B$ to denote $\Pi x : A.B$ when x does not occur free in B.

We use $[M/x]$ to denote substitution of M for x. We assume familiarity with β-reduction and the following pairing reductions:

$(\pi_1) : \pi_1(\langle M, N\rangle) \to_{\pi_1} M \qquad (\pi_2) : \pi_2(\langle M, N\rangle) \to_{\pi_2} N$

When type-checking, we restrict to $\beta\pi_1\pi_2$-normal terms. If such a normal term M is neither of the form $(\lambda x.M_1)$ nor $\langle M_1, M_2\rangle$, we say M is an *extraction*. We use E, F, E^1, E^2, ... to denote extractions. We could optionally include η-reduction and a surjective pairing reduction reducing $\langle \pi_1(M), \pi_2(M)\rangle$ to M, but these are not needed for type checking the signature considered in this paper. (One can enable or disable such reductions in Scunak using flags.)

As usual, Σ denotes a signature $c^1 : A^1, \ldots, c^n : A^n$. Similarly, Γ denotes a context $x^1 : B^1, \ldots, x^m : B^m$. We assume (but do not discuss) validity of signatures and contexts.

In order to account for proof irrelevance, the main judgments in the Scunak type theory are $\Gamma \vdash M \sim N \uparrow A$ (checking normal terms M and N are equal at type A) and $\Gamma \vdash E \sim F \downarrow A$ (extracting a type A at which extractions E and F are equal). Rules for such judgments are given in [1]. Since we will not need proof irrelevance in this paper, we can give a simplified typing judgment and rely on structure equality of normal forms of terms. We let $M^\downarrow$ and $A^\downarrow$ denote the normal form of types and terms, respectively. Since terms are untyped, normal forms do not always exist. In the cases we consider in this paper, normal forms exist. The type judgments we consider here are the following:

- $\Gamma \vdash_\Sigma M \uparrow A$ (Check normal term M has type A.)
- $\Gamma \vdash_\Sigma E \downarrow A$ (Extract type A for extraction E.)
- $\Gamma \vdash_\Sigma A : \mathit{Type}$ (Check A is a valid type.)

The corresponding rules are given in Figures 1 and 2.

It is important to note that this is a simplification of the actual type-checking performed in Scunak. The term $(\lambda P \lambda \phi \lambda u \lambda v \lambda w.w)$ can be checked to inhabit type $(\Pi P : \texttt{prop}.\Pi \phi : (\texttt{pf}\ P \to \texttt{prop}).\Pi u : (\texttt{pf}\ P).\Pi v : (\texttt{pf}\ P).\Pi w : (\texttt{pf}\ (\phi\, u)).\texttt{pf}\ (\phi\, v))$ by making use of proof irrelevance. (In particular, the proofs u and v can be considered the same.) However, the term does not inhabit the type using the simplified form of typing presented here. While semantically proof irrelevance is vital for class types to correspond to classes, in the Scunak signatures considered so far, proof irrelevance has rarely actually been used during type checking. Even when proof irrelevance is used, its use can often be eliminated fairly easily. In the first construction of functions from sets in Scunak, proof irrelevance was used a few times, but these occurrences were eliminated by slightly modifying a few declarations.

Naturally, there are several important meta-theoretic questions one could investigate regarding the Scunak type theory. Is it possible that a type is inhabited by a non-normal term, but inhabited by no normal term? The answer to this question

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \downarrow A} \qquad \frac{c : A \in \Sigma}{\Gamma \vdash c \downarrow A} \qquad \frac{\Gamma \vdash E \downarrow (\Pi x : A.B) \quad \Gamma \vdash M \uparrow A}{\Gamma \vdash (EM) \downarrow ([M/x]B)}$$

$$\frac{\Gamma \vdash E \downarrow \texttt{class}\ \phi}{\Gamma \vdash \pi_2(E) \downarrow \texttt{pf}\ (\phi\, \pi_1(E))} \qquad \frac{\Gamma \vdash E \downarrow \texttt{class}\ \phi}{\Gamma \vdash \pi_1(E) \downarrow \texttt{obj}}$$

$$\frac{\Gamma \vdash E \downarrow B \quad B \in \{\texttt{obj}, \texttt{prop}\}}{\Gamma \vdash E \uparrow B} \qquad \frac{\Gamma \vdash E \downarrow \texttt{pf}\ M \quad M^{\downarrow} = N}{\Gamma \vdash E \uparrow \texttt{pf}\ N}$$

$$\frac{\Gamma, z : A \vdash [z/u]M \uparrow [z/x]B \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash (\lambda u M) \uparrow (\Pi x : A.B)} \qquad \frac{\Gamma, z : A \vdash (Ez) \uparrow [z/x]B \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash E \uparrow (\Pi x : A.B)}$$

$$\frac{\Gamma \vdash_\Sigma M_1 \uparrow \texttt{obj} \quad \Gamma \vdash_\Sigma M_2 \uparrow \texttt{pf}\ (\phi\, M_1)}{\Gamma \vdash_\Sigma \langle M_1, M_2 \rangle \uparrow \texttt{class}\ \phi}$$

$$\frac{\Gamma \vdash_\Sigma \pi_1(E) \uparrow \texttt{obj} \quad \Gamma \vdash_\Sigma \pi_2(E) \uparrow \texttt{pf}\ (\phi\, \pi_1(E))}{\Gamma \vdash_\Sigma E \uparrow \texttt{class}\ \phi}$$

Fig. 1. Rules for Typing Judgments without Proof Irrelevance

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, z : A \vdash [z/x]B : \mathit{Type} \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash (\Pi x : A.B) : \mathit{Type}} \qquad \frac{}{\Gamma \vdash \texttt{obj} : \mathit{Type}}$$

$$\frac{}{\Gamma \vdash \texttt{prop} : \mathit{Type}} \qquad \frac{\Gamma \vdash M \uparrow \texttt{prop}}{\Gamma \vdash \texttt{pf}\ M : \mathit{Type}} \qquad \frac{\Gamma \vdash M \uparrow (\texttt{obj} \to \texttt{prop})}{\Gamma \vdash \texttt{class}\ M : \mathit{Type}}$$

Fig. 2. Simplified Rules for Valid Types

is trivially "no", since only normal terms can be judged to inhabit a type given the algorithmic typing rules in Figure 2. Meta-theoretic issues such as normalization and subject reduction become interesting once one considers a typing judgment for arbitrary terms. One can then consider whether the algorithmic typing judgment is complete with respect to the more general judgment. One can also consider semantics for types and terms. We leave such issues for future work. At the moment the emphasis of the research is on investigating the naturality of encoding formal mathematics in the Scunak type theory.

A Scunak signature can be translated into a Twelf or Automath signature. In both Twelf and Automath, one begins by declaring three basic type families corresponding to `obj`, `prop` and `pf`. When translating to Twelf or Automath, any occurrences of class types are removed by Currying. So long as the Scunak signature can be type-checked using the simplified typing system above (i.e., proof irrelevance is not needed), the resulting Twelf and Automath files should be well-typed. In the signature described below, we have managed to remove all essential uses of proof irrelevance so that the corresponding Twelf and Automath files do type check. (Actually, one must explicitly add `%abbrev` to some Twelf abbreviations by hand, but this is a separate issue.)

3 Specifying a Set Theory

One can give a signature of constants and abbreviations for Scunak in PAM files. The PAM syntax allows a mixture of set theoretic and type theoretic notations. (PAM stands for "pseudo-Automath" since some of the notation is similar to Automath. However, the PAM syntax is also significantly different from Automath.) To demonstrate the PAM syntax, we describe a PAM file for a form of set theory starting from certain axioms and ending with a definition of functions as functional relations. We begin by describing the constants in the signature which correspond to the axiomatic kernel of the set theory. Similar encodings of a variety of foundational systems for mathematics in Automath are discussed in [10].

Throughout a PAM file, one can specify local parameters. For example,

```
[M:prop][N:prop][y:obj][z:obj][A:set][B:set][C:set]
```

Intuitively, this declaration of parameters means: "Let M and N be propositions, y and z be objects, and A, B and C be sets." (Note that `obj` and `set` are synonyms, standing for the same basic type `obj`.)

The declaration

```
(not M):prop.
```

introduces a new constant `not` of type `prop` $\rightarrow$ `prop` into the signature. (Note the use of the parameter M of type `prop` as an argument.)

In order to obtain classical logic, we can declare an excluded middle proof by cases rule as follows:

```
[case1:|- M -> |- N]
[case2:|- (not M) -> |- N]
(xmcases M N case1 case2):|- N.
```

The parameters `case1` and `case2` correspond to the two premises of the rule. Note that `|- N` is the PAM syntax for the type (`pf` N). The type of `xmcases` is

$$\Pi M : \texttt{prop}.\Pi N : \texttt{prop}.(\texttt{pf}\ M \rightarrow \texttt{pf}\ N) \rightarrow (\texttt{pf}\ (\texttt{not}\, M) \rightarrow \texttt{pf}\ N) \rightarrow \texttt{pf}\ N$$

We also declare the usual elimination rule for negation.

```
(notE M N):|- M -> |- (not M) -> |- N.
```

The usual introduction rule for negation, as well as the proof by contradiction rule, can be derived using `xmcases` and `notE`.

Negation and the two rules above translate into the following Twelf code

```
not : prop -> prop.
xmcases : {M:prop} {N:prop} (pf M -> pf N)
                     -> (pf (not M) -> pf N) -> pf N.
notE : {M:prop} {N:prop} pf M -> pf (not M) -> pf N.
```

as well as corresponding Automath code. Since class types have not yet been used, the Scunak, Twelf, and Automath versions are very similar.

One may expect to see more propositional connectives (such as conjunction or implication) in the signature. However, once we include the set theory constructors

and axioms, we can actually define these connectives. We will show such definitions later.

The basic relations in set theory are equality and membership.

```
(eq y z):prop.
(in A z):prop.
```

In PAM syntax, one can write `(y==z)` for `(eq y z)` and `(z::A)` for `(in A z)`. Note that `(in A z)` intuitively represents the proposition $z \in A$. The reason the arguments are reversed is so that the η-short form `(in A)`, an extraction of type `obj` $\rightarrow$ `prop`, represents the "class" of all members of `A`.

An equality elimination rule corresponding to replacing equals by equals is included in the signature. We omit this here.

The rule for set extensionality is declared as follows.

```
[AsubB:{x:obj}{u:|- (x::A)}|- (x::B)]
[BsubA:{x:obj}{u:|- (x::B)}|- (x::A)]
(setext A B AsubB BsubA):|- (A==B).
```

The type of the parameter `BsubA` is $\Pi x : \texttt{obj}.\Pi u : \texttt{pf}\ (\texttt{in}\ B\,x).\texttt{pf}\ (\texttt{in}\ A\,x)$. Intuitively, this corresponds to a premise stating that every element of B is an element of A. That is, B is a subset of A. However, note that this represents the assertion that B is a subset of A at the *type* level, not at the level of propositions. We will reuse the parameter `BsubA` when declaring the rules for powerset.

At this point, we can begin describing the basic set constructors and the rules (or axioms) corresponding to each such constructor.

There is an empty set. We encode this axiom simply by declaring a constant `emptyset` of type `obj`.

```
emptyset:obj.
```

In PAM syntax, one can write `{}` for `emptyset`. If some y is in the empty set, then every proposition M holds.

```
[yinempty:|- (y::{})]
(emptysetE y yinempty M):|- M.
```

We can adjoin y to the set A to obtain the set $y; A$ (or, $\{y\} \cup A$).

```
(setadjoin y A):obj.
```

In PAM syntax, `(y;A)` represents `(setadjoin y A)`. There is special PAM syntax for finite enumerated sets which expands into `emptyset` and `setadjoin`. One can use `{x1,...,xn}` (intuitively, the finite set $\{x_1, \ldots, x_n\}$) to represent the term `(setadjoin x1 ... (setadjoin xn emptyset))`. In particular, `{y}` and `{y,z}` correspond to the terms `(setadjoin y emptyset)` and `(setadjoin y (setadjoin z emptyset))`, respectively. We omit the three rules for introducing and eliminating `setadjoin`.

The power set of A is a set. There are two rules for introducing and eliminating the powerset. (Note the reuse of the parameter `BsubA` declared above.)

```
(powerset A):obj.
(powersetI A B BsubA):|- (B::(powerset A)).
```

```
(powersetE A B z):|- (B::(powerset A)) -> |- (z::B) -> |- (z::A).
```

The union of A (intuitively, $\bigcup A$) is a set. There are two corresponding rules, omitted here.

```
(setunion A):obj.
```

Finally, we come to the most interesting axiom: separation. We can state this as follows. For any property $\psi(x)$ of elements $x \in A$, there is a set $\{x \in A | \psi(x)\}$.

```
[psi:A -> prop]
(dsetconstr A psi):obj.
```

We have declared the parameter `psi` to have type `A -> prop`. However, technically, `A` is a term, not a type. In PAM syntax one is allowed to use an extraction as a type, so long as the extraction has type `obj` or `obj` $\rightarrow$ `prop`. In this case, `A` has type `obj`. So, Scunak assumes the intention is for `A` to be the class type `class (in` A`)`.[2] Technically, the type of `psi` is (`class (in` A`)`) $\rightarrow$ `prop` and the type of `dsetconstr` is ΠA : `obj`.((`class (in` A`)`) $\rightarrow$ `prop`) $\rightarrow$ `obj`.

In PAM syntax, we write `{x:A|M}` for `(dsetconstr A (\x.M))`, where a backslash is PAM syntax for a λ binder.

Note that `dsetconstr` makes explicit use of a class type. Consequently, in the translations to Twelf and Automath, ψ becomes a function of two arguments: an object x_1 and a proof x_2 that x_1 is in A. In Twelf, we have

```
dsetconstr : {A:obj} ({x1:obj} pf (in A x1) -> prop) -> obj.
```

We omit the proof rules for `dsetconstr`.

It is important that in the set construction above, $\psi(x)$ can make use of the fact that $x \in A$ (as opposed to x being simply an object). This allows one to specify sets such as $\{x \in (\Re \setminus \{0\}) | \frac{x^2-1}{x} = 0\}$ where one must know $x \neq 0$ in order to construct the term representing $\frac{x^2-1}{x}$.

These axioms are sufficient to describe all hereditary finite sets. If one adds an axiom of infinity, one essentially obtains a form of Mac Lane set theory (Zermelo set theory with bounded quantifiers).

4 From Set Theory Axioms to Binary Relations

Starting from the axioms of set theory described above, one can define the usual propositional connectives as well as bounded quantification. Also, one can construct pairs and define binary relations as certain sets of pairs. This provides the infrastructure for defining functions (at the object level). We describe this infrastructure below.

First, we can define `true` and `false` as $\emptyset \in \{\emptyset\}$ and $\emptyset \in \emptyset$, respectively.

```
true:prop=({}::{{}}).
false:prop=({}::{}).
```

The important properties of `true` and `false` hold. Namely, there are terms inhabiting `pf true` and ΠP : `prop`.`pf false` $\rightarrow$ `pf` P.

[2] This is a concrete example justifying reversing the usual order of arguments of `in`.

For any proposition M, $\{x \in \{\emptyset\} | M\}$ is $\{\emptyset\}$ if M is true and $\emptyset$ if M is false. Using this set, we can embed the type of propositions into the type of objects.

```
(prop2set M):obj={x:{{}}|M}.
```

Using `prop2set`, we can define disjunction, implication and conjunction. The types corresponding to the usual natural deduction rules for these connectives are inhabited.

```
(or M N):prop=({{}}::{prop2set M,prop2set N}).
(imp M N):prop=((not M) | N).
(and M N):prop=(not (M => (not N))).
```

In PAM syntax, we can write `(M | N)`, `(M => N)`, and `(M & N)` for `(or M N)`, `(imp M N)`, and `(and M N)`, respectively.

If A is a set and $\psi(x)$ is a property of elements of A, then $\{x \in A | \psi(x)\} = A$ iff $\psi(x)$ holds for all $x \in A$. Similarly, $\{x \in A | \psi(x)\} \neq \emptyset$ iff $\psi(x)$ holds for some $x \in A$. We use these facts to define bounded quantifiers.

```
(dall A psi):prop=({x:A|psi x}==A).
(dex A psi):prop=(not ({x:A|psi x}=={})).
```

In PAM syntax, we write `(forall x:A . M)` and `(exists x:A . M)` as syntactic sugar for `(dall A (\x.M))` and `(dex A (\x.M))`, respectively.

In fact, `dall` and `dex` are bounded, *dependent* quantifiers. We can use the fact that x is in the set A in the construction of the proposition $x \in A$. Thus, we can sensibly represent a proposition such as $\exists x \in (\Re \setminus \{0\}).\frac{x^2-1}{x} = 0$.

Using bounded quantification, we can define subset.

```
(subset A B):prop=(forall x:A . (x::B)).
```

In PAM syntax, we can write `(A <= B)` for `(subset A B)`.

Binary union $A \cup B$ is defined as $\bigcup\{A, B\}$.

```
(binunion A B)=(setunion {A,B}).
```

In PAM syntax, we can write `(A \cup B)` for `(binunion A B)`.

A set A is a singleton if there is some x such that $A = \{x\}$. Since we only have bounded quantification, we must give a set in which that x must live. That is, we do not have a term corresponding to the proposition $\exists x.(A = \{x\})$. Instead we must use an appropriate set B and represent the proposition as $\exists x \in B.(A = \{x\})$. In this case, an appropriate choice of B is obvious: A.

```
(singleton A):prop=(exists x:A . (A=={x})).
```

Since `singleton` has type `obj` → `prop`, `class singleton` is a valid class type. In the PAM syntax, we can simply use the extraction `singleton` as a type:

```
[S:singleton]
```

Note that if S be a member of this class, then $\pi_1(S)$ has type `obj` and $\pi_2(S)$ has type `pf` (`singleton` $\pi_1(S)$). In PAM syntax, one never explicitly writes π_1 and π_2 operators. If `S` is used where a term of type `obj` is expected, Scunak reconstructs the term π_1(`S`). If `S` is used where a term of type `pf` (`singleton` π_1(`S`)) is expected, Scunak reconstructs π_2(`S`). In particular, we write the proposition

$(\bigcup S) \in S$ as `((setunion S)::S)` in PAM syntax. The reconstructed term is (`in` π_1(`S`) (`setunion` π_1(`S`))). We can declare a *claim* (i.e., a signature element for which a definition *will* be declared) called `theprop` of this proof type.

```
(theprop S):|- ((setunion S)::S)?
```

There is a term inhabiting this type, which we omit here. Once one gives the term as the definition (i.e., proof) of `theprop`, then `theprop` is an abbreviation and no longer a claim.

Using `theprop`, we can define a dependently typed description operator `the` as follows:

```
(the S):(in S)=<(setunion S),theprop S>.
```

Once the type and term are reconstructed, `the` has type

$$\Pi S : (\texttt{class singleton}).\texttt{class } (\texttt{in } \pi_1(S))$$

and is defined by the term ($\lambda S.\langle$(`setunion` π_1(`S`)), (`theprop S`)$\rangle$). With the typing rules in Figure 1 and the given types of `setunion` and `theprop`, one can easily verify that the term indeed inhabits the type. Intuitively, given a singleton set S, (`the` S) is the unique member of S.

We can define a quantifier for unique existence using the singleton predicate..

```
(ex1 A psi):prop=(singleton {x:A|psi x}).
```

In PAM syntax, we write `(exists1 x:A . M)` for `(ex1 A (\x.M))`.

A set A is a Kuratowski pair if there exist u and v such that $A = \{\{u\}, \{u, v\}\}$. To define this notion using bounded quantification, we make use of $\bigcup A$ as a bound. One can prove that if any such u and v exist, they must inhabit $\bigcup A$.

```
(iskpair A):prop=(exists u:(setunion A) .
                  (exists v:(setunion A) . (A=={{u},{u,v}}))).
```

Given any objects y and z, $\{\{y\}, \{y, z\}\}$ is a Kuratowski pair. We can prove this and form an abbreviation `kpairiskpair`. Using such an abbreviation, we can define an operation `kpair` which takes two objects y and z and returns a member of the class type of Kuratowski pairs.

```
(kpair y z):iskpair=<{{y},{y,z}},kpairiskpair y z>.
```

In PAM syntax, we write `<<y,z>>` for the Kuratowski pair of `y` and `z`.

Using Kuratowski pairs, we can define the Cartesian product $A \times B$ of two sets A and B as follows:

```
(cartprod A B):obj
 ={x:powerset (powerset (A \cup B))|
    (exists u:A . (exists v:B . (x==<<u,v>>)))}.
```

In PAM syntax we write `(A \times B)` for `(cartprod A B)`.

We have already used the notation `{x:A|psi x}` for denoting $\{x \in A|\psi(x)\}$ in PAM syntax. When working with functions, we will need to consider sets of pairs. Informally, we can write $\{(u, v) \in A \times B|\phi(u, v)\}$. In order to support a corresponding PAM notation, we define a dependent set of pairs constructor.

```
[phi:A -> B -> prop]
(dpsetconstr A B phi):obj
 ={x:(A \times B)|
    (exists u:A . (exists v:B . ((phi u v) & (x==<<u,v>>))))}.
```

In PAM syntax, we write `{<<u,v>>:A \times B|M}` as syntactic sugar equivalent to `(dpsetconstr A B (\u v.M))`. (A single backslash in PAM notation binds a list of variables.)

Finally, we define the notion of a binary relation on two sets A and B in the usual way.

```
[R:obj]
(breln A B R):prop=(R<=(A \times B)).
```

This gives all the infrastructure necessary to define set-theoretic functions.

5 Representing Functions as Objects

Let A, B, and R be sets. We say R is a function from A to B if R is a binary relation on A and B and forall $x \in A$ there is a unique $y \in B$ such that the pair of x and y is in R. In PAM syntax, we can make this abbreviation as follows.

```
[A:set][B:set][R:obj]
(func A B R):prop
 =((breln A B R)&(forall x:A . (exists1 y:B . (<<x,y>>::R)))).
```

As before, Scunak reconstructs the π_1 operations. Note that `(<<x,y>>::R)` is PAM syntax for the term (`in R` π_1(`kpair` π_1(`x`) π_1(`y`))).

Since `(func A B)` has type `obj` $\rightarrow$ `prop`, `class (func A B)` is a valid class type. Let f have this type and let x have type A.

```
[f:(func A B)]
[x:A]
```

Using the definition of `func`, we can prove the set represented in PAM notation as `{y:B|<<x,y>>::f}` is a singleton. In the signature, `funcImageSingleton` is an abbreviation corresponding to this fact. Hence, the pair (in PAM syntax)

```
<{y:B|<<x,y>>::f},(funcImageSingleton A B f x)>
```

is of class type `class singleton`. Applying the description operator `the`, we obtain a member of `{y:B|<<x,y>>::f}`. One can prove the first component of `(the <{y:B|<<x,y>>::f},(funcImageSingleton A B f x)>)` is in `B`. In the signature, `apProp` abbreviates such a proof. Given this information, we can define an object level application as follows:

```
(ap A B f x):B
 =<(the <{y:B|<<x,y>>::f},(funcImageSingleton A B f x)>),
   (apProp A B f x)>.
```

The type of `ap` is

$$\Pi A : \texttt{obj}.\, \Pi B : \texttt{obj}.\, (\texttt{class}\ \ (\texttt{func}\, A\, B)) \rightarrow (\texttt{class}\ \ (\texttt{in}\ \ A)) \rightarrow \texttt{class}\ \ (\texttt{in}\ \ B)$$

It is perhaps instructive to compare this to the Twelf version of `ap` obtain by translating from Scunak. Since `ap` returns a member of the class type `class (in B)`, there are two corresponding Twelf abbreviations. (Due to a use of `%abbrev`, the description operator `the` is expanded in terms of `setunion` in the Twelf version.)

```
%abbrev
ap : {A:obj} {B:obj} {f:obj}
            pf (func A B f) -> ({x:obj} pf (in A x) -> obj)
   = [A:obj] [B:obj] [f:obj] [fp:pf (func A B f)]
     [x:obj] [xp:pf (in A x)]
 (setunion
  (dsetconstr B ([y:obj] [yp:pf (in B y)] in f (kpair x y)))).

ap_pf :
   {A:obj} {B:obj} {f:obj} pf (func A B f)
    -> ({x:obj} pf (in A x)
        -> pf (in B (setunion (dsetconstr B
                                ([y:obj] [yp:pf (in B y)]
                                 in f (kpair x y))))))
   = [A:obj] [B:obj] [f:obj] [fp:pf (func A B f)]
     [x:obj] [xp:pf (in A x)] apProp A B f fp x xp.
```

Note that in Twelf, `ap` is a function of six arguments instead of four. In particular, the object `f` is separated from the proof `fp` that `f` is a function from `A` to `B`. Likewise, the object `x` is separated from the proof `xp` that `x` is a member of `A`. Intuitively, the Twelf abbreviation `ap` returns the object corresponding to $f(x)$ and the Twelf abbreviation `ap_pf` returns the proof that $f(x)$ is in B.

Similarly, we can define an object-level λ-abstraction operator. Intuitively, this reifies a meta-level function g from A to B to be an object-level function from A to B. Let g have type (`class (in A)`) $\rightarrow$ (`class (in B)`). In PAM syntax, we write `[g:A -> B]`. We can prove the set of pairs represented in PAM syntax by `{<<x,y>>:A \times B|((g x)==y)}` is a function from A to B. We abbreviate this proof as `lamProp`. Using this, we can define the abstraction operator `lam` as follows.

```
(lam A B g):(func A B)
 =<{<<x,y>>:A \times B|((g x)==y)},(lamProp A B g)>.
```

The type of `lam` is

$$\Pi A : \texttt{obj}.\, \Pi B : \texttt{obj}.(\texttt{class}\ \ (\texttt{in}\, A) \rightarrow \texttt{class}\ \ (\texttt{in}\, B)) \rightarrow \texttt{class}\ \ (\texttt{func}\, A\, B)$$

Note that the types of `ap` and `lam` have the form one expects when coding simply typed λ-calculus using higher-order abstract syntax. In particular, `ap` takes an object-level function in `class (func A B)` to a meta-level function `class A` $\rightarrow$ `class B` and `lam` takes such a meta-level function to such an object-level function. However, the intention is quite different. We are not encoding *syntax* of simply typed λ-terms, but the standard set theoretic *semantics* of simply typed

λ-terms. Consequently, we can prove properties which hold in such standard models. For example, we can prove functional extensionality and soundness of β- and η-conversion.

Functional extensionality states that two functions $f, k : A \to B$ are equal if they return the same value on all $x \in A$.q We can declare functional extensionality as a claim `funcext` in PAM syntax.

```
[k:(func A B)]
[eqfkx:{x:A}|- ((ap A B f x)==(ap A B k x))]
(funcext A B f k eqfkx):|- (f==k)?
```

In the PAM file, the proof (i.e., definition) is given following the declaration of the claim. We omit this proof term here.

Finally, we can prove the object-level versions of β-equality and η-equality. We omit the proof terms and only show the declarations of the claims.

```
(beta1 A B g x):|- ((ap A B (lam A B g) x)==(g x))?
(eta1 A B f):|- ((lam A B (ap A B f))==f)?
```

6 Comparing the Signatures

The construction of functions starting from the given axioms of set theory can be encoded in Scunak by giving a signature of 23 constants and 112 abbreviations. By Currying, one obtains corresponding Twelf and Automath signatures. Each of these signatures contains 3 declarations for the type families, 23 constants and 116 abbreviations. In particular, 4 of the Scunak abbreviations (`the`, `kpair`, `ap`, and `lam`) return a class type and therefore correspond to 8 abbreviations in Twelf and Automath. In Twelf, 11 abbreviations must be declared using `%abbrev` since there is no strict occurrence of some argument. Each of the three systems can type check the signature in less than a second.

7 Conclusion

Scunak provides a convenient way to specify a set theory and represent mathematics within the set theory. Two of the reasons for the naturality of mathematics represented in Scunak are class types and the PAM syntax. Class types allow one to treat arbitrary predicates (set-theoretic classes) as subtypes of the type of (untyped) mathematical objects. The PAM syntax allows one to give types and terms in a reasonably natural way.

References

[1] Brown, C. E., *Combining Type Theory and Untyped Set Theory*, in: *IJCAR 2006*, Seattle, Washington, 2006, to appear.

[2] Brown, C. E., *Verifying and Invalidating Textbook Proofs using Scunak*, in: *Mathematical Knowledge Management, MKM 2006*, Wokingham, England, 2006, to appear.

[3] Byliński, C., *Functions and their basic properties*, Journal of Formalized Mathematics **1** (1989), `http://mizar.org/JFM/Vol1/funct_1.html`.

[4] de Bruijn, N. G., *A survey of the project AUTOMATH*, in: J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980 pp. 579–606.

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

[6] Megill, N., *Metamath Home Page*, `http://au.metamath.org/index.html`.

[7] Paulson, L. C., *Set Theory for Verification: I. From Foundations to Functions*, Journal of Automated Reasoning **11** (1993), pp. 353–389.

[8] Pfenning, F. and C. Schürmann, *System Description: Twelf–A Meta-Logical Framework for Deductive Systems*, in: H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence **1632** (1999), pp. 202–206.

[9] Wiedijk, F., *A new implementation of Automath*, J. Autom. Reasoning **29** (2002), pp. 365–387.

[10] Wiedijk, F., *Is ZF a hack? Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics*, Journal of Applied Logic **4** (2006), to appear.

Synthesis of moduli of uniform continuity by the Monotone Dialectica Interpretation in the proof-system MinLog

Mircea-Dan Hernest[1,2]

Laboratoire d'Informatique (LIX)
École Polytechnique
F-91128 Palaiseau - FRANCE

Abstract

We extract on the computer a number of moduli of uniform continuity for the first few elements of a sequence of closed terms t of Gödel's $\mathbf{T}$ of type $(\mathbb{N}\to\mathbb{N})\to(\mathbb{N}\to\mathbb{N})$. The generic solution may then be quickly inferred by the human. The automated synthesis of such moduli proceeds from a proof of the hereditarily extensional equality ($\approx$) of t to itself, hence a proof in a weakly extensional variant of Berger-Buch-holz-Schwichtenberg's system Z of $t \approx_{(\mathbb{N}\to\mathbb{N})\to(\mathbb{N}\to\mathbb{N})} t$. We use an implementation on the machine, in Schwichtenberg's `MinLog` proof-system, of a non-literal adaptation to Natural Deduction of Kohlenbach's monotone functional interpretation. This new version of the Monotone Dialectica produces terms in `NbE`-normal form by means of a recurrent partial `NbE`-normalization. Such partial evaluation is strictly necessary.

Keywords: Program extraction from (classical) proofs, Complexity of extracted programs, Monotone Dialectica Interpretation, Proof- and program-extraction system `MinLog`, Gödel's functional interpretation, Proof Mining, Partial Evaluation.

1 The *monotone* functional *Dialectica* interpretation

Kohlenbach's monotone variant of Gödel's functional (aka "Dialectica") interpretation was introduced in [18] as an optimization of Gödel's original term extraction technique[3] from [8]. The main feature of this "monotone Dialectica interpretation" is the extraction of Howard majorants [14] (or, equally, Bezem strong majorants [6])[4] for some exact realizers[5]. In the mathematical practice this operation turns out to be much simpler[6] than the synthesis of some actual exact realizers by the pure Gödel's Dialectica interpretation from [8,1].

[1] *Project LogiCal* - Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA et Université Paris-Sud - `FRANCE`

[2] Email: `danher@lix.polytechnique.fr`

[3] Paper [1] provides a nice survey in English which includes the extensions to full Analysis.

[4] In this paper we actively use only Howard's variant of majorization, originally introduced in [14] (see also [20,21]), which is presented in Definition 1.3 below.

[5] Which are not effectively produced, but their *strong* existence is ensured intuitionistically.

[6] See, e.g., [21] and [22] for two comprehensive surveys of the wide range of mathematical application of this purely proof-theoretical technique.

This paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

Definition 1.1 [Base Arithmetic for Monotone Dialectica program-extraction] We denote by $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$ the weakly extensional variant (see [10]) of Berger-Buchholz-Schwichtenberg's system Z (introduced in [2], see also [24]) to which the *strong* $\exists$ quantifier was added (together with its defining axioms, see [10,24]) and also all the necessary *monotonic* elements were added, namely the functional inequality constant $\geq$ together with the axioms governing its usual behaviour [7].

Note that the system $\mathtt{WeZ}^{\exists}$, i.e., $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$ without the monotonic elements (which was denoted $\mathtt{WE{-}Z}^{-}$ in [10]) is a Natural Deduction formulation of the weakly extensional Heyting Arithmetic in all finite types $\mathtt{WE{-}HA}^{\omega}$ from Section 1.6.12 of [26].

Definition 1.2 [Extended Arithmetic for extraction by Monotone Dialectica] We denote by $\mathtt{WeZ}^{\exists+}_{\mathtt{m}}$ the extension of $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$ with the Axiom of Choice, the Independence of Premises for universal premises and Markov's Principle (axiom) [8].

Definition 1.3 [Section 2 of [14], adapted to the **T** presentation from [10] [9]] *Howard's majorizability relation* $\succeq$ is defined over the **T** type structure by

$$\begin{aligned} x \succeq_{\mathbb{N}} y &:\equiv \mathtt{at}(\geq x\, y) \\ x \succeq_{\rho\tau} y &:\equiv \forall z_1^{\rho}, z_2^{\rho}\,(z_1 \succeq_{\rho} z_2 \to x z_1 \succeq_{\tau} y z_2) \quad , \end{aligned}$$

where $\geq$ is the usual inequality boolean function on $\mathbb{N} \times \mathbb{N}$ defined in [10] and "$\mathtt{at}$" is the boolean, unary and unique predicate of $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$, also defined in [10].

The monotone Dialectica interpretation (abbreviated "MD-interpretation" and even shorter, MDI) is a recursive syntactic translation from proofs in $\mathtt{WeZ}^{\exists+}_{\mathtt{m}}$ [10] to proofs in $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$ such that the positive occurrences of the strong $\exists$ and the negative occurrences of $\forall$ in the proof's conclusion formula get effectively (either Howard or Bezem) majorized at each of the proof-recursion steps [11] by terms in Gödel's **T**. These *majorizing terms* are also called "the programs extracted by" the MDI and (if only the extracted terms are wanted) this translation process is also referred to as "Monotone Dialectica program-extraction".

Definition 1.4 [*Association of boolean terms to quantifier-free formulas*] By quantifier-free formula we understand a formula built from prime formulas $\mathtt{at}(t^{bool})$ and $\bot$ by means of $\wedge$, $\to$ and, if $\exists$ is available, also $\vee$. Such formulas are decidable in $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$. There exists a unique bijective association of boolean terms to quantifier-free formulas $A_0 \mapsto \mathtt{t}_{A_0}$ such that $\mathtt{WeZ}^{\exists}_{\mathtt{m}} \vdash A_0 \leftrightarrow \mathtt{at}(\mathtt{t}_{A_0})$.

The MD-interpretation of proofs includes the following translation of formulas:

---

[7] See Section 3.1 of [10] for details - our system $\mathtt{WeZ}^{\exists}_{\mathtt{m}}$ here was there denoted by $\mathtt{WE{-}Z}^{-}_{\mathtt{m}}$.

[8] See, e.g., Section 2.3 of [10] for the detailed definitions of these axioms (plus comments).

[9] Please beware of the typo in the corresponding definition from Section 3.1 of [10].

[10] This can be extended to fully classical proofs, modulo some double-negation translation.

[11] This is exactly *the point* of Kohlenbach's MD-interpretation from [18], in contrast to his precursor of the MDI from [16] which first extracts the effective Gödel's Dialectica *exact realizers* and subsequently majorizes them via the algorithms of either Howard [14] or Bezem [6].

Definition 1.5 [**The MD-interpretation of formulas**] Recursively defined:

$$
\begin{aligned}
A^{\mathtt{MD}} &:\equiv A_{\mathtt{MD}} :\equiv \mathtt{at}(\mathtt{t}_A) \text{ for quantifier-free formulas } A \\
(A \wedge B)^{\mathtt{MD}} &:\equiv \exists \underline{x},\underline{u}\,\forall \underline{y},\underline{v}\,[\,(A\wedge B)_{\mathtt{MD}} :\equiv A_{\mathtt{MD}}(\underline{x};\underline{y};\underline{a}) \wedge B_{\mathtt{MD}}(\underline{u};\underline{v};\underline{b})\,] \\
(\exists z A(z,\underline{a}))^{\mathtt{MD}} &:\equiv \exists z^\dagger,\underline{x}\,\forall \underline{y}\,[\,(\exists z A(z,\underline{a}))_{\mathtt{MD}}(z^\dagger,\underline{x};\underline{y};\underline{a}) :\equiv A_{\mathtt{MD}}(\underline{x};\underline{y};z^\dagger,\underline{a})\,] \\
(\forall z A(z,\underline{a}))^{\mathtt{MD}} &:\equiv \exists \underline{X}\,\forall z^\dagger,\underline{y}\,[\,(\forall z A(z,\underline{a}))_{\mathtt{MD}}(\underline{X};z^\dagger,\underline{y};\underline{a}) :\equiv A_{\mathtt{MD}}(\underline{X}(z^\dagger);\underline{y};z^\dagger,\underline{a})\,] \\
(A \to B)^{\mathtt{MD}} &:\equiv \exists \underline{Y},\underline{U}\,\forall \underline{x},\underline{v}\,[\,(A\to B)_{\mathtt{MD}} :\equiv A_{\mathtt{MD}}(\underline{x};\underline{Y}(\underline{x},\underline{v})) \to B_{\mathtt{MD}}(\underline{U}(\underline{x});\underline{v})\,]
\end{aligned}
$$

where $\cdot \mapsto \cdot^\dagger$ is a mapping which assigns to every given variable z a completely new variable $z^\dagger$ which has the same type of z. The free variables of $A^{\mathtt{MD}}$ are exactly the free variables of A.

Theorem 1.6 (Majorant realizer synthesis by the MD-interpretation) [12] There exists an algorithm which, given at input a Natural Deduction proof $\mathcal{P} : \{C^i(a_i)\}_{i=1}^n \vdash A(a)$ [hence of the conclusion formula A, whose free variables form the *tuple* a, from the *undischarged* assumption formulas $\{C^i\}_{i=1}^n$] in $\mathtt{WeZ}_{\mathtt{m}}^{\exists+}$, it produces at output the following (below let $\underline{a} :\equiv a_1,\ldots,a_n,a$):

(i) the tuples of terms $\{T_i[\underline{a}]\}_{i=1}^n$ and $T[\underline{a}]$, whose free variables are among $\underline{a}$

(ii) the tuples of variables $\{x_i\}_{i=1}^n$ and y, all together with

(iii) the following verifying proof in $\mathtt{WeZ}_{\mathtt{m}}^{\exists}$ (below let $\underline{x} :\equiv x_1,\ldots,x_n$):

$$
\begin{aligned}
\mathcal{P}_{\mathtt{MD}} : \emptyset \vdash \exists Y_1,\ldots Y_n, X\,[\,&\textstyle\bigwedge_{i=1}^n (\lambda \underline{a}.\,T_i) \succeq Y_i \wedge (\lambda \underline{a}.\,T) \succeq X \wedge \\
&\forall \underline{a},\underline{x},y\,(\,\{\textstyle\bigwedge_{i=1}^n C^i_{\mathtt{MD}}(x_i;Y_i(\underline{a},\underline{x},y);a_i)\} \to A_{\mathtt{MD}}(X(\underline{a},\underline{x});y;a)\,)\,]
\end{aligned}
$$

Moreover, variables $\underline{x}$ and y do not occur in $\mathcal{P}$ (they are all completely new). Hence $\underline{x}$ and y also do not occur free in the *extracted* terms $\{T_i\}_{i=1}^n$ and T.

Proof: See [11] for a sketch of the proof (in Natural Deduction) or [18,21] for full proofs of the equivalent original formulations in the Hilbert-style setting. □

Remark 1.7 The MD-translated proof $\mathcal{P}_{\mathtt{MD}}$ is also called the *verifying proof* since it arithmetically verifies the fact that the MD-extracted programs actually *majorize* some (strong, intuitionistically proven to exist) realizers of the MD-interpretation of the conclusion formula of the proof at input.

Gödel's Dialectica interpretation becomes far more complicated when it has to face Contraction, which in Natural Deduction amounts to the discharging of more than one copy of an un-cancelled assumption in an Implication Intro- duction $\dfrac{[A]\ldots/B}{A\to B}$. This is because, for the contractions which are relevant

[12] This theorem was conjectured (in a weaker form) already in Section 3.1 of [10].

to Dialectica [13], the contraction formula A becomes [14] part of the raw (not yet normalized) realizing term. A number of such *D-relevant* contraction formulas, which would not be part of the executed finally strongly normalized extracted term, can be eliminated already at the extraction stage, see [12] for such an example. Unfortunately, such an a priori elimination during extraction of some of the contractions (which we named "redundant" in [12]) is not always possible, see also [12] for such a negative example. The MD-interpretation simplifies the Dialectica treatment of all non-redundant relevant contractions and therefore represents an important complexity improvement of the extracted program whenever such "persistent" contractions occur in the proof at input.

2 The minimal arithmetic HeExtEq proof in MinLog

MinLog is an interactive proof- and program-extraction system developed by H. Schwichtenberg and members of the logic group at the University of Munich. It is based on first order Natural Deduction calculus and uses as primitive *minimal* rather than classical or intuitionistic logic. See [9,25] for full details.

The hereditarily-extensional-equality test-case (abbreviated HeExtEq) was suggested by U. Kohlenbach as an interesting example for the application of the Monotone Dialectica program extraction from proofs, see Chapter 8 of [21]. In fact it had been carried out at a theoretical level already in Chapter 5 of [20] by means of the precursor of the Monotone Dialectica introduced in [16]. The treatment in [21] is even more platonic, by means of a good number of meta-theorems. We took the challenge to use a machine extraction in order to analyze on the computer a number of concrete instances of the HeExtEq example.

Definition 2.1 [[26], Section 2.7.2, adapted to the **T** presentation from [10]] The *extensional equality* at type $\sigma \equiv \sigma_1 \dots \sigma_n \mathbb{N}$, denoted $=_\sigma$, is defined by

$$\begin{aligned} x =_{\mathbb{N}} y &:\equiv \mathtt{at}(= xy) \\ x =_\sigma y &:\equiv \forall z_1^{\sigma_1} \dots z_n^{\sigma_n} (xz_1 \dots z_n =_{\mathbb{N}} yz_1 \dots z_n) \quad , \end{aligned}$$

where $=$ is defined in [10] as the usual equality boolean function on $\mathbb{N} \times \mathbb{N}$. It is immediate that $x =_{\rho\tau} y \equiv \forall z^\rho (xz =_\tau yz)$. As a parallel with the majorizability relation (see Definition 1.3), the *hereditarily extensional equality* is defined over the **T** type structure by

$$\begin{aligned} x \approx_{\mathbb{N}} y &:\equiv x =_{\mathbb{N}} y \\ x \approx_{\rho\tau} y &:\equiv \forall z_1^\rho, z_2^\rho (z_1 \approx_\rho z_2 \to xz_1 \approx_\tau yz_2) \quad , \end{aligned}$$

Definition 2.2 [Minimal Arithmetic] We denote by $\mathtt{WeZ}_\mathtt{m}$ the system $\mathtt{WeZ}^\exists$ without the strong $\exists$ and also without the Ex-Falso-Quodlibet axiom $\bot \to F$, hence with an underlying Minimal Logic (in the sense of [15]) substructure.

[13] Not all *logical* contractions are relevant for the Dialectica interpretations, see [12] for a short account of this issue or [11] for full details.

[14] Via the boolean term associated (see Definition 1.4) to the *MD-radical* formula $A_{\mathtt{MD}}$ (a quantifier-free formula) which is at its turn associated to the formula A via Definition 1.5.

Proposition 2.3 ([20] - 5.13 or [21] - 8.17, adapted)
Let t^{ρ} be a closed term of Gödel's $\mathbf{T}$. Then $\mathtt{WeZ}_{\mathtt{m}} \vdash t \approx_{\rho} t$.

Proof: By induction on the combinatorial structure of t, since closed terms of Gödel's $\mathbf{T}$ can be expressed [15] as built by application only (i.e., without λ-abstraction) from 0, Suc, Gödel's recursor $\mathcal{R}$ and combinators Σ and Π. $\Box$

Corollary 2.4 ([20,21]) Let $t^{(\mathbb{N}\to\mathbb{N})\to(\mathbb{N}\to\mathbb{N})}$ be a closed $\mathbf{T}$-term. Since

$$\mathtt{WeZ}_{\mathtt{m}} \vdash \forall x^{\mathbb{N}\to\mathbb{N}}, y^{\mathbb{N}\to\mathbb{N}} [\, x =_{\mathbb{N}\to\mathbb{N}} y \leftrightarrow x \approx_{\mathbb{N}\to\mathbb{N}} y \,]$$

it immediately follows that

$$\mathtt{WeZ}_{\mathtt{m}} \vdash \forall x^{\mathbb{N}\to\mathbb{N}}, y^{\mathbb{N}\to\mathbb{N}} [\, x =_{\mathbb{N}\to\mathbb{N}} y \to t(x) =_{\mathbb{N}\to\mathbb{N}} t(y) \,] \quad .$$

Proposition 2.5 ([20] - 5.15 or [21] - 8.19, adapted) Let $t^{(\mathbb{N}\to\mathbb{N})\to(\mathbb{N}\to\mathbb{N})}$ be a closed term of Gödel's $\mathbf{T}$. Then t is uniformly continuous on every ball $B_y :\equiv \{x^{\mathbb{N}\to\mathbb{N}} \,|\, \forall z^{\mathbb{N}}.\, y(z) \succeq_{\mathbb{N}} x(z)\}$ with a modulus of uniform continuity which is effectively synthesizable (uniformly in $y^{\mathbb{N}\to\mathbb{N}}$) as a closed term $\widetilde{t}(y)^{\mathbb{N}\to\mathbb{N}}$ of $\mathbf{T}$, i.e., one can extract (by MD-interpretation) a closed $\mathbf{T}$-term $\widetilde{t}^{\,(\mathbb{N}\to\mathbb{N})\to(\mathbb{N}\to\mathbb{N})}$ s.t.:

$$\mathtt{WeZ}_{\mathtt{m}} \vdash \forall y \forall x_1, x_2 \in B_y \forall k^{\mathbb{N}} [\bigwedge_{i=0}^{\widetilde{t}(y)(k)} x_1(i) =_{\mathbb{N}} x_2(i) \to \bigwedge_{j=0}^{k} t(x_1)(j) =_{\mathbb{N}} t(x_2)(j)]$$

Proof: Straightforward from Corollary 2.4 and Theorem 1.6, see [20,21] for details (in the Hilbert-style setting) of the proof originally introduced in [17]. $\Box$

The HeExtEq example was implemented in MinLog [9] in the sense that a minimal arithmetic MinLog proof of

$$\forall x^{\mathbb{N}\to\mathbb{N}}, y^{\mathbb{N}\to\mathbb{N}} [\, x =_{\mathbb{N}\to\mathbb{N}} y \to t(x) =_{\mathbb{N}\to\mathbb{N}} t(y) \,]$$

is mechanically generated for each particular $\mathbf{T}$-term $t^{(\mathbb{N}\to\mathbb{N})\to(\mathbb{N}\to\mathbb{N})}$ by a Scheme [23] procedure which takes as argument such a concrete MinLog $\mathbf{T}$-term t.

3 The *light* Monotone Dialectica *interpretation*

Our approach for the MinLog extraction of the generic modulus of uniform continuity $\widetilde{t}$, given the concrete MinLog term t is different from the letter of Proposition 2.5. It amounts in fact to the design of a new variant of the MD-interpretation, which combines those features of the pre-existing versions due to Kohlenbach [16] which turn out to be useful on the machine.

[15] Lemma 2.6 of [20] gives such a *syntactic* translation from λ-terms to combinatorial terms.

[16] We distinguish three such variants of the Monotone Dialectica interpretation, which were introduced in (chronologically ordered) [16], [18] and finally [19]. See also Zucker's chapter VI in [26], particularly its sections 8.3-6, for a raw, unformalized and quite primitive form of MD-interpretation.

We here name *light Monotone Dialectica* (abbreviated LMD-interpretation and even shorter, LMDI) this optimization of Kohlenbach's MD-interpretation for the extraction of majorants in NbE-normal form [17]. Hence the particularity of the new light MD-interpretation is the production of terms in normal form, which is *the goal* of the automated, machine program-extraction.

The key features of this novel form of MD-interpretation are the following:

(i) The terms extracted at each step of the recursion over the input proof structure are neither exact realizers, nor majorants, but *partial majorants*, in the sense that only the persistent contractions are treated like in [18].

(ii) An NbE-normalization (see [3,4,5] for the original NbE) of such extracted partial majorants is carried out for optimization purposes after the proof mining of the conclusion at each Implication Elimination (aka Modus Ponens) application. This recurrent form of partial normalization turns out to bring a huge improvement w.r.t. the one single final call-by-value NbE normalization process in situations of long sequences of nested Modus Ponens. We named this technique [18] "Normalization during Extraction" (abbreviated "NdE"), see [13] for a short account. The HeExtEq proof (described in Section 2 above) does actually contain quite long sequences of nested Modus Ponens.

(iii) The final such extracted partial majorant is NbE-normalized and then its majorant is built like in [16], but using the majorant for Gödel's recursor $\mathcal{R}$ from [19].

4 Machine results for the HeExtEq case-study in MinLog

We used our light Monotone Dialectica MinLog extraction modules which are available within the special [19] MinLog distribution [9]. We applied the LMDI extraction on the MinLog HeExtEq proof for the following concrete instances of the term t:

- The simple sum: $f, k \mapsto f(0) + \cdots + f(k)$.
- The double sum: $f, k \mapsto f(f(0)) + \cdots + f(f(k))$.
- The triple sum: $f, k \mapsto f(f(f(0))) + \cdots + f(f(f(k)))$.

In the case of the simple sum, the machine output is, as expected, the identity function, regardless of the actual f, hence the functional $f, k \mapsto k$. Also for the double sum, the outcome is the expected one, namely

$$f, k \;\; \mapsto \;\; max\{k\,,\, f(0)\,,\, \ldots\,,\, f(k)\} \quad .$$

[17] Here "NbE" is the usual acronym for "Normalization by Evaluation". See [3,4,5] for the original call-by-value NbE normalization technique.

[18] Which is a recurrent form of Partial Evaluation. See the volume [7] for accounts of the partial evaluation programming methodology.

[19] Our Dialectica modules are for the moment not compatible with the official MinLog distribution from [25].

On the contrary, for the triple sum, the mathematician needs to work a good number of minutes to produce the following *optimal* result

$$f, k \;\mapsto\; max\{k\,,\, f(0)\,,\, f(1)\,,\, \ldots\,,\, f(max\{k\,,\, f(0)\,,\, f(1)\,,\, \ldots\,,\, f(k)\})\} \qquad (1)$$

The machine produces in less than one minute an output which can be isomorphically adapted for display as follows:

$$f, k \;\mapsto\; max\{k\,,\, f(0)\,,\, \ldots\,,\, f(k)\,,\; max\{f(0)\,,\, \ldots\,,\, f(max\{f(0)\,,\, \ldots\,,\, f(k)\})\}\} \qquad (2)$$

It is easy to notice that the machine-yielded expression (2) is immediately equivalent to the more human expression (1). Note also that in the context of a pointwise continuity demand, the optimal answer would be

$$f, g, k \;\mapsto\; max\{k\,,\, f(0)\,,\, f(1)\,,\, \ldots\,,\, f(k)\,,\, max\{f(f(0))\,,\, f(f(1))\,,\, \ldots\,,\, f(f(k))\}\}$$

which is strictly lower than the machine (or human) optimal output for the case of uniform continuity. In fact, while first trying to solve by brain the triple sum problem, we first erroneously thought that this were a modulus of uniform continuity, which is not the case. We later produced (1) by simplifying the machine outcome (2) and after some checks we realized the error. Hence we could produce a correct answer only with the help of the computer extraction.

Notwithstanding, right now a pattern can be noticed by the human in the solution of the `HeExtEq` problem for terms $t_l :\equiv \lambda f, k.\, f^{(l)}(0) + \cdots + f^{(l)}(k)$, with $f^{(l)}(i) :\equiv f(f \ldots (f(i)))$, where f appears l times on the right-hand side. We write again the above moduli of uniform continuity for t_l, with $l := 1, 2, 3$:

$$\begin{aligned}
\widetilde{t_1}(f,k) &\equiv k\\
\widetilde{t_2}(f,k) &\equiv max\{k, f(0), \ldots, f(\widetilde{t_1}(f,k))\}\\
\widetilde{t_3}(f,k) &\equiv max\{k, f(0), \ldots, f(\widetilde{t_2}(f,k))\}\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{aligned}$$

We thus immediately infer the generic (recursive) solution for every $l \in \mathbb{N}$:

$$\widetilde{t_{l+1}}(f,k) \;\equiv\; max\{k, f(0), \ldots, f(\widetilde{t_l}(f,k))\}$$

The verification that $\widetilde{t_l}$ is the optimal modulus of uniform continuity for t_l is now an easy exercise, which we leave to the reader (see [11] for the solution).

5 Conclusions and future work

More such `MinLog` extractions of moduli of uniform continuity for other various concrete instances of the input term t can and ought to be performed. The light `MD`-interpretation should be mathematically formalized, in synthesis with the *light* optimization of Gödel's Dialectica from [10]. It might be that the latter improvement applies also in the case of the `HeExtEq` proof. This issue should be researched

with high priority. Also a complete mathematical formulation of the Normalization during Extraction (`NdE`) ought to be given.

Acknowledgement

We would like to thank Prof. U. Kohlenbach for having suggested to us that the `HeExtEq` example may produce interesting results on the Computer. We also thank Prof. H. Schwichtenberg for having suggested to us that the already available formulations of the Monotone (or Bounded) Dialectica may not be satisfying enough from the computer-applied viewpoint.

References

[1] Avigad, J. and S. Feferman, *Gödel's functional ('Dialectica') interpretation*, in: S. Buss, editor, *Handbook of Proof Theory*, Studies in Logic and the Foundations of Mathematics **137**, Elsevier, 1998, pp. 337–405.

[2] Berger, U., W. Buchholz and H. Schwichtenberg, *Refined program extraction from classical proofs*, Annals of Pure and Applied Logic **114** (2002), pp. 3–25.

[3] Berger, U. and H. Schwichtenberg, *An inverse of the evaluation functional for typed λ–calculus*, in: R. Vemuri, editor, *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Los Alamitos, 1991, pp. 203–211.

[4] Berger, U., M. Eberl and H. Schwichtenberg, *Normalization by Evaluation*, in: B. Möller and J. Tucker, editors, *Prospects for Hardware Foundations*, LNCS **1546**, Springer Verlag, 1998, pp. 117–137.

[5] Berger, U., M. Eberl and H. Schwichtenberg, *Term rewriting for normalization by evaluation*, Information and Computation **183** (2003), pp. 19–42, International Workshop on Implicit Computational Complexity (ICC'99).

[6] Bezem, M., *Strongly majorizable functionals of finite type: A model for bar-recursion containing discontinuous functionals*, J. of Symb. Logic **50** (1985), pp. 652–660.

[7] Danvy, O., R. Glück and P. Thiemann, editors, "Partial Evaluation. Dagstuhl Castle, Germany, February 1996", LNCS **1110**, Springer Verlag, 1996.

[8] Gödel, K., *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes*, Dialectica **12** (1958), pp. 280–287.

[9] Hernest, M.-D., *`MinLog` for Dialectica program-extraction*, Free software - code source @ http://www.brics.dk/~danher/MinLogForDialectica, For the official `MinLog` see [25].

[10] Hernest, M.-D., *Light Functional Interpretation*, LNCS **3634**, Springer Verlag, 2005, pp. 477 – 492, Computer Science Logic - CSL'2005.

[11] Hernest, M.-D., *Feasible programs from (non-constructive) proofs by the light (monotone) Dialectica interpretation*, PhD Thesis, École Polytechnique `and` University of Munich (2006), In preparation, draft available @ http://www.brics.dk/~danher/teza/.

[12] Hernest, M.-D., *Light Dialectica program extraction from a classical Fibonacci proof*, in: *Proceedings of DCM'06 at ICALP'06*, Electronic Notes in Theoretical Computer Science (ENTCS, 2007), 10pp., Accepted for publication, Downloadable @ http://www.brics.dk/~danher/.

[13] Hernest, M.-D., *NdE - Normalization during Extraction*, Regular Abstract, Local Proceedings of CiE'06 (Computability in Europe 2006), available in the author's web-page, see http://www.brics.dk/~danher/. Full paper in preparation.

[14] Howard, W., *Hereditarily majorizable functionals of finite type*, In [26], appendix chapter, pages 454–461.

[15] Johansson, I., *Der Minimalkalkül, ein reduzierter intuitionistischer Formalismus*, Compositio Matematica **4** (1936), pp. 119–136.

[16] Kohlenbach, U., *Effective bounds from ineffective proofs in analysis: an application of functional interpretation and majorization*, J. of Symb. Logic **57** (1992), pp. 1239–1273.

[17] Kohlenbach, U., *Pointwise hereditary majorization and some applications*, Archive for Mathematical Logic **31** (1992), pp. 227–241.

[18] Kohlenbach, U., *Analysing proofs in Analysis*, in: W. Hodges, M. Hyland, C. Steinhorn and J. Truss, editors, *Logic: from Foundations to Applications, Keele, 1993*, European Logic Colloquium (1996), pp. 225–260.

[19] Kohlenbach, U., *Mathematically strong subsystems of analysis with low rate of growth of provably recursive functionals*, Archive for Mathematical Logic **36** (1996), pp. 31–71.

[20] Kohlenbach, U., *Proof interpretations*, Technical report BRICS LS-98-1, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark (1998), Free downloadble @ http://www.brics.dk/LS/Abs/BRICS-LS-Abs/BRICS-LS-Abs.html.

[21] Kohlenbach, U., *Proof Interpretations and the Computational Content of Proofs*, Latest version in the author's web page (April 2006), vii + 420pp.

[22] Kohlenbach, U. and P. Oliva, *Proof mining: a systematic way of analysing proofs in Mathematics*, Proc. of the Steklov Institute of Mathematics **242** (2003), pp. 136–164.

[23] Cadence Research Systems, *(Petite) Chez Scheme*, http://www.scheme.com (2006).

[24] Schwichtenberg, H., *Minimal logic for computable functions*, Lecture course on program-extraction from (classical) proofs. Available in the `MinLog` distribution [25].

[25] Schwichtenberg, H. and Others, *Proof- and program-extraction system `MinLog`*, Free software - code source and documentation @ http://www.minlog-system.de/.

[26] Troelstra, A., editor, "Metamathematical investigation of intuitionistic Arithmetic and Analysis", Lecture Notes in Mathematics (LNM) **344**, Springer-Verlag, 1973.