CADE-21

The 21^{st} Conference on Automated Deduction

Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice

(LFMTP'07)

Editors:
Brigitte Pientka, Carsten Schürmann

Bremen, Germany, July 15^{th} 2007

CADE-21 Organization:

Conference Chair: Michael Kohlhase (Jacobs University Bremen)
Program Chair: Frank Pfenning (Carnegie Mellon University)
Workshop Chair: Christoph Benzmüller (University of Cambridge)
Local Organization: Event4 Event Management

Preface

These are the Proceedings of the Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP'07. The LFMTP workshop series emerged from the LFM workshop series on "Logical Frameworks and Meta-Languages" and the MERλIN workshop series on "MEchanized Reasoning about Languages with variable BInding".

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 has been the focus of considerable research over the last two decades, using competing and sometimes incompatible basic principles. This workshop brings together designers, implementors, and practitioners to discuss all aspects of logical frameworks.

We received 13 submissions to the workshop of which the committee decided to accept 10. The programme also includes one invited talk by Randy Pollack, LFCS, University of Edinburgh. The papers included in these workshop proceedings were peer-reviewed by the program committee.

The editors would like to thank the committee and external reviewers for their excellent work. LFMTP07 is held on July 15th, 2007 in association with CADE-21, the Conference on Automated Deduction in Bremen. We would like to thank the CADE-21 workshop chair Christoph Benzmüller and the CADE-21 conference chair Michael Kohlhase for all their support and help with the organization of this workshop, and Andrei Voronkov for letting us use his easychair online refereeing software. It has been a pleasure to work with all of you. Thank you.

July 15th, 2007
Brigitte Pientka
Carsten Schürmann

Table of Contents

Author Index

Invited Talk: Locally Nameless Representation and Nominal Isabelle

Randy Pollack[1]

Edinburgh University, U.K.

The idea that bound variables and free (global) variables should be represented by distinct syntactic classes of names goes back at least to Gentzen and Prawitz. In the early 1990's, McKinna and Pollack formalized a large amount of the metatheory of Pure Type Systems with a representation using two classes of names. This work developed a style for formalizing reasoning about binding which was heavy, but worked reliably. However, the use of names for bound variables is not a perfect fit to the intuitive notion of binding, so I suggested (1994) that the McKinna–Pollack approach to reasoning with two species of variables also works well with a representation that uses names for global variables, and de Bruijn indices for bound variables. This *locally nameless* representation, in which alpha equivalence classes have exactly one element, had previously been used in Huet's Constructive Engine and by Andy Gordon. The locally nameless representation with McKinna–Pollack style reasoning has recently been used by several researchers (with several proof tools) for solutions to the POPLmark Challenge.

In this talk I discuss the current state of play with locally nameless representation and suitable styles of reasoning about it. Using Urban's nominal Isabelle package, it is possible to get the boilerplate definitions and lemmas about freshness of names and swapping/permuting names for free. From nominal Isabelle and other work there are new appraches to proving the strengthened induction principles that reasoning with names requires. There are also some downsides to the new ideas, which I will point out. Among my examples is an isomorphism (proved in nominal Isabelle) between nominal representation and locally nameless representation of lambda terms.

[1] Email: rpollack@inf.ed.ac.uk

Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax

Alberto Momigliano,[a] Alan J. Martin,[b] Amy P. Felty[c,b]

[a] *LFCS, School of Informatics, University of Edinburgh, U.K.& DSI, University of Milan, Italy*
Email: amomigl1@inf.ed.ac.uk

[b] *Department of Mathematics and Statistics, University of Ottawa, Canada*
Email: {amart045,afelty}@site.uottawa.ca

[c] *School of Information Technology and Engineering (SITE), University of Ottawa, Canada*

Abstract

Logical frameworks supporting *higher-order abstract syntax* (HOAS) allow a direct and concise specification of a wide variety of languages and deductive systems. Reasoning about such systems within the same framework is well-known to be problematic. We describe the new version of the Hybrid system, implemented on top of Isabelle/HOL (as well as Coq), in which a de Bruijn representation of λ-terms provides a definitional layer that allows the user to represent object languages in HOAS style, while offering tools for reasoning about them at the higher level. We briefly describe how to carry out two-level reasoning in the style of frameworks such as Linc, and briefly discuss our system's capabilities for reasoning using tactical theorem proving and principles of induction and coinduction.

Keywords: higher-order abstract syntax, interactive theorem proving, induction, variable binding, Isabelle/HOL

1 Introduction

We give a system presentation of Hybrid [4] (`http://hybrid.dsi.unimi.it/`), in coincidence with the release of its new and first official version, as well as with the porting to Isar and Coq [2,1]. Hybrid is a package that introduces a binding operator that (1) allows a direct expression of λ-abstraction in full higher-order abstract syntax (HOAS) style, and (2) is *defined* in such a way that expanding its definition results in the conversion of a term to its de Bruijn representation [8]. The latter makes Hybrid's specifications compatible with principles of (co)induction, available in standard proof assistants. The basic idea is inspired by the work of Gordon [10], where bound variables are presented to the user as strings. Instead of strings, we use a binding operator (LAM) defined using λ-abstraction at the meta-level. Hybrid provides a library of operations and lemmas to reason on the HOAS level, hiding the details of the de Bruijn representation. Hybrid originated as a (meta)language on top of Isabelle/HOL for reasoning over languages with bindings, aiming to enrich a traditional inductive setting with a form of HOAS. It soon became apparent

that it could also provide definitional support for *two-level* reasoning as proposed in our previous work [9]; the latter aimed to endow Coq (in that case) with the style of reasoning of frameworks such as $FO\lambda^{\Delta\mathbb{N}}$ [12], Linc [17], and, to a lesser extent, Twelf [16]. Hybrid is defined as an Isabelle/HOL theory (approx. 150 lines of functional definitions and 400 lines of statements and proofs). In contrast to other approaches such as the Theory of Contexts [11], our Isabelle/HOL theory does not contain any axioms, which would require external justification.

Previous work has described Hybrid applied to a variety of object languages (OLs) and their (meta)properties, as detailed in Section 4. Here we concentrate on describing some new features of the system infrastructure, as well as recalling the two-level approach. We briefly mention some initial tactical support for reasoning in this setting, and describe how we state the adequacy of Hybrid's encodings.

The main improvement of this version is an overall reorganization of the infrastructure, based on the internalization of the set of *proper* terms: those are the subset of de Bruijn terms that are well-formed (in the sense that all indices representing bound variable occurrences in a term have a corresponding binder in the term), eliminating the need for adding well-formedness annotations in OL judgments. We recall that to represent syntax in the presence of binders, we use a predicate (abstr) that recognizes the *parametric* part of the function space between OL expressions. The crucial injectivity property of the Hybrid binder LAM:

$$\mathsf{abstr}\ f \Longrightarrow (\mathsf{LAM}\, x.\, f\, x = \mathsf{LAM}\, x.\, g\, x) = (f = g).$$

strengthens our previous version by requiring only one of f and g to satisfy this condition (instead of both), thus simplifying the elimination rules for inductively defined OL judgments.

Notation: An Isabelle/HOL type declaration has the form $s :: [\, t_1, \ldots t_n \,] \Rightarrow t$. Free variables are implicitly universally quantified. We use $\equiv$ and $\bigwedge$ for *equality by definition* and universal meta-quantification. We use the usual logical symbols for the connectives of Isabelle/HOL. A rule (a sequent) with premises $H_1 \ldots H_n$ and conclusion C will be represented as $[\![\, H_1; \ldots ; H_n \,]\!] \Longrightarrow C$. The keyword *Inductive* introduces an inductive relation in Isabelle/HOL; similarly for *datatype*. We freely use infix notations, without explicit declarations.

2 Using Hybrid

The objective of Hybrid is to provide support for HOAS via an *approximation* of the following datatype definition, which is not well-formed in an inductive setting:

$$\mathit{datatype}\ \alpha\ \mathit{expr} = \mathsf{CON}\ \alpha \mid \mathsf{VAR}\ \mathit{var} \mid \mathit{expr}\ \$\ \mathit{expr} \mid \mathsf{LAM}\ (\underline{\mathit{expr}} \Rightarrow \mathit{expr})$$

where *var* is a countably infinite set of free variables, and the type parameter α is used to supply object-language-specific constants. (It will henceforth be omitted, except where instantiated.) The problem is, of course, LAM, whose argument type involves a negative occurrence (underlined) of *expr*. Since it is possible to formalize Cantor's diagonal argument in Isabelle/HOL, such a function cannot be injective, and thus cannot be a constructor of a datatype.

However, for HOAS it is neither necessary nor desirable for LAM to be used with arbitrary Isabelle/HOL functions as arguments: only those functions that use their arguments *generically* are needed. These functions will be called *abstractions*; in the Hybrid system, they are recognized by a predicate $\mathsf{abstr} :: [\, \mathit{expr} \Rightarrow \mathit{expr} \,] \Rightarrow \mathit{bool}$. Functions that are not abstractions are called *exotic*,

for example $\lambda x.\,\mathsf{if}\ x = \mathsf{CON}\ a\ \mathsf{then}\ (x\ \$\ x)\ \mathsf{else}\ x$ for some OL constant a. The possibility of introducing such functions would break the adequacy of any second-order encoding.

As described in Section 3, Hybrid defines a type *expr* together with functions of the appropriate types to replace the constructors of the problematic datatype definition above. Distinctness of the constructors and injectivity of CON, VAR, and \$ are proved. Two more properties would be needed for a datatype: injectivity of LAM and an induction principle. In Hybrid, the former is weakened by the addition of an abstr premise, while the latter uses a nonstandard LAM case to allow the induction hypothesis to have the type $expr \Rightarrow bool$ despite the presence of HOAS:

$$[\![\ \ldots; \bigwedge v.\, P\,(e\,(\mathsf{VAR}\ v)) \Longrightarrow P\,(\mathsf{LAM}\ e)\ ]\!] \Longrightarrow P\,(u :: expr)$$

Thus, the type *expr* is only a "quasi-datatype", and it is necessary to impose abstr conditions wherever LAM is used. Hybrid provides lemmas for proving the resulting abstr subgoals, so that basic reasoning about *expr* is similar to a true datatype. However, primitive recursion on *expr* is not (currently) available. Also, induction involves the use of explicit free variables, which HOAS techniques normally seek to avoid, and this complicates the reasoning; thus, induction on OL judgments is preferred.

From the user's point of view Hybrid provides a form of HOAS where: object level constants correspond to expressions of the form CON c; bound object-level variables correspond to (bound) meta variables in expressions of the form LAM $v.\,e$; subterms are combined with \$; and free object-level variables may be represented as VAR i (although typical examples will not use this feature).

2.1 Example

As a small example (space limitations), we consider the encoding of the types of $F_{<:}$, the subtyping language of the PoplMark challenge [6] as an OL. Types have the form $\top$ (the maximum type), $\tau_1 \to \tau_2$ (type of functions), or $\forall x{<:}\tau_1.\tau_2$ (bound universal type). In the latter, x is a type variable possibly occurring in τ_2, which can be instantiated with subtypes of τ_1. To represent the OL types, we define:

$$\begin{aligned} datatype\ con &= cTOP \mid cARR \mid cUNI \qquad uexp = con\ expr \\ \mathsf{top} &\equiv \mathsf{CON}\ cTOP \\ t_1\ \mathsf{arrow}\ t_2 &\equiv \mathsf{CON}\ cARR\ \$\ t_1\ \$\ t_2 \\ \mathsf{univ}\,t_1\,(x.\,t_2\ x) &\equiv \mathsf{CON}\ cUNI\ \$\ t_1\ \$\ \mathsf{LAM}\,x.\,t_2\ x. \end{aligned}$$

Note that *uexp* is introduced to abbreviate an instantiated version of the "quasi-datatype" *expr*, where α is replaced by the above type *con* that is introduced specifically for this OL.

To illustrate the representation of judgments of an OL, we consider rules for well-formed types, where Γ is a list of distinct type variables.

$$\frac{x \in \Gamma}{\Gamma \vdash x} \qquad \frac{}{\Gamma \vdash \top} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma, x \vdash \tau_2}{\Gamma \vdash \forall x{<:}\tau_1.\tau_2}$$

The standard Twelf-style encoding of the right premise of the last rule would be $\bigwedge x.\ \underline{\mathsf{isTy}\ x} \Longrightarrow \mathsf{isTy}\ (T_2\ x)$, where isTy is a meta-level predicate introduced to repre-

sent this OL judgment. Note the negative occurrence (underlined) of the predicate being defined (the same problem as before, but at the predicate level). *Two-level reasoning* is introduced to address this problem. In particular, a *specification logic* (SL) is defined inductively in Isabelle/HOL, which is in turn used to drive the encoding of the OL as an inductive set of Prolog-like clauses, avoiding any negative occurrences at the meta-level.

A Hybrid user can specify his/her own SL, but we envision a library of such logics that a user can choose from. Indeed, several such logics have been encoded to date. We can view our realization of the two-level approach as a way of "fast prototyping" HOAS logical frameworks. We can quickly implement and experiment with a potentially interesting SL, without the need to develop all the building blocks of a usable new framework, such as unification algorithms, type inference or proof search; instead we rely on the ones provided by Isabelle/HOL.

To illustrate, we choose a simple SL, a sequent formulation of a fragment of first-order minimal logic with backchaining, adapted from [12]. Its syntax can be encoded directly with an Isabelle/HOL datatype:

$$\mathit{datatype}\ \mathit{oo} = \mathsf{tt} \mid \langle \mathit{atm} \rangle \mid \mathit{oo}\ \ \mathsf{and}\ \ \mathit{oo} \mid \mathit{atm}\ \ \mathsf{imp}\ \ \mathit{oo} \mid \mathsf{all}\ (\mathit{uexp} \Rightarrow \mathit{oo})$$

where *atm* is a parameter used to represent atomic predicates of the OL and $\langle \_ \rangle$ coerces atoms into propositions. We use the symbol $\rhd$ for the sequent arrow of the SL, in this case decorated with natural numbers to allow reasoning by (complete) induction on the height of a proof. The inference rules of the SL are represented as the following Isabelle/HOL inductive definition:

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

The backward arrow in $A \longleftarrow G$ in the last rule is used to encode OL judgments as logic programming style clauses. To reason about OLs, a small set of structural rules of the SL is proved once and for all, such as weakening and cut elimination. To complete our example OL, we define *atm* as a datatype with a single constructor $\mathsf{isTy}\ \mathit{uexp}$, thus representing the OL well-formedness judgment '$\vdash$' at the specification level, and encode the OL inference rules as the following definition of $(\_ \longleftarrow \_)$:

$$\begin{aligned}
\mathit{Inductive}\ _ \longleftarrow _\ ::\ [\,\mathit{atm}, \mathit{oo}\,] \Rightarrow \mathit{bool} \\
\Longrightarrow \mathsf{isTy}\ \mathsf{top} \longleftarrow \mathsf{tt} \\
\Longrightarrow \mathsf{isTy}\ (T_1\ \mathsf{arrow}\ T_2) \longleftarrow \langle \mathsf{isTy}\ T_1 \rangle\ \mathsf{and}\ \langle \mathsf{isTy}\ T_2 \rangle \\
[\![\, \mathsf{abstr}\ T_2 \,]\!] \Longrightarrow \mathsf{isTy}\ (\mathsf{univ}\, T_1\ (x.\, T_2\ x)) \longleftarrow \\
\qquad \langle \mathsf{isTy}\ T_1 \rangle\ \mathsf{and}\ (\mathsf{all}\, x.\, (\mathsf{isTy}\ x)\ \mathsf{imp}\ \langle \mathsf{isTy}\ (T_2\ x) \rangle)
\end{aligned}$$

Note the negative occurrence of isTy now embedded in the SL. We remark that Hybrid is (currently) untyped in the sense that OL syntax is encoded as terms of

type *uexp*; i.e., there is no new type introduced to represent well-formed types of $F_{<:}$. For this reason, the isTy predicate is necessary to identify the required subset of *uexp*.

For any OL represented in Hybrid, it is important to show that both terms and judgments are adequately encoded. For our example, this means showing that there is a bijection between object-level types of $F_{<:}$ and the subset of terms of type *uexp* formed from only variables, the constants top, arrow, and univ, and Isabelle/HOL λ-abstractions (the latter of which can only appear as the second argument of univ). We write ε_X for the *encoding* function from OL terms with free variables in X to terms in *uexp*, and δ_X for its inverse *decoding*. We also need to show that substitution commutes with the encoding. For OL judgments, the proof obligation is that whenever $x_1, \ldots, x_n \vdash t$ holds in the OL, then for some i, $(\mathsf{isTy}\ x_1, \ldots, \mathsf{isTy}\ x_n) \rhd_i \mathsf{isTy}\ (\varepsilon_{x_1,\ldots,x_n}(t))$ is provable. Conversely, whenever $(\mathsf{isTy}\ x_1, \ldots, \mathsf{isTy}\ x_n) \rhd_i \mathsf{isTy}\ T$ is provable, then T is in the domain of $\delta_{x_1,\ldots,x_n}$ and $x_1, \ldots, x_n \vdash \delta_{x_1,\ldots,x_n}(T)$ holds in the OL. It is also important to show the adequacy of SL encodings. For the SL presented here, we can adapt the proof from [12]. Since our version is defined inductively, the inversion properties of this definition play a central role in the proof.

2.2 Tactical support

We chose to develop Hybrid as a package, rather than a standalone system mainly to exploit all the reasoning capabilities that a mature proof assistant can provide, in particular support for tactical theorem proving. Contrast this with a system such as Twelf, where proofs are coded as logic programs and post hoc checked for correctness. At the same time, our aim is to try to retain some of the conciseness of a language such as LF, which for us means hiding most of the administrative reasoning concerning variable binding and contexts. Because of the "hybrid" nature of our approach, this cannot be completely achieved, but some simple-minded tactics go a long way into mechanizing most of boilerplate scripting. While in the previous version of the system we employed specific tactics to recognize *proper* terms and *abstractions*, now this is completely delegated to Isabelle's simplification. Thus, we can concentrate on assisting two-level reasoning, which would otherwise be encumbered by the indirection in accessing OL specifications via the SL. Luckily, Twelf-like reasoning consists, at a high-level, of three basic steps: inversion, backchaining (filling, in Twelf's terminology) and recursion. This corresponds to highly stereotyped proof scripts that we have abstracted into:

(i) an *inversion* tactic `defL_tac`, which goes through the SL and applies as an elimination rule one of the OL clauses. This is complemented by the eager application of other *safe* elimination rules (viz. invertible SL rules such as conjunction elimination). This contributes to keeping the SL overhead to a minimum;

(ii) a dual *backchaining* tactic `defR_tac`; the latter is integrated into the tactic `2lprolog_tac`, which performs automatic depth first search (or other searches supported by Isabelle) on Prolog-like goals;

(iii) a *complete induction* tactic, to be fired when given the appropriate derivation height by the user.

These tactics have been tested most extensively on the minimal SL, while more human intervention is required when using sub-structural logics (such as *Olli* [15]), given the non-deterministic nature of their context management.

3 Definition of Hybrid in Isabelle/HOL

The Hybrid system defines the type *expr* in terms of an Isabelle/HOL datatype *dB* that uses de Bruijn indices to represent bound variables:

$$\mathit{datatype}\ \alpha\ \mathit{dB} = \mathsf{CON}'\,\alpha \mid \mathsf{VAR}'\,\mathit{var} \mid \mathsf{BND}'\,\mathit{bnd} \mid \mathit{dB}\ \$'\ \mathit{dB} \mid \mathsf{ABS}'\,\mathit{dB} \mid \mathsf{ERR}'$$

where the type *bnd* of de Bruijn indices is defined to be the natural numbers, and the type parameter α is the same as for *expr*.

Each occurrence of $\mathsf{BND}'\,i$ refers to the variable implicitly bound by the $(i+1)^{\text{th}}$ enclosing ABS' node. If there are not enough enclosing ABS' nodes, then it is called a *dangling index*. A term without dangling indices is called *proper*, and *expr* should consist of the proper terms of type *dB*; but since the subterms of a proper term are not always proper, a more general notion of *level* is needed. Thus, Hybrid defines a predicate $\mathsf{level} :: [\mathit{nat}, \mathit{dB}] \Rightarrow \mathit{bool}$ by primitive recursion; the meaning of $\mathsf{level}\,i\,s$ is that the term s would have no dangling indices if enclosed in at least i ABS' nodes.

Using Isabelle/HOL's *typedef* mechanism, the type *expr* is defined as a *bijective image* of the set $\{\, s :: \mathit{dB} \mid \mathsf{level}\,0\,s \,\}$, with inverse bijections $\mathsf{dB} :: \mathit{expr} \Rightarrow \mathit{dB}$ and $\mathsf{expr} :: \mathit{dB} \Rightarrow \mathit{expr}$. In effect, *typedef* makes *expr* a subtype of *dB*, but since Isabelle/HOL's type system does not have subtyping, the conversion function dB must be explicit. The notation $\ulcorner s \urcorner = \mathsf{dB}\ s$ and $\llcorner s \lrcorner = \mathsf{expr}\ s$ will be used below, although dB and expr will still be used when referring to them as functions[1]. At this point three of the four constructors of *expr* can be defined:

$$\mathsf{CON}\ a \equiv \llcorner \mathsf{CON}'\,a \lrcorner \qquad \mathsf{VAR}\ n \equiv \llcorner \mathsf{VAR}'\,n \lrcorner \qquad s\ \$\ t \equiv \llcorner \ulcorner s \urcorner\ \$'\ \ulcorner t \urcorner \lrcorner$$

To define the predicate abstr and the remaining constructor LAM, it is helpful first to explicitly represent the structure of abstractions. Thus, Hybrid defines a polymorphic datatype *dB_fn*:

$$\mathit{datatype}\ (\beta, \alpha)\ \mathit{dB\_fn} = \mathsf{ATOM}^*\,(\beta \Rightarrow \alpha\ \mathit{dB}) \mid \mathsf{CON}^*\,\alpha \mid \mathsf{VAR}^*\,\mathit{var} \mid \mathsf{BND}^*\,\mathit{bnd} \mid \mathit{dB\_fn}\ \$^*\ \mathit{dB\_fn} \mid \mathsf{ABS}^*\,\mathit{dB\_fn} \mid \mathsf{ERR}^*$$

(where the type parameter α will once again be left implicit), together with a function $\text{-}_\star :: \beta\ \mathit{dB\_fn} \Rightarrow (\beta \Rightarrow \mathit{dB})$ that maps the constructors of $\beta\ \mathit{dB\_fn}$ to corresponding constructors of *dB* applied pointwise, e.g., $(S\ \$^*\ T)_\star = \lambda x.\,(S_\star\,x)\ \$'\ (T_\star\,x)$, except that $(\mathsf{ATOM}^*\,f)_\star = f$.

A function is called *ordinary* if it is the image under $\text{-}_\star$ of a term whose root node is not ATOM^*, and a term of type $\beta\ \mathit{dB\_fn}$ is called *full* if in all of its occurrences of $\mathsf{ATOM}^*\,f$, the function f is not ordinary. Every function $f :: \beta \Rightarrow \mathit{dB}$ can be written uniquely in the form $T_\star$ for some full term $T :: \beta\ \mathit{dB\_fn}$: the non-$\mathsf{ATOM}^*$ constructors represent the common structure of $f\,x$ for all values of x, while the

[1] The function expr in the Isabelle/HOL theory is actually modified from the one provided by *typedef*, to produce a well-behaved result even when presented with a term of nonzero level. In particular, we will have $\ulcorner s\ \$\ t \urcorner = \ulcorner s \urcorner\ \$'\ \ulcorner t \urcorner$ without any level assumption.

ATOM^* constructors represent the places where $f\,x$ depends on x. That is, $\text{-}_\star$ is bijective on full terms; its inverse shall be denoted $\mathsf{dB\_fn} :: (\beta \Rightarrow dB) \Rightarrow \beta\ dB\_fn$.

Establishing this bijection is a significant part of the Isabelle/HOL theory, and it allows functions on $expr \Rightarrow dB$ to be defined by primitive recursion, and their properties proved by induction, on $expr\ dB\_fn$ [2].

Now abstr is defined by $\mathsf{abstr}\ f \equiv \mathsf{Abstr}\ f^\star$, where $f^\star = \mathsf{dB\_fn}\ (\mathsf{dB} \circ f)$ and the auxiliary predicate Abstr is defined by primitive recursion on $expr\ dB\_fn$; the essential case is $\mathsf{Abstr}\ (\mathsf{ATOM}^*\ f) = (f = \mathsf{dB})$. Note that $\mathsf{ATOM}^*\,\mathsf{dB} = (\lambda\, x.\, x)^\star$ in $f^\star$ stands in for the bound metavariable in f.

Similarly, LAM is defined by $\mathsf{LAM}\ f \equiv \mathsf{Lambda}\ f^\star$, where

$$\mathsf{Lambda}\ S \equiv \mathsf{if}\ \mathsf{Abstr}\ S\ \mathsf{then}\ \llcorner \mathsf{ABS}'\,(\mathsf{Lbind}\ 0\ S)\lrcorner\ \mathsf{else}\ \llcorner \mathsf{ERR}'\lrcorner.$$

The conditional construction serves to distinguish LAM of an abstraction from LAM of an exotic function. The function $\mathsf{Lbind} :: [\,bnd, expr\ dB\_fn\,] \Rightarrow dB$ is defined by primitive recursion:

$$\begin{aligned}
\mathsf{Lbind}\ i\ (\mathsf{ATOM}^*\ f) &= \mathsf{BND}'\,i \\
\mathsf{Lbind}\ i\ (S\ \$^*\ T) &= (\mathsf{Lbind}\ i\ S)\ \$'\ (\mathsf{Lbind}\ i\ T) \\
\mathsf{Lbind}\ i\ (\mathsf{ABS}^*\ S) &= \mathsf{ABS}'\,\big(\mathsf{Lbind}\ (\mathsf{Suc}\ i)\ S\big)
\end{aligned}$$

where $\mathsf{Lbind}\ i\ S = S_\star\ \mathsf{arbitrary}$ in the remaining cases. Note that the Abstr condition ensures that all occurrences of $\mathsf{ATOM}^*\ f$ passed to Lbind have $f = \mathsf{dB}$.

With these definitions, statements such as the following are provable:

$$\begin{aligned}
&\mathsf{LAM}\,x.\,\mathsf{LAM}\,y.\,\mathsf{CON}\ c\ \$\ x\ \$\ y\ \$\ \mathsf{VAR}\ 3 = \\
&\qquad \llcorner \mathsf{ABS}'\,(\mathsf{ABS}'\,(\mathsf{CON}'\,c\ \$'\ \mathsf{BND}'\,1\ \$'\ \mathsf{BND}'\,0\ \$'\ \mathsf{VAR}'\,3))\lrcorner
\end{aligned}$$

Indeed, application of dB or expr triggers simplification rules that convert between HOAS and de Bruijn form.

4 Conclusion

Materials related to Hybrid, including source code, case studies and previous papers, can be found at `http://hybrid.dsi.unimi.it/`. Ready-to-use SLs are minimal and ordered linear logic. Several case studies have been carried out, only the first three being one-level:

- Encodings and proofs of simple properties of quantified propositional formulae (conversion to normal forms), and of the higher-order π-calculus (structural congruence and reaction rules) [4].
- A Howe-style proof that applicative bisimulation in the lazy λ-calculus is a congruence [13].
- A subject reduction theorem [5] for the intermediate language MIL-lite of the MLj compiler.
- The two-level approach with Coq as the meta-language is first introduced in [9];

[2] In previous versions of Hybrid [4], a related induction principle on $dB \Rightarrow dB$, called abstraction_induct, was used directly. A generalization of it is still used in establishing the bijection. Other instances of dB_fn may be useful in generalizing abstr to functions of more than one variable.

subject reduction and uniqueness of typing of Mini-ML are re-proved and compared to the proofs in McDowell's thesis.

- In [14] we verified the correctness of a compiler for (a fragment) of Mini-ML into an environment machine. To deal with recursion more succinctly, we enriched the language with Milner & Tofte's non-well-founded closures, and checked, via coinduction, a type preservation result.
- Properties of continuation machines are investigated in [15] with an ordered linear logic as SL, e.g. internalizing the instruction stack in the ordered context.

Future work will tackle the issue of formulating SLs capable of performing induction over *open* terms, required, for example, to complete the PoplMark challenge. We also plan to add primitive recursion principles for defining functions directly on the higher-order syntax, following on [5, 7]. On the practical side, we are looking into presenting Hybrid as a *"lightweight" HOAS package*, as opposed to Urban's nominal package [18], which is more concerned with a machine assisted reconstruction of the informal "Barendregt" style of mathematical reasoning in presence of binders. Our package would include some facilities that would automatically turn a user signature into appropriate Hybrid-based Isabelle/HOL theories, in the spirit of OTT [3]. The aim is to aid the user's focus on the problem at hand by further separating him from the machinery of defining an OL, such as CON instantiation, simplifier setup and customization of the tactics we have discussed earlier.

Acknowledgement

Felty and Martin acknowledge the support of the Natural Sciences and Engineering Research Council of Canada and the University of Ottawa. Momigliano is partially supported by the European Project Mobius within the frame of IST 6th Framework.

References

[1] *The Coq proof assistant, v.8.0*, `http://coq.inria.fr/`.

[2] *Isabelle/Isar 2005*, `http://isabelle.in.tum.de/Isar`.

[3] *Ott*, `http://www.cl.cam.ac.uk/~pes20/ott/`.

[4] Ambler, S., R. Crole and A. Momigliano, *Combining higher order abstract syntax with tactical theorem proving and (co)induction*, in: *Fifteenth International Conference on Theorem Proving in Higher-Order Logics*, Springer-Verlag LNCS **2342**, 2002, pp. 13–30.

[5] Ambler, S. J., R. L. Crole and A. Momigliano, *A definitional approach to primitive recursion over higher order abstract syntax*, in: *ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding* (2003), pp. 1–11.

[6] Aydemir, B. E. et al., *Mechanized metatheory for the masses: the* PoplMark *challenge*, in: *Eighteenth International Conference on Theorem Proving in Higher-Order Logics*, Springer-Verlag LNCS **3605**, 2005, pp. 50–65.

[7] Capretta, V. and A. Felty, *Combining de Bruijn indices and higher-order abstract syntax in Coq*, in: *Proceedings of TYPES 2006*, Springer-Verlag LNCS **4502**, 2007, pp. 63–77.

[8] de Bruijn, N. G., *Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem*, Indagationes Mathematicæ **34** (1972), pp. 381–392.

[9] Felty, A., *Two-level meta-reasoning in Coq*, in: *Fifteenth International Conference on Theorem Proving in Higher-Order Logics*, Springer-Verlag LNCS **2342**, 2002, pp. 198–213.

[10] Gordon, A., *A mechanisation of name-carrying syntax up to α-conversion*, in: *Higher Order Logic Theorem Proving and its Applications*, Springer-Verlag LNCS **780**, 1993, pp. 414–426.

[11] Honsell, F., M. Miculan and I. Scagnetto, *An axiomatic approach to metareasoning on nominal algebras in HOAS*, in: *28th International Colloquium on Automata, Languages and Programming*, Springer-Verlag LNCS **2076**, 2001, pp. 963–978.

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

[13] Momigliano, A., S. Ambler and R. Crole, *A Hybrid encoding of Howe's method for establishing congruence of bisimilarity*, Electronic Notes in Theoretical Computer Science **70** (2002), pp. 60–75.

[14] Momigliano, A. and S. J. Ambler, *Multi-level meta-reasoning with higher order abstract syntax*, in: *Sixth International Conference on Foundations of Software Science and Computational Structures*, Springer-Verlag LNCS **2620**, 2003, pp. 375–391.

[15] Momigliano, A. and J. Polakow, *A formalization of an ordered logical framework in Hybrid with applications to continuation machines*, in: *ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding* (2003), pp. 1–9.

[16] Pfenning, F. and C. Schürmann, *System description: Twelf — a meta-logical framework for deductive systems*, in: *Sixteenth International Conference on Automated Deduction*, Springer-Verlag LNCS **1632**, 1999, pp. 202–206.

[17] Tiu, A., "A Logical Framework for Reasoning about Logical Specifications," Ph.D. thesis, Pennsylvania State University (2004).

[18] Urban, C. and C. Tasson, *Nominal techniques in Isabelle/HOL*, in: *Twentieth International Conference on Automated Deduction*, Springer-Verlag LNCS **3632**, 2005, pp. 38–53.

A Bidirectional Refinement Type System for LF

William Lovas[1] Frank Pfenning[2]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

We present a system of refinement types for LF in the style of recent formulations where only canonical forms are well-typed. Both the usual LF rules and the rules for type refinements are bidirectional, leading to a straightforward proof of decidability of type-checking even in the presence of intersection types. Because we insist on canonical forms, structural rules for subtyping can now be derived rather than being assumed as primitive. We illustrate the expressive power of our system with several examples in the domain of logics and programming languages.

Keywords: LF, refinement types, subtyping, dependent types, intersection types

1 Introduction

LF was created as a framework for defining logics [6]. Since its inception, it has been used to formalize reasoning about a number of deductive systems (see [13] for an introduction). In its most recent incarnation as the Twelf metalogic [14], it has been used to encode and mechanize the metatheory of programming languages that are prohibitively complex to reason about on paper [3,9].

It has long been recognized that some LF encodings would benefit from the addition of a subtyping mechanism to LF [12,2]. In LF encodings, judgements are represented by type families, and many subsyntactic relations and judgemental inclusions can be elegantly represented via subtyping.

Prior work has explored adding subtyping and intersection types to LF via *refinement types* [12]. Many of that system's metatheoretic properties were proven indirectly by translation into other systems, though, giving little insight into a notion of adequacy or an implementation strategy. We present here a refinement type system for LF based on the modern *canonical forms* approach, and by doing so we obtain direct proofs of important properties like decidability.

[1] Email: wlovas@cs.cmu.edu
[2] Email: fp@cs.cmu.edu

In canonical forms-based LF, only β-normal η-long terms are well-typed — the syntax restricts terms to being β-normal, while the typing relation forces them to be η-long. Since standard substitution might introduce redexes even when substituting a normal term into a normal term, it is replaced with a notion of *hereditary substitution* that contracts redexes along the way, yielding another normal term. Since only canonical forms are admitted, type equality is just α-equivalence, and typechecking is manifestly decidable.

Canonical forms are exactly the terms one cares about when adequately encoding a language in LF, so this approach loses no expressivity. Since all terms are normal, there is no notion of reduction, and thus the metatheory need not directly treat properties related to reduction, such as subject reduction, Church-Rosser, or strong normalization. All of the metatheoretic arguments become straightforward structural inductions, once the theorems are stated properly.

By introducing a layer of refinements distinct from the usual layer of types, we prevent subtyping from interfering with our extension's metatheory. We also follow the general philosophy of prior work on refinement types [5,4] in only assigning refined types to terms already well-typed in pure LF, ensuring that our extension is conservative.

In the remainder of the paper, we describe our refinement type system alongside several illustrative examples (Section 2). Then we explore its metatheory and give proof sketches of important results, including decidability (Section 3). We note that our approach leads to subtyping only being defined on atomic types, but we show that subtyping at higher types is already present in our system by proving that the usual declarative rules are sound and complete with respect to an intrinsic notion of subtyping (Section 4). Finally, we discuss some related work (Section 5) and summarize our results (Section 6).

2 System and Examples

We present our system of LF with Refinements, LFR, through several examples. In what follows, R refers to atomic terms and N to normal terms. Our atomic and normal terms are exactly the terms from canonical presentations of LF.

$$\begin{array}{rl} R ::= c \mid x \mid R\ N & \text{atomic terms} \\ N, M ::= R \mid \lambda x.\, N & \text{normal terms} \end{array}$$

In this style of presentation, typing is defined bidirectionally by two judgements: $R \Rightarrow A$, which says atomic term R *synthesizes* type A, and $N \Leftarrow A$, which says normal term N *checks* against type A. Since λ-abstractions are always checked against a given type, they need not be decorated with their domain types.

Types are similarly stratified into atomic and normal types.

$$\begin{array}{rl} P ::= a \mid P\ N & \text{atomic type families} \\ A, B ::= P \mid \Pi x{:}A.\, B & \text{normal type families} \end{array}$$

The operation of hereditary substitution, written $[N/x]_A$, is a partial function which computes the normal form of the standard capture-avoiding substitution of

N for x. It is indexed by the putative type of x, A, to ensure termination, but neither the variable x nor the substituted term N are required to bear any relation to this type index for the operation to be defined. We show in Section 3 that when N and x *do* have type A, hereditary substitution is a total function on well-formed terms.

Our layer of refinements uses metavariables Q for atomic sorts and S for normal sorts. These mirror the definition of types above, except for the addition of intersection and "top" sorts.

$$\begin{array}{rll} Q & ::= s \mid Q\,N & \text{atomic sort families} \\ S, T & ::= Q \mid \Pi x{::}S{\sqsubset}A.\, T \mid \top \mid S_1 \wedge S_2 & \text{normal sort families} \end{array}$$

Sorts are related to types by a refinement relation, $S \sqsubset A$ ("S refines A"), discussed below. A term of type A can be assigned a sort S only when $S \sqsubset A$. We occasionally omit the "$\sqsubset A$" from function sorts when it is clear from context.

2.1 *Example: Natural Numbers*

For the first running example we will use the natural numbers in unary notation. In LF, they would be specified as follows

$$nat : \mathsf{type}. \qquad zero : nat. \qquad succ : nat \to nat.$$

Suppose we would like to distinguish the odd and the even numbers as refinements of the type of all numbers.

$$even \sqsubset nat. \qquad odd \sqsubset nat.$$

The form of the declaration is $s \sqsubset a$ where a is a type family already declared and s is a new sort family. Sorts headed by s are declared in this way to refine types headed by a. The relation $S \sqsubset A$ is extended through the whole sort hierarchy in a compositional way.

Next we declare the sorts of the constructors. For zero, this is easy:

$$zero :: even.$$

The general form of this declaration is $c :: S$, where c is a constant already declared in the form $c : A$, and where $S \sqsubset A$. The declaration for the successor is slightly more difficult, because it maps even numbers to odd numbers and vice versa. In order to capture both properties simultaneously we need to use *intersection sorts*, written as $S_1 \wedge S_2$. [3]

$$succ :: even \to odd \;\wedge\; odd \to even.$$

In order for an intersection to be well-formed, both components must refine the same type. The nullary intersection $\top$ can refine any type, and represents the maximal refinement of that type. [4]

$$\frac{s \sqsubset a \in \Sigma}{s\,N_1 \ldots N_k \sqsubset a\,N_1 \ldots N_k} \qquad \frac{S \sqsubset A \qquad T \sqsubset B}{\Pi x{::}S.\, T \sqsubset \Pi x{:}A.\, B} \qquad \frac{S_1 \sqsubset A \qquad S_2 \sqsubset A}{S_1 \wedge S_2 \sqsubset A} \qquad \frac{}{\top \sqsubset A}$$

[3] Intersection has lower precedence than arrow.

[4] As usual in LF, we use $A \to B$ as shorthand for the dependent type $\Pi x{:}A.\, B$ when x does not occur in B.

Canonical LF	LF with Refinements
$\dfrac{\Gamma, x{:}A \vdash N \Leftarrow B}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{:}A.\, B}$	$\dfrac{\Gamma, x{::}S \sqsubset A \vdash N \Leftarrow T}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{::}S \sqsubset A.\, T}$ **(Π-I)**
$\dfrac{\Gamma \vdash R \Rightarrow P' \qquad P' = P}{\Gamma \vdash R \Leftarrow P}$	$\dfrac{\Gamma \vdash R \Rightarrow Q' \qquad Q' \leq Q}{\Gamma \vdash R \Leftarrow Q}$ **(switch)**
$\dfrac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$ $\qquad$ $\dfrac{c{:}A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$	$\dfrac{x{::}S \sqsubset A \in \Gamma}{\Gamma \vdash x \Rightarrow S}$ **(var)** $\qquad$ $\dfrac{c :: S \in \Sigma}{\Gamma \vdash c \Rightarrow S}$ **(const)**
$\dfrac{\Gamma \vdash R \Rightarrow \Pi x{:}A.\, B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash R\, N \Rightarrow [N/x]_A\, B}$	$\dfrac{\Gamma \vdash R \Rightarrow \Pi x{::}S \sqsubset A.\, T \qquad \Gamma \vdash N \Leftarrow S}{\Gamma \vdash R\, N \Rightarrow [N/x]_A\, T}$ **(Π-E)**

To show that the declaration for *succ* is well-formed, we establish that $\mathit{even} \to \mathit{odd} \wedge \mathit{odd} \to \mathit{even} \sqsubset \mathit{nat} \to \mathit{nat}$.

The *refinement relation* $S \sqsubset A$ should not be confused with the usual *subtyping relation*. Although each is a kind of subset relation, they are quite different: Subtyping relates two types, is contravariant in the domains of function types, and is transitive, while refinement relates a sort to a type, so it does not make sense to consider its variance or whether it is transitive. We will discuss subtyping below and in Section 4.

Now suppose that we also wish to distinguish the strictly positive natural numbers. We can do this by introducing a sort *pos* refining *nat* and declaring that the successor function yields a *pos* when applied to anything, using the maximal sort.

$\mathit{pos} \sqsubset \mathit{nat}. \qquad \mathit{succ} :: \cdots \wedge \top \to \mathit{pos}.$

Since we only sort-check well-typed programs and *succ* is declared to have type $\mathit{nat} \to \mathit{nat}$, the sort $\top$ here acts as a sort-level reflection of the entire *nat* type.

We can specify that all odds are positive by declaring *odd* to be a subsort of *pos*.

$\mathit{odd} \leq \mathit{pos}.$

Although any ground instance of *odd* is evidently *pos*, we need the subsorting declaration to establish that variables of sort *odd* are also *pos*.

Now we should be able to verify that, for example, *succ* (*succ zero*) $\Leftarrow$ *even*. To explain how, we analogize with pure canonical LF. Recall that atomic types have the form $a\ N_1 \ldots N_k$ for a type family a and are denoted by P. Arbitrary types A are either atomic (P) or (dependent) function types ($\Pi x{:}A.\, B$). Canonical terms are then characterized by the rules shown in the left column above.

There are two typing judgements, $N \Leftarrow A$ which means that N checks against A (both given) and $R \Rightarrow A$ which means that R synthesizes type A (R given as input, A produced as output). Both take place in a context Γ assigning types to variables. To force terms to be η-long, the rule for checking an atomic term R only checks it at an atomic type P. It does so by synthesizing a type P' and comparing it to the

given type P. In canonical LF, all types are already canonical, so this comparison is just α-equality.

On the right-hand side we have shown the corresponding rules for sorts. First, note that the format of the context Γ is slightly different, because it declares sorts for variables, not just types. The rules for functions and applications are straightforward analogues to the rules in ordinary LF. The rule **switch** for checking atomic terms R at atomic sorts Q replaces the equality check with a subsorting check and is the only place where we appeal to subsorting (defined below). For applications, we use the type A that refines the type S as the index parameter of the hereditary substitution.

Subsorting is exceedingly simple: it only needs to be defined on atomic sorts, and is just the reflexive and transitive closure of the declared subsorting relationship.

$$\frac{s_1 \leq s_2 \in \Sigma}{s_1\ N_1 \ldots N_k \leq s_2\ N_1 \ldots N_k} \qquad \frac{}{Q \leq Q} \qquad \frac{Q_1 \leq Q' \qquad Q' \leq Q_2}{Q_1 \leq Q_2}$$

The sorting rules do not yet treat intersections. In line with the general bidirectional nature of the system, the introduction rules are part of the *checking* judgement, and the elimination rules are part of the *synthesis* judgement.

$$\frac{\Gamma \vdash N \Leftarrow S_1 \qquad \Gamma \vdash N \Leftarrow S_2}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2}\ (\wedge\textbf{-I}) \qquad \frac{}{\Gamma \vdash N \Leftarrow \top}\ (\top\textbf{-I})$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_1}\ (\wedge\textbf{-E}_1) \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_2}\ (\wedge\textbf{-E}_2)$$

Note that although LF type systhesis is unique, sort synthesis is not, due to the intersection elimination rules.

Now we can see how these rules generate a deduction of *succ* (*succ zero*) $\Leftarrow$ *even*. The context is always empty and therefore omitted. To save space, we abbreviate *even* as e, *odd* as o, *pos* as p, *zero* as z, and *succ* as s, and we omit reflexive uses of subsorting.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash s \Rightarrow e \to o \wedge (o \to e \wedge \top \to p)}}{\vdash s \Rightarrow o \to e \wedge \top \to p}}{\vdash s \Rightarrow o \to e} \qquad \dfrac{\dfrac{\dfrac{}{\vdash s \Rightarrow e \to o \wedge (\ldots)}}{\vdash s \Rightarrow e \to o} \qquad \dfrac{}{\vdash z \Rightarrow e}}{\vdash s\ z \Rightarrow o} \qquad \dfrac{}{\vdash z \Leftarrow e}}{\vdash s\ z \Leftarrow o}}{\vdash s\ (s\ z) \Rightarrow e}}{\vdash s\ (s\ z) \Leftarrow e}$$

Using the $\wedge$**-I** rule, we can check that *succ zero* is both odd and positive:

$$\frac{\vdots}{\vdash s\ z \Leftarrow o} \qquad \frac{\vdots}{\vdash s\ z \Leftarrow p}$$
$$\vdash s\ z \Leftarrow o \wedge p$$

Each remaining subgoal now proceeds similarly to the above example.

To illustrate the use of sorts with non-trivial type *families*, consider the definition of *double* in LF.

$\mathit{double} : \mathit{nat} \to \mathit{nat} \to \mathsf{type}.$
$\mathit{dbl\text{-}zero} : \mathit{double}\ \mathit{zero}\ \mathit{zero}.$
$\mathit{dbl\text{-}succ} : \Pi X{:}\mathit{nat}.\, \Pi Y{:}\mathit{nat}.\, \mathit{double}\ X\ Y \to \mathit{double}\ (\mathit{succ}\ X)\ (\mathit{succ}\ (\mathit{succ}\ Y)).$

With sorts, we can now directly express the property that the second argument to *double* must be even. But to do so, we require a notion analogous to *kinds* that may contain sort information. We call these *classes* and denote them by L.

$$\begin{array}{lr} K ::= \mathsf{type} \mid \Pi x{:}A.\, K & \text{kinds} \\ L ::= \mathsf{type} \mid \Pi x{::}S \sqsubset A.\, L \mid \top \mid L_1 \wedge L_2 & \text{classes} \end{array}$$

Classes L mirror kinds K, and they have a refinement relation $L \sqsubset K$ similar to $S \sqsubset A$. (We elide the rules here.) Now, the general form of the $s \sqsubset a$ declaration is $s \sqsubset a :: L$, where $a : K$ and $L \sqsubset K$; this declares sort constant s to refine type constant a and to have class L.

We reuse the type name *double* as a sort, as no ambiguity can result. As before, we use $\top$ to represent a *nat* with no additional restrictions.

$\mathit{double} \sqsubset \mathit{double} :: \top \to \mathit{even} \to \mathsf{type}.$
$\mathit{dbl\text{-}zero} :: \mathit{double}\ \mathit{zero}\ \mathit{zero}.$
$\mathit{dbl\text{-}succ} :: \Pi X{::}\top.\, \Pi Y{::}\mathit{even}.\, \mathit{double}\ X\ Y \to \mathit{double}\ (\mathit{succ}\ X)\ (\mathit{succ}\ (\mathit{succ}\ Y)).$

After these declarations, it would be a *sort error* to pose a query such as "`?-` *double X* (*succ* (*succ* (*succ zero*)))." before any search is ever attempted. In LF, queries like this could fail after a long search or even not terminate, depending on the search strategy.

The tradeoff for such precision is that now sort checking itself is non-deterministic and has to perform search because of the choice between the two intersection elimination rules. As Reynolds has shown, this non-determinism causes intersection type checking to be PSPACE-hard [16], even for normal terms as we have here [15]. Using techniques such as focusing, we believe that for practical cases they can be analyzed efficiently for the purpose of sort checking. [5]

2.2 *A Second Example: The λ-Calculus*

As a second example, we use an intrinsically typed version of the call-by-value simply-typed λ-calculus. This means every object language expression is indexed by its object language type. We use sorts to distinguish the set of *values* from the set of arbitrary *computations*. While this can be encoded in LF in a variety of ways, it is significantly more cumbersome.

```
tp : type.          % the type of object language types
➪ : tp → tp → tp.   % object language function space
%infix right 10 ➪ .
```

[5] The present paper concentrates primarily on decidability, though, not efficiency.

$exp : tp \to \mathsf{type}.$ % the type of expressions
$cmp \sqsubset exp.$ % the sort of computations
$val \sqsubset exp.$ % the sort of values
$val \leq cmp.$ % every value is a (trivial) computation

$lam :: (val\ A \to cmp\ B) \to val\ (A \Rightarrow B).$
$app :: cmp\ (A \Rightarrow B) \to cmp\ A \to cmp\ B.$

In the last two declarations, we follow Twelf convention and leave the quantification over A and B implicit, to be inferred by type reconstruction. Also, we did not explicitly declare a type for *lam* and *app*. We posit a front end that can recover this information from the refinement declarations for *val* and *cmp*, avoiding redundancy.

The most interesting declaration is the one for the constant *lam*. The argument type $(val\ A \to cmp\ B)$ indicates that *lam* binds a variable which stands for a value of type A and the body is an arbitrary computation of type B. The result type $val\ (A \Rightarrow B)$ indicates that any λ-abstraction is a value. Now we have, for example (parametrically in A and B): $A{::}\top \sqsubset tp, B{::}\top \sqsubset tp \vdash lam\ \lambda x.\, lam\ \lambda y.\, x \Leftarrow val\ (A \Rightarrow (B \Rightarrow A))$.

Now we can express that evaluation must always returns a value. Since the declarations below are intended to represent a logic program, we follow the logic programming convention of reversing the arrows in the declaration of *ev-app*.

$eval :: cmp\ A \to val\ A \to \mathsf{type}.$
$ev\text{-}lam :: eval\ (lam\ \lambda x.\, E\ x)\ (lam\ \lambda x.\, E\ x).$
$ev\text{-}app :: \Pi E_1' {::} (val\ A \to cmp\ A).$
$\quad eval\ (app\ E_1\ E_2)\ V$
$\quad \leftarrow eval\ E_1\ (lam\ \lambda x.\, E_1'\ x)$
$\quad \leftarrow eval\ E_2\ V_2$
$\quad \leftarrow eval\ (E_1'\ V_2)\ V.$

Sort checking the above declarations demonstrates that evaluation always returns a value. Moreover, due to the explicit sort given for E_1', the declarations also ensure that the language is indeed call-by-value: it would be a sort error to ever substitute a computation for a *lam*-bound variable, for example, by evaluating $(E_1'\ E_2)$ instead of $(E_1'\ V_2)$ in the *ev-app* rule. An interesting question for future work is whether type reconstruction can recover this restriction automatically—if the front end were to assign E_1' the "more precise" sort $cmp\ A \to cmp\ A$, then the check would be lost.

2.3 A Final Example: The Calculus of Constructions

As a final example, we present the Calculus of Constructions. Usually, there is a great deal of redundancy in its presentation because of repeated constructs at the level of objects, families, and kinds. Using sorts, we can enforce the stratification and write typing rules that are as simple as if we assumed the infamous *type* : *type*.

$term : \mathsf{type}.$ % terms at all levels

$hyp \sqsubset term.$ % hyperkinds (the classifier of "kind")
$knd \sqsubset term.$ % kinds
$fam \sqsubset term.$ % families
$obj \sqsubset term.$ % objects

$tp :: hyp \wedge knd.$
$pi :: fam \to (obj \to fam) \to fam \wedge$ % dependent function types, $\Pi x{:}A.\, B$
$\quad fam \to (obj \to knd) \to knd \wedge$ % type family kinds, $\Pi x{:}A.\, K$
$\quad knd \to (fam \to fam) \to fam \wedge$ % polymorphic function types, $\forall \alpha{:}K.\, A$
$\quad knd \to (fam \to knd) \to knd.$ % type operator kinds, $\Pi \alpha{:}K_1.\, K_2$
$lm :: fam \to (obj \to obj) \to obj \wedge$ % functions, $\lambda x{:}A.\, M$
$\quad fam \to (obj \to fam) \to fam \wedge$ % type families, $\lambda x{:}A.\, B$
$\quad knd \to (fam \to obj) \to obj \wedge$ % polymorphic abstractions, $\Lambda \alpha{:}K.\, M$
$\quad knd \to (fam \to fam) \to fam.$ % type operators, $\lambda \alpha{:}K.\, A$
$ap :: obj \to obj \to obj \wedge$ % ordinary application, $M\ N$
$\quad fam \to obj \to fam \wedge$ % type family application, $A\ M$
$\quad obj \to fam \to obj \wedge$ % polymorphic instantiation, $M\ [A]$
$\quad fam \to fam \to fam.$ % type operator instantiation, $A\ B$

The typing rules can now be given non-redundantly, illustrating the implicit overloading afforded by the use of intersections. We omit the type conversion rule and auxiliary judgements for brevity.

$of :: knd \to hyp \to \mathsf{type} \wedge$
$\quad fam \to knd \to \mathsf{type} \wedge$
$\quad obj \to fam \to \mathsf{type}.$

$of\text{-}tp :: of\ tp\ tp.$

$of\text{-}pi :: of\ (pi\ T_1\ \lambda x.\, T_2\ x)\ tp$
$\quad \leftarrow of\ T_1\ tp$
$\quad \leftarrow (\Pi x{::}term.\, of\ x\ T_1 \to of\ (T_2\ x)\ tp).$
$of\text{-}lm :: of\ (lm\ U_1\ \lambda x.\, T_2\ x)\ (pi\ U_1\ \lambda x.\, U_2\ x)$
$\quad \leftarrow of\ U_1\ tp$
$\quad \leftarrow (\Pi x{::}term.\, of\ x\ U_1 \to of\ (T_2\ x)\ (U_2\ x)).$
$of\text{-}ap :: of\ (ap\ T_1\ T_2)\ (U_1\ T_2)$
$\quad \leftarrow of\ T_1\ (pi\ U_2\ \lambda x.\, U_1\ x)$
$\quad \leftarrow of\ T_2\ U_2.$

Intersection types also provide a degree of modularity: by deleting some conjuncts from the declarations of *pi*, *lm*, and *ap* above, we can obtain an encoding of any point on the λ-cube.

3 Metatheory

In this section, we present some metatheoretic results about our framework. These follow a similar pattern as previous work using hereditary substitutions [17,11,7]. To conserve space, we omit proofs that are similar to those from prior work, and

only sketch novel results. We refer the interested reader to the companion technical report [10], which contains complete proofs of all thoerems.

3.1 Hereditary Substitution

Hereditary substitution is defined judgementally by inference rules. The only place β-redexes might be introduced is when substituting a normal term N into an atomic term R: N might be a λ-abstraction, and the variable being substituted for may occur at the head of R. Therefore, the judgements defining substitution into atomic terms are the only ones of interest.

First, we note that the type index on hereditary substitution need only be a simple type to ensure termination. To that end, we denote simple types by α and define an erasure to simple types $(A)^-$.

$$\alpha ::= a \mid \alpha_1 \to \alpha_2 \qquad (a\ N_1 \ldots N_k)^- = a \qquad (\Pi x{:}A.\, B)^- = (A)^- \to (B)^-$$

We write $[N/x]^{\mathrm{n}}_A\, M = M'$ as short-hand for $[N/x]^{\mathrm{n}}_{(A)^-}\, M = M'$.

We denote substitution into atomic terms by two judgements: $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R'$, for when the head of R is *not* x, and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N', \alpha')$, for when the head of R *is* x. The former is just defined compositionally; the latter is defined by two rules:

$$\frac{}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, x_0 = (N_0, \alpha_0)}\ \textbf{(rn-var)} \qquad \frac{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_1 = (\lambda x.\, N_1, \alpha_2 \to \alpha_1) \quad [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2' \quad [N_2'/x]^{\mathrm{n}}_{\alpha_2}\, N_1 = N_1'}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_1\ N_2 = (N_1', \alpha_1)}\ \textbf{(rn-}\beta\textbf{)}$$

The rule **rn-var** just returns the substitutend N_0 and its putative type index α_0. The rule **rn-**β applies when the result of substituting into the head of an application is a λ-abstraction; it avoids creating a redex by hereditarily substituting into the body of the abstraction.

A simple lemma establishes that these two judgements are mutually exclusive.

Lemma 3.1

(i) *If* $[N/x]^{\mathrm{rr}}_A\, R = R'$, *then the head of* R *is not* x.

(ii) *If* $[N/x]^{\mathrm{rn}}_A\, R = (N', \alpha')$, *then the head of* R *is* x.

Proof. By induction over the given derivation. □

Substitution into normal terms has two rules for atomic terms R, one which calls the "rr" judgement and one which calls the "rn" judgement.

$$\frac{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R'}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R = R'}\ \textbf{(subst-n-atom)} \qquad \frac{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (R', a')}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R = R'}\ \textbf{(subst-n-atomhead)}$$

Note that the latter rule requires both the term and the type returned by the "rn" judgement to be atomic.

Every other syntactic category's substitution judgement is defined compositionally.

3.2 Decidability

A hallmark of the canonical forms/hereditary substitution approach is that it allows a decidability proof to be carried out comparitively early, before proving anything about the behavior of substitution, and without dealing with any complications introduced by β/η-conversions inside types. Ordinarily in a dependently typed calculus, one must first prove a substitution theorem before proving typechecking decidable. (See [8] for a typical non-canonical account of LF definitional equality.)

If only canonical forms are permitted, then type equality is just α-convertibility, so one only needs to show decidability of substitution in order to show decidability of typechecking. Since LF encodings represent judgements as type families and proof-checking as typechecking, it is comforting to have a decidability proof that relies on so few assumptions.

Lemma 3.2 *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$, *then* α' *is a subterm of* α_0.

Proof. By induction on the derivation of $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$. In rule **rn-var**, α' is the same as α_0. In rule **rn-β**, our inductive hypothesis tells us that $\alpha_2 \to \alpha_1$ is a subterm of α_0, so α_1 is as well. □

Theorem 3.3 (Decidability of Substitution) *Hereditary substitution is decidable. In particular:*

(i) *Given* N_0, x_0, α_0, *and* R, *either* $\exists R'.\, [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$, *or* $\nexists R'.\, [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$,

(ii) *Given* N_0, x_0, α_0, *and* R, *either* $\exists (N', \alpha').\, [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$, *or* $\nexists (N', \alpha').\, [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$,

(iii) *Given* N_0, x_0, α_0, *and* N, *either* $\exists N'.\, [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N'$, *or* $\nexists N'.\, [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N'$,

and similarly for other syntactic categories

Proof. By lexicographic induction on the type subscript α_0, the main subject of the substitution judgement, and the clause number. For each rule defining hereditary substitution, the premises are at a smaller type subscript, or if the same type subscript, then a smaller term, or if the same term, then an earlier clause. The case for rule **rn-β** relies on Lemma 3.2 to know that α_2 is a strict subterm of α_0. □

Theorem 3.4 (Decidability of Subsorting) *Given* Q_1 *and* Q_2, *it is decidable whether or not* $Q_1 \leq Q_2$.

Proof. Since the subsorting relation $Q_1 \leq Q_2$ is just the reflexive, transitive closure of the declared subsorting relation $s_1 \leq s_2$, it suffices to compute this closure and check whether the heads of Q_1 and Q_2 are related by it. □

We prove decidability of typing by exhibiting a deterministic algorithmic system that is equivalent to the original. Instead of synthesizing a single sort for an atomic term, the algorithmic system synthesizes an intersection-free list of sorts, Δ.

$$\Delta ::= \cdot \mid \Delta, Q \mid \Delta, \Pi x{::}S{\sqsubset}A.\, T$$

One can think of Δ as the intersection of all its elements. Instead of applying intersection eliminations, the algorithmic system eagerly breaks down intersections

using a "split" operator, leading to a deterministic "minimal-synthesis" system.

$$\begin{array}{cc} \mathrm{split}(Q) = Q & \mathrm{split}(S_1 \wedge S_2) = \mathrm{split}(S_1), \mathrm{split}(S_2) \\ \mathrm{split}(\Pi x{::}S \sqsubset A.\, T) = \Pi x{::}S \sqsubset A.\, T & \mathrm{split}(\top) = \cdot \end{array}$$

$$\frac{c{::}S \in \Sigma \qquad c{:}A \in \Sigma}{\Gamma \vdash c \Rrightarrow \mathrm{split}(S)} \qquad \frac{x{::}S \sqsubset A \in \Gamma}{\Gamma \vdash x \Rrightarrow \mathrm{split}(S)} \qquad \frac{\Gamma \vdash R \Rrightarrow \Delta \qquad \Gamma \vdash \Delta \mathbin{@} N = \Delta'}{\Gamma \vdash R\, N \Rrightarrow \Delta'}$$

The rule for applications uses an auxiliary judgement $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ which computes the possible types of $R\, N$ given that R synthesizes to all the sorts in Δ. It has two key rules:

$$\frac{}{\Gamma \vdash \cdot \mathbin{@} N = \cdot} \qquad \frac{\Gamma \vdash \Delta \mathbin{@} N = \Delta' \qquad \Gamma \vdash N \Lleftarrow S \qquad [N/x]^{\mathrm{s}}_{A}\, T = T'}{\Gamma \vdash (\Delta, \Pi x{::}S \sqsubset A.\, T) \mathbin{@} N = \Delta', \mathrm{split}(T')}$$

The other rules force the judgement to be defined when neither of the above two rules apply. Finally, to tie everything together, we define a new checking judgement $\Gamma \vdash N \Lleftarrow S$ that makes use of the algorithmic synthesis judgement; it looks just like $\Gamma \vdash N \Leftarrow S$ except for the rule for atomic terms.

$$\frac{\Gamma \vdash R \Rrightarrow \Delta \qquad Q' \in \Delta \qquad Q' \leq Q}{\Gamma \vdash R \Lleftarrow Q}$$

This new algorithmic system is manifestly decidable.

Theorem 3.5 *Algorithmic type checking is decidable. In particular:*

(i) *Given* Γ *and* R*, there is a unique* Δ *such that* $\Gamma \vdash R \Rrightarrow \Delta$*.*

(ii) *Given* Γ*,* N*, and* S*, it is decidable whether or not* $\Gamma \vdash N \Lleftarrow S$*.*

(iii) *Given* Γ*,* Δ*, and* N*, there is a unique* Δ' *such that* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$*.*

Proof. By induction on the term, R or N, the clause number, and the sort S or the list of sorts Δ. For each rule, the premises are either known to be decidable, or at a smaller term, or if the same term, then an earlier clause, or if the same clause, then either a smaller S or a smaller Δ. □

Note that the algorithmic synthesis system *always* outputs *some* Δ; if the given term has no sort, then the output will be $\cdot$.

It is straightforward to show that the algorithm is sound and complete with respect to the original bidirectional system.

Theorem 3.6 (Soundness of Algorithmic Typing)

(i) *If* $\Gamma \vdash R \Rrightarrow \Delta$*, then for all* $S \in \Delta$*,* $\Gamma \vdash R \Rightarrow S$*.*

(ii) *If* $\Gamma \vdash N \Lleftarrow S$*, then* $\Gamma \vdash N \Leftarrow S$*.*

(iii) *If* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$*, and for all* $S \in \Delta$*,* $\Gamma \vdash R \Rightarrow S$*, then for all* $S' \in \Delta'$*,* $\Gamma \vdash R\, N \Rightarrow S'$*.*

Proof. By straightforward induction on the given derivation. □

Lemma 3.7 *If* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ *and* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\Pi x{::}S \sqsubset A.\, T \in \Delta$ *and* $\Gamma \vdash N \Lleftarrow S$ *and* $[N/x]^{\mathrm{s}}_{A}\, T = T'$, *then* $\mathrm{split}(T') \subseteq \Delta'$.

Proof. By straightforward induction on the derivation of $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$. □

Theorem 3.8 (Completeness for Algorithmic Typing)

(i) *If* $\Gamma \vdash R \Rightarrow S$, *then* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathrm{split}(S) \subseteq \Delta$.

(ii) *If* $\Gamma \vdash N \Leftarrow S$, *then* $\Gamma \vdash N \Lleftarrow S$.

Proof. By straightforward induction on the given derivation. In the application case, we make use of the fact that $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ is always defined and apply Lemma 3.7. □

Decidability theorems and proofs for other syntactic categories' formation judgements are similar, so we omit them.

3.3 Identity and Substitution Principles

Since well-typed terms in our framework must be canonical, that is β-normal and η-long, it is non-trivial to prove $S \to S$ for non-atomic S, or to compose proofs of $S_1 \to S_2$ and $S_2 \to S_3$. The Identity and Substitution principles ensure that our type theory makes logical sense by demonstrating the reflexivity and transitivity of entailment. Reflexivity is witnessed by η-expansion, while transitivity is witnessed by hereditary substitution.

The Identity Principle effectively says that synthesizing (atomic) objects can be made to serve as checking (normal) objects. The Substitution Principle dually says that checking objects may stand in for synthesizing assumptions, that is, variables.

Theorem 3.9 (Substitution) *If* $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$, *and* $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}}\ \mathsf{ctx}$, *and* $\Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}} \vdash S \sqsubset A$, *and* $\Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$, *then* $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ *and* $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}\ \mathsf{ctx}$, *and* $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ *and* $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash S' \sqsubset A'$, *and* $[N_0/x_0]^{\mathrm{n}}_{A_0}\, N = N'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$, *and similarly for other syntactic categories.*

Proof. The staging of the substitution theorem is somewhat intricate. First, we strengthen its statement to one that does not presuppose the well-formedness of the context or the classifying types, but instead presupposes that substitution is defined on them. This strengthened statement may be proven by induction on $(A_0)^-$ and the derivations being substituted into. In the application case, we require a lemma about how hereditary substitutions compose, analogous to the fact that for ordinary substitution, $[N_0/x_0]\,[N_2/x_2]\,N = [[N_0/x_0]\,N_2/x_2]\,[N_0/x_0]\,N$. □

A more in-depth discussion of the proof of substitution for core canonical LF can be found in [7]. The story for LFR is quite similar, and is detailed in the companion technical report [10].

Theorem 3.10 (Expansion) *If* $\Gamma \vdash S \sqsubset A$ *and* $\Gamma \vdash R \Rightarrow S$, *then* $\Gamma \vdash \eta_A(R) \Leftarrow S$.

Proof. By induction on S. The $\Pi x{:}A_2.\, A_1$ case relies on the auxiliary fact that $[\eta_{A_2}(x)/x]^{\mathrm{s}}_{A_2}\, S_1 = S_1$. □

Corollary 3.11 (Identity) *If* $\Gamma \vdash S \sqsubset A$, *then* $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow S$.

4 Subsorting at Higher Sorts

Our bidirectional typing discipline limits subsorting checks to a single rule, the **switch** rule when we switch modes from checking to synthesis. Since we insist on only typing canonical forms, this rule is limited to atomic sorts Q, and consequently, subsorting need only be defined on atomic sorts.

As it turns out, though, the usual variance principles and structural rules for subsorting at higher sorts are admissible with respect to an intrinsic notion of higher-sort subsorting. The simplest way of formulating this intrinsic notion is as a variant of the identity principle: S is a subtype of T if $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$. This notion is equivalent to a number of other alternate formulations, including a subsumption-based formulation and a substitution-based formulation.

Theorem 4.1 (Alternate Formulations of Subsorting) *The following are equivalent:*

(i) *If* $\Gamma \vdash R \Rightarrow S_1$, *then* $\Gamma \vdash \eta_A(R) \Leftarrow S_2$.

(ii) $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$.

(iii) *If* $\Gamma \vdash N \Leftarrow S_1$, *then* $\Gamma \vdash N \Leftarrow S_2$.

(iv) *If* $\Gamma_{\mathrm{L}}, x{::}S_2 \sqsubset A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ *and* $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_1$, *then* $\Gamma_{\mathrm{L}}, [N_1/x]^{\gamma}_A \Gamma_{\mathrm{R}} \vdash [N_1/x]^{\mathrm{n}}_A N \Leftarrow [N_1/x]^{\mathrm{s}}_A S$.

Proof. Using Identity and Substitution, and the fact that $[N/x]^{\mathrm{n}}_A \eta_A(x) = N$.

i $\Longrightarrow$ *ii*: By rule, $\Gamma, x{::}S_1 \sqsubset A \vdash x \Rightarrow S_1$. By i, $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$.

ii $\Longrightarrow$ *iii*: Suppose $\Gamma \vdash N \Leftarrow S_1$. By ii, $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$. By Theorem 3.9 (Substitution), $\Gamma \vdash [N/x]^{\mathrm{n}}_A \eta_A(x) \Leftarrow S_2$. Thus, $\Gamma \vdash N \Leftarrow S_2$.

iii $\Longrightarrow$ *iv*: Suppose $\Gamma_{\mathrm{L}}, x{::}S_2 \sqsubset A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ and $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_1$. By iii, $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_2$. By Theorem 3.9 (Substitution), $\Gamma_{\mathrm{L}}, [N_1/x]^{\gamma}_A \Gamma_{\mathrm{R}} \vdash [N_1/x]^{\mathrm{n}}_A N \Leftarrow [N_1/x]^{\mathrm{s}}_A S$.

iv $\Longrightarrow$ *i*: Suppose $\Gamma \vdash R \Rightarrow S_1$. By Theorem 3.10 (Expansion), $\Gamma \vdash \eta_A(R) \Leftarrow S_1$. By Corollary 3.11 (Identity), $\Gamma, x{::}S_2 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$. By iv, $\Gamma \vdash [\eta_A(R)/x]^{\mathrm{n}}_A \eta_A(x) \Leftarrow S_2$. Thus, $\Gamma \vdash \eta_A(R) \Leftarrow S_2$. □

All of the rules in Fig. 1 are sound with respect to this intrinsic notion of subsorting.

Theorem 4.2 *If* $S \leq T$, *then* $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$.

Proof. By induction, making use of the alternate formulations given by Theorem 4.1. □

The soundness of the rules in Fig. 1 demonstrates that any subsumption relationship you might want to capture with them is already captured by our checking and synthesis rules. More interesting is the fact that the usual rules are *complete* with respect to our intrinsic notion. Space limitations preclude more than a brief overview here; the companion technical report contains a detailed account.

We demonstrate completeness by appeal to an algorithmic subtyping system very similar to the algorithmic typing system from Section 3.2. This system is characterized by two judgements: $\Delta \leq S$ and $\Delta \mathbin{@} (N :: \Delta_1) = \Delta_2$. With the

$$\boxed{S_1 \leq S_2}$$

$$\frac{}{S \leq S}\ \textbf{(refl)} \qquad \frac{S_1 \leq S_2 \qquad S_2 \leq S_3}{S_1 \leq S_3}\ \textbf{(trans)} \qquad \frac{S_2 \leq S_1 \qquad T_1 \leq T_2}{\Pi x{::}S_1.\, T_1 \leq \Pi x{::}S_2.\, T_2}\ \textbf{(S-}\Pi\textbf{)}$$

$$\frac{}{S \leq \top}\ \textbf{(}\top\textbf{-R)} \qquad \frac{T \leq S_1 \qquad T \leq S_2}{T \leq S_1 \wedge S_2}\ \textbf{(}\wedge\textbf{-R)} \qquad \frac{S_1 \leq T}{S_1 \wedge S_2 \leq T}\ \textbf{(}\wedge\textbf{-L}_1\textbf{)} \qquad \frac{S_2 \leq T}{S_1 \wedge S_2 \leq T}\ \textbf{(}\wedge\textbf{-L}_2\textbf{)}$$

$$\frac{}{\top \leq \Pi x{::}S.\, \top}\ \textbf{(}\top/\Pi\textbf{-dist)} \qquad \frac{}{(\Pi x{::}S.\, T_1) \wedge (\Pi x{::}S.\, T_2) \leq \Pi x{::}S.\, (T_1 \wedge T_2)}\ \textbf{(}\wedge/\Pi\textbf{-dist)}$$

Fig. 1. Derived structural rules for subsorting.

appropriate definition, we can prove the following by induction on the type A and the derivation $\mathcal{E}$.

Theorem 4.3 *Suppose* $\Gamma \vdash R \Rightarrow A$. *Then:*

(i) *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathcal{E} :: \Gamma \vdash \eta_A(R) \Lleftarrow S$, *then* $\Delta \leq S$.

(ii) *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathcal{E} :: \Gamma \vdash \Delta_0 \mathbin{@} \eta_A(R) = \Delta'$, *then* $\Delta_0 \mathbin{@} (\eta_A(R) :: \Delta) = \Delta'$.

From this and Theorem 3.8 we obtain a completeness theorem:

Theorem 4.4 *If* $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Lleftarrow T$, *then* $\mathrm{split}(S) \leq T$.

Finally, we can complete the triangle by showing that the algorithmic formulation of subtyping implies the original declarative formulation:

Theorem 4.5 *If* $\mathrm{split}(S) \leq T$, *then* $S \leq T$.

5 Related Work

The most closely related work is [12], which also sought to extend LF with refinement types. We improve upon that work by intrinsically supporting a notion of canonical form. Also closely related in Aspinall and Compagnoni's work on subtyping and dependent types [2,1]. The primary shortcoming of their work is its lack of intersection types, which are essential for even the simplest of our examples.

6 Summary

In summary, we have exhibited a variant of the logical framework LF with a notion of subtyping based on refinement types. We have demonstrated the expressive power of this extension through a number of realistic examples, and we have shown several metatheoretic properties critical to its utility as a logical framework, including decidability of typechecking.

Our development was drastically simplified by the decision to admit only canonical forms. One effect of this choice was that subsorting was only required to

be judgementally defined at base sorts; higher-sort subsorting was derived through an η-expansion-based definition which we showed sound and complete with respect to the usual structural subsorting rules.

There are a number of avenues of future exploration. For one, it is unclear how subsorting and intersection sorts will interact with the typical features of a metalogical framework, including type reconstruction, unification, and proof search, to name a few; these questions will have to be answered before refinement types can be integrated into a practical implementation. It is also worthwhile to consider adapting the refinement system to more expressive frameworks, like the Linear Logical Framework on the Concurrent Logical Framework.

References

[1] David Aspinall. Subtyping with power types. In Peter Clote and Helmut Schwichtenberg, editors, *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2000.

[2] David Aspinall and Adriana B. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2):273–309, 2001.

[3] Karl Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th Annual Symposium on Principles of Programming Languages (POPL '03)*, pages 198–212, New Orleans, Louisiana, January 2003. ACM Press.

[4] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, May 2005. Available as Technical Report CMU-CS-05-110.

[5] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110.

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

[7] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 2007. To appear. Available from `http://www.cs.cmu.edu/~drl/`.

[8] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 6:61–101, January 2005.

[9] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In Matthias Felleisen, editor, *Proceedings of the 34th Annual Symposium on Principles of Programming Languages (POPL '07)*, pages 173–184, Nice, France, January 2007. ACM Press.

[10] William Lovas and Frank Pfenning. A bidirectional refinement type system for LF. Technical Report CMU-CS-07-127, Department of Computer Science, Carnegie Mellon University, 2007.

[11] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

[12] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[13] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[14] Frank Pfenning and Carsten 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)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[15] John C. Reynolds. Even normal forms can be hard to type. Unpublished, marked Carnegie Mellon University, December 1, 1989.

[16] John C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28, 1996.

[17] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Coercive subtyping via Mappings of Reduction Behaviour

Paul Callaghan[1,2]

Department of Computer Science
University of Durham
Durham, UK.

Abstract

This paper reports preliminary work on a novel approach to Coercive Subtyping that is based on relationships between reduction behaviour of source and target types in coerced terms. Coercive Subtyping is a superset of record-based subtyping, allowing so-called coercion functions to carry the subtyping. This allows many novel and powerful forms of subtyping and abbreviation, with applications including interfaces to theorem provers and programming with dependent type systems. However, the use of coercion functions introduces non-trivial overheads, and requires difficult proof of properties such as coherence in order to guarantee sensible results. These points restrict the practicality of coercive subtyping.

We begin from the idea that coercing a value v from type U to a type T intuitively means that we wish to compute with v as if it was a value in T, not that v must be converted into a value in T. Instead, we explore how to compute on U in terms of computation on T, and develop a framework for mapping computations on some T to computations on some U via a simple extension of the elimination rule of T. By exposing how computations on different types are related, we gain insight on and make progress with several aspects of coercive subtyping, including (a) distinguishing classes of coercion and finding reasons to deprecate use of some classes; (b) alternative techniques for proving key properties of coercions; (c) greater efficiency from implementations of coercions.

Keywords: Coercive subtyping, type theory, inductive families, implementation.

1 Introduction

Coercive subtyping is an abbreviation mechanism which handles mismatches of type between a value and its context of use with implicit insertion of functions to bridge the gap. It has many natural uses, such as providing a form of record subtyping on nested algebraic structures or conversions between simple types. In the context of dependent types and inductive families, coercions can be used for many other interesting purposes, not least modelling of semantics of words in natural languages and providing a bridge between simply and dependently typed data. As a running example, consider the coercions between conventional lists and sized vectors. A

[1] This work is supported by the TYPES Working Group, a coordination action in the EU 6'th Framework programme.

[2] Email: mailto:p.c.callaghan@durham.ac.uk

vector can be used wherever a list is expected by implicitly coercing the vector with function $v2l$, or in the reverse direction with $l2v$ (though note the dependency to find the size of the vector):

$$v2l \ : \ (A : \mathit{Type})(n : \mathbb{N}) \ \mathit{Vec} \ A \ n \rightarrow \mathit{List} \ A$$

$$l2v \ : \ (A : \mathit{Type}) \qquad (l : \mathit{List} \ A) \ \mathit{Vec} \ A \ (\mathit{length} \ A \ l)$$

However, the use of coercions relies on non-trivial proofs of key properties such as coherence (uniqueness of result up to conversion) and transitivity elimination. Progress has been made on some sub-classes of coercions (between restricted groups of types), but establishing results *beyond* these classes and establishing them in a satisfactory (i.e., elegant and/or natural) way is an open problem. For practical purposes, such as flexible implementation within proof tools, such proofs should ideally be automatable and not require the user to provide difficult justifications. There are also issues of efficiency: the standard formulation implies some overheads during computation, e.g. conversion of at least some of a vector to a list before any computation can proceed. A final concern is with how intermediate computations on coerced values appear to the user of proof tools.

One view is that the standard formulation of coercive subtyping is too strong, in the sense that it allows very powerful coercion functions to be conceived but which are problematic for practical use. This view is based on the author's experience of earlier implementation and experimentation with many kinds of coercion function. The author also prefers that use of this intuitively 'natural' mechanism of coercion should not be limited by the need to consider or to understand non-trivial details of how classes of coercion interact. Too much complexity and the claim of being a flexible abbreviation mechanism is significantly weakened.

This paper explores one novel alternative. We start from the idea that coercion means that we wish to use a value v in source type U as if it was a value in target type T, or more specifically, that we wish to compute with v in terms of T. This intention does not force us to convert v into a representation in T. Instead, a mapping is constructed between the computation behaviours of U and of T in terms of the elimination operators of both types which allows easy transformation of operations on T into operations on U. This clearly avoids converting v to T, but we suggest there are several other important benefits, including (a) the links between source and target type are made explicit and are in terms which are fundamental to both types, so it is thus clearer why and how U can be treated as T; (b) making such information more explicit has several benefits in establishing key properties, not least simplifying some of the proofs; (c) it provides a framework to discuss and characterise different proposed coercions, and possibly to impose meaningful structure on sets of coercions.

The following summarises the key ideas and contributions. We first observe that summation on lists (of $\mathbb{N}$) can be translated to summation on vectors by modification of the arguments of the elimination operator. In fact, all functions on lists can be converted to functions on vectors in exactly the same way. The modification reflects how the types are related and this is independent of actual functions. We aim to use this relationship to allow computation on a source value in terms of the target type,

without converting the source value. A particular form of elimination operator is proposed to encode this relationship in a clear way: operators $\mathbf{E}_{XY}$ should take the parameter, motive, and branch function arguments of the eliminator for type X, but take a value in type Y as the elimination target, and should be defined in terms of $\mathbf{E}_Y$. For the vector-to-list example, this means:

$$\frac{C : List\ A \to Type \quad f_0 : C\ (nil_A) \quad f_1 : (a : A)(l : List\ A)C\ l \to C\ (cons_A\ a\ l)}{\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 : (n : \mathbb{N})(v : Vec\ A\ n)C\ (co_V\ A\ n\ v)}$$

$$\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 =_{\mathrm{df}} \mathbf{E}_V\ A\ ([n : \mathbb{N}][v : Vec\ A\ n]C\ (co_V\ A\ n\ v))\ f_0\ ([m : \mathbb{N}][x : A][v : Vec\ A\ m]f_1\ x\ (co_V\ A\ m\ v))$$

Which other combinations of types support the definition of a $\mathbf{E}_{XY}$ term? A conservative answer is the combinations whose conversion (from Y to X) is invertible. This covers many useful conversions, including functorial maps and natural conversions between different container types.

To use these terms, we propose that coercion becomes a two-stage process. Initially, a coerced term is marked (with a form of constructor), and when this reaches an eliminator of the target type, the $\mathbf{E}_{XY}$ term is used to map the computation to the source type. Hence no conversion (of value) takes place, and nothing happens until a computation acts on the coerced value. This process is clearly type-safe, although formal metatheory relating to canonicity and convertibility remain to be studied as further work. (Some preliminary remarks are made.) We briefly consider how the $\mathbf{E}_{XY}$ terms affect the proof of relevant properties. The key point is that composition via $\mathbf{E}_{XY}$ terms eventually reduces to elimination on the source type.

We first review the background type theory and some relevant earlier results on Coercive Subtyping. The central idea is presented through an expansion of the list and vector examples, then formalised as a relationship between elimination operators. Consequences of this formalisation are then explored, through further examples. The paper ends with comments on metatheory and implementation.

2 Preliminaries

2.1 Inductive families and their elimination

This paper concerns relationships between inductive families which are not tied to specific type theories, hence we assume a 'vanilla' dependent type theory. Briefly, there is a dependent product type $(x : K)K'$ (where x may occur free in K'), and we often write $K \to K'$ for dependent products with no dependency. *Type* is the sort of all types, i.e. $A : Type$ means A is a type. Notation $[x : K]k$ denotes λ-terms. The system includes an η rule, i.e. $[x : K]f\ x = f : (x : K)K',\ x \notin FV(f)$.

Inductive types [7,8,11] may be introduced through a schema [12], summarised as the grammar $\Theta = X \mid (x : K)\Theta \mid \Phi \to \Theta$ and $\Phi = (\overline{x} : \overline{K})X$. It identifies a class of inductive types which recurse through strictly positive operators and specifies how the elimination operators and computation rules are formed for each type. X is a placeholder for the type being defined. Small types K, K_i are those which don't contain *Type*. Inductive type T is constructed from a sequence $\overline{\Theta}$ which represents the types of T's constructors. The types of the constructors and the elimination

operator, and the computation rules for the constructors, are constructed by analysis of the schemata. Inductive types can also be *parametrized*, e.g. to give polymorphic lists. The type of the elimination operator and associated computation rules are as follows. (We adopt elements of natural deduction style from [17].)

$$\frac{C : List\ A \to Type \quad f_0 : C\ (nil\ A) \qquad f_1 : (x : A)(xs : List\ A)C\ xs \to C\ (cons\ A\ x\ xs)}{\mathbf{E}_L\ A\ C\ f_0\ f_1 : (z : List\ A)C\ z}$$

$$\mathbf{E}_L\ A\ C\ f_0\ f_1\ (nil\ A) = f_0$$

$$\mathbf{E}_L\ A\ C\ f_0\ f_1\ (cons\ A\ x\ xs) = f_1\ x\ xs\ (\mathbf{E}_L\ A\ C\ f_0\ f_1\ xs)$$

The result type of the elimination is determined by the *motive* argument (C in these examples) [17]. Computation over the constructors are handled by the "case functions" or "branch functions" (f_0 and f_1 above), one for each constructor. Finally, we have the 'target' z to eliminate, and the corresponding result $C\ z$.

Inductive families [8] are a generalisation of inductive types, where a *family* of types is inductively defined. An extended schema [12] replaces constant X with an indexed form $X\ \overline{q}$ and modifies the construction of operators to insert indices at appropriate places. These indices are different in nature from the parameters above: parameters are fixed for any instance of a type, but the indices may vary inside the value depending on how it has been constructed.

A standard example is the family of vectors, $Vec : \mathbb{N} \to Type$, indexed by length, with constructors $vnil : (A : Type)\,Vec\ A\ zero$ and $vcons : (A : Type)(n : \mathbb{N})A \to Vec\ A\ n \to Vec\ A\ (succ\ n)$. The cons operation only extends a vector by one unit of size. There are no other ways to build vectors. These constraints are reflected in the type and behaviour of the elimination operator.

$$\frac{A : Type \quad C : (n : \mathbb{N})\,Vec\ A\ n \to Type \quad f_0 : C\ zero\ (vnil\ A) \qquad f_1 : (n : \mathbb{N})(x : A)(xs : Vec\ A\ n)C\ n\ xs \to C\ (succ\ n)\ (vcons\ A\ n\ x\ xs)}{\mathbf{E}_V\ A\ C\ f_0\ f_1 : (m : \mathbb{N})(z : Vec\ A\ m)C\ m\ z}$$

$$\mathbf{E}_V\ A\ C\ f_0\ f_1\ zero\ (vnil\ A\ ) = f_0$$

$$\mathbf{E}_V\ A\ C\ f_0\ f_1\ (succ\ n)\ (vcons\ A\ n\ x\ xs) = f_1\ n\ x\ xs\ (\mathbf{E}_L\ A\ C\ f_0\ f_1\ n\ xs)$$

2.1.1 Computation on inductive families

This section ends with some examples of definitions via elimination operators. These terms will be used in examples of coercion execution later.

- $plus =_{\mathrm{df}} [x, y : \mathbb{N}]\mathbf{E}_{\mathbb{N}}\ ([n : \mathbb{N}]\mathbb{N})\ y\ ([m : \mathbb{N}][p : \mathbb{N}]succ\ p)\ x : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
- Summing up a list of numbers, $sum_L : List\ \mathbb{N} \to \mathbb{N}$, and $sum_L =_{\mathrm{df}} \mathbf{E}_L\ \mathbb{N}\ ([l : List\ \mathbb{N}]\mathbb{N})\ zero\ ([x : \mathbb{N}][l : List\ \mathbb{N}][s : \mathbb{N}]plus\ x\ s)$

- Converting a vector to a list, $v2l : (A : Type)(n : \mathbb{N})\, Vec\ A\ n \to List\ A$, where

$$v2l =_{\text{df}} [A : Type][n : \mathbb{N}]\mathbf{E}_V\ A\ ([m : \mathbb{N}][v : Vec\ A\ m]List\ A)\ (nil\ A)$$
$$([m : \mathbb{N}][x : \mathbb{N}][l : Vec\ A\ m][t : List\ A]cons\ A\ x\ t)$$

- The opposite direction (a list to a vector) is less simple. We must provide size information, hence the definition of *length*. Term $l2v$ is often described as a *dependent coercion* [16], where the target type depends on the source value.

$$\begin{aligned} length : &\quad (A : Type)\ List\ A \to \mathbb{N} \\ &=_{\text{df}} [A : Type]\ \mathbf{E}_L\ A\ ([l : List\ A]\mathbb{N})\ zero\ ([x : A][l : List\ A][t : \mathbb{N}]succ\ t) \\ l2v\ : &\quad (A : Type)\ (l : List\ A)\, Vec\ A\ (length\ A\ l) \\ &=_{\text{df}} [A : Type]\ \mathbf{E}_L\ A\ ([l : List\ A]\, Vec\ A\ (length\ A\ l))\ (vnil\ A) \\ &\qquad ([x : A][t : List\ A]vcons\ A\ (length\ A\ t)\ x) \end{aligned}$$

2.2 Coercive subtyping

The wider conception of coercive subtyping in this paper derives from [13]. A coercion is a function $c : K_0 \to K$, which lifts an object of type K_0 to type K. The meaning of coercion use may be expressed via the *coercive definition rule* [13]:

$$\frac{f : (x : K)K' \qquad k_0 : K_0 \qquad K_0 <_c K}{f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

This says that term $f(k_0)$ abbreviates and is definitionally equal to a term where the coercion is made explicit, namely $f(c(k_0))$, when a coercion c exists to lift object k_0 to the type expected by the functional operation f. Notice that coercions are only used in a context where the expected type is known, i.e. where we know both K_0 and K. This paper does not consider coercions on higher types, e.g. contravariant sub-typing on dependent products.

Users can declare 'primitive' coercions, and 'derived' coercions can be synthesised by combining new coercions with old. The conventional transitivity rule generates $A <_h C$ with $h = g \circ f$ from $A <_f B$ and $B <_g C$. We also allow 'nesting' of coercions, e.g. lifting of coercions on element types to coercions over lists or vectors [6]. There are limits on what is allowed as a coercion: the resulting coercion set must satisfy *coherence*, the property that coercions between any two types are unique up to conversion. In any sizeable example, it is not unusual that several coercion terms may be derivable for a given source and target, particularly if arising from different combinations of transitivity and nesting of coercions. Thus we must be sure that all possibilities lead to the same result. So-called "coherence checking" of a set of coercions is in principle undecidable, and thus an interesting question for practical implementations. Coherence is a major problem for parametrized coercions, which are essential for realistic (i.e., large-scale) use. A related problem is elimination of transitivity of coercion formation, to avoid unbounded search.

Forms of coercive subtyping have been implemented in Lego [3], Coq [19], Plastic [5], and Matita [2]. The coherence checking provided in [3] and [19] is decidable because both use a restricted form of coercion (based on syntactic matching of head

type constructor). Matita's implementation is based on Coq though it supports a wider range of coercions by virtue of its more powerful handling of multiple inheritance via pullbacks in the coercion graph [18]. Plastic's implementation is more experimental: it allows full conversion tests to be used, and provides very powerful forms of coercion, but there are open problems to solve.

Several meta-theoretic results have been established over type theories such as UTT, subject to coherence. Elimination of transitivity in sub-typing has been proved for subsets of inductive types (and families) [9,10]. (The limitation is to non-recursive container types whose coercions are defined using projections rather than by direct elimination, and this enables various proofs to go through. The current work can be understood as a generalisation of this technique to recursive types.) Some work has explored weaker notions of transitivity [9]. Issues of structural subtyping in parametrized types are considered in [14]; proofs of coherence and transitivity are obtained by *extending* the underlying framework with new equalities that represent functorial properties of individual parametrized types, e.g. $map\ f \circ map\ g = map\ (f \circ g)$ for lists. More tentatively, efforts like "Observational type theory" [1], which carefully add forms of extensionality to intensional type theories, may also provide sufficient leverage for some of the problems discussed here.

Coercive subtyping generalises previous notions of subtyping. There are many applications, many of which provide useful abbreviations that ease use of complex constructions. The classic example is the use of coercions between levels of algebraic structure, where one may supply a value representing a group where (e.g.) a set is required: one just 'forgets' the additional structure. This facility has been much used in substantial formalization work. In the richer context of dependent type theory, coercions allow novel and interesting forms of subtyping, e.g. in representation of Natural Language lexical semantics [15], or in providing a bridge between easy-to-use simple types and more precise dependent types [6].

Coercions are also useful in programming with inductive families: coercions may be lifted functorially over inductive types, e.g. coercing a List of element type A to a List of element type B given a coercion $c : A \to B$ [13,4]. We are also finding interesting applications through regular use in Plastic. The parametrized coercion from function spaces (i.e., non-dependent Π-types) to the (dependent) Π-types is proving very useful: N-ary functions can be written in a simple notation but then coerced automatically to the required complex type. Subtyping also has applications with universes [5]. Such coercions help to simplify interfaces to proof tools.

2.3 Discussion

Coercive subtyping is potentially a powerful and useful framework. It has shown benefits in recent formalization work, and shows promise as a tool in user interfaces for proof systems and in programming with dependent types.

There are significant limitations, however. Current implementations require restrictions on coercions in order to guarantee key properties. For theory, some progress has been made beyond the implemented classes of coercion, but only for certain subsets and the strength of results is not uniform across the subsets. (A uniform treatment is arguably easier for users to understand, e.g. they don't have

to learn a mixture of restrictions or analyze which ones might have applied when their abbreviation is rejected!) There are important classes of conversions for which no results have been established (e.g. between related container classes). It is important to make progress on these issues if coercive subtyping is to justify the claim of being a good abbreviation mechanism.

Why do these problems arise? The author's view is that the conventional presentation of Coercive Subtyping is too powerful, in the sense of not adequately limiting how it is used, and that many problems arise by trying to understand or to control the power at later stages. The conventional presentation also seems too divorced from how computation works on inductive families in an intensional theory.

One possibility is to find a more constrained presentation that provides a better balance between flexibility and ease of establishing key properties.

3 A different approach

3.1 A motivating example

Consider the coercion $v2l$ from vectors to lists (section 2.1.1) and its use when sum_L over lists is applied to vectors of numbers, e.g. in $sum_L\ <1,2,3>$. (For convenience, numerals denote equivalent values in the $\mathbb{N}$ type, and inferrable arguments are often omitted.) The conventional approach implies a conversion of the vector $<1,2,3>$ to a list, giving overheads of three new *cons* nodes to be allocated and time taken to do the extra work. This is wasteful, especially with the intuition that the folding could, in some sense, just traverse the vector structure and pick out the useful information. Can sum_V (sum on vectors) be defined in terms of how summation works on lists? The composition of sum_L with $v2l$ works, but a better (e.g. more efficient) version is possible when we have access to the arguments to the elimination operator inside sum_L. That is, when:

$$\begin{aligned} sum_L &=_{\text{df}} \mathbf{E}_L\ \mathbb{N}\ C\ zero\ f_1 \\ C &=_{\text{df}} [x : List\ \mathbb{N}]\mathbb{N} \\ f_1 &=_{\text{df}} [x : \mathbb{N}][xs : List\ \mathbb{N}][h : \mathbb{N}]plus\ x\ h \end{aligned}$$

then we can define sum_V as the following, assuming coercion function [3] $v2l$:

$$\begin{aligned} sum_V &=_{\text{df}} \mathbf{E}_V\ \mathbb{N}\ C'\ zero\ f_1' \\ C' &=_{\text{df}} [n : \mathbb{N}][v : Vec\ A\ n]C\ (v2l\ A\ n\ v) \\ f_1' &=_{\text{df}} [n : \mathbb{N}][x : A][v : Vec\ A\ n][h : C'\ n\ v]f_1\ x\ (v2l\ A\ n\ v)\ h \end{aligned}$$

Thus we have defined sum_V purely by modification of the arguments of the eliminator in sum_L. This sum_V is convertible with the direct definition on vectors. In fact, given arbitrary C, f_0, f_1, we can transfer any elimination on lists to one over vectors. We aim to use this kind of transformation as the basis for coercive subtyping.

[3] It appears here that we still retain the coercions, but this is informal notation representing the coercion of sub-values via the new mechanism, and will be made more precise later. Notice that in this example, both C and f_1 will discard the coerced terms passed to them. This is true for all 'iterative' or folding computations (as opposed to primitive recursive ones).

3.2 Characterisation of applicable coercions

For which pairs of types can this transformation be done? And how is the transformation calculated? In this paper, we give a conservative answer, based on prior existence of a coercion function, i.e. a term generated by standard schema (e.g. for functorial maps [14]) *or* nominated by the user. (It may be possible to calculate the smallest function between the a given pair of types that unambiguously preserves appropriate information, if it exists. We leave this question for another paper.)

Given an existing coercion term, the criterion is that for each constructor of the target type, we can uniquely determine how it maps to source type constructors, and hence how the relevant elimination argument for the target type constructors should be transformed. This covers all functions whose injectivity is decidable, and so encompasses functorial maps and many of the natural transformations between 'similar' datatypes (similar in the sense that none of the core data content is lost). It does not, however, include terms like first projection on dependent pairs (Σ-types) – see section 3.8. We also have to transform the *motive* (denoted C, C' in the examples); this is more straightforward and can be determined by inspection of the family indices of the types concerned.

As a first attempt, the transformations will be expressed as a particular fixed form of eliminator: the parameters, motive and branches (i.e. functions for each constructor) will be based on the target type, but indices and the value to eliminate over will be based on the source type. The head symbol of the function body will also be the eliminator of the source type. Defining this as a well-typed term also guarantees the type correctness of a transformation. For the vector to list example, this means a term $\mathbf{E}_{LV}$ as defined below. (Such terms are henceforth named $\mathbf{E}_{XY}$, where X is the target type and Y the source type.)

$$\frac{C : \mathit{List}\ A \to \mathit{Type} \quad f_0 : C\ (\mathit{nil}_A) \quad f_1 : (a : A)(l : \mathit{List}\ A)C\ l \to C\ (\mathit{cons}_A\ a\ l)}{\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 : (n : \mathbb{N})(v : \mathit{Vec}\ A\ n)C\ (\mathit{co}_V\ A\ n\ v)}$$

$$\begin{aligned}\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 =_{\mathrm{df}} \mathbf{E}_V\ A\ ([n : \mathbb{N}][v : \mathit{Vec}\ A\ n]C\ (\mathit{co}_V\ A\ n\ v))\ f_0 \\ ([m : \mathbb{N}][x : A][v : \mathit{Vec}\ A\ m][H : C\ (\mathit{co}_V\ A\ m\ v)] \\ f_1\ x\ (\mathit{co}_V\ A\ m\ v)\ H)\end{aligned}$$

The term contains occurrences of symbol co_V in places where where conversions from vectors to lists may still be required, specifically in the motive's main argument and in branch functions where recursive sub-values must be converted. For now, co_V should be read as a constructor rather than as a function: the reasons for this are explained in section 3.3.

The definition of $\mathbf{E}_{LV}$ is not a case of shifting a problem elsewhere or of introducing circularity. Firstly, it is correct wrt. type checking – that's the result type we must expect since C etc are not yet known. Secondly, consider the three possible futures for each occurrence: (a) the coerced value is discarded without examining it; (b) some elimination is performed on it; or (c) no computation occurs, e.g. in the case of C being a type constructor or a variable. Cases (a) and (b) are certainly fine, since either way the coerced term will eventually disappear. Note that case (a) tends to be more frequent, i.e. C doesn't often examine the elimination target. For

case (b), the elimination will be over lists, and the coerced (sub-)term is handled in the same way as the original term. In case (c), the blocked term will only be tested with conversion. The interaction with conversion is considered later.

Observe that $\mathbf{E}_{LV}$ precisely explains how to transform computations on lists to computations on vectors, and does so in a way that is central to the meaning of the types – via their eliminators. This statement of coercibility is arguably more meaningful than just providing some function to convert from one to the other, in the sense that it shows *why* and *how* coercion is possible. Such structure is very useful in establishing proofs of relevant properties (section 3.5), compared to the conventional conversions which are effectively opaque and unanalysable in an intensional setting. The proposal is to use the $\mathbf{E}_{XY}$ terms as the basis of coercive subtyping.

Currently, terms like $\mathbf{E}_{LV}$ are defined manually by inspection of the conventional coercion terms (e.g. from $v2l$). Automatic derivation by inversion of such terms or by analysis of the inductive schemata of two types is planned as further work.

3.3 Modifying the ι-reduction mechanism

We now consider how these transformations are used. Firstly, instead of inserting a coercion function when a type mismatch is found, we insert a marker representing a coerced term. Currently, the marker bears a label for the source type and the parameters and indices such that the resulting term is well-typed, e.g. a coerced vector v is represented as $(co_V\ A\ n\ v)$, where the label is assigned the type $(A : \mathit{Type})(n : \mathbb{N})\,\mathit{Vec}\ A\ n \to \mathit{List}\ A$. Note that well-typedness ensures that the target type is known and does not need to be labelled explicitly. This constant wrapper *delays action* on the coerced value until we know what computation is to be performed on it. To 'activate' the coercion, the marker must reach an elimination operator of the target type, at which point the appropriate $\mathbf{E}_{XY}$ term is used. We represent this by adding a further computation rule for Lists:

$$\mathbf{E}_L\ A\ C\ f_0\ f_1\ \ (co_V\ A\ n\ v) \mapsto \mathbf{E}_{LV} A\ C\ f_0\ f_1\ n\ v$$

Should other coercion transformations be required (subject to conditions, e.g. that a suitable $\mathbf{E}_{XY}$ term can be defined and that coherence is preserved), they will each result in an additional computation rule in a form similar to the last line. Note that the form of the new reductions reflects those for decoding Tarski-style universes [5], e.g. $\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a) : \mathit{Type}$, where decoding of a lifted name is handled by decoding the unlifted name.

3.4 Canonicity and interaction with conversion

There are two aspects to canonicity: first, what happens when coerced values are compared with other values in the target type; and second, the extent to which the coercion markers act as constructors for the target type.

The first situation concerns questions such as the convertibility of a list with a vector value coerced to a list, i.e. $n : \mathbb{N}, v : \mathit{Vec}\ A\ n, l : \mathit{List}\ A \vdash (co_V\ A\ n\ v) = l$, or convertibility of values coerced from two different source types into the same target type. In the standard approach, (a) the coercion function performs the

transformation on non-blocked terms and convertibility can proceed; or (b) the term is blocked, e.g. as a variable, so the coercions remain in place and conversion fails unless the coerced values are convertible.

It is different in the new scheme, because the transformation is implicit in some elimination, and convertibility (by itself) does not add eliminations. But there is a computation that can be applied safely to a term to remove the coercion at its head: the *identity elimination* for the target type. Note that this does not imply significant overheads for convertibility: in a sense, it performs a conversion that was delayed from an earlier time, and performs work that is identical to the conventional transformation function. [4] In cases where the reduction is blocked, we are in no worse a situation than the same case in the original scheme.

To clarify, the proposal is add reductions (not equivalences) of the form below, where $C_{Id(X)}$ is the identity elimination motive for inductive type X and $\overline{f_{Id(X)}}$ the identity branch functions for X. This form is necessary to ensure that the coercion marker is removed. Such reductions will only apply to coerced terms. Note that such reductions are similar to η rules on inductive types, and that η rules are not problem-free [12, p. 198], but the limitation to coercions and only to complete delayed coercions may prove sufficient. (This will be investigated as further work.)

$$co_Y\ \overline{p}\ \overline{q}\ y \mapsto \mathbf{E}_{XY}\ \overline{p}\ C_{Id(X)}\ \overline{f_{Id(X)}}\ \overline{q}\ y$$

The second situation is less clear. The question is whether and to what extent coerced terms, particularly blocked ones (which can't progress with computation), are to be taken as canonical values of the target type. Subtyping does blur the notion of canonicity. An answer to this question has bearing on how reductions of the form above are used. We leave this issue for further work.

A related question is how much of the coercion process is revealed to the user, particularly in the context of a proof assistant where intermediate terms may be partially computed. Do we show details of the source type, of the target type, or a mixture? What is suitable for helping the user understand the state of his or her proof? One possibility is that reduction of $\mathbf{E}_{XY}$ should be extended to compute out the $\mathbf{E}_Y$ portions, i.e. to extract data from the source value and show a term containing original branch functions and extracted data. For example, $sum_L < 1, 2, 3 >$ could be reduced directly to something like $plus\ 1\ (plus\ 2\ (plus\ 3\ 0))$. The labelling framework from Epigram [17] may also help: it is designed to give clear information about computations at a level above raw eliminators. We also remark on the parallels between *views* in Epigram and the source-target relationship in coercions, in that one can provide coercions from a source type to its view types.

3.5 Compositions of $\mathbf{E}_{XY}$ terms and transitivity

We now consider how the $\mathbf{E}_{XY}$ terms behave under composition. The transitivity rule of coercive subtyping forms new coercions by composition, and coherence requires a check that new coercions are consistent with existing coercions. A related case is where values are coerced to one type and in the course of later computa-

[4] That is, $\mathbf{E}_{LV}\ A\ C_{Id(L)}\ \overline{f_{Id(L)}}\ n\ v = v2l\ A\ n\ v$, and this is easily proved. This property shows that the relevant $\mathbf{E}_{XY}$ term was correctly constructed, so it is a useful check to make.

tion are coerced again, possibly back to the original type, that is where a term $co_Y\ \overline{p'}\ \overline{q'}\ (co_Z\ \overline{p''}\ \overline{q''}\ v)$ is formed ($\overline{p}$ and $\overline{q}$ represent parameters and indices).

As a concrete example, consider the coercions between lists and vectors, and the cases of list-vector-list and vector-list-vector. Intuitively, the result should be an identity operation on lists (resp. vectors). In general, such "identity coercions" are excluded in coercive subtyping. One practical reason is that the resulting composition is in general not intensionally equal to identity, meaning that we would have to choose one direction (X to Y) or the other, but can not have both. We will examine the two combinations of $\mathbf{E}_{LV}$ (defined earlier) and $\mathbf{E}_{VL}$ (defined below).

$$\frac{\begin{array}{l}A : \mathit{Type} \quad C : (n : \mathbb{N})\, \mathit{Vec}\ A\ n \to \mathit{Type} \quad f_0 : C\ \mathit{zero}\ (\mathit{vnil}\ A\ \mathit{zero}) \\ f_1 : (m : \mathbb{N})(a : A)(v : \mathit{Vec}\ A\ m)C\ m\ v \to C\ (\mathit{succ}\ m)\ (\mathit{vcons}\ A\ m\ a\ v))\end{array}}{\mathbf{E}_{VL}\ A\ C\ f_0\ f_1 : (z : \mathit{List}\ A)C\ (\mathit{length}\ A\ z)\ (co_L\ A\ l)}$$

$$\begin{array}{l}\mathbf{E}_{VL}\ A\ C\ f_0\ f_1 =_{\mathrm{df}} \mathbf{E}_L\ A\ ([l : \mathit{List}\ A]C\ (\mathit{length}\ A\ l)\ (co_L\ A\ l))\ f_0 \\ \qquad\qquad ([x : A][t : \mathit{List}\ A]f_1\ (\mathit{length}\ A\ t)\ x\ (co_V\ A\ m\ v))\end{array}$$

We consider the composition of $\mathbf{E}_{LV}$ with $\mathbf{E}_{VL}$ under arbitrary A, C, f_0, f_1 and a coerced list value l. It shows the effect of two coercions in sequence. The composition is not direct, i.e. does not take the usual form $f\ \overline{a} \circ g\ \overline{b}$, since we will require reduction of $\mathbf{E}_{LV}$ to compute changed arguments for $\mathbf{E}_{VL}$. In the third line, $\mathbf{E}_{VL}$ is 'activated' to transfer the computation on vectors to one on the (coerced) list.

$$\begin{array}{ll}& \mathbf{E}_{LV}\ A\ C\ f_0\ f_1\ (\mathit{length}\ A\ l)\ (co_L\ A\ l) \\ =_{\delta\beta} & \mathbf{E}_V\ A\ ([n : \mathbb{N}][v : \mathit{Vec}\ A\ n]C\ (co_V\ A\ n\ v))\ f_0 \\ & \qquad ([m : \mathbb{N}][a : A][v : \mathit{Vec}\ A\ m][H : C\ (co_V\ A\ m\ v)]f_1\ a\ (co_V\ A\ m\ v)\ H) \\ & \qquad (co_L\ A\ l) \\ =_{co} & \mathbf{E}_{VL}\ A\ ([n : \mathbb{N}][v : \mathit{Vec}\ A\ n]C\ (co_V\ A\ n\ v))\ f_0 \\ & \qquad ([m : \mathbb{N}][a : A][v : \mathit{Vec}\ A\ m][H : C\ (co_V\ A\ m\ v)]f_1\ a\ (co_V\ A\ m\ v)\ H)\ l \\ =_{\delta\beta} & \mathbf{E}_L\ A\ ([k : \mathit{List}A]C\ (co_V\ A\ (\mathit{length}\ A\ k)\ (co_L\ A\ k)))\ f_0 \\ & \qquad ([a : A][k : \mathit{List}A][H : C\ (co_V\ A\ (\mathit{length}\ A\ k)\ (co_L\ A\ k))] \\ & \qquad\quad f_1\ a\ (co_V\ A\ (\mathit{length}\ A\ k)\ (co_L\ A\ k))) \\ & \qquad l\end{array}$$

We require that this computation is identical to that done directly on l, so compare it with term $\mathbf{E}_L\ A\ C\ f_0\ f_1\ l$. There are disagreements in the motive and step-case function, shown here as required conversions (in η-long form):

$$\begin{array}{l}C = [k : \mathit{List}\ A]C\ (co_V\ A\ (\mathit{length}\ A\ k)\ (co_L\ A\ k)) \\ f_1 = [a : A][k : \mathit{List}A][H : C\ (co_V\ A\ (\mathit{length}\ A\ k)\ (co_L\ A\ k))] \\ \qquad f_1\ a(co_V\ A\ (\mathit{length}\ A\ k)\ (co_L\ A\ k))\ H\end{array}$$

The problem here is the equation $k = co_V\ A\ (length\ A\ k)\ (co_L\ A\ k)$, i.e. whether the two coercions together correspond to an identity. However, note that co_V and co_L are constructors and they only have meaning when some elimination is applied. Section 3.3 suggested application of the relevant identity elimination. This still leaves the problem of $k = \mathbf{E}_L\ A\ C_{Id(X)}\ \overline{f_{Id(X)}}\ k$, i.e. of whether the identity elimination really is an identity elimination. For now, we take it as a premiss of the proposition and conclude this: if the identity elimination(s) are identities for conversion, then we can prove coherence of the composed coercions.

The other direction is less straightforward, from vectors to lists to vectors as the composition of $\mathbf{E}_{VL}$ with $\mathbf{E}_{LV}$. Under arbitrary A, C, f_0, f_1, and coerced vector v of size n, the critical equations are:

$$C\ n\ v \quad = C\ (length\ A\ (co_V\ A\ n\ v)) \quad (co_L\ A\ (co_V\ A\ n\ v))$$

$$f_1\ m\ x\ v\ = f_1\ (length\ A\ (co_V\ A\ m\ v))\ x\ (co_L\ A\ (co_V\ A\ m\ v))$$

Assuming the properties of identity eliminations suggested above, convertibility is blocked here only by the vector length component, i.e. $(length\ A\ (co_V\ A\ n\ v)) = n$ is not derivable intensionally. It can be proved extensionally by induction over n or v. Note that this is still a computation applied to a coercion, but reduction via $\mathbf{E}_{LV}$ does not help: the $\mathbf{E}_V$ elimination can not be reduced further. [5]

To summarise, a composition of $\mathbf{E}_{LV}$ with $\mathbf{E}_{VL}$ is intensionally equal to the uncoerced computation under assumption of properties of identity eliminations. The opposite direction is blocked by an equation on the vector's size parameter. We suggest that coherence is provable in a similar way for a wider range of types for which $\mathbf{E}_{XY}$ terms can be defined: (a) all inductive types (i.e. no family indices); (b) inductive families with non-recursive indices; (c) inductive families where the $\mathbf{E}_{XY}$ terms do not exhibit dependencies across the result of composition. The justification is that the $\mathbf{E}_{XY}$ terms avoid recursion on intermediate values and the critical terms to check (arguments to the elimination operators) are convertible by virtue of how the $\mathbf{E}_{XY}$ are constructed. Note that this conjecture covers both compositions of coercions within the same type (including the functorial coercions studied in [14]), AND the more general compositions involving two or three different types (on which no work has yet been done). The issues of inductive family parameter behaviour in composed coercions requires further study.

The value of such a result (when formally proved) is to provide an intensional and simple to automate method to check coherence of a wider range of coercion combinations, particularly those arising from transitive closure of coercions.

3.6 Preliminary remarks on metatheory

The mechanism proposed in section 3.3 is a modification of how coercions are specified and of their reduction behaviour, with subsequent effects on how key properties are stated. The modification exposes details of how computation on one type is mapped to computation on the other, and delays the action of 'coercion' until it is

[5] It may possible to get round this particular case by introducing a coercion from vectors to their lengths, i.e. to express computation on $\mathbb{N}$ in terms of a vector traversal and thus avoid the elimination on vectors, but this is hardly a general solution.

known what computation to apply (else applies an identity elimination).

The new ι-reductions are sound wrt. types because of the fixed form of the $\mathbf{E}_{XY}$ terms and the way in which they are used in ι-reductions. (Indeed, preliminary experiments have been developed and checked in Plastic [5].) Correctness of the definition of $\mathbf{E}_{XY}$ terms can be stated by comparison with the conventional conversion function (possibly from which the definition has been extracted), and easily proved by applying the identity elimination. That is, the $\mathbf{E}_{XY}$ term is correct if the original conversion function is correct. (In future, we may require $\mathbf{E}_{XY}$ terms to be defined from the structure of the relevant inductive schemata, which will give a stronger guarantee of correctness.)

Use of the default identity elimination in conversion appears safe, in the sense that redexes arise because of a delayed computation and the reduction effectively executes the conversion of representation originally specified by the user (again, this gives rise to a proof obligation which is easily proved by induction). This redex is thus equivalent to the situation in the standard approach of forcing reduction on an implicitly coerced value, and relevant results from the literature should apply.

3.7 Implementation details, and efficiency

Preliminary experiments have been carried out in Plastic [5], a system which implements Luo's LF with Coercive Subtyping. Several $\mathbf{E}_{XY}$ terms have been defined and the additional ι-reductions (of eliminators) simulated via 'back door' access to the inductive families implementation. (This back door allows non-standard inductive types to be defined manually. The proposed reductions are checked for type safety, but termination is not checked.) Relevant proofs have been developed in Plastic, where feasible, or the reasons for failure analysed.

The full mechanism, including identity elimination reductions in conversion, will be straightforward to implement. Identity eliminators can be derived from schemata. The types of $\mathbf{E}_{XY}$ terms can be generated automatically from the relevant schemata. Derivation of definitions of $\mathbf{E}_{XY}$ terms may be possible for simple cases, else the user can develop the definition by refinement. The existing algorithms for generating and matching coercions will not need changes.

Improvements in efficiency of coercion execution are expected. Firstly, there is no intermediate structure to be built and then traversed: computations are applied directly to the original data (this technique has parallels with deforestation in functional programming). Secondly, the $\mathbf{E}_{XY}$ computations can be improved in several ways, not least some form of partial evaluation or Normalization by Evaluation on the branch functions to avoid repeated work later, or the collapsing of chains of coercions (e.g. projections on algebraic structures) to simpler functions.

3.8 Further examples

We briefly consider two different examples. Firstly, projections from Σ-types. Some authors suggest π_1 as a useful coercion, though one recent study identifies problematic interactions of this coercion with functorial mappings on Σ [9]. That work suggests a two-stage application of coercions, effectively constraining how these two groups of coercion can be composed: π_1 is used only in the second stage, after all

other applicable coercions have been inserted.

$$\Sigma : (A : Type)(B : A \to Type)\, Type$$
$$\pi_1 : (A : Type)(B : A \to Type)\Sigma\ A\ B \qquad \to A$$
$$\pi_2 : (A : Type)(B : A \to Type)(s : \Sigma\ A\ B) \to B\ (\pi_1\ A\ B\ s)$$

In the new framework, can π_1 be expressed as a coercion? The first question is whether (and how) we can transfer computation on A to computation on the Σ value. Nothing is known about A, hence the answer is negative and we reject π_1 as a coercion: it makes no sense.[6] Note that the functorial mappings on Σ *are* expressible as $\mathbf{E}_{XY}$-style terms (see below for a similar example).

The second example involves lifting a coercion over a container data type, specifically lifting a coercion on element types to a coercion on lists of that element. Such a rule can be applied recursively, hence used to convert arbitrarily nested lists. (Such recursive coercions have been implemented in Plastic for the conventional approach [6,4].) The basic form of $\mathbf{E}_{LL}$ is given below, expressing computation on $List\ A$ values in terms of $List\ B$ computations. Notice that two extra parameters are needed: the type of the source list elements and a coercion function on the elements. The precise representation of such 'nested' coercions has not been decided; for now, the simplest representation is chosen. The key detail below is that the modified step-case function applies the element conversion function to the head element before proceeding. Relevant coherence properties hold intensionally for this term, similarly to the $\mathbf{E}_{LV} - \mathbf{E}_{VL}$ composition. Observe that $\mathbf{E}_{LL}$ pre-composes with $\mathbf{E}_{LV}$ etc.

$$\begin{aligned}
\mathbf{E}_{LL} : \ & (B : Type)(C : [l : List\ B]\, Type)(f_0 : C\ nil) \\
& (f_1 : (x : B)(xs : List\ B)(H : C\ xs)C\ (cons\ B\ x\ xs)) \\
& (A : Type)(f : A \to B)(l : List\ A)C\ (co_{LL}\ A\ B\ f\ l) \\
\mathbf{E}_{LL} = \mathbf{E}_L\ & A([l : List\ A]C\ (co_{LL}\ A\ B\ f\ l))\ f_0 \\
& ([x : A][xs : List\ A]f_1\ (f\ x)\ (co_{LL}\ A\ B\ f\ xs))
\end{aligned}$$

3.9 Discussion and future work

This work is still in early stages, but early results are promising and there are several interesting extensions to pursue. The work contributes in several ways: (a) characterising an important subset of coercion functions; (b) enabling proof of key properties of this subset; (c) supporting greater efficiency of coercion use. Progress has been made towards simpler proofs on functorial coercions, and towards new proofs for more general cases of coercion combination (e.g. compositions involving several distinct types) that were previously identified as problematic. All of these aspects are important to promote Coercive Subtyping as a useful and practical abbreviation mechanism.

[6] It may be interesting to consider a weaker conversion term, say of type $(A \to C) \to \Sigma\ A\ B \to C$ which reflects that A will be transformed to C. Investigating such terms and the possibility of integration with the main mechanism is planned as further work.

Future work includes formal proofs, full implementation in Plastic to enable larger studies, and further study on the class of $\mathbf{E}_{XY}$ terms - including automatic derivation and the study of restrictions (e.g. indices of inductive families). Coercions that extend compatibility of inductive family indices appear particularly interesting, e.g. $Eq\ m\ n \to Vec\ A\ m \to Vec\ A\ n$.

Acknowledgements

Thanks to the referees and Zhaohui Luo for their interesting and useful comments.

References

[1] Altenkirch, T. and C. McBride, *Towards observational type theory*, Manuscript, available online (2006).

[2] Asperti, A., C. Sacerdoti Coen, E. Tassi and S.Zacchiroli, *Crafting a proof assistant* (2006), submitted for publication to the TYPES06 Post Proceedings.

[3] Bailey, A., "The Machine-checked Literate Formalisation of Algebra in Type Theory," Ph.D. thesis, University of Manchester (1998).

[4] Callaghan, P., *Coercions in Plastic*, Lecture notes, TYPES Summer School 2002 (2002), `http://www-sop.inria.fr/certilab/types-sum-school02/`.

[5] Callaghan, P. and Z. Luo, *An Implementation of LF with Coercive Subtyping & Universes*, Journal of Automated Reasoning (Special Issue on Logical Frameworks) **27** (2001), pp. 3–27.

[6] Callaghan, P., Z. Luo and J. Pang, *Object languages in a type-theoretic meta-framework*, in: *Proc. Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, 2001.

[7] Coquand, T. and C. Paulin-Mohring, *Inductively defined types*, LNCS **417** (1990).

[8] Dybjer, P., *Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics*, in: G. Huet and G. Plotkin, editors, *Logical Frameworks* (1991).

[9] Luo, Y., "Coherence and Transitivity in Coercive Subtyping," Ph.D. thesis, U. of Durham (2004).

[10] Luo, Y. and Z. Luo, *Coherence and transitivity in coercive subtyping*, in: *Proc. 8th Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, 2001, pp. 249–265, (LNAI 2250).

[11] Luo, Z., *A unifying theory of dependent types: the schematic approach*, Proc. Symposium on Logical Foundations of Computer Science (Logic at Tver'92), LNCS 620 (1992).

[12] Luo, Z., "Computation and Reasoning: A Type Theory for Computer Science," OUP, 1994.

[13] Luo, Z., *Coercive subtyping*, Journal of Logic and Computation **9** (1999), pp. 105–130.

[14] Luo, Z. and R. Adams, *Structural subtyping for inductive types with functorial equality rules*, Math. Struct. in Comp. Science (2007), (to appear).

[15] Luo, Z. and P. Callaghan, *Coercive subtyping and lexical semantics (ext'd abstr.)*, LACL'98 (1998).

[16] Luo, Z. and S. Soloviev, *Dependent coercions*, The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science **29** (1999).

[17] McBride, C. and J. McKinna, *The view from the left*, J. Functional Programming **14** (2004), pp. 69–111.

[18] Sacerdoti Coen, C., *A Presentation of Matita* (2006), slides from talk at CHIT/CHAT Workshop.

[19] Saïbi, A., "Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories," Ph.D. thesis (1999).

Focusing the inverse method for LF: a preliminary report

Brigitte Pientka and Xi Li[1]

School of Computer Science
McGill University
Montreal, Canada

Florent Pompigne

ENS Cachan
94235 Cachan cedex, France

Abstract

In this paper, we describe a proof-theoretic foundation for bottom-up logic programming based on uniform proofs in the setting of the logical framework LF. We present a forward uniform proofs calculus which is a suitable foundation for the inverse method for LF and prove its correctness. We also present some preliminary results of an implementation for the Horn Fragment as part of the logical framework Twelf, and compare its performance with the tabled logic programming engine.

1 Introduction

Logic programming is typically thought of as a backward proof search method where we start with the query and apply backchaining. We first try to find a clause head which unifies with a given query and then try to solve its subgoals. Proof-theoretically backchaining in logic programming can be elegantly explained by uniform proofs [MNPS91] which serves as a foundation for higher-order logic programming systems such as as λ-Prolog [NM99], Twelf [PS99], or Isabelle [Pau86]. These frameworks provide a general meta-language for the specification and implementation of formal systems, and execution of these specifications is based on the operational semantics of backchaining logic programming. However, the backchaining semantics has also several disadvantages. Many straightforward specifications may not be directly executable, thus requiring more complex and sometimes less efficient implementations and performance may be severely hampered by redundant computation. The tabled logic programming engine [Pie02a,Pie05] in Twelf

[1] Email: bpientka@cs.mcgill.ca

addresses these concerns. It allows the logic programming interpreter to memoize subcomputations and re-use their result later, thereby eliminating infinite and redundant computation. However, a critical potential bottleneck in this system is that the memo-table may grow large and there are a large number of suspended goals. The overhead of freezing and storing a given proof search state such that it can be resumed later on is substantial.

An alternative to backchaining in logic programming is forward chaining where we start with some axioms, and then satisfy the subgoals to conclude new facts. This idea of forward chaining has been exploited in bottom-up logic programming as found for example in magic sets [Ram91]. Forward logic programming has potentially many advantages over the more traditional backwards logic programming approaches, since suspending computation and storing a global state, as in tabled logic programming, is completely unnecessary. It provides a sound and complete proof search procedure, where only true statements are generated and only those must be stored. Forward chaining in this sense can be naturally explained by proof search based on the inverse method (see for example [CPP06]). In this paper, we lay the foundation for exploring forward chaining in the logical framework Twelf, and present a forward uniform proof calculus together with its correctness proof. Building on this theoretical discussion, we discuss how to turn theory into a practical implementation and report on our experience with a prototype for the Horn fragment.

This paper is structured: Section 2 we introduces briefly the syntax of LF, and Section 3 gives some example specification in LF. Section 4 we present uniform calculus together with a lifted version which has meta-variables. In Section 5, we present a forward uniform proof system together with a lifted version which is a suitable basis for inverse method for LF. Section 6, we discuss implementation issues, and report on some some preliminary results for the Horn fragment and compare it to the tabled higher-order logic programming engine.

2 Background: The logical framework LF

Our main interest in this paper is in designing a forward inverse method prover for the logical framework Twelf. Twelf supports the specification of deductive systems, given via axioms and inference rules, together with the proofs about them, and has been extensively used over the past few years in several applications. The theoretical foundation for Twelf is the logical framework LF [HHP93]. The LF language, a dependently typed lambda-calculus, can be briefly described as follows:

$$
\begin{array}{ll}
\text{Kinds } K & ::= \mathsf{type} \mid \Pi x{:}A.K \\
\text{Types } A & ::= a\ M_1 \ldots M_n \mid A_1 \rightarrow A_2 \mid \Pi x : A_1.A_2 \\
\text{Normal Objects } M & ::= \lambda x.M \mid R \\
\text{Neutral Objects } M & ::= x \mid c \mid R\ M
\end{array}
$$

We follow recent formulations which only concentrate on characterizing normal forms [NPP06], however this is not strictly necessary. Objects provided by the logi-

cal framework LF include lambda-abstraction, application, constants and variables. To preserve canonical forms in the presence of substitution, we rely on hereditary substitutions as defined in [NPP06]. Types classify objects, and range over type constants a which may be indexed by objects $M_1 \dots M_n$, as well as non-dependent and dependent function types. Viewing types as propositions, LF types can be interpreted as logical propositions. Atomic type $a\ M_1 \dots M_n$ correspond to an atomic proposition, non-dependent function type $A_1 \to A_2$ corresponds to an implication, and the dependent function type $\Pi x{:}A.B$ can be interpreted as the universal quantifier. We will use types and formulas interchangeably.

3 Example: Bounded polymorphic subtyping

As a motivating example which illustrates also many challenges we face when designing an inverse method prover for Twelf, we consider bounded subtype polymorphism (see also Ch. 26 [Pie02b]). In this system, we enrich polymorphic types such as $\forall\alpha.T$ with a subtype relation and refine the universal quantifier to carry a subtyping constraint. This example was proposed as part of the POPLmark challenge [ABF+05] to study different meta-theoretic properties about bounded subtype polymorphism. Here our focus is primarily in executing the given specification and experimenting with it. The syntax of types can be defined as follows:

$$\begin{array}{lll} \text{Types} & T & ::= \text{top} \mid \alpha \mid T_1 \Rightarrow T_2 \mid \forall\alpha \leq T_1.T_2 \\ \text{Context} & \Gamma & ::= \cdot \mid \Gamma, w{:}\alpha \leq T \end{array}$$

In $\forall\alpha \leq T_1.T_2$, the type variable α only binds occurrences of α in T_2. The typing context Γ keeps track of constraints such as $\alpha \leq T$. Next, we describe a subtyping algorithm using the judgment:

$$\Gamma \vdash T \leq S \qquad \text{Type } T \text{ is a subtype of } S \text{ in the context } \Gamma$$

$$\frac{}{\Gamma \vdash T \leq \text{top}}\ \text{sa-top} \qquad \frac{\alpha \leq T \in \Gamma}{\Gamma \vdash \alpha \leq T}\ \text{sa-hyp} \qquad \frac{}{\Gamma \vdash \alpha \leq \alpha}\ \text{sa-ref-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 \Rightarrow S_2 \leq T_1 \Rightarrow T_2}\ \text{sa-arr} \qquad \frac{\Gamma \vdash \alpha \leq U \quad \Gamma \vdash U \leq V}{\Gamma \vdash \alpha \leq V}\ \text{sa-tr-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma, w{:}\alpha \leq T_1 \vdash S_2 \leq T_2}{\Gamma \vdash \forall\alpha \leq S_1.S_2 \leq \forall\alpha \leq T_1.T_2}\ \text{sa-all}^{\alpha,w}$$

The description is algorithmic in the sense that general rules for reflexivity and transitivity are admissible, and for each type constructor, top, $\forall$ and $\Rightarrow$ there is one rule which can be applied. However, it is worth pointing out that while the presented characterization has pleasant meta-theoretic properties, it does not eliminate all non-determinism. While the rule for transitivity is restricted to type variables on the left side of the subtyping relation, we can satisfy the left premise with four possible rules, i.e. the rule sa-top, sa-hyp, sa-ref-tvar, and sa-tr-tvar. However, only

the rule sa-hyp is really fruitful. A crucial question therefore is not only how we can implement this formal system in the logical framework, but also what is the right paradigm to execute this implementation.

We begin by encoding the object-language of polymorphic types in LF using higher-order abstract syntax, i.e. type variables α in the object language will be represented as variables in the meta-language. This is standard practice.

```
tp:type.                          arr: tp -> tp -> tp.
top: tp.                          all: tp -> (tp -> tp) -> tp.
```

We define an LF type called `tp`, with the constructors `top`, `arr`, and `all`. The type for the constructor `all` takes in two arguments. The first argument stands for the bound and has type `tp`, while the second argument represents the body of the forall-expression and is represented by the function type (`tp -> tp`).

Next we consider the implementation of the subtyping relation. Since we represent variables of the object language implicitly, we cannot generically represent sa-ref and sa-tr where both these rules are applicable for all type variables. Instead of a general variable rule, we will add *rules for reflexivity and transitivity for each type variable*. Reflexivity and transitivity rules are dynamically introduced for each type variable.

We are now ready to show the encoding of these subtyping rules in LF. We first define the constant `sub` which describes the subtyping relation. Next, we represent each inference rule in the object-language as a clause consisting of nested universal quantifiers and implications. Upper-case letters denote logic variables which are implicitly bound by a Π-quantifier at the outside.

```
sub : tp -> tp -> type.
sa_top : sub S top.
sa_arr : sub S2 T2 -> sub T1 S1
          -> sub (arr S1 S2) (arr T1 T2) .
sa_all : (Πa:tp.
            (ΠU.ΠV.sub U V ->  sub a U -> sub a V) ->
            sub a T1 -> sub a a ->
            sub (S2 a) (T2 a))
         -> sub T1 S1
         -> sub (all S1 (λa.(S2 a))) (all T1 (λa.(T2 a))).
```

Using a higher-order logic programming interpretation based on backchaining, we can read the clause `sa_arr` as follows: To prove the goal `sub (arr S1 S2) (arr T1 T2)`, we must prove `sub T1 S1` and then `sub S2 T2`. Similarly we can read the clause `sa_all`: To prove `sub (all S1 (λa.(S2 a))) (all T1 (λa.(T2 a)))`, we need to prove first `sub T1 S1`, and then assuming `tr:ΠU.ΠV. sub U V -> sub a U -> sub a V`, `w:sub a T1`, and `ref:sub a a`, prove that `sub (S2 a) (T2 a)` is true where `a` is a new parameter of type `tp`.

Unfortunately, this specification cannot directly be executed using the backward logic programming engine, since the transitivity rule does not eliminate all non-determinism. Tabled logic programming memoizes previously encountered subgoals and allows us to reuse the results later on. This enables us to execute the specification for bounded subtype polymorphism. However, tabled logic programming has

also a substantial overhead of storing encountered goals together with their answer substitution, and freezing and suspending computation to resume it later.

In this paper, we explore an alternative paradigm, a forward logic programming. This means we start from some axioms and then apply the given rules in a forward direction. In this example, `sub T top` is an axiom and so is for example, `sub a a` for any variable `a`. The clause `sa_arr` is then interpreted in a forward direction as follows: Given a proof for `sub T1 S1` and a proof for `sub S2 T2`, we can derive `sub (arr S1 S2) (arr T1 T2)`. We present first a theoretical foundation for forward proof search, and then outline the basic idea and challenges when implementing an inverse method search engine based on it. Finally, we conclude with a discussion of some preliminary results of our current prototype implementation. While our prototype only concentrates on the Horn fragment, it nevertheless provides some interesting preliminary results and analysis especially when compared to tabled logic programming.

4 Uniform Proofs

A standard proof-theoretic characterization for backchaining in logic programming is based on uniform proofs [MNPS91]. The essential idea in uniform proofs is to chain all invertible rules eagerly, and postpone the non-invertible rules lead to a uniform sequent calculus. In a uniform calculus, we distinguish between uniform phase, where we apply all the invertible rules, and focusing phase, where we pick and focus on a non-invertible rule. We can characterize uniform proofs by two main judgments:

$\Gamma \Longrightarrow A$	There is a *uniform proof* for A from the assumptions in Γ
$\Gamma \gg A \Longrightarrow P$	There is a *focused proof* for the atom P focusing on the proposition A using the assumptions in Γ

Next, we present a proof system characterizing uniform proofs.

$$\frac{\Gamma \gg A \Longrightarrow P \quad A \in \Gamma}{\Gamma \Longrightarrow P}\ \mathsf{choose} \qquad \frac{}{\Gamma \gg P \Longrightarrow P}\ \mathsf{hyp}$$

$$\frac{\Gamma, c{:}A_1 \Longrightarrow A_2}{\Gamma \Longrightarrow A_1 \to A_2} \to \mathsf{R} \qquad \frac{\Gamma \Longrightarrow A_1 \quad \Gamma \gg A_2 \Longrightarrow P}{\Gamma \gg A_1 \to A_2 \Longrightarrow P} \to \mathsf{L}$$

$$\frac{\Gamma, x{:}A \Longrightarrow B}{\Gamma \Longrightarrow \Pi x{:}A.B}\ \Pi\mathsf{R} \qquad \frac{\Gamma \gg [M/x]A \Longrightarrow P \quad \Gamma \vdash M : A}{\Gamma \gg \Pi x{:}A.B \Longrightarrow P}\ \Pi\mathsf{L}$$

We note that our context Γ keeps track of dynamic assumptions which are introduced in the rule $\to$R and can be used during proof search as well as parameter assumptions which are introduced in the rule ΠR but cannot be used in proof search. Our goal is to enforce that every proposition is well-typed, so in the sequent $\Gamma \Longrightarrow A$ we have that A is a well-formed type in the context Γ, and similarly in the sequent

$\Gamma \gg A \Longrightarrow P$ we have that A and P are a well-formed types in the context Γ. Moreover, we require in the rule $\Pi\mathsf{L}$ that M has type A in the context Γ.

In practice, we typically do not guess the correct instantiation for the universally quantified variables in the $\Pi\mathsf{L}$ rule, but introduce a meta-variable which will be instantiated with unification later. Previously[Pie03] , we have been advocating the use of meta-variables. Meta-variables are associated with a postponed substitution σ which is applied as soon as we know what the meta-variable stands for. A formal treatment for meta-variables based on contextual modal types can be found in [Pie03,NPP06]. This allows us to formally distinguish between the ordinary bound variables introduced by $\Pi\mathsf{R}$ or a λ-abstraction and meta-variables u which are subject to instantiation. An advantage of this approach is that we localize dependencies while allowing in-place updates. Moreover, we can present all meta-variables that appear in a given term in a linear order and ensure that the types and contexts of meta-variables further to the right may mention meta-variables. When a meta-variable is introduced it is created as $u[\mathsf{id}_\Gamma]$ meaning it can depend on all the bound variables occurring in Γ. During search Γ is concrete and id_Γ will be unfolded. Moreover, we can easily characterize all the meta-variables occurring in a formula or sequent. The distinction between ordinary bound variables and meta-variables provides a clean basis for describing proof search. We will therefore enrich our lambda-calculus with first-class meta-variables denoted by u.

$$\begin{array}{ll} \text{Neutral Terms } M & ::= \ldots \mid u[\sigma] \\ \text{Meta-variable context } \Delta & ::= \cdot \mid \Delta, u{::}A[\Psi] \end{array}$$

The type of a meta-variable is $A[\Psi]$ denoting an object M which has type A in the context Ψ. We briefly highlight how contextual substitution into types and objects-level terms is defined to give an intuition, but refer the interested reader to [NPP06] for more details. We write $[\![\hat{\Psi}.M/u]\!]$ for replacing a meta-variable u with an object M. $\hat{\Psi}$ characterizes the ordinary bound variables occurring in M. This explicit listing of the bound variables occurring in M is necessary because of α-renaming issues and can be eliminated in an implementation. We only show contextual substitution into objects here.

$$\begin{array}{lll} [\![\hat{\Psi}.M/u]\!](\lambda y.N) & = \lambda y.N' & \text{if } [\![\hat{\Psi}.M/u]\!]N = N' \\ [\![\hat{\Psi}.M/u]\!](u[\sigma]) & = M' & \text{if } [\![\hat{\Psi}.M/u]\!]\sigma = \sigma' \text{ and } [\sigma'/\Psi]M = M' \\ [\![\hat{\Psi}.M/u]\!](u'[\sigma]) & = u'[\sigma'] & \text{if } u' \neq u \text{ and } [\![\hat{\Psi}.M/u]\!]\sigma = \sigma' \\ [\![\hat{\Psi}.M/u]\!](R\ N) & = (R'\ N') & \text{if } [\![\hat{\Psi}.M/u]\!]R = R' \text{ and } [\![\hat{\Psi}.M/u]\!](N) = N' \\ [\![\hat{\Psi}.M/u]\!](x) & = x & \\ [\![\hat{\Psi}.M/u]\!](c) & = c & \end{array}$$

We note that there are no side-conditions necessary when substituting into λ-abstraction, since the objects M we substitute for u is closed with respect to $\hat{\Psi}$. When we encounter a meta-variable $u[\sigma]$, we first apply $[\![\hat{\Psi}.M/u]\!]$ to the substitution

σ yielding σ' and then replace u with M and apply the substitution σ'. Note because of α-renaming issues we must possibly rename the domain of σ'.

Simultaneous contextual substitution can be defined following similar principles. A simultaneous contextual substitution maps the meta-variables in its domain Δ' to another meta-variable context Δ which describes its range. More formally we can define simultaneous contextual substitutions as well-typed as follows:

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \Delta; \Psi \vdash M : A}{\Delta \vdash (\theta, \hat{\Psi}.M/u) : \Delta', u{::}A[\Psi]}$$

Finally, we are in the position to give a uniform calculus which introduces meta-variables in the rule $\Pi\mathsf{L}$, and delays their instantiation to the hyp rule where we rely on higher-order unification to find the correct instantiation. Since higher-order unification is undecidable in general we restrict it to the pattern fragment.

$\Delta; \Gamma \Longrightarrow A/(\theta, \Delta')$	There is a *uniform proof* for A from the assumptions in Γ where θ is a contextual substitution which instantiates the meta-variable in Δ and has range Δ'
$\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta')$	There is a *focused proof* for the atom P focusing on the proposition A using the assumptions in Γ where θ is a contextual substitution which instantiates the meta-variable in Δ and has range Δ'

In the rule $\Pi\mathsf{L}$ we introduce a new meta-variable $u[\mathsf{id}_\Gamma]$ of type $A[\Gamma]$. This means we introduce a meta-variable whose instantiation can depend on all the parameters occurring in Γ. In the hypothesis rule, we rely on higher-order pattern unification to find the most general unifier θ of P' and P, s.t. $[\![\theta]\!]P' = [\![\theta]\!]P$.

$$\frac{\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta') \quad A \in \Gamma}{\Delta; \Gamma \Longrightarrow P/(\theta, \Delta')} \qquad \frac{\Delta; \Gamma \vdash P' \doteq P/(\theta, \Delta')}{\Delta; \Gamma \gg P' \Longrightarrow P/(\theta, \Delta')}$$

$$\frac{\Delta; \Gamma, A_1 \Longrightarrow A_2/(\theta, \Delta')}{\Delta; \Gamma \Longrightarrow A_1 \to A_2/(\theta, \Delta')} \qquad \frac{\Delta; \Gamma \Longrightarrow A_1/(\theta_1, \Delta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \gg [\![\theta_1]\!]A_2 \Longrightarrow [\![\theta_1]\!]P/(\theta_2, \Delta_2)}{\Delta; \Gamma \gg A_1 \to A_2 \Longrightarrow P/([\![\theta_2]\!]\theta_1, \Delta_2)}$$

$$\frac{\Delta; \Gamma, x{:}A \Longrightarrow B/(\theta, \Delta')}{\Delta; \Gamma \Longrightarrow \Pi x{:}A.B\ (\theta, \Delta')} \qquad \frac{\Delta, u{::}A[\Gamma]; \Gamma \gg [u[\mathsf{id}_\Gamma]/x]B \Longrightarrow P/((\theta, \hat{\Gamma}.M/u), \Delta')}{\Delta; \Gamma \gg \Pi x{:}A.B \Longrightarrow P/(\theta, \Delta')}$$

Next, we prove that this system is sound and complete with the uniform proofs where we guess the correct instantiation for the meta-variables.

Theorem 4.1 (Soundness)

(i) *If* $\Delta; \Gamma \Longrightarrow A/(\theta, \Delta')$ *then for any grounding substitution* $\cdot \vdash \rho : \Delta'$ *we have* $\cdot; [\![\rho]\!][\![\theta]\!]\Gamma \Longrightarrow [\![\rho]\!][\![\theta]\!]A$.

(ii) *If* $\Delta;\Gamma \gg A \Longrightarrow P/(\theta,\Delta')$ *then for any grounding substitution* $\cdot \vdash \rho : \Delta'$ *we have* $\cdot;[\![\rho]\!][\![\theta]\!]\Gamma \gg [\![\rho]\!][\![\theta]\!]A \Longrightarrow [\![\rho]\!][\![\theta]\!]P$.

Proof. By structural induction on the first derivation (see also [Pie03]). □

Theorem 4.2 (Completeness)

(i) *If* $\cdot;[\![\rho]\!]\Gamma \Longrightarrow [\![\rho]\!]A$ *for a modal substitution* ρ, *s.t.* $\cdot \vdash \rho : \Delta$
then $\Delta;\Gamma \Longrightarrow A/(\Delta',\theta)$ *for some* θ *and* $\rho = [\![\rho']\!]\theta$ *for some* ρ' *s.t.* $\cdot \vdash \rho' : \Delta'$.

(ii) *If* $\cdot;[\![\rho]\!]\Gamma \gg [\![\rho]\!]A \Longrightarrow [\![\rho]\!]P$ *for a modal substitution* ρ *s.t.* $\cdot \vdash \rho : \Delta$
then $\Delta;\Gamma \gg A \Longrightarrow P/(\Delta',\theta)$ *for some* θ *and* $\rho = [\![\rho']\!]\theta$ *for some* ρ', *s.t.* $\cdot \vdash \rho' : \Delta'$.

Proof. Simultaneous structural induction on the first derivation (see also [Pie03]).□

5 Inverse method and focusing

An interesting alternative to backward proof search, is forward proof search based on the inverse method. This has potentially many advantages. While backward logic programming based depth first search is incomplete and requires backtracking, forward search provides a complete search strategy without backtracking. Similar to tabling, it also allows us to execute some specifications which were not previously executable. Next, we present a forward uniform proof system where we guess the correct instantiation. We derive this forward calculus from the uniform proof system presented earlier which models backchaining. Hence our system will only distinguish between the left focusing and right uniform phase. To obtain a more general proof-theoretic foundation, one could distinguish between a right-focusing and a left-focusing phase (see for example [CPP06]). Finally, we describe a lifted version with meta-variables.

$\Gamma \overset{f}{\Longrightarrow} A$	A has *forward uniform proof* using the assumptions in Γ
$\Gamma \gg A \overset{f}{\Longrightarrow} P$	P has a *forward focused proof* focusing on the proposition A using the assumptions in Γ

The context Γ is now interpreted differently, in that sequents $\Gamma \overset{f}{\Longrightarrow} A$ and $\Gamma \gg A \overset{f}{\Longrightarrow} P$ assert that all assumptions in Γ as well as A, if the sequent is focused are needed to prove the conclusion. General weakening is thus disallowed but incorporated in the rule $\mathsf{f}{\rightarrow}\mathsf{R}_2$. Since our context Γ keeps track of dynamic assumption and parameters, we do not require it to be completely empty in the rule f-ax. Instead we can think of it as the strongest context in which P is well-typed. Since we want to preserve that contexts Γ are well-typed, we must make sure in the rule $\mathsf{f}{\rightarrow}\mathsf{L}$ that the union two context Γ_1 and Γ_2 is well-typed and preserves the present dependencies between parameter assumptions and dynamic assumptions. The rule f-drop was called in the backwards uniform calculus choose. In the forward direction there is no choice but rather we must drop the formula out of the focus.

$$\frac{\Gamma \gg A \overset{f}{\Longrightarrow} P}{\Gamma \cup \{A\} \overset{f}{\Longrightarrow} P}\ \mathsf{f\text{-}drop} \qquad \frac{}{\Gamma \gg P \overset{f}{\Longrightarrow} P}\ \mathsf{f\text{-}ax}$$

$$\frac{\Gamma, c{:}A_1 \overset{f}{\Longrightarrow} A_2}{\Gamma \overset{f}{\Longrightarrow} A_1 \to A_2}\ \mathsf{f{\to}R_1} \qquad \frac{\Gamma \overset{f}{\Longrightarrow} A_2}{\Gamma \overset{f}{\Longrightarrow} A_1 \to A_2}\ \mathsf{f{\to}R_2} \qquad \frac{\Gamma_1 \overset{f}{\Longrightarrow} A_1 \quad \Gamma_2 \gg A_2 \overset{f}{\Longrightarrow} P}{\Gamma_1 \cup \Gamma_2 \gg A_1 \to A_2 \overset{f}{\Longrightarrow} P}\ \mathsf{f{\to}L}$$

$$\frac{\Gamma, x{:}A \overset{f}{\Longrightarrow} B}{\Gamma \overset{f}{\Longrightarrow} \Pi x{:}A.B}\ \mathsf{f\Pi R} \qquad \frac{\Gamma \gg [M/x]B \overset{f}{\Longrightarrow} P \quad \Gamma \vdash M : A}{\Gamma \gg \Pi x{:}A.B \overset{f}{\Longrightarrow} P}\ \mathsf{f\Pi L}$$

Next, we prove soundness and completeness of this forward uniform calculus.

Theorem 5.1 (Soundness)

(i) *If* $\Gamma \overset{f}{\Longrightarrow} A$ *then* $\Gamma \Longrightarrow A$

(ii) *If* $\Gamma \gg A \overset{f}{\Longrightarrow} P$ *then* $\Gamma \gg A \Longrightarrow P$.

Proof. Straightforward structural induction. □

Theorem 5.2 (Completeness)

(i) *If* $\Gamma \Longrightarrow A$ *then* $\Gamma' \overset{f}{\Longrightarrow} A$ *where* $\Gamma' \subseteq \Gamma$.

(ii) *If* $\Gamma \gg A \Longrightarrow P$ *then* $\Gamma' \gg A \overset{f}{\Longrightarrow} P$ *where* $\Gamma' \subseteq \Gamma$

Proof. Straightforward structural induction. □

The forward uniform proof system presented gives rise to a proof search method based on the inverse method central to which is the notion of subformula. We outline the notion of subformulas and present a lifted calculus following the development set out in [DV01]. We adapt the standard definition of subformulas to the higher-order setting where objects may contain meta-variables. The immediate free subformula of the negative occurrence of the formula $\Pi x{:}A.B$ in the context Γ is then $[u[\mathsf{id}_\Gamma]/x]B$. The immediate ground subformula of the negative occurrence of the formula $\Pi x{:}A.B$ in the context Γ is $[M/x]B$. Free signed subformulas and its immediate signed subformulas are defined inductively as follows:

signed subformula	free signed subformula	immediate signed subformula
$(A \to B)^-$	A^+, B^-	A^+, B^-
$(A \to B)^+$	A^-, B^+	A^-, B^+
$(\Pi x{:}A.B)^-$	$([u[\mathsf{id}_\Gamma]/x]B)^-$	$([M/x]B)^-$
$(\Pi x{:}A.B)^+$	$([a/x]B)^+$	$([a/x]B))^+$

Definition 5.3 [Subformula property]

(i) Every derivation of a uniform sequent $\Gamma \Longrightarrow A$ consists of signed ground subformulas of signed formulas in Γ^- and A^+.

(ii) Every derivation of a focused $\Gamma \gg A \Longrightarrow P$ consists of signed ground subformulas of signed forumlas in Γ^- and A^+.

Theorem 5.4 (Ground subformula property of uniform proofs)

(i) *Let $\mathcal{D}$ be a derivation of a signed uniform sequent $\Gamma^- \Longrightarrow A^+$ then every signed uniform sequent $\Gamma_0^- \Longrightarrow A_0^+$ or signed focused sequent $\Gamma_1^- \gg A_1^- \Longrightarrow P_1$ occurring in $\mathcal{D}$ fulfills the subformula property, i.e. $[\Gamma_0^-, A_0^+] < [\Gamma^-, A^+]$ or $[\Gamma_1^-, A_1^-, P_1^+] < [\Gamma^-, A^+]$.*

(ii) *Let $\mathcal{D}$ be a derivation of a signed focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$ then every signed uniform sequent $\Gamma_0^- \Longrightarrow A_0^+$ or signed focused sequent $\Gamma_1^- \gg A_1^- \Longrightarrow P_1$ occurring in $\mathcal{D}$ fulfills the subformula property, i.e. $[\Gamma_0^-, A_0^+] < [\Gamma^-, A^-, P^+]$ or $[\Gamma_1^-, A_1^-, P_1^+] < [\Gamma^-, A^-, P^+]$.*

Proof. By routine inspection of the inference rules for uniform and focused proofs.□

Thus when we search for a proof of a particular signed sequent $\Gamma \Longrightarrow A$ or $\Gamma \gg A \Longrightarrow P$ resp. we can restrict our search to sequents consisting of signed subformulas of $[\Gamma^-, A^+]$. When $[\Gamma^-, A^+]$ contains quantifiers, it may have an infinite number of signed subformulas, so the subformula property does not restrict the search space good enough. However any signed formula has only a finite number of free signed subformulas.

Next, we consider *free signed subformula property.* We will often represent signed subformulas of a given uniform sequent $\Gamma^- \Longrightarrow A^+$ in the form $[\![\theta]\!]\Gamma_0^- \Longrightarrow [\![\theta]\!]A_0^+$, where θ is a substitution from the meta-variables Δ to some ground instance, i.e. $\cdot \vdash \theta : \Delta$ and $\Delta; \Gamma_0^- \Longrightarrow A_0^-$. We call this the representation *via free signed subformula*. Similarly we often represent signed subformulas of a given focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$ in the form $[\![\theta]\!]\Gamma_0^- \gg [\![\theta]\!]A^- \Longrightarrow [\![\theta]\!]P^+$. Moreover, we often write $S = [\Gamma^-, A+]$ as an abbreviation for the sequent $\Gamma \Longrightarrow A$, and $\Delta \vdash S$ as an abbreviation for $\Delta; \Gamma \Longrightarrow A$.

Lemma 5.5 *Let $S_0 = [\Gamma_0^-, A_0^+]$, and $S_1 = [\Gamma_1^-, A_1^+]$ be free signed subformulas s.t. $\Delta_0 \vdash S_0$ and $\Delta_1 \vdash S_1$. Then $[\Gamma_1^-, A_1^+] < [\Gamma_0^-, A_0^+]$ i.e. S_1 is a signed subformula of S_0, iff $S_1 = [\![\theta]\!]S$ for some signed sequent S s.t. S is a free signed subformula of S_0, where $\Delta_1 \vdash \theta : \Delta$ and $\Delta \vdash S$ and $\Delta \cap \Delta_0 = \emptyset$.*

Proof. By inspection of the definition of signed subformulas. □

Every signed subformula of a closed signed formula can be obtained from a free signed subformula by applying a contextual substitution. We now reformulate the subformula property.

Corollary 5.6 (Free subformula property) *Let $\mathcal{D}$ be a derivation of a closed signed uniform sequent $S = \Gamma^- \Longrightarrow A^+$ or closed signed focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$. Every signed sequent occurring in $\mathcal{D}$ has the form $[\![\theta]\!]S_0$ for a free signed sequent S_0 of S and a substitution θ s.t. $\Delta \vdash S_0$ and $\cdot \vdash \theta : \Delta$.*

Suppose we want to check the provability of a closed signed sequent S. By the

previous corollary, we can restrict signed formulas occurring in the derivation to signed sequents of the form $[\![\theta]\!]S_0$ where S_0 is a free signed sequent of S s.t. $\Delta \vdash S_0$ and $\cdot \vdash \theta : \Delta$. Since this applies to axioms as well, every axiom has the form $\Gamma \gg [\![\theta]\!]([\![\rho]\!]P) \rightarrow [\![\theta]\!]P'$ where P and P' are atomic free signed subformulas of the sequent S and $[\![\theta]\!]([\![\rho]\!]P) = [\![\theta]\!]P'$, and ρ is a renaming of meta-variables occurring in P, and Γ characterizes the parameters occurring in $[\![\theta]\!]P'$ and $[\![\theta]\!]([\![\rho]\!]P)$ respectively. For any given P, P' there may be an infinite number of such axioms because of different choices for substitutions θ, but there is only a finite number of pairs of free signed sequents. We can choose a most general axiom that represents all axioms.

We will now introduce a forward calculus $\mathcal{F}^A$ for the inverse method with meta-variables. The calculus is based on the idea of representing sequents through free subformulas and using most general unifiers. Since higher-order unification is only decidable for patterns, we restrict our attention for now to this fragment. A sequent S in the original forward calculus for closed sequents, is an instance of a sequent $[\![\theta]\!]S_0$ in the calculus $\mathcal{F}^A$ if there exists a grounding substitution ρ s.t. $[\![\rho]\!][\![\theta]\!]S_0 = S$. Unlike more standard presentation where we associate a substitution θ with each of the formulas in Γ and the conclusion A, we will associate a substitution θ with a sequent. The judgment $(\Gamma \rightarrow A) \cdot \theta$ denotes a sequent where $[\![\theta]\!]\Gamma \rightarrow [\![\theta]\!]A$. This will be easier to implement, and models more closely our prototype.

$$\frac{(\Delta; \Gamma \gg A \overset{f}{\Longrightarrow} P) \cdot \theta}{(\Delta; \Gamma \cup \{A\} \overset{f}{\Longrightarrow} P) \cdot \theta} \qquad \frac{\Delta; \Gamma \vdash [\![\rho]\!]P' \doteq P/\theta}{(\Delta; \Gamma \gg [\![\rho]\!]P' \overset{f}{\Longrightarrow} P) \cdot \theta}$$

$$\frac{(\Delta; \Gamma \overset{f}{\Longrightarrow} B) \cdot \theta}{(\Delta; \Gamma \overset{f}{\Longrightarrow} (A \rightarrow B)) \cdot \theta} \qquad \frac{(\Delta; \Gamma, c{:}A_1 \overset{f}{\Longrightarrow} A_2) \cdot \theta}{(\Delta; \Gamma \overset{f}{\Longrightarrow} (A_1 \rightarrow A_2)) \cdot \theta}$$

$$\frac{(\Delta_1; \Gamma_1 \overset{f}{\Longrightarrow} A_1) \cdot \theta_1 \qquad (\Delta_2; \Gamma_2 \gg A_2 \overset{f}{\Longrightarrow} P) \cdot \theta_2}{(((\Delta_1 \cup \Delta_2); \Gamma_1 \cup \Gamma_2) \gg (A_1 \rightarrow A_2) \overset{f}{\Longrightarrow} P) \cdot [\![\theta]\!]\theta_1'} \quad \begin{array}{r} \text{where } \mathrm{ext}(\Delta_1 \cup \Delta_2, \theta_1) = \theta_1' \\ \mathrm{ext}(\Delta_1 \cup \Delta_2, \theta_2) = \theta_2' \\ \mathsf{mgu}(\theta_1', \theta_2') = \theta \end{array}$$

$$\frac{(\Delta; \Gamma, x{:}A \overset{f}{\Longrightarrow} B) \cdot \theta}{(\Delta; \Gamma \overset{f}{\Longrightarrow} (\Pi x{:}A.B) \cdot \theta} \qquad \frac{(\Delta, u{::}A[\Gamma]; \Gamma \gg [u[\mathsf{id}_\Gamma]/x]B \overset{f}{\Longrightarrow} P) \cdot (\theta, \hat{\Gamma}.M/u) \quad u \text{ is new}}{(\Delta; \Gamma \gg (\Pi x{:}A.B) \overset{f}{\Longrightarrow} P) \cdot \theta}$$

In the hypothesis rule where we unify the assumption $[\![\rho]\!]P'$ with P we keep a context Γ which describes the parameters occurring in P' and P. As mentioned earlier, typical formulations of forward calculi require the context to be empty, since they do not keep track explicitly of the parameters introduced during proof search. Due to the dependent nature of our calculus, and the fact that we would like to preserve that all propositions are well-typed, we keep track of parameters explicitly and allow the context Γ in this hypothesis rule to be non-empty. Our intention is that Γ describes all the parameters occurring in P and P'. This is largely straightforward. In the implication left rule, we must union not only the assumptions in Γ_1 and in Γ_2, but we also must union the meta-variables occurring in both branches. Since meta-variables occurring in both branches of the proof, may have been instantiated differently, we must reconcile their different instantiations in

θ_1 and θ_2 by unifying them. Before we can unify them we first extend them with identity substitution s.t. they share the same domain. This extension is denoted with $\text{ext}(\Delta_1 \cup \Delta_2, \theta_1) = \theta_1'$ and $\text{ext}(\Delta_1 \cup \Delta_2, \theta_2) = \theta_2'$ respectively.

Theorem 5.7 (Soundness)

(i) *If* $(\Gamma \stackrel{f}{\Longrightarrow} A) \cdot \theta$ *then for any grounding substitution* ρ*, we have* $[\![\rho]\!]([\![\theta]\!]\Gamma) \stackrel{f}{\Longrightarrow} [\![\rho]\!][\![\theta]\!]A$.

(ii) *If* $(\Gamma \gg A \stackrel{f}{\Longrightarrow} P) \cdot \theta$ *then for any grounding substitution* ρ*, we have* $[\![\rho]\!]([\![\theta]\!]\Gamma) \gg [\![\rho]\!]([\![\theta]\!]A) \stackrel{f}{\Longrightarrow} [\![\rho]\!][\![\theta]\!]P$.

Proof. Proof by induction on the first derivation. □

Theorem 5.8 (Completeness) *Suppose* $\Gamma \stackrel{f}{\Longrightarrow} [\![\theta]\!]A$ *(resp.* $\Gamma \gg [\![\theta]\!]A \stackrel{f}{\Longrightarrow} [\![\theta]\!]P$*) and* $\Gamma = [\![\theta_1]\!]A_1, \ldots [\![\theta_n]\!]A_n$ *where* $A^+, A_1^-, \ldots A_n^-$ *are signed free subformulas of the goal. Then there exist a substitution* θ' *and a grounding substitution* ρ *such that:*

(i) $(A_1, \ldots A_n \stackrel{f}{\Longrightarrow} A) \cdot \theta'$ *(resp.* $(A_1, \ldots A_n \gg A \stackrel{f}{\Longrightarrow} P) \cdot \theta'$*)*

(ii) $[\![\rho]\!][\![\theta']\!](A_i) = \theta_i(A_i)$ *and* $[\![\rho]\!][\![\theta']\!](A) = [\![\theta]\!](A)$ *(resp. and* $[\![\rho]\!][\![\theta']\!](P) = [\![\theta]\!](P)$*)*

Proof. Proof by induction on the first derivation. □

6 Implementation of an inverse method prover for LF

In this section, we discuss the implementation of an inverse method prover for LF by considering the example given earlier. The first step in the inverse method is computation of subformulas. Given a signed formula we compute the set $\mathcal{N}$ of negative subformulas and the set $\mathcal{P}$ of positive subformulas. Each subformula is denoted as $\Delta; \Gamma \vdash a\ M_1 \ldots M_n$ where Δ characterizes the meta-variables, and Γ describes the parameters occurring in $a\ M_1 \ldots M_n$ Given the set $\mathcal{N}$ of negative subformulas and the set $\mathcal{P}$ of positive subformulas, we can generate a *focused axioms*, if a negative subformula unifies with a positive subformula. We compute the minimal set $\mathcal{F}$ of *focused axioms* by checking forward and backward subsumption of newly generated axiom. Following Chaudhuri *et al.*, our implementation creates big-step derived rules by chaining all the focused rules together to form a focused thread and chaining all the uniform rules together to form a uniform thread. Our compiled rules are therefore of the following form

$$\frac{\stackrel{f}{\Longrightarrow} P_1 \quad \ldots \quad \stackrel{f}{\Longrightarrow} P_n}{\stackrel{f}{\Longrightarrow} P}$$

After this pre-compilation phase is finished, we delete the focused axiom, and search over the set $\mathcal{F}$ of uniform facts and the set $\mathcal{R}$ of pre-compiled derived rules.

6.1 Top-level of the inverse method

Next, we must iterate over the set $\mathcal{F}$ of uniform facts and the set $\mathcal{R}$ of pre-compiled derived rules to generate new facts by forward chaining. Essentially we need to plug

the facts into the open premises to generate new facts. There are essentially two possible loop structures which we both briefly discuss. Both of these two loops have been implemented and tested for the Horn fragment.

Iteration over facts The first loop follows essentially ideas used by K. Chaudhuri in his implementation of the inverse method for linear logic [Cha06]. We pick a fact f from the set $\mathcal{F}$ of fact and then use this fact f to generate new pre-instantiated rules and new facts from the set $\mathcal{R}$. Given a rule with the premises $P_1, \ldots, P_n$, we try to unify each P_i with the fact f and generate all pre-instantiated rules for this given fact f. If the fact f unifies with k premises, then we generate possibly up to $2^k - 1$ pre-instantiated rules where k is less than n. If k is equal to n, i.e. all premises can be satisfied, a new fact P is generated which is added to the set $\mathcal{F}$ if there is no fact f' in $\mathcal{F}$ s.t. P is an instance of f'. The set of rules therefore may grow exponentially during execution. However, an advantage is that every fact f will be chosen only once, and only once we unify it with a given premise. We terminate if no new facts have been generated.

Iterate over rules In this alternative implementation, we keep the two sets of facts $\mathcal{F}$ and $\mathcal{F}_n$ and iterate over the set $\mathcal{R}$ of rules. Initially, all facts generated during the pre-compilation phase are in the set $\mathcal{F}_n$ and $\mathcal{F}$ is empty. Given a rule with the premises $P_1, \ldots, P_n$, we try to find a fact f from the set $\mathcal{F}_n$ which unifies with P_1 up to P_n. If we succeed in unifying with P_i, we continue to search over the set $\mathcal{F}$ and $\mathcal{F}_n$ to find instantiations of the remaining premises. If all the premises are unifiable with some fact f, we generate a new fact P which is temporarily added to a set $\mathcal{F}'$, if there is no fact f' in $\mathcal{F}$, $\mathcal{F}_n$ or $\mathcal{F}'$ s.t. P is an instance of f'. This stage will terminate if all rules have been tried with the facts from $\mathcal{F}_n$. Now we add $\mathcal{F}_n$ to the set of facts $\mathcal{F}$ and $\mathcal{F}'$ will be used as our new set of facts $\mathcal{F}_n$. In this loop, the size of $\mathcal{R}$ remains constant. On the other hand, we may unify multiple times a given premise P_i with a given fact from $\mathcal{F}$. We terminate if no new facts are generated, i.e. $\mathcal{F}'$ is empty.

6.2 Experimental results

So far we have completed a prototype for the Horn fragment of LF. In this section, we discuss our preliminary experience and compare the performance with the tabled logic programming engine. We will pay particular attention to the two different implementation strategies of the inverse method. To evaluate and understand the current limitations, we will concentrate here on two examples, the first one computes the Fibonacci numbers, and the second one parses propositional formulas. All experiments are done on a machine with the following specifications: 3.4GHz Intel Pentium, 4.0GB RAM. We are using SML of New Jersey 110.55 under the Linux distribution Gentoo 16.14.under Linux. Times are measured in seconds, and the ∞ indicates we terminated the process after 30min.

Fibonacci example Computing the Fibonacci numbers is an interesting example, because a depth-first search will yield an exponential algorithm. Memoization allows us to re-use the computation of previous subgoals, and we expect its performance to be linear. Similarly, forward search has the potential of re-using results, and should yield a linear time algorithm. We compare the two different implementations for

the inverse method, and the tabled logic programming engine.

			IR	IF	Tab	
k	fib(k)	Facts	Time	Time	Time	♯Entries
14	377	377(add) + 14(fib)	1.48	2.75	0.46 (0.08)	403
15	610	610(add) + 15(fib)	4.41	102.37	1.210 (0.07)	638
16	987	987(add) + 16(fib)	11.19	∞	3.135 (0.10)	1017
17	1597	1597(add) + 17(fib)	34.10	∞	6.861 (0.10)	1629
18	2584	2584(add) + 18(fib)	193.79	∞	139.826 (0.16)	2618

IF describes the inverse method where we iterate over facts and generate pre-instantiated rules, and IR denotes the inverse method where we iterate over the rules and the number of rules remains constant. The column Tab lists the runtime when all predicates are tabled. In parenthesis, we list the time if we selectively table only the fib predicate. The number of rules generated by the IF loop is 1470 for $k = 14$ and 2302 for $k = 15$. This is a staggering number compared to the 2 rules used in the IR loop. The high number of rules generated also yields a severe performance penalty. Tabling still outperforms inverse method search, even if we table all predicates in the program. As we can see, there is a severe penalty for tabling if we do not table selectively. In fact, selective tabling yields the best performance and does also outperform depth-first search.

Parsing Parsing algorithms are interesting since we typically would like to mix right and left recursive program clauses to model the right and left associativity properties of implications, conjunctions and disjunction. Clauses for conjunction and disjunction are left recursive, while the program clause for implication is right recursive. This program is not executable via depth-first search, and we compare the performance between the two implementations of the inverse method and tabling.

	IR		IF		Tab	
tokens	time	♯facts	time	♯fact	time	♯entries
5	0.860	138	0.109	2214	0.016	6
7	1.359	138	29.828	3702	0.015	10
9	1.016	138	33.391	3846	0.032	10
11	∞		∞		0.171	18

While the number of rules generated by the IF loop is not quite as large as for Fibonacci, it is still substantial. For 3 tokens, we generate 54 rules up to 182 rules for 9 tokens. This is compared to 13 rules which are generated during the pre-compilation phase in the IR method. These results clearly demonstrate that tabling cannot easily be outperformed. The inverse method is costly, and especially in the implementation IF the number of facts is growing substantially more. Our other implementation of the inverse method where the number of rules remain constant

has fair performance, although it cannot rival tabling.

To gain a better understanding of where the bottleneck lies in the inverse method implementation compared to a tabled implementation, we measured the number of unification failures. Unification is at the heart of proof search, and its performance affects in a crucial way the global efficiency of each of these applications. This is especially the case for the inverse method, since we rely on it to instantiate premises of rules, and to check for subsumption, i.e. is a newly derived uniform fact subsumed by an existing uniform fact. In the parsing example for example, we have over 3 million unification failures during subsumption checking, and over 21,000 unification failures when unifying a premise with a given fact. Let us contrast this to tabled logic programming where we count 70 unification failures all of which are in fact handled by the linear assignment algorithm. To check whether a new subgoal is already in the table no higher-order subsumption check is performed since we only check for α-variance. This strikingly illustrates that the performance of unification has a much greater impact on the inverse method than on tabled proof search.

7 Future Work and Conclusion

We presented the basis for an inverse method prover for the logical framework LF. Following standard development, we presented a forward uniform proof calculus and lifted it to allow for subformulas which may contain meta-variables. While we concentrate here on the logical framework LF, which is the basis of *Twelf*, it seems possible to apply the presented approach to λProlog [NM88] or Isabelle [Pau86], which are based on hereditary Harrop formulas. Moreover, we proved the correctness of forward uniform proof calculus. Finally, we discuss challenges when implementing an inverse method prover for the logical framework LF.

In the future we intend to extend our implementation of the inverse method to hereditary Harrop formulas and cover the full higher-order fragment. To achieve a basic implementation seems not that difficult, however to build an inverse method prover with competitive performance we must tackle several issues. The first issue is efficient higher-order unification which seems central to the inverse method. Related to this issue is the fact that our theoretical development and implementation only deals with higher-order patterns where unification is decidable. To handle the full fragment of higher-order terms, we carefully need to revisit the issue of constraints.

Another important question is how to bound the inverse method search. While we do get a decision procedure when we execute the parsing algorithm with tabling, the inverse method does not directly yield a decision procedure. One way of addressing this problem may be to incorporate ideas from Chaudhuri *et al.* [CPP06] and distinguish not only between left focusing and uniform proofs, but also introduce a right focusing phase. As observed in [CPP06], this may have a substantial effect on performance. However, it remains unclear how to in general classify atoms as being left or right biased or mix the two biases. Extending the given theoretical framework to consider different bias for atoms is in principle possible.

Finally an important question is how to bring some goal-directed search into the inverse method. While the subformula property restricts the proof search on the level of formulas, it does not restrict the possible instantiations for the objects

occurring in formulas. This has been already observed in the logic programming community and lead to the development of magic sets [Ram91]. Magic sets transform the original program in such a way that a forward chaining logic programming engine is goal-directed and will only generate the relevant subgoals for a given query. Incorporating magic sets into the inverse method could substantially reduce the number of generated intermediate goals, and only generate relevant subgoals thereby yielding a competitive engine compared to backward chaining logic programming.

References

[ABF+05] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Oxford, UK, August 22-25*, volume 3603 of *Lecture Notes in Computer Science(LNCS)*, pages 50–65. Springer, 2005.

[Cha06] Kaustuv Chaudhuri. The focused inverse method for linear logic. Technical report, Department of Computer Science,, December 2006. CMU-CS-06-162.

[CPP06] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. In U. Furbach and N. Shankar, editors, *Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, USA*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 2006.

[DV01] Anatoli Degtyarev and Andrei Voronkov. The inverse method. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 179–272. Elsevier and MIT Press, 2001.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[NPP06] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. *ACM Transactions on Computational Logic (accepted, to appear in 2007)*, page 56 pages, 2006.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pie02a] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.

[Pie02b] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pie03] Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Sciences, Carnegie Mellon University, December 2003. CMU-CS-03-185.

[Pie05] Brigitte Pientka. Tabling for higher-order logic programming. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction (CADE), Talinn, Estonia*, volume 3632 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.

[PS99] Frank Pfenning and Carsten 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)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.

[Ram91] Raghu Ramakrishnan. Magic templates: a spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3-4):189–216, 1991.

Formalising in Nominal Isabelle Crary's Completeness Proof for Equivalence Checking

Julien Narboux[1] and Christian Urban[2]

TU Munich, Germany

Abstract

In the book on Advanced Topics in Types and Programming Languages, Crary illustrates the reasoning technique of logical relations in a case study about equivalence checking. He presents a type-driven equivalence checking algorithm and verifies its completeness with respect to a definitional characterisation of equivalence. We present in this paper a formalisation of Crary's proof using Isabelle/HOL and the nominal datatype package.

Keywords: logical relations, proof assistants, formalisations, Isabelle/HOL, nominal logic work.

1 Introduction

Logical relations are a powerful reasoning technique for establishing properties about programming languages. The idea of logical relations goes back to Tait [8] and is usually employed for showing strong normalisation results. However this technique has wide applicability. Crary illustrates this by using a logical relation argument to prove completeness of an equivalence checking algorithm [3]. One reason for formalising proofs involving logical relations is that they are fairly intricate: First they require a logic that is sufficiently strong (see comment in [4, Page 58]). Also in the final step of such proofs, one has to establish by induction a property under a closing substitution. These substitutions might, however, interfere with binders and one has to be careful that the proof covers all cases that are required by the induction. We will show in this formalisation that there are a few places where one has to pay attention to this issue and that the strong induction principles [10] that have the variable convention already built in are quite convenient to get the formal arguments through.

There have already been a number of formalisations of proofs involving logical relations. For example Altenkirch [1] formalises the usual strong normalisation proof for

[1] Email: narboux(at)in.tum.de
[2] Email: urbanc(at)in.tum.de

System F in the theorem prover LEGO. To our knowledge all these formalisations use de Bruijn indices to represent α-equated terms. We attribute this to the fact that proofs using logical relations heavily rely on terms being a representation for α-equivalence classes. We assume that this is the reason why a formalisation based on a concrete (un-quotioned) representation has never been attempted.

One practical reason why we do not wish to formalise Crary's proof using de Bruijn indices is that we like to stay as faithful as possible to the source and thus do not need to invent any of the formal arguments ourselves. This intention materialised quite a bit in our formalisation, except in one place where we developed a completely different argument than the one Crary had mind, but did not completely spell out its details (we found this out after we completed the formalisation by communicating with Crary about our proof). Even so we also had to spend considerable work to implement the informal rules presented by Crary and to justify that our implementation captures the intended behaviour of these rules.

Our formal proof is carried out in Isabelle/HOL and relies much on the infrastructure provided by the nominal datatype package [9,10,11]. This package uses many ideas from the nominal logic work by Pitts [6]. The ability to directly define in the nominal datatype package α-equivalent terms and obtain automatically recursion combinators and strong induction principles that have the usual variable convention already built was of great help in our formalisation. There is one place were we had to derive manually some infrastructure, which we hope can be derived automatically in the future. In the rest of the paper we give a guided tour through our formalisation.

2 Terms, Types and Substitution

Terms, types and substitutions are relatively standard and follow closely Crary's notes. Terms are given by the grammar

Definition 2.1 (Terms)

$trm ::= Var\ name \mid App\ trm\ trm \mid Lam\ name.trm \mid Const\ nat \mid Unit$

where in the *Lam*-clause, as usual, a variable is bound; there is also an infinite supply of constants all represented by natural numbers. By stating this definition in the nominal datatype package we immediately obtain α-equivalent terms. Types are given by the grammar

Definition 2.2 (Types) $ty ::= TBase \mid TUnit \mid ty \rightarrow ty$

where there is no binding. We define the usual size function for types (details omitted), as this will be the measure over which we define the logical relation later on.

The most important operation we need for our terms is that of applying simultaneous substitutions, which we represent as finite lists of (*name*,*trm*)-pairs. Crary defines them as functions from some set of variables to terms. One reason for our choice is that it is easier to deal with finitary structures in the nominal datatype package than with infinite ones (functions are considered as infinitary structures and would require additional theorem prover code). Using our list representation we define:

Definition 2.3 (Simultaneous Substitution)

$$\begin{array}{lcll} \theta(\mathit{Var}\ x) & = & \mathit{lookup}\ \theta\ x & \\ \theta(\mathit{App}\ t_1\ t_2) & = & \mathit{App}\ \theta(t_1)\ \theta(t_2) & \\ \theta(\mathit{Lam}\ x.t) & = & \mathit{Lam}\ x.\theta(t) & \text{provided } x\ \#\ \theta \\ \theta(\mathit{Const}\ n) & = & \mathit{Const}\ n & \\ \theta(\mathit{Unit}) & = & \mathit{Unit} & \end{array}$$

where in the first clause we use the auxiliary function *lookup* defined by the clauses:

$$\begin{array}{lcl} \mathit{lookup}\ []\ x & = & \mathit{Var}\ x \\ \mathit{lookup}\ ((y, T)::\theta)\ x & = & \mathit{if}\ x = y\ \mathit{then}\ T\ \mathit{else}\ \mathit{lookup}\ \theta\ x \end{array}$$

Single substitutions are a derived concept by defining $e[x::=e'] \stackrel{\text{def}}{=} [(x, e')](e)$ with $[(x, e')]$ being a singleton list.

Note that in the *Lam*-clause we attach the side-condition about x being fresh for θ (written $x\ \#\ \theta$), which is equivalent to x being not free in the list of (*name*,*trm*)-pairs. Despite imposing this side-condition, the definition above yields a total function, since we work with α-equivalence classes where renamings are always possible. Because we define a function over the structure of α-equated terms, we must be careful to not introduce any inconsistencies [9]. The reason is that we can specify functions over the structure of such terms that do not respect α-equivalence (for example the function that calculates the bound names of a term or returns the immediate subterms) and consequently lead to inconsistencies in Isabelle/HOL. In our formalisation this means that we have to give two four-line proofs that ensure that simultaneous substitutions respect α-equivalence.

3 Typing and Definitional Equivalence

Next, we define the typing judgements for our terms. In order to stay faithful to Crary's notes we introduce the notion for when a typing context Γ is *valid*, namely when it includes only a single association for every variable occurring in Γ. Again we use lists to represent these typing contexts; this time because Isabelle/HOL does not provide out-of-the-box a type of finite sets. Using the lists we can define the notion of validity by the two rules:

$$\frac{}{\mathit{valid}\ []} \qquad \frac{\mathit{valid}\ \Gamma \qquad x\ \#\ \Gamma}{\mathit{valid}\ ((x, T)::\Gamma)}$$

where we attach in the second rule the side-condition that x must be fresh for Γ, which in case of our typing contexts is equivalent to x not occurring in Γ. The typing rules are then defined as:

$$\frac{\mathit{valid}\ \Gamma \qquad (x, T) \in \Gamma}{\Gamma \vdash \mathit{Var}\ x : T}\text{T-Var} \qquad \frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash \mathit{App}\ e_1\ e_2 : T_2}\text{T-App}$$

$$\frac{x\ \#\ \Gamma \qquad (x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \mathit{Lam}\ x.t : T_1 \rightarrow T_2}\text{T-Lam}$$

$$\frac{\mathit{valid}\ \Gamma}{\Gamma \vdash \mathit{Const}\ n : \mathit{TBase}}\text{T-Const} \qquad \frac{\mathit{valid}\ \Gamma}{\Gamma \vdash \mathit{Unit} : \mathit{TUnit}}\text{T-Unit}$$

where we ensure that only valid contexts appear in typing judgements by including validity in the rules for variables and Units. To preserve validity in the rule T-Lam, we have the side-condition that x must be fresh for Γ. (We can infer this freshness condition also from

the premise $(x, T_1)::\Gamma \vdash t : T_2$ and the fact that in typing-judgements contexts are always valid, but this requires a side-lemma.) In rule T-Var we use the notation $(x, T) \in \Gamma$ to stand for list-membership.

The completeness of the typing algorithm is proved with respect to some rules characterising definitionally the equivalence between typed terms. The corresponding judgements Crary is using for this are of the form $\Gamma \vdash s \equiv t : T$ where s and t are terms and T is a type. We formalise his rules of definitional equivalence as follows:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t : T}\text{Q-Refl} \qquad \frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T}\text{Q-Symm} \qquad \frac{\Gamma \vdash s \equiv t : T \qquad \Gamma \vdash t \equiv u : T}{\Gamma \vdash s \equiv u : T}\text{Q-Trans}$$

$$\frac{\Gamma \vdash s_1 \equiv t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash \mathit{App}\ s_1\ s_2 \equiv \mathit{App}\ t_1\ t_2 : T_2}\text{Q-App}$$

$$\frac{x\ \#\ \Gamma \qquad (x, T_1)::\Gamma \vdash s_2 \equiv t_2 : T_2}{\Gamma \vdash \mathit{Lam}\ x.s_2 \equiv \mathit{Lam}\ x.t_2 : T_1 \rightarrow T_2}\text{Q-Abs}$$

$$\frac{x\ \#\ (\Gamma, s_2, t_2) \qquad (x, T_1)::\Gamma \vdash s_1 \equiv t_1 : T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash \mathit{App}\ (\mathit{Lam}\ x.s_1)\ s_2 \equiv t_1[x::=t_2] : T_2}\text{Q-Beta}$$

$$\frac{x\ \#\ (\Gamma, s, t) \qquad (x, T_1)::\Gamma \vdash \mathit{App}\ s\ (\mathit{Var}\ x) \equiv \mathit{App}\ t\ (\mathit{Var}\ x) : T_2}{\Gamma \vdash s \equiv t : T_1 \rightarrow T_2}\text{Q-Ext}$$

$$\frac{\Gamma \vdash s : \mathit{TUnit} \qquad \Gamma \vdash t : \mathit{TUnit}}{\Gamma \vdash s \equiv t : \mathit{TUnit}}\text{Q-Unit}$$

Validity of the typing contexts are implied by the validity in the typing rules, which are included in the premises of Q-Refl and Q-Unit, and by having the side-condition about x being fresh for Γ in Q-Abs, Q-Beta and Q-Ext.

Comparing our rules with the ones given by Crary, slightly unusual are the side-conditions $x\ \#\ (s_2, t_2)$ in the rule Q-Beta and $x\ \#\ (s, t)$ in the rule Q-Ext. In the former case we can relatively easily show that our Q-Beta is equivalent to

$$\frac{(x, T_1)::\Gamma \vdash s_1 \equiv t_1 : T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash \mathit{App}\ (\mathit{Lam}\ x.s_1)\ s_2 \equiv t_1[x::=t_2] : T_2}\text{Q-Beta'}.$$

However this requires explicit α-conversions and the fact that all typing contexts in the definitional equivalence judgements are valid. In light of this equivalence, the question arises why we insist on the more restricted rule: The reason is that based on those constraints the nominal datatype package can automatically derive a strong induction principle that has the variable convention already built in. This will be very convenient in some proofs later on. To do the same without those constraints is possible, but slightly more laborious.

In case of Q-Ext the side-conditions represent the fact that the extensionality rule should hold for a fresh variable x only. By imposing $x\ \#\ (\Gamma, s, t)$ we can show that Q-Ext is equivalent to

$$\frac{\forall x.\ x\ \#\ \Gamma \longrightarrow (x, T_1)::\Gamma \vdash \mathit{App}\ s\ (\mathit{Var}\ x) \equiv \mathit{App}\ t\ (\mathit{Var}\ x) : T_2}{\Gamma \vdash s \equiv t : T_1 \rightarrow T_2}\text{Q-Ext'}$$

The argument for this uses the some/any-property from [6] and relies on the fact that the definitional equivalence is equivariant; by this we mean it is invariant under swapping of variables, namely $\Gamma \vdash s \equiv t : T$ implies $(x\ y)\cdot\Gamma \vdash (x\ y)\cdot s \equiv (x\ y)\cdot t : T$ for all x and y (see

[10,11] for more details). The side-conditions in Q-Ext are not explicitly given by Crary and the equivalence with Q-Ext' gave us confidence to have captured with them the "idea" of an extensionality rule.

4 The Equivalence Checking Algorithm

One feature of Crary's equivalence checking algorithm is that it includes a fair amount of optimisations, in the sense that in some circumstances two lambda terms are not completely normalised but only transformed into a weak-head normal-form. For this Crary introduces the following four rules:

$$\frac{}{App\ (Lam\ x.t_1)\ t_2 \leadsto t_1[x::=t_2]}\text{QAR-Beta} \qquad \frac{t_1 \leadsto t_1'}{App\ t_1\ t_2 \leadsto App\ t_1'\ t_2}\text{QAR-App}$$

$$\frac{s \leadsto t \qquad t \Downarrow u}{s \Downarrow u}\text{QAN-Reduce} \qquad \frac{t \not\leadsto}{t \Downarrow t}\text{QAN-Normal}$$

The algorithm is then defined by two mutual recursive judgements, called respectively *algorithmic term equivalence* and *algorithmic path equivalence*. The former is written as $\Gamma \vdash s \Leftrightarrow t : T$ and the latter as $\Gamma \vdash s \leftrightarrow t : T$. Their rules are

$$\frac{s \Downarrow p \qquad t \Downarrow q \qquad \Gamma \vdash p \leftrightarrow q : TBase}{\Gamma \vdash s \Leftrightarrow t : TBase}\text{QAT-Base}$$

$$\frac{x\ \#\ (\Gamma, s, t) \qquad (x, T_1)::\Gamma \vdash App\ s\ (Var\ x) \Leftrightarrow App\ t\ (Var\ x) : T_2}{\Gamma \vdash s \Leftrightarrow t : T_1 \to T_2}\text{QAT-Arrow}$$

$$\frac{valid\ \Gamma}{\Gamma \vdash s \Leftrightarrow t : TUnit}\text{QAT-One}$$

$$\frac{valid\ \Gamma \qquad (x, T) \in \Gamma}{\Gamma \vdash Var\ x \leftrightarrow Var\ x : T}\text{QAP-Var}$$

$$\frac{\Gamma \vdash p \leftrightarrow q : T_1 \to T_2 \qquad \Gamma \vdash s \Leftrightarrow t : T_1}{\Gamma \vdash App\ p\ s \leftrightarrow App\ q\ t : T_2}\text{QAP-App}$$

$$\frac{valid\ \Gamma}{\Gamma \vdash Const\ n \leftrightarrow Const\ n : TBase}\text{QAP-Const}$$

following quite closely Crary's definition. One difference, however, is the inclusion of the validity predicate in the rules QAT-One, QAP-Var and QAP-Const ensuring that only valid typing contexts appear in the judgements. Another, more interesting, difference is the fact that by imposing the side-condition $x\ \#\ (s, t)$ in the rule rule QAT-Arrow we explicitly restricting the algorithm to consider only fresh variables. Recall that we imposed a similar restriction in the rule Q-Beta given in Sec. 3. There, however, the side-condition was innocuous as we could show that the rule *with* the side-condition is equivalent to the one *without* the side-condition. With rule QAT-Arrow the situation is different—the side-condition is a "real" restriction, meaning that

$$\frac{x\ \#\ \Gamma \qquad (x, T_1)::\Gamma \vdash App\ s\ (Var\ x) \Leftrightarrow App\ t\ (Var\ x) : T_2}{\Gamma \vdash s \Leftrightarrow t : T_1 \to T_2}$$

and QAT-Arrow are *not* interderivable. (The reason for this is that in the judgement $\Gamma \vdash s \Leftrightarrow t : T$ the free variables of s and t do not necessarily need to be contained in Γ. Therefore we cannot infer from $x \# \Gamma$ that $x \# (s, t)$ holds, as we did with Q-Beta.) While this restriction seems reasonable from an algorithmic point of view, it will turn out that it is actually crucial in our proofs: in order to get the inductions through for the properties of transitivity and monotonicity for the rules given above, we like to assume a sort of variable convention for x. That means we like to structure our argument so that the x in case of QAT-Arrow is fresh not just for Γ, s and t, but also for some other variables specified in the lemma at hand. This is very much like the informal reasoning using the variable convention, except that x in QAT-Arrow is not a binder. Still the nominal datatype package is able to derive automatically a strong induction principle, which allows us later on to make the reasoning with the variable convention completely formal. One proviso for deriving this strong induction principle is however that we formulate the QAT-Arrow as we have (essentially we have to make sure that the variable x does not occur freely in the conclusion of the corresponding rule; for more details we refer again to [10]). To see the improvement we obtain with the strong induction principle, consider the usual induction principle that comes with the rules specified above:

$$\frac{\begin{array}{l}\ldots\\ \forall x\,\Gamma\,s\,t\,T_1\,T_2.\\ \quad x\#(\Gamma,s,t)\ \wedge\ P_1\,((x:T_1)::\Gamma)\,(App\,s\,(Var\,x))\,(App\,t\,(Var\,x))\,T_2\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow P_1\,\Gamma\,s\,t\,(T_1 \to T_2)\\ \ldots\end{array}}{\Gamma \vdash s \Leftrightarrow t : T \longrightarrow P_1\,\Gamma\,s\,t\,T \qquad \Gamma \vdash s \leftrightarrow t : T \longrightarrow P_2\,\Gamma\,s\,t\,T}$$

This induction principle states that if one wants to prove two properties P_1 and P_2 by mutual induction over the rules for algorithmic term equivalence and algorithmic path equivalence, then one can assume in the QAT-Arrow the side-condition $x\#(\Gamma, s, t)$ and P_1 for the premise, and one has to establish P_1 for the conclusion. The strong induction principle is similar

$$\frac{\begin{array}{l}\ldots\\ \forall x\,\Gamma\,s\,t\,T_1\,T_2\,c.\\ \quad x\#c\ \wedge\ x\#(\Gamma,s,t)\ \wedge\ (\forall c.\ P_1\,c\,((x:T_1)::\Gamma)\,(App\,s\,(Var\,x))\,(App\,t\,(Var\,x))\,T_2)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow P_1\,c\,\Gamma\,s\,t\,(T_1 \to T_2)\\ \ldots\end{array}}{\Gamma \vdash s \Leftrightarrow t : T \longrightarrow P_1\,c\,\Gamma\,s\,t\,T \qquad \Gamma \vdash s \leftrightarrow t : T \longrightarrow P_2\,c\,\Gamma\,s\,t\,T}$$

except that it includes an induction context c in the properties P_1 and P_2, and we can assume that in the QAT-Arrow-case the x is fresh with respect to this induction context (see highlighted box). Over this induction context we have control when we set up an induction: if we want to employ the variable convention in our formal proofs, we just need to instantiate this induction context appropriately.

Before we can describe our proofs in detail we need two more definitions. We need to formalise Crary's notion of logical equivalence, written $\Gamma \vdash s\ is\ t : T$, and the logical equivalence of two simultaneous substitutions, say θ_1 and θ_2, over a context Γ. The latter is a derived concept and will be written as $\Gamma' \vdash \theta\ is\ \theta'\ over\ \Gamma$. The former is defined by recursion over the size of the types. The clauses are as follows:

$\Gamma \vdash s \text{ is } t : \mathit{TUnit} \stackrel{\text{def}}{=} \mathit{true}$

$\Gamma \vdash s \text{ is } t : \mathit{TBase} \stackrel{\text{def}}{=} \Gamma \vdash s \Leftrightarrow t : \mathit{TBase}$

$\Gamma \vdash s \text{ is } t : (T_1 \to T_2) \stackrel{\text{def}}{=} \forall \Gamma' s' t'.\ \Gamma \subseteq \Gamma' \wedge \mathit{valid}\ \Gamma' \wedge \Gamma' \vdash s' \text{ is } t' : T_1 \longrightarrow$
$\Gamma' \vdash (\mathit{App}\ s\ s') \text{ is } (\mathit{App}\ t\ t') : T_2$

using in the last clause the notion of a weaker context, written $\Gamma \subseteq \Gamma'$ (for Γ' to be weaker than Γ, every (name,type)-pair in Γ must also appear in Γ'). Logical equivalence for simultaneous substitutions over a context Γ is defined as

$$\Gamma' \vdash \theta \text{ is } \theta' \text{ over } \Gamma \stackrel{\text{def}}{=} \forall x\ T.\ (x,T) \in \mathit{set}\ \Gamma \longrightarrow \Gamma' \vdash \theta(\mathit{Var}\ x) \text{ is } \theta'(\mathit{Var}\ x) : T$$

With this we have all necessary definitions in place.

5 Proofs

The first mayor property we need to establish is transitivity for algorithmic term equivalence and algorithmic path equivalence. These proofs are not detailed in Crary's notes and we diverged in our formalisation from the proofs he had in mind. We first show that type unicity holds for algorithmic path equivalence

Lemma 5.1 (Type Unicity)
If $\Gamma \vdash s \leftrightarrow t : T$ and $\Gamma \vdash s \leftrightarrow u : T'$ then $T = T'$.

and subsequently show symmetry for both the algorithmic path equivalence and the algorithmic term equivalence.

Lemma 5.2 (Algorithmic Symmetry)
If $\Gamma \vdash s \Leftrightarrow t : T$ then $\Gamma \vdash t \Leftrightarrow s : T$.
If $\Gamma \vdash s \leftrightarrow t : T$ then $\Gamma \vdash t \leftrightarrow s : T$.

Both proofs are by relatively straightforward inductions over $\Gamma \vdash s \Leftrightarrow t : T$ and $\Gamma \vdash s \leftrightarrow t : T$. This then allows us to prove transitivity, where we need the strong induction principle in order to get the induction through.

Lemma 5.3 (Algorithmic Transitivity)
If $\Gamma \vdash s \Leftrightarrow t : T$ and $\Gamma \vdash t \Leftrightarrow u : T$ then $\Gamma \vdash s \Leftrightarrow u : T$.
If $\Gamma \vdash s \leftrightarrow t : T$ and $\Gamma \vdash t \leftrightarrow u : T$ then $\Gamma \vdash s \leftrightarrow u : T$.

Proof. By mutual induction over $\Gamma \vdash s \Leftrightarrow t : T$ and $\Gamma \vdash s \leftrightarrow t : T$ where we instantiate the induction context with the term u. In the QAP-App-case we then have the induction hypotheses

$$ih_1: \quad \forall u.\ \Gamma \vdash q \leftrightarrow u : T_1 \to T_2 \longrightarrow \Gamma \vdash p \leftrightarrow u : T_1 \to T_2$$

$$ih_2: \quad \forall u.\ \Gamma \vdash t \Leftrightarrow u : T_1 \longrightarrow \Gamma \vdash s \Leftrightarrow u : T_1$$

and the assumptions

$$(i)\text{: } \Gamma \vdash \mathit{App}\ q\ t \leftrightarrow u : T_2 \quad \text{and} \quad (ii)\text{: } \Gamma \vdash p \leftrightarrow q : T_1 \to T_2$$

From the first assumption we obtain by inversion of the typing rule an r, T'_1 and v such that

$$(iii)\colon \Gamma \vdash q \leftrightarrow r : T'_1 \to T_2 \quad (iv)\colon \Gamma \vdash t \Leftrightarrow v : T'_1$$

and $u = App\ r\ v$ hold. From the second assumption we obtain $\Gamma \vdash q \leftrightarrow p : T_1 \to T_2$ by symmetry of $\leftrightarrow$ (Lemma 5.2), and then can use this and (iii) to find out by the type unicity of $\leftrightarrow$ (Lemma 5.1) that $T'_1 \to T_2 = T_1 \to T_2$ holds. This in turn implies that $T'_1 = T_1$, which allows us to use (iii) and (iv) in the induction hypotheses. This gives us

$$\Gamma \vdash s \Leftrightarrow v : T_1 \quad \text{and} \quad \Gamma \vdash p \leftrightarrow r : T_1 \to T_2\ .$$

Hence we know that $\Gamma \vdash App\ p\ s \leftrightarrow u : T_2$ holds by the rule QAP-App and the equation $u = App\ r\ v$.

The case QAT-Base uses the fact that normalisation produces unique results, that is if $t \Downarrow q$ and $t \Downarrow q'$ then $q = q'$.

In the QAT-Arrow case we have $\Gamma \vdash t \Leftrightarrow u : T_1 \to T_2$ and thus can infer that the judgement $(x, T_1)::\Gamma \vdash App\ t\ (Var\ x) \Leftrightarrow App\ u\ (Var\ x) : T_2$ holds. By induction we obtain further that $(x, T_1)::\Gamma \vdash App\ s\ (Var\ x) \Leftrightarrow App\ u\ (Var\ x) : T_2$ holds. Finally we can infer the proof obligation in this case, namely $\Gamma \vdash s \Leftrightarrow u : T_1 \to T_2$, provided we know $x \# (\Gamma,s,u)$. The freshness for Γ and s is given by the side-conditions of QAT-Arrow. The freshness for u is given by the strong induction principle (recall that we instantiated the induction context with u). Thus we are done. □

Next we prove closure under weak-head reductions, but we restrict our argument to the single step case (Crary proves closure under multiple steps) as this is easier to prove (actually it can be derived automatically by Isabelle's automatic search tools) and is sufficient for our formalisation.

Lemma 5.4 (Algorithmic Weak-Head Closure)
If $\Gamma \vdash s \Leftrightarrow t : T$ and $s' \rightsquigarrow s$ and $t' \rightsquigarrow t$ then $\Gamma \vdash s' \Leftrightarrow t' : T$.

This lemma is by a simple induction over $\Gamma \vdash s \Leftrightarrow t : T$. The following lemma establishes a kind of weakening property for the judgements of the algorithm.

Lemma 5.5 (Algorithmic Monotonicity)
If $\Gamma \vdash s \Leftrightarrow t : T$ and $\Gamma \subseteq \Gamma'$ and $valid\ \Gamma'$ then $\Gamma' \vdash s \Leftrightarrow t : T$.
If $\Gamma \vdash s \leftrightarrow t : T$ and $\Gamma \subseteq \Gamma'$ and $valid\ \Gamma'$ then $\Gamma' \vdash s \leftrightarrow t : T$.

Proof. By mutual induction using the strong induction principle. This time we instantiate the induction context with Γ'. The only interesting case (that is the one which is not automatic) analyses the rule QAT-Arrow: There we have by assumption $\Gamma \subseteq \Gamma'$ from which we can infer $(x,T_1)::\Gamma \subseteq (x,T_1)::\Gamma'$. In order to apply the induction hypotheses, we need the fact that $valid\ ((x, T_1)::\Gamma')$ holds. At this point the usual induction would start to become ugly since explicit renamings need to be performed. However we based our argument on the strong induction principle with the induction context being instantiated with Γ'. This gives us $x \# \Gamma'$ from which we can easily infer the desired fact. We can then conclude in this case with appealing to the induction hypotheses. □

The next lemma will help us to establish the fact that logical equivalence implies algorithmic equivalence.

Lemma 5.6 (Algorithmic Path Equivalence implies Weak-Head-Normal Form)
If $\Gamma \vdash s \leftrightarrow t : T$ then $s \not\rightsquigarrow$ and $t \not\rightsquigarrow$.

This is by straightforward induction on $\Gamma \vdash s \leftrightarrow t : T$. The main lemma in Crary's proof is then stated as follows (where we had to include in our formal version of this lemma that Γ is valid).

Lemma 5.7 (Main Lemma)
If $\Gamma \vdash s \; is \; t : T$ and $valid \; \Gamma$ then $\Gamma \vdash s \Leftrightarrow t : T$.
If $\Gamma \vdash p \leftrightarrow q : T$ then $\Gamma \vdash p \; is \; q : T$.

Proof. The proof is by simultaneous induction over T generalising over Γ, s, t, p and q. The non-trivial case is for $T = T_1 \rightarrow T_2$. For the first property we have the induction hypotheses

$$ih_1: \quad \forall \Gamma \; s \; t. \; \Gamma \vdash s \; is \; t : T_2 \wedge valid \; \Gamma \longrightarrow \Gamma \vdash s \Leftrightarrow t : T_2$$

$$ih_2: \quad \forall \Gamma \; s \; t. \; \Gamma \vdash s \leftrightarrow t : T_1 \longrightarrow \Gamma \vdash s \; is \; t : T_1$$

and the assumptions $\Gamma \vdash s \; is \; t : T_1 \rightarrow T_2$ and $valid \; \Gamma$. We choose a fresh x (fresh w.r.t. Γ, s and t). We can thus derive that $valid \; ((x, T_1)::\Gamma)$ holds and hence $(x, T_1)::\Gamma \vdash Var \; x \leftrightarrow Var \; x : T_1$. From this we can derive $(x, T_1)::\Gamma \vdash Var \; x \; is \; Var \; x : T_1$ using the second induction hypothesis. Using the our assumptions we can then derive $(x, T_1)::\Gamma \vdash App \; s \; (Var \; x) \; is \; App \; t \; (Var \; x) : T_2$ which by the first induction hypothesis leads to $(x, T_1)::\Gamma \vdash App \; s \; (Var \; x) \Leftrightarrow App \; t \; (Var \; x) : T_2$. Because we chosen x to be fresh, we can then derive $\Gamma \vdash s \Leftrightarrow t : T_1 \rightarrow T_2$, as needed. The proof for the other property uses Lemma 5.5, but we omit the details. □

In his notes Crary carefully designs the logical equivalence so that it has the following property:

Lemma 5.8 (Logical Monotonicity)
If $\Gamma \vdash s \; is \; t : T$ and $\Gamma \subseteq \Gamma'$ and $valid \; \Gamma'$ then $\Gamma' \vdash s \; is \; t : T$.

whose proof is by induction on the definition of the logical equivalence, appealing in the *TBase*-case to Lemma 5.5. From logical monotonicity we can deduce the following corollary:

Corollary 5.9 (Logical Monotonicity for Substitutions)
If $\Gamma' \vdash \theta \; is \; \theta' : \Gamma$ and $\Gamma' \subseteq \Gamma''$ and $valid \; \Gamma''$ then $\Gamma'' \vdash \theta \; is \; \theta' : \Gamma$.

The next three lemmas infer some properties about logical equivalence needed in the fundamental theorems. They are all by relatively routine inductions over the type T, so we only state them here.

Lemma 5.10 (Logical Symmetry)
If $\Gamma \vdash s \; is \; t : T$ then $\Gamma \vdash t \; is \; s : T$.

Lemma 5.11 (Logical Transitivity)
If $\Gamma \vdash s \; is \; t : T$ and $\Gamma \vdash t \; is \; u : T$ then $\Gamma \vdash s \; is \; u : T$.

Lemma 5.12 (Logical Weak Head Closure)
If $\Gamma \vdash s \; is \; t : T$ and $s' \leadsto s$ and $t' \leadsto t$ then $\Gamma \vdash s' \; is \; t' : T$.

Note that in Lemma 5.12 we prove again only the case of closure under single weak-head reductions since this is sufficient for the the fundamental theorems, which are shown next.

Theorem 5.13 (Fundamental Theorem 1)
If $\Gamma \vdash t : T$ and $\Gamma' \vdash \theta \; is \; \theta' : \Gamma$ and *valid* Γ' then $\Gamma' \vdash \theta(t) \; is \; \theta'(t) : T$.

Proof. By induction over the definition of $\Gamma \vdash t : T$. We use the strong induction principle for typing and instantiate the induction context so that binders avoid the substitutions θ and θ'. This will give us the two facts

$$(*) \quad (x, s)::\theta(t) = \theta(t)[x::=s] \qquad \text{and} \qquad (x, s)::\theta'(t) = \theta'(t)[x::=s]$$

which state how we can pull apart a simultaneous substitution such that we obtain a separate single substitution. These facts will be crucial in our induction argument since the left-hand sides correspond to what we have by the induction hypotheses and the right-hand sides will correspond to what we have to prove. These facts do, however, not hold for general x, only for ones that are fresh for the substitution. Since we can assume that x is fresh for θ and θ', our argument goes through smoothly. In the T-Lam-case we have the induction hypothesis

$$ih: \quad \forall \Gamma' \, \theta \, \theta'. \; \Gamma' \vdash \theta \; is \; \theta' : (x, T_1)::\Gamma \wedge valid \; \Gamma' \longrightarrow \Gamma' \vdash \theta(t_2) \; is \; \theta'(t_2) : T_2$$

and we can assume $\Gamma' \vdash \theta \; is \; \theta' : \Gamma$ and further that $x \; \# \; (\Gamma, \theta, \theta')$ (the first freshness assumption comes from the T-Lam rule; the second and third from the strong induction). We need to show that $\Gamma' \vdash \theta(Lam \; x.t_2) \; is \; \theta'(Lam \; x.t_2) : T_1 \rightarrow T_2$ holds. For this it is sufficient to show for all Γ'', s' and t' that

$$\Gamma'' \vdash App \; (Lam \; x.\theta(t_2)) \; s' \; is \; App \; (Lam \; x.\theta'(t_2)) \; t' : T_2$$

whereby we can assume that $\Gamma' \subseteq \Gamma''$, $\Gamma'' \vdash s' \; is \; t' : T_1$ and *valid* Γ''. From these assumptions we obtain by Lemma 5.8 that $\Gamma'' \vdash \theta \; is \; \theta' : \Gamma$ holds and by the freshness conditions also that $\Gamma'' \vdash (x, s')::\theta \; is \; (x, t')::\theta' : (x, T_1)::\Gamma$ (we proved that logical equivalence can be so extended in a side-lemma). Now by induction hypothesis we can infer that

$$\Gamma'' \vdash (x, s')::\theta(t_2) \; is \; (x, t')::\theta'(t_2) : T_2$$

holds. Now we can apply the facts mentioned under $(*)$ to obtain

$$\Gamma'' \vdash \theta(t_2)[x::=s'] \; is \; \theta'(t_2)[x::=t'] : T_2$$

Since we know that

$$App \; (Lam \; x.\theta(t_2)) \; s' \leadsto \theta(t_2)[x::=s']$$
$$App \; (Lam \; x.\theta'(t_2)) \; t' \leadsto \theta'(t_2)[x::=t']$$

hold, we can apply Lemma 5.12 to conclude with $\Gamma'' \vdash App \; (Lam \; x.\theta(t_2)) \; s' \; is \; App \; (Lam \; x.\theta'(t_2)) \; t' : T_2$. This completes, the proof as the T-Lam-case is the only non-automatic case in our formal proof. □

The second fundamental lemma shows that logical equivalence is closed under simultaneous substitutions.

Theorem 5.14 (Fundamental Theorem 2)
If $\Gamma \vdash s \equiv t : T$ and $\Gamma' \vdash \theta \; is \; \theta' : \Gamma$ and *valid* Γ' then $\Gamma' \vdash \theta(s) \; is \; \theta'(t) : T$.

Proof. By strong induction over the definition of the definitional equivalence $\Gamma \vdash s \equiv t : T$. The induction context is again instantiated with θ and θ'. There are several interesting

cases. However we only show the cases for Q-Abs, Q-Beta and Q-Ext.

In the first case we have the induction hypothesis

$$\textit{ih}: \quad \forall \Gamma'\, \theta\, \theta'.\ \Gamma' \vdash \theta \textit{ is } \theta' : (x, T_1)::\Gamma \wedge \textit{valid } \Gamma' \longrightarrow \Gamma' \vdash \theta(s_2) \textit{ is } \theta'(t_2) : T_2$$

and need to show that

$$\Gamma' \vdash \theta(\textit{Lam } x.s_2) \textit{ is } \theta'(\textit{Lam } x.t_2) : T_1 \rightarrow T_2$$

holds. Because by the strong induction principle, we can assume that $x \# (\theta, \theta')$, we are able to immediately move the substitutions under the lambdas, i.e. we have to proceed with showing

$$\Gamma' \vdash \textit{Lam } x.\theta(s_2) \textit{ is Lam } x.\theta'(t_2) : T_1 \rightarrow T_2.$$

This can be done by establishing $\Gamma'' \vdash \textit{App } (\textit{Lam } x.\theta(s_2))\ s' \textit{ is App } (\textit{Lam } x.\theta'(t_2))\ t' : T_2$ for all Γ'', s' and t'. The reasoning is similar to Theorem 5.13 and therefore omitted.

In the second case we need to establish that $\Gamma' \vdash \theta(\textit{App } (\textit{Lam } x.s_1)\ s_2) \textit{ is } \theta'(t_1[x:=t_2]) : T_2$ holds. Again, by the convenience afforded by the strong induction principle we can immediately move the substitution inside the terms, that is we have to show

$$\Gamma' \vdash \textit{App } (\textit{Lam } x.\theta(s_1))\ \theta(s_2) \textit{ is } \theta'(t_1)[x:=\theta'(t_2)] : T_2$$

We omit the other details, because they just amount to using the induction hypotheses and adjusting substitutions appropriately.

In the third case we do not have additional freshness assumptions about θ and θ' (we do not need them in this case). However, the side-conditions about x being fresh for s and t will turn out to be crucial. The reason is that we can then simplify the terms

$$(**) \quad (x, s')::\theta(s) = \theta(s) \qquad \text{and} \qquad (x, t')::\theta'(t) = \theta'(t)$$

The induction hypothesis in this case is

$$\forall \Gamma'\, \theta\, \theta'.\ \Gamma' \vdash \theta \textit{ is } \theta' \textit{ over } (x,T_1)::\Gamma \wedge \textit{valid } \Gamma' \longrightarrow \Gamma' \vdash \theta(\textit{App } s\ (\textit{Var } x)) \textit{ is } \theta'(\textit{App } t\ (\textit{Var } x)) : T_2.$$

and we have the assumptions that $\Gamma' \vdash \theta \textit{ is } \theta' : \Gamma$, $\textit{valid } \Gamma'$ and $x \# (\Gamma, s, t)$. We show that $\Gamma' \vdash \theta(s) \textit{ is } \theta'(t) : T_1 \rightarrow T_2$ holds which by the assumption about the validity of Γ' amounts to showing that

$$\Gamma'' \vdash \textit{App } \theta(s)\ s' \textit{ is App } \theta'(t)\ t' : T_2$$

holds for all Γ'', s' and t', using the assumption about $\Gamma' \subseteq \Gamma''$, $\Gamma'' \vdash s' \textit{ is } t' : T_1$ and $\textit{valid } \Gamma''$. Using Lemma 5.8 we can infer that

$$\Gamma'' \vdash \theta \textit{ is } \theta' : \Gamma$$

holds, from which we obtain

$$\Gamma'' \vdash (x, s')::\theta \textit{ is } (x, t')::\theta' : (x, T_1)::\Gamma.$$

Using the induction hypothesis gives us then

$$\Gamma'' \vdash (x, s')::\theta(\textit{App } s\ (\textit{Var } x)) \textit{ is } (x, t')::\theta'(\textit{App } t\ (\textit{Var } x)) : T_2.$$

Moving the substitutions inside and using the facts $(**)$ we can conclude with

$$\Gamma'' \vdash App\ \theta(s)\ s'\ is\ App\ \theta'(t)\ t' : T_2$$

This completes the proof. □

Completeness of the algorithm is now a simple consequence of the Theorem 5.14 by using the fact that $\Gamma \vdash Var\ x\ is\ Var\ x : T$ holds by Lemma 5.7 and that $\Gamma \vdash [] \ is\ [] : \Gamma$ holds.

Corollary 5.15 (Completeness)
If $\Gamma \vdash s \equiv t : T$ then $\Gamma \vdash s \Leftrightarrow t : T$.

Thus we have formally verified that the algorithm says "yes" for all equivalent terms. The soundness property is left as an exercise in [3]. We have not formalised this part.

6 About the Formalisation

We can generally remark that having a formalised proof allows one to quickly test changes whether they affect the whole proof. This proved convenient for testing if lemmas or definitions need to be strengthened or can be weakened. Having the formal proof at our disposal also made it easy to compile this paper, as Isabelle has an extensive infrastructure for using formal definitions in papers and providing sanity checks. This is especially useful to keep formalisations and papers synchronised. The inductive rules and the statements of the lemmas and theorems presented in this paper have been generated from the formal definitions.

More specifically we can say that our formalisation follows a good deal the informal reasoning of Crary (see Figure 1 which shows the first fundamental lemma as an example in the Isar proof language [12]). The strong induction principles proved crucial in order to get the inductions through. Such strong induction principles are derived automatically for any nominal datatype (which can at the moment only include lambda-type of bindings, but they can occur iterated and can bind different kinds of variables). The strong induction principles are also derived automatically for inductive definition satisfying certain freshness conditions (see [10]).

The only sore point we see in our formalisation is the lack of automation in inversion lemmas. While this is not a serious problem in the formalisation of Crary's chapter (we only need one such inversion lemma and that can be proved in 5 lines), it can be painful in other formalisations. We hope this problem can be solved in the future. To see what the issues are, re-consider the T-Lam-rule:

$$\frac{x\ \#\ \Gamma \qquad (x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash Lam\ x.t : T_1 \rightarrow T_2}\text{T-Lam}$$

and assume that we have given the typing judgement $\Gamma \vdash Lam\ x.t : T$. In in formal reasoning we can match this judgement against all typing rules, which is only successful in case of T-Lam. The informal matching would then produce that there exists an T_1 and T_2 such that $T = T_1 \rightarrow T_2$ and that $(x, T_1)::\Gamma \vdash t : T_2$ as well as $x\ \#\ \Gamma$ hold. However, this is not how we can proceed in the nominal datatype package, where terms are α-equivalent classes. There we obtain for the assumption $\Gamma \vdash Lam\ x.t : T$ the "matcher" that there exists Γ', x', t', T_1' and T_2' so that $\Gamma = \Gamma'$, $Lam\ x.t = Lam\ x'.t'$ and $T = T_1' \rightarrow T_2'$. As properties we obtain $\Gamma' \vdash Lam\ x'.t : T'$ and $x'\ \#\ \Gamma'$. Solving these equation would be no

theorem *fundamental-theorem-1*:
assumes a_1: $\Gamma \vdash t : T$
and a_2: $\Gamma' \vdash \theta$ *is* θ' *over* Γ
and a_3: *valid* Γ'
shows $\Gamma' \vdash \theta(t)$ *is* $\theta'(t) : T$
using a_1 a_2 a_3
proof (*nominal-induct* Γ t T *avoiding*: θ θ' *arbitrary*: Γ' *rule*: *typing.strong-induct*) *(**)*
case (*T-Lam* x Γ T_1 t_2 T_2 θ θ' Γ')
have *vc*: $x \# \theta$ $x \# \theta'$ **by** *fact* *(variable convention)*
have *fs*: $x \# \Gamma$ **by** *fact* *(freshness condition from the rule)*
have asm_1: $\Gamma' \vdash \theta$ *is* θ' *over* Γ **by** *fact*
have *ih*: $\bigwedge \theta\ \theta'\ \Gamma'.\ [\![\Gamma' \vdash \theta$ *is* θ' *over* $(x,T_1)::\Gamma$; *valid* $\Gamma']\!] \Longrightarrow \Gamma' \vdash \theta(t_2)$ *is* $\theta'(t_2) : T_2$
by *fact* *(induction hypothesis)*
show $\Gamma' \vdash \theta(Lam\ x\ .\ t_2)$ *is* $\theta'(Lam\ x\ .\ t_2) : T_1 \rightarrow T_2$ **using** *vc* *(*)*
proof (*simp*, *intro strip*) *(unfolding the definition of logical equivalence)*
fix Γ'' s' t'
assume *sub*: $\Gamma' \subseteq \Gamma''$
and asm_2: $\Gamma'' \vdash s'$ *is* $t' : T_1$
and *val*: *valid* Γ''
from asm_1 *val sub* **have** $\Gamma'' \vdash \theta$ *is* θ' *over* Γ **using** *logical-subst-monotonicity* **by** *blast*
with asm_2 *vc fs* **have** $\Gamma'' \vdash (x,s')::\theta$ *is* $(x,t')::\theta'$ *over* $(x,T_1)::\Gamma$ *(*)*
using *equiv-subst-ext* **by** *blast*
with *ih val* **have** $\Gamma'' \vdash ((x,s')::\theta)(t_2)$ *is* $((x,t')::\theta')(t_2) : T_2$ **by** *auto*
with *vc* **have** $\Gamma'' \vdash \theta(t_2)[x::=s']$ *is* $\theta'(t_2)[x::=t'] : T_2$ **by** (*simp add*: *psubst-subst*) *(*)*
moreover
have *App* $(Lam\ x\ .\ \theta(t_2))\ s' \leadsto \theta(t_2)[x::=s']$ **by** *auto*
moreover
have *App* $(Lam\ x\ .\ \theta'(t_2))\ t' \leadsto \theta'(t_2)[x::=t']$ **by** *auto*
ultimately show $\Gamma'' \vdash$ *App* $(Lam\ x\ .\ \theta(t_2))\ s'$ *is* *App* $(Lam\ x\ .\ \theta'(t_2))\ t' : T_2$
using *logical-weak-head-closure* **by** *auto*
qed
qed (*auto*) *(all other cases are automatic)*

Fig. 1. The complete formalised proof of the first fundamental lemma (Lemma 5.13) in the readable Isar proof-language. In the places marked with a single star, one appeals in informal reasoning to the variable convention about the binder x. This variable convention is given in our proof by the strong induction principle and by declaring that x should avoid θ and θ' (see line marked with two stars). The fact *logical-subst-monotonicity* is Corollary 5.9; *equiv-subst-ext* establishes that for a fresh x one can extend the logical equivalence of simultaneous substitutions; and *psubst-subst* is a lemma that allows us to pull apart a simultaneous substitution in order to obtain a single substitution. We can do this provided the variable convention about x holds.

problem if we had term-constructors that are injective (that is a characteristics of standard, unquotioned datatypes). However, our constructors for α-equivalence classes are clearly *not* injective. What we have to do is to analyse $Lam\ x.t = Lam\ x'.t'$ according to the built-in notion of the nominal datatype package for α-equivalence. We end up with two cases: one is simple and the other needs explicit renamings. However these reasoning maneuvers should really be performed automatically by the nominal datatype package.

7 Conclusion

We presented a formalisation of Crary's case study about logical relations. This is in addition to the usual strong normalisation proof of the simply-typed lambda-calculus, which has been part of the nominal datatype package for quite some time. It remains to be seen whether the nominal datatype package is up to the task of formalising strong normalisation for System F, where also types have binders. In this case the definition of logical relations is not completely trivial like in the completeness proof we presented above.

We are aware of work by Schürmann and Sarnat about formalising logical relation proofs in Twelf [7]. This involves a clever trick of implementing an object logic in Twelf and coding the logical relation proof in this object logic. We unfortunately do not know how convenient this style of reasoning is. We are also aware that Aydemir et al [2] use a locally nameless approach (which goes back to work by McKinna and Pollack [5]) to representing binders and work on formalising programming language theory. It would be interesting to compare in detail our formalisation and the approach taken by Aydemir et al. Our initial opinion is that in our formalisation we do not have to deal with the concepts of *open* and *closed* terms; and that we do not have to discard any *pre-terms*.

The sources of our formalisation are included in the nominal datatype package (see `http://isabelle.in.tum.de/nominal/`). From the web-page of the first author one can also download a longer version of the documented proofs.

Acknowledgements:
We thank Karl Crary for the discussions about his proof. We are also very grateful to Carsten Schürmann who made us aware of typos and omissions in an early version of this paper.

References

[1] T. Altenkirch. A Formalization of the Strong Normalisation Proof for System F in LEGO. In *Proc. of TLCA*, volume 664 of *LNCS*, pages 13–28, 1993.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, and S. Weirich. Engineering Aspects of Formal Metatheory, 2007. Submitted for publication.

[3] K. Crary. Logical Relations and a Case Study in Equivalence Checking. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 223–244. MIT Press, 2005.

[4] R. Harper and D. Licata. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 2007. To appear.

[5] J. McKinna and R. Pollack. Pure Type Systems Formalized. In *Proc. of the International Conference on Typed Lambda Calculi and Applications (TLCA)*, number 664 in LNCS, pages 289–305. Springer-Verlag, 1993.

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

[7] C. Schürmann and J. Sarnat. Towards a Judgemental Reconstruction of Logical Relation Proofs. Submitted, 2007.

[8] W. W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[9] C. Urban 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)*, volume 4130 of *LNAI*, pages 498–512, 2006.

[10] C. Urban, S. Berghofer, and M. Norrish. Barendregt's Variable Convention in Rule Inductions. In *Proc. of the 21th International Conference on Automated Deduction (CADE)*, 2007. To appear.

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

[12] M. Wenzel. 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, pages 167–184, 1999.

Towards Formalizing Categorical Models of Type Theory in Type Theory

Alexandre Buisse[1]

Department of Computer Science and Engineering
Chalmers University of Technology
Rännvägen 6, S-41296 Göteborg

Peter Dybjer[2]

Department of Computer Science and Engineering
Chalmers University of Technology
Rännvägen 6, S-41296 Göteborg

Abstract

This note is about work in progress on the topic of "internal type theory" where we investigate the internal formalization of the categorical metatheory of constructive type theory in (an extension of) itself. The basic notion is that of a category with families, a categorical notion of model of dependent type theory. We discuss how to formalize the notion of category with families inside type theory and how to build initial categories with families. Initial categories with families will be term models which play the role of canonical syntax for dependent type theory. We also discuss the formalization of the result that categories with finite limits give rise to categories with families. This yields a type-theoretic perspective on Curien's work on "substitution up to isomorphism". Our formalization is being carried out in the proof assistant Agda 2 developed at Chalmers.

1 Introduction

Most work on the metatheory of constructive type theory use standard informal mathematical metalanguage. Although such metatheory often have an intuitive constructive character it is striking that most authors rely on classical set-theoretic notions when explaining concepts rigorously. For example, when building models of constructive type theory it is common to first build a partial interpretation function mapping raw terms to their meaning, and only afterwards show that the meaning of well-typed terms is defined. The constructive meaning of this is not obvious, at least if we use constructive type theory itself as the metalanguage. The functions in this theory are all total, so partial functions need to be encoded as total functions, and this will have formal repercussions. Another example is the treatment

[1] Email: buisse@cs.chalmers.se
[2] Email: peterd@cs.chalmers.se

of the inductive-recursive definitions which are needed in certain model constructions and normalization proofs. Although such definitions are constructively valid [12,13,14,15] most authors rely on their interpretation in set theory [23,3,1,2] and this also has formal repercussions.

In this project we plan to show that it is possible to rely entirely on constructive notions on the metalevel. The idea of doing such constructive model theory goes back to Martin-Löf [19]. However, Martin-Löf relied on an informal constructive metalanguage, while we here are more specific and work with suitable versions of Martin-Löf type theory as formal metalanguages. We are checking our proofs in the proof assistant Agda 2 which is currently under development by Ulf Norell at Chalmers. In this way we are turning *constructive metamathematics* into *metaprogramming*. A proof of an abstract result such as "any category with finite limits is a category with families" turns into a "compiler" which maps any input data structure representing a category with finite limits into an output data structure representing a category with families. The type-system will ensure that this compiler is correct. When we program this compiler in the Agda system we can actually run it on concrete examples.

It is worth-while pointing out that Martin-Löf type theory is intended to be a full-scale language for constructive mathematics just as Zermelo-Fraenkel set theory is a full-scale language for classical mathematics. Just as it might be necessary to postulate the existence of certain additional large cardinals in order to discuss the metatheory of set theory inside set theory, we will here need to add certain analogues of large cardinals to constructive type theory.

Previous work on constructive model theory which use a formal constructive metalanguage includes Pollack's work on Lego in Lego [22] and Barras' work on Coq in Coq [4]. Both authors deal with the usual lambda calculus based syntax of constructive type theory. Here we will instead base our work on a categorical notion of model of type theory. The formal system of type theory will be represented abstractly as an *initial category with families (with extra structure)*. A category with families (cwfs) [11] is a notion of model of (the most basic rules of) dependent type theory, and the initial such is the "term model". There are several reasons for choosing a categorical approach. One of them is to achieve more "canonical" results. Syntactic approaches tend to depend on a number of detailed choices. Should we choose ordinary named variables or de Bruijn's nameless ones, or should we use de Bruijn levels? Is the rule of substitution a primitive or derived rule? Etc. Categorical notions tend to be more stable and canonical, although it must be admitted that certain representation issues remain. Other arguments for basing our approach on category theory are well-known from categorical type theory: we get a clear notion of model, access to many powerful results in category theory, and a mathematically elegant approach which hopefully also leads to a more economical formalization.

Our work builds on the second author's paper "Internal Type Theory" [11]. In that paper the notion of a category with families was introduced as a notion of model of dependent type theory with a particularly straightforward connection to syntax. The formalization of cwfs inside type theory is also discussed. Such a formalization is called an "internal type theory" since it is analogous to the notion

of *internal category*. In any category with suitable extra structure (finite limits) we can define what it means to be an internal category object. Similarly, in any cwf with extra structure (modelling Π-types, Σ-types and universes) we can define a notion of internal cwf.

Our work will rely on previous experience of formalization of category theory inside constructive type theory, see for example the development of elementary category theory in Huet and Saibi's book on Constructive Category Theory [18].

Plan of the note.

In Section 2 we present the notion of a cwf in classical metalanguage. In Section 3 we explain some of the features of the proof assistant Agda 2 which we use for our formalization. In Section 4 we present the usual approach to the formalization of categories inside type theory, continue with the formalization of the category of families of sets, and then arrive at the notion of category with families inside type theory. In Section 5 we sketch how to formalize the result that categories with finite limits give rise to cwfs. We discuss the close relationship to Curien's paper "Substitution up to Isomorphism" [7] and contrast it to a similar result by Hofmann [16] formulated using classical categorical notions. In Section 6 we discuss the construction of initial categories with families.

2 Categories with families

Categories with families (cwfs) [11,17] are variants of Cartmell's *categories with attributes* [16,21]. The point of the reformulation is to get a more direct link to the syntax of dependent types. In particular we avoid reference to pullbacks, which give rise to a conditional equation when formalized in a straightforward way. Cwfs can therefore be formalized as a generalized algebraic theory in Cartmell's sense with clear similiarities to Martin-Löf's *substitution calculus* for type theory [20].

Let **Fam** be the category of families of sets, where an *object* is a family of sets $(B(x))_{x\in A}$ and a *morphism* with source $(B(x))_{x\in A}$ and target $(B'(x'))_{x'\in A'}$ is a pair consisting of a function $f : A \to A'$ and a family of functions $g(x) : B(x) \to B'(f(x))$ indexed by $x \in A$.

The components of a cwf are named after the corresponding syntactic notions.

Definition 2.1 A *category with families* consists of the following four parts:

- A base *category* C. Its objects are called *contexts* and its morphisms are called *substitutions*.
- A *functor* $T : C^{op} \to \mathbf{Fam}$. We write $T(\Gamma) = (\Gamma \vdash A)_{A\in Type(\Gamma)}$, where Γ is an object of C, and call it the family of *terms* indexed by *types* in context Γ. Moreover, if γ is a morphism of C then the two components of $T(\gamma)$ interpret substitution in types and terms respectively. We write $A[\gamma]$ for the application of the first component to a type A and $a[\gamma]$ for the application of the second component to a term a.
- A *terminal object* $[\,]$ of C called the *empty context*.
- A *context comprehension* operation which to an object Γ of C and a type $A \in$

$Type(\Gamma)$ associates an object $\Gamma; A$ of C; a morphism $\mathrm{p}_{\Gamma,A} : \Gamma; A \to \Gamma$ of C (the *first projection*); and a term $\mathrm{q}_{\Gamma,A} \in \Gamma; A \vdash A[\mathrm{p}_{\Gamma,A}]$ (the *second projection*). The following universal property holds: for each object Δ in C, morphism $\gamma : \Delta \to \Gamma$, and term $a \in \Delta \vdash A[\gamma]$, there is a unique morphism $\theta = \langle \gamma, a \rangle : \Delta \to \Gamma; A$, such that $p \circ \theta = \gamma$ and $q[\theta] = a$.

A basic example of a cwf is obtained by letting $C = \mathrm{Set}$, the category of small sets, $Type(\Gamma)$ be the set of Γ-indexed small sets, and

$$\begin{aligned} \Gamma \vdash A &= \prod_{x \in \Gamma} A(x) \\ A[\delta](x) &= A(\delta(x)) \\ a[\delta](x) &= a(\delta(x)) \\ [\,] &= 1 \\ \Gamma; A &= \sum_{x \in \Gamma} A(x) \end{aligned}$$

Definition 2.2 Let (C, T) denote a cwf with base category C and functor T. A *morphism of cwfs* with source (C, T) and target (C', T') is a pair (F, σ), where $F : C \to C'$ is a functor and $\sigma : T \to T'F$ is a natural transformation, such that terminal object and context comprehension are preserved on the nose.

Small cwfs and morphisms of cwfs form a category **Cwf**.

As already mentioned the notion of a category with families can be formalized as a *generalized algebraic theory* in the sense of Cartmell [6]. It is instructive to look at the rules of this theory, but we do not have room in this short note to display them, and refer the reader to Dybjer [11].

3 The proof assistant Agda 2

We will work in Martin-Löf's constructive type theory and use the proof assistant Agda 2 for our formalization. Agda 2 is an implementation of Martin-Löf's logical framework with support for adding new inductively defined sets and recursively defined functions using pattern matching. It is thus suitable for implementing various fragments of Martin-Löf type theory. It can also be viewed as a dependently typed programming language. Its syntax is quite close to Haskell. The main difference to Haskell (and other standard functional languages such as OCAML) is that it has dependent types. It is important to point out that Agda does not itself force the user to define only well-founded data types and terminating functions, although at a later stage such termination and well-foundedness checkers will be available. Currently it is up to the user to make sure that a specific logical discipline is followed and to explain and justify this discipline.

If we remove Agda's dependent types, we get a language rather close to Haskell. However, Agda doesn't have the implicit Hindley-Milner polymorphism of Haskell. In Haskell we have for example the polymorphic identity function

```
id :: a -> a
```

stating that for any type `a` we have an identity funciton on it. In Agda we have to

explicitly quantify over the type Set of small types. We write

$$\mathbf{id} : (A : Set) \to A \to A$$

which means that for any small type A we have an identity function $\mathbf{id}\, A : A \to A$. (In general Agda uses the notation $(x : \alpha) \to \beta$ for the type of functions which map an object x of type α into a result in β, where the result type β may depend on x.) This is why one says that Agda is an implementation of Martin-Löf's *monomorphic* type theory: **id** has a unique type.

However, it is cumbersome to work in monomorphic type theory since one has to manipulate large expressions. Therefore Agda allows you to suppress certain arguments when they can be inferred from the context. We call such arguments *implicit*. For example, whenever the function **id** gets a second argument a, Agda tries to infer the first argument A, which is the type of a. The user can inform Agda that the first argument of **id** is such an implicit argument by enclosing it in braces:

$$\mathbf{id} : \{A : Set\} \to A \to A$$

Thus, during type-checking of an expression $\mathbf{id}\, a$ the system will know that a is the second argument and try to infer the first argument.

Sometimes a user may want to give an implicit argument explicitly, for example, in situations where the system cannot infer it. Such explicit implicit arguments are enclosed in braces. For example, in

$$\mathbf{id}\, \{Bool\} : Bool \to Bool$$

the user has made the first argument $Bool$ explicit.

Agda 2 has a notion of **record**, that is a type of tuples with named components (field). To instantiate it, one needs to instantiate all the fields. A record is really a module, and thus the field A of a record r of type R can be accessed with the syntax $R.A\ r$: declaring the record type R automatically creates a module with the same name and one projection function for each field, which take as their first argument a structure of type R.

Here is an example. A record for equivalence relations on a given set A consists of four fields: a binary relation on A together with proofs of reflexivity, symmetry, and transitivity. Formally:

$$\begin{aligned}
&\mathbf{record}\ Equivalence\ (A\ :\ Set)\ :\ Set_1\ \mathbf{where}\\
&\qquad _==_ \quad : A \to A \to Set\\
&\qquad refl \quad : \{x\ :\ A\} \to x\ ==\ x\\
&\qquad sym \quad : \{x\ y\ :\ A\} \to x\ ==\ y \to y\ ==\ x\\
&\qquad trans \quad : \{x\ y\ z\ :\ A\} \to x\ ==\ y \to y\ ==\ z \to x\ ==\ z
\end{aligned}$$

Note that the record is a *large* type, that is, a member of the universe Set_1.

The central notion of a *setoid*, that is, a set with an equivalence relation, will be used extensively:

$$\begin{aligned}
&\mathbf{record}\ Setoid\ :\ Set_1\ \mathbf{where}\\
&\qquad car \quad : Set\\
&\qquad rel \quad : Equivalence\ car
\end{aligned}$$

To get a nicer notation, we'll define a function to access directly to the carrier of a setoid:

$$|\_| : (S : Setoid) \to Set$$
$$|S| = Setoid.carS$$

4 Categories with families in type theory

We will follow the recipe for formalizing categories with families in type theory (*internal cwfs*) described in Dybjer [11]. It is well-known how to formalize basic categorical notions such as category, functor, natural transformation, etc [18] in type theory. It is also well-known how to formalize the type-theoretic analogue **Set** of the category of sets in type theory. The objects of this category are *setoids* (or *E-sets*) that is sets with equivalence relations. The arrows are functions respecting equivalence relations.

The crucial issue for the formalization of cwfs is the formalization of the category **Fam**, and we follow the approach in the paper Internal Type Theory [11]. (We will however also investigate an alternative proof-irrelevant definition of setoid-indexed families used in a recent implementation of the category of setoids in Agda 2 by Thierry Coquand and Ulf Norell.)

Once we have defined the category **Fam** it is straightforward to formalize the rest of the cwf-structure in type theory. Note that if a cwf is formally represented as a quadruple consisting of the base category, the family valued functor, the terminal object, and context comprehension, where each of these components itself is a tuple, we can "flatten" this structure into a tuple where each component corresponds to a rule in a substitution calculus for type theory. This calculus is closely related to the calculus of explicit substitutions used by Curien [7]. Like this calculus there is an explicit construction for the type conversion rule [11]. We can thus see how the type-theoretic perspective gives a rational reconstruction of Curien's calculus.

4.1 Categories

A category in type theory consists of a set of objects, hom-setoids for each pair of objects, an identity arrow for each object, composition respecting equivalence of the arrows in the hom-setoids, and proofs of the identity and associativity laws. (Note that we have explicit proof objects for each law.)

$$\begin{array}{rlll}
\textbf{open} & Setoid & & \\
\textbf{record} & Cat & : & Set_2 \ \textbf{where} \\
& Obj & : & Set_1 \\
& _ \longrightarrow _ & : & Obj \to Obj \to Setoid \\
& id & : & \{A : Obj\} \to |A \longrightarrow A| \\
& _ \circ _ & : & \{A\ B\ C : Obj\} \to |B \longrightarrow C| \to |A \longrightarrow B| \to \\
& & & \quad |A \longrightarrow C| \\
& \ldots & &
\end{array}$$

$$
\begin{array}{l}
idL \quad : \{A\ B\ :\ Obj\}\ \{f\ :\ A \longrightarrow B\} \rightarrow \_ == \_\ (rel(A \longrightarrow B)) \\
\qquad\quad (id \circ f)\ f \\
\ldots
\end{array}
$$

We have chosen to formalize a notion of *locally small* category where hom-setoids must be "small" (the carrier is a set), but an object can be "large" (a member of the universe Set_1 of large sets). Examples of such locally small categories is the category **Set** of setoids and the category **Fam** of setoid-indexed families of setoids. Note also the type of locally small categories is "very large" (a member of a second universe Set_2 of very large sets).

4.2 The category ***Fam***

To define the category **Fam**, we need the notions of setoid-indexed family of setoids, and of morphism between such (the respecitve objects and arrows of **Fam**). We have seen in section 3 how to define a setoid as a record. The next step is to define a morphism between setoid as a function between the carriers together with a proof that it maps equivalent elements to equivalent elements:

$$
\begin{array}{l}
\textbf{record}\ \_ \Rightarrow \_\ (S_1\ S_2\ :\ Setoid)\ :\ Set\ \textbf{where} \\
\qquad map\ : |S_1| \rightarrow |S_2| \\
\qquad stab\ : \{x\ y : |S_1|\} \rightarrow x ==^{3} y \rightarrow (map\ x) == (map\ y)
\end{array}
$$

Identity and composition of setoid morphisms are easy to add. For instance,

$$
\begin{array}{l}
id\ :\ \{S : Setoid\} \rightarrow S \Rightarrow S \\
id = \textbf{record}\ \{map = \backslash x \rightarrow x\ \ ;\ stab = \backslash p \rightarrow p\}
\end{array}
$$

We also add an extensional equality $==_{\Rightarrow}$:

$$
\begin{array}{l}
\_ ==_{\Rightarrow} \_\ : \{S_1\ S_2 : Setoid\} \rightarrow S_1 \Rightarrow S_2 \rightarrow S_1 \Rightarrow S_2 \rightarrow Set \\
F_1 ==_{\Rightarrow} F_2 = (\textbf{forall}\ x \rightarrow (map\ F_1\ x)\ ==_{\Rightarrow}\ (map\ F_2\ x)) \rightarrow True
\end{array}
$$

We are now ready to define the notion of a family of setoids indexed by a given setoid S: a fibre map that indexes setoids by elements of the carrier of the indexing setoid, a reindexing function ι that maps the equivalence relation of the indexing setoid into the indexed setoids and proofs that this reindexing function is coherent with the fact that the relation is an equivalence.

$$
\begin{array}{lll}
\textbf{record}\ SetoidFam\ (S\ :\ Setoid)\ :\ Set_1\ \textbf{where} & & \\
\qquad fibre & : |S| \rightarrow Setoid & \\
\qquad \iota & : \{x\ x' : |S|\} \rightarrow x == x' \rightarrow (fibre\ x') \Rightarrow (fibre\ x) & \\
\qquad idcoh & : \{x : |S|\} \rightarrow \iota\ (refl\ (rel\ S)\ \{x\}) ==_{\Rightarrow} id\ \{fibre\ x\} & \\
\qquad symcoh_L & : \{x\ y : |S|\} \rightarrow (p : x == y) \rightarrow & \\
\qquad\qquad & \iota\ (sym\ (rel\ S)\ \{x\}\ \{y\}\ p) \circ (\iota\ p) ==_{\Rightarrow} id &
\end{array}
$$

[3] Here we redefined $\_ == \_$ with the type $\{S : Setoid\} \rightarrow Equivalence.\_ == \_\ (rel\ S)$

$$
\begin{aligned}
&\mathit{symcoh}_R \;:\; \ldots \\
&\mathit{transcoh} \;:\; \{x\ y\ z : |S|\} \to (p : x == y) \to (p' : x == z) \to \\
&\qquad \iota\ (\mathit{trans}\ (\mathit{rel}\ S)\ \{x\}\ \{y\}\ \{z\}\ p\ p') ==_{\Rightarrow} (\iota\ p) \circ (\iota\ p')
\end{aligned}
$$

The last step is to define what a morphism between objects of type $SetoidFam$ is. There is one map for each component: a map $indexmap$ between the index setoids, a map $fibremap$ between the fibres, and a proof ιmap that reindexing commutes with the map between fibres (see the upper part of the figure below):

$$
\begin{aligned}
\textbf{record}\quad \_ \Longrightarrow \_ \quad & \{S_1\ S_2 :\ Setoid\} \\
& (F_1 : SetoidFam\ S_1)\ (F_2 : SetoidFam\ S_2) :\ Set_1\ \textbf{where} \\
\mathit{indexmap} &: S_1 \Rightarrow S_2 \\
\mathit{fibremap} &: (x : |S_1|) \to (\mathit{fibre}\ F_1\ x) \Rightarrow (\mathit{fibre}\ F_2\ ((\mathit{map}\ \mathit{indexmap}\ x)) \\
\iota map \quad &: \{x\ x' : |S_1|\} \to (p : x == x') \to ((\mathit{fibremap}\ x) \circ (\iota\ F_1\ p)) \\
& ==_{\Rightarrow} (\iota\ F_2\ (\mathit{stab}\ \mathit{indexmap}\ p)) \circ (\mathit{fibremap}\ x')
\end{aligned}
$$

The figure 1 illustrates the structure of a morphism in **Fam**.

The rest of the properties needed for **Fam** to be a category are then straightforward, though particularily tedious.

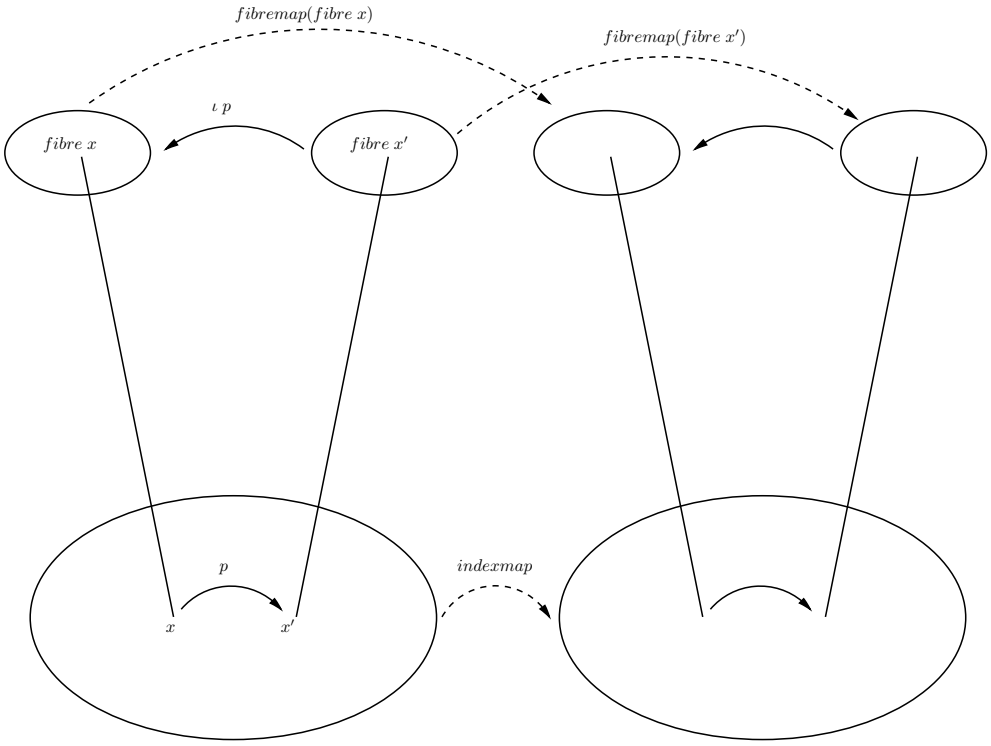

Fig. 1. Representation of $F_1 \Longrightarrow F_2$

4.3 Cwfs

Just as in the classical definition of a cwf in Section 2, a cwf inside type theory is a quadruple consisting of a base category C, a functor $T : C^{op} \to \mathbf{Fam}$, a terminal object of C, and a context comprehension. Above we have outlined the type theoretic definition of a category and of the category **Fam**. The type-theoretic definition of a functor is well-known [18]. Furthermore, it is clear what a terminal object is type-theoretically, and we can express the structure of context comprehension type-theoretically. All this leads to the definition of cwf inside type theory (i.e. the notion of internal cwf), although we do not have room to display the details.

We remark that our formalization of locally small cwfs in type theory has only used a logical framework with Π-types, records (we could equivalently use Σ-types), and the universes Set, Set_1, and Set_2. (We don't need Set_2 if we only want to formalize small cwfs.)

4.4 Cwfs with extra structure

The notion of cwf just models the most basic structure of dependent types: context and variable formation and substitution in types and terms. Therefore we usually want to work with cwfs with extra structure corresponding to adding type formers (Π, Σ, universes, natural numbers, etc) to dependent type theory. This does not give rise to any further formalization problems. See the Internal Type Theory paper [11] for further explanation.

5 Categories with finite limits are cwfs

Here we outline the proof inside type theory that categories with finite limits are cwfs. This proof will help us understand how cwfs relate to standard ideas in categorical type theory: why types can be modelled as projection arrows, why terms can be modelled as sections of these projections, and why substitution in types can be modelled by pullbacks. We will discuss the type-theoretic perspective on the problem of "substitution-up-to-isomorphism" and show the similarity with Curien's approach [7]. We will also contrast it to Hofmann [16], who used standard categorical notions assuming set-theoretic metalanguage.

5.1 Categories with finite limits in type theory

Categories with finite limits can be formalized as categories with terminal objects and pullback. (A category with terminal objects and pullbacks has all finite limits.) As a type-theoretic structure a pullback is a function that given three objects and two arrows constructs an object, two arrows, proofs of commutativity of the square, and a proof of the universal property. Here is the formalization in Agda.

$$
\begin{array}{rl}
\textbf{record} \quad IsPull & \{A\ B\ C\ D : Obj\}(f : A \longrightarrow B)(g : A \longrightarrow C)(f' : C \longrightarrow D) \\
& (g' : B \longrightarrow D)(square : g' \circ f == f' \circ g) : Set_1\ \textbf{where} \\
h \quad : (A' : & Obj)(h_1 : A' \longrightarrow C)(h_2 : A' \longrightarrow B)(tr : f' \circ h_1 == g' \circ h_2) \\
& \to \exists!\ \backslash(h : A' \longrightarrow A) \to (g \circ h == h_1) \wedge (f \circ h == h_2)
\end{array}
$$

$$
\begin{array}{l}
\textbf{record}\ \mathit{Pullback}\,\{B\ C\ D : \mathit{Obj}\}(g' : B \longrightarrow D)(f' : C \longrightarrow D) : \mathit{Set}_1\ \textbf{where} \\
\qquad\begin{array}{rl}
A & : \mathit{Obj} \\
f & : A \longrightarrow B \\
g & : A \longrightarrow C \\
\mathit{square} & : g' \circ f == f' \circ g \\
\mathit{pull} & : \mathit{isPull}\ f\ g\ f'\ g'\ \mathit{square}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{record}\ \mathit{PullCat} : \mathit{Set}_2\ \textbf{where} \\
\qquad \mathit{pullprop} : \{B\ C\ D : \mathit{Obj}\}(g' : B \longrightarrow D)(f' : C \longrightarrow D) \to \mathit{pullback}\ g'\ f'
\end{array}
$$

5.2 Slice categories

We shall recover the structure of cwfs by modelling types by objects in slice categories, and by modelling substitution in types by (the object part of) the pullback functor between slice categories.

Given any category C and an object Γ of that category we can construct the slice category C/Γ. The objects are pairs of objects A in C and arrows $f : A \longrightarrow C$. The proof that C/Γ is a category is quite easily derived from the fact that C is also a category.

5.3 Cwfs from categories with finite limits

We get the cwf structure from a category with finite limits in the following way:

- The base categories are the same.
- The set of types in a context Γ is the set of objects of the slice category C/Γ. Equality of types is isomorphism in the slice category.
- The set of terms of a given type A in context Γ is the set of sections of the arrow in C with target Γ which models A. Equality of terms is inherited from equality of arrows in the base category. (Proofs that these arrows are sections are not relevant to the equality.)
- Substitution in types is obtained by the pullback construction.
- Substitution in terms is also extracted from the pullback.
- Etc.

We here show part of the Agda formalization of how substitution in types is modelled by the pullback construction (types are omitted when not important).

An object of the slice category C/Γ is an arrow, but in order to typecheck, we need to also specify the codomain of this arrow:

$$
\begin{array}{l}
\textbf{record}\ \mathit{SlObj}\,(\Gamma : \mathit{Obj}) : \mathit{Set}_1\ \textbf{where} \\
\qquad\begin{array}{rl}
\mathit{dom} & : \mathit{Obj} \\
\mathit{arr} & : \mathit{dom} \longrightarrow \Gamma
\end{array}
\end{array}
$$

A section of an arrow $f : \Delta \to \Gamma$ is

$$
\begin{array}{l}
\textbf{record}\ \mathit{Section}\ (\Gamma : \mathit{Obj})\ (A : \mathit{SlObj}\ \Gamma) : \mathit{Set}\ \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{sect} & : \Gamma \longrightarrow (\mathit{dom}\ A) \\
\mathit{id}_L & : (\mathit{arr}\ A) \circ \mathit{sect} == \mathit{id}
\end{array}
\end{array}
$$

We are now ready to proceed:

$$
\begin{array}{l}
\mathit{Context} = \mathit{Obj} \\
\mathit{Subst}\ \Gamma\ \Delta = \Gamma \longrightarrow \Delta \\
\mathit{Type}\ \Gamma = \mathit{SlObj}\ \Gamma \\
\mathit{Term}\ \Gamma\ A = \mathit{Section}\ \Gamma\ A
\end{array}
$$

$$
\begin{array}{l}
\mathit{subst}\ : \{\Gamma\ \Delta :\ \mathit{Context}\} \to \mathit{Type}\ \Gamma \to \mathit{Subst}\ \Delta\ \Gamma \to \mathit{Type}\ \Delta \\
\mathit{subst}\ \{\Gamma\}\ \{\Delta\}\ T\ g = \textbf{let}\ p = \mathit{pullprop}\ (\mathit{arr}\ \Gamma\ T)\ g\ \textbf{in} \\
\qquad \mathit{record}\ \ \{\mathit{dom} = \mathit{pullback}.A\ p;\ \mathit{arr} = \mathit{pullback}.g\ p\}
\end{array}
$$

We can compare our interpretation to the approach taken in the paper "Substitution up to Isomorphism" by Curien [7]. Like us, Curien interprets equality of types as isomorphism in the slice category. Another similarity is that he uses an explicit substitution calculus for dependent type theory not unlike our initial cwf which has an explicit constructor for applications of the rule of type equality.

This approach can be contrasted to Hofmann's work on interpreting type theory in locally cartesian closed categories [16]. In this work he shows how to construct a category with attributes from a category with finite limits using a technique due to Bénabou [5]. Since categories with attributes are equivalent to categories with families this ought to be highly relevant to our work. However, Hofmann uses standard category theory relying on set-theoretic metalanguage, and his notion of category with attributes is a "strict" one, just as our set-theoretic notion of cwf in Section 2. To show that in this classical setting categories with finite limits form cwfs, we cannot just interpret substitution as "chosen" pullbacks, unless this choice satisfies the laws of substitution in types *up to equality*. Hofmann states that it is an open problem to find such a choice. When working in type-theoretic metalanguage on the other hand we have the freedom to interpret equality of types as isomorphism of objects, and thus there is no need for Bénabou's construction.

However, what we gain when avoiding Bénabou's construction we have to pay back when constructing cwfs (with extra structure) from syntax and proving their initiality. This work is similar to the *coherence problem* discussed by Curien.

6 Initial cwfs (with extra structure)

If we work in set-theoretic metalanguage initial cwfs exist. This is a corollary of a theorem of Cartmell [6] who showed that any generalized algebraic theory has an initial model in an appropriate categorical sense.

We shall discuss two ways of constructing initial cwfs with extra structure inside type theory. Without the extra structure the initial cwf is trivial; it is nothing but the category with one object and one arrow, where the family valued functor returns the empty family of sets. To get interesting extra structure we postulate the

existence of Π and Σ and a universe of small types. We call this an *LF-cwf*, since it is the categorical analogue of Martin-Löf's logical framework. We remark that the discussion below is not very dependent on the exact choice of extra structure, except that some properties will rely on normalization.

6.1 Strongly typed version

This version is obtained by taking the definition of an LF-cwf as a record and turn it into an inductive definition. The notion of LF-cwf specifies seven different families of sets, one corresponding to each of the seven forms of judgement. Each of these will turn into a formation rule for seven inductively defined sets of "derivations" of judgements. The notion of LF-cwf furthermore specifies a number of different operations each corresponding to a rule of inference. Each of these operations will become a constructor in the inductive definition of the initial cwf.

For instance, contexts are defined by the grammar $\Gamma ::= [\,] \mid \Gamma \triangleright A$ where A is a type. Correspondingly there will be two constructors for contexts in the initial cwf, with the following types formalized in Agda:

```
mutual
  data Ctxt : Set where
    [] : Ctxt
    ▷ : (Γ : Ctxt) → Type Γ → Ctxt
  data Type : Ctxt → Set where
    ...
```

However, it is important to remark that this inductive definition falls outside the standard schema of mutual inductive definitions in constructive type theory [10]. Nevertheless, we believe that it is a constructively meaningful definition. As part of our investigation we plan to generalize the schema in [10,15] to cover that schema, and also to provide set-theoretic semantics by extending [9].

6.2 The category of cwfs in type theory.

Although the above seems like a reasonable candidate for a strongly typed notion of *term model* of type theory, we would like to prove formally in type theory that we have an initial LF-cwf, that is, that it forms an initial object in the category of LF-cwfs. In Section 2 we defined a notion of cwf morphism which preserves chosen structure "on the nose". However,the type-theoretic definition of a category does not equip objects with a notion of equality. The natural notion of equality of objects is isomorphism, and hence we would like to use a notion of cwf morphism which preserves the cwf structure up to isomorphism. To spell out the definition of the category of cwfs and construct an initial object (together with the unique arrow to another object) is another part of our project. Given such a definition it should be straightforward to see that the above strongly typed version is initial since it means that each construction is interpreted as the corresponding notion in a given cwf. In a sense the elimination principle *is* the unique arrow from the initial cwf to an arbitrary cwf, at least roughly speaking, cf e g the proof of the correspondence between initiality and elimination principles in [14].

6.3 Raw term version

An alternative definition of the initial cwf can be obtained by first defining raw contexts, raw types, etc.

mutual

data $RawCtxt : Set$ **where**

$[\,] : RawCtxt$

$\triangleright : RawCtxt \to RawType \to RawCtxt$

data $RawType : Set$ **where**

$\ldots$

As a second step we define a predicate "valid context" on $RawCtxt$, a binary relation "valid type" between $RawCtxt$ and $RawType$, etc. In this way we give a mutual inductive definition of all the seven forms of judgement viewed as predicates on raw notions.

Finally, we would like to show that this also yields an initial cwf by defining a cwf-structure-preserving map into an arbitary cwf, and to show the uniqueness of this map.

7 Conclusion and future work

As already mentioned this is work in progress. An auxiliary aim is to test the suitability of the new proof assistant Agda 2 for the purpose of formalizing category theory. Agda 2.0.0 was just released (June 2007) and still lacks many features of a more mature system such as Coq or even its predecessors AgdaLight, Agda 1 and Alfa. For example, there is still no support for equational reasoning and automatic proof construction. The implicit argument feature of Agda 2 is used heavily in this work, but we have encountered some performance problems.

After completing the formalizations described in this paper, we would like to add more structure to categories with families. In particular we would like to formalize the full Seely-Curien [24,7] interpretation of Martin-Löf type theory (understood as categories with families with extra structure modelling Π- and Σ-types and extensional equality types) in locally cartesian closed categories.

Another direction of future research would be to formalize key metatheoretical results of Martin-Löf type theory such as decidability of equality and type-checking based on categories with families [1,2]. This is related to the work by Danielsson [8] who presented such a formalization of a normalization by evaluation result in the system AgdaLight, a precursor of the Agda 2 system. Danielsson did however not base his work on a categorical presentation of dependent type theory.

References

[1] A. Abel, K. Aehlig, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In *"23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIII*, Electronic Notes in Theoretical Computer Science, pages 17–40. Elsevier, 2007.

[2] A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for Martin-Lf type theory with equality judgements. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, July 2007. To appear.

[3] P. Aczel. *Frege Structures and the Notions of Proposition, Truth, and Set*, pages 31–59. North-Holland, 1980.

[4] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.

[5] J. Benabou. Fibered categories and the foundations of naive category theory. *J. Symb. Log*, 50(1):10–37, 1985.

[6] J. Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[7] P.-L. Curien. Substitution up to isomorphism. *Fundamenta Informaticae*, 19(1,2):51–86, 1993.

[8] N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Proceedings of the TYPES meeting 2006*. Springer-Verlag, 2007.

[9] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[10] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.

[11] P. Dybjer. Internal type theory. In *TYPES '95, Types for Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, pages 120–134. Springer, 1996.

[12] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 2000.

[13] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, April 1999.

[14] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1–47, 2003.

[15] P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 2006.

[16] M. Hofmann. On the interpretation of type theory in locally cartesian closed categories. In L. Pacholski and J. Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*. Springer, 1994.

[17] M. Hofmann. Syntax and semantics of dependent types. In A. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1996. To appear.

[18] G. Huet and A. Saibi. Constructive category theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg*, January 1995.

[19] P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109, 1975.

[20] P. Martin-Löf. Substitution calculus. Notes from a lecture given in Göteborg, November 1992.

[21] A. M. Pitts. Categorical logic. In *Handbook of Logic in Computer Science*. Oxford University Press, 1997. Draft version of article to appear.

[22] R. Pollack. *The Theory of Lego A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[23] D. S. Scott. Combinators and classes. In C. Böhm, editor, *Lambda-Calculus and Computer Science Theory*, volume 37, pages 1–26, 1975.

[24] R. A. G. Seely. Locally cartesian closed categories and type theory. *Proceedings of the Cambridge Philosophical Society*, 95:33–48, 1984.

Signature Compilation for the Edinburgh Logical Framework[1]

Michael Zeller, Aaron Stump, and Morgan Deters

Computational Logic Group
Computer Science and Engineering Dept.
Washington University in St. Louis
St. Louis, Missouri, USA

Abstract

This paper describes the Signature Compiler, which can compile an LF signature to a custom proof checker in either C++ or Java, specialized for that signature. Empirical results are reported showing substantial improvements in proof-checking time over existing LF checkers on benchmarks.

Keywords: Edinburgh LF, signature compilation

1 Introduction

The Edinburgh Logical Framework (LF) provides a flexible meta-language for deductive systems in several application domains [1]. A well-known example is for proof-carrying code [2]. Another example is for proofs produced from decision procedures [5]. A single LF type checker can be used to check proofs in any deductive system defined by an LF signature (a list of typing declarations and definitions). LF implementations like Twelf work in an interpreting manner: first the LF signature is read, and then proofs can be checked with respect to it [4]. This system description (LFMTP 2007 Category C) describes the Signature Compiler ("sc") tool, which supports a compiling approach to LF type checking: an LF signature is translated to a custom proof checker specialized for that signature. Signature compilation emits checkers that run much faster than existing interpreting checkers on benchmark proofs, as shown in Section 3, including proofs produced by a QBF solver for QBF benchmark formulas. The Signature Compiler is publicly available from the "Software" section of `http://cl.cse.wustl.edu`. For space reasons, this paper must assume familiarity with LF and its Twelf syntax.

[1] This work supported by the U.S. National Science Foundation under grant numbers CCF-0448275 and CNS-0551697.

```
o : type.     trm : type.

== : trm -> trm -> o.     imp : o -> o -> o.

%infix left 3 ==.    %infix left 5 imp.

pf : o -> type.

impi : {p:o} {q:o} (pf p -> pf q) -> pf (p imp q).
mp : pf (P imp Q) -> pf P -> pf Q.
```

Fig. 1. Fragment of example LF signature

2 The Signature Compiler

The intended use of sc is for generating backend checkers, which are optimized for the case when the proof successfully checks. Thus, sc does not report useful error information for failed proofs. Also, backend checkers allow (untrusted) proofs to contain additional definitions, but not additional declarations, which might subvert the deductive system defined by the (trusted) signature. The ideal case for use of sc is when many proofs expressed with respect to the same signature need to be checked efficiently. In such a case, reuse of the custom checker generated by sc makes up for the time needed for signature compilation.

The Signature Compiler parses an LF signature in Twelf syntax, and generates all the source files required for a proof checker that checks proofs expressed with respect to that signature. The Signature Compiler supports fully explicit LF in Twelf syntax, without type-level λ-abstractions (a common restriction, not essential for sc), and where constants declared in the signature must be fully applied when used. The checkers emitted by the Signature Compiler, but not sc itself, also support a form of implicit LF, in which holes ("_") can be written in place of arguments to constants c from the signature, as long as the values of those holes can be determined by unification in the higher-order pattern fragment from the types of other arguments to c. Support for more aggressive compression schemes must remain to future work (cf. [3]). The Signature Compiler is written in around 3000 lines of C++ and can generate custom checkers in both C++ and Java.

Figure 1 gives part of a standard LF signature for an example logic with equality, implication, and universal quantification. This logic is used for the benchmarks below. For space reasons, the figure focuses just on implication; see the Appendix for the complete signature. Infix directives in Twelf syntax, used after the declaration of "`imp`" in the figure, are supported by Signature Compiler, and by the emitted checkers.

The Signature Compiler emits code for custom parsers for each signature it compiles. Neither the emitted parsers nor sc itself relies on parser or lexer generators, since such reliance would increase the size of the trusted computing base, and make it more difficult to support infix directives in proofs. Simple lexer generation – in particular, creating an inlined trie – is performed by sc for lexing efficiency in the emitted checkers. The representation of terms is optimized by generating code for custom classes for each expression declared or defined in the signature. The parser generates instances of these classes when parsing. Binding expressions (λ- and Π-expressions) are parsed in such a way that each bound variable is represented as a distinct instance of a `DefExpr` class, with all uses of the variable represented as

```
case /*===*/ X61o61o_EXPR: {
  /*===*/ X61o61oExpr *e = (/*===*/ X61o61oExpr *)_e;
  if( (areEqualNuke(computeType(e->e1),
                    new /*trm=*/ XtrmExpr()) &&
       areEqualNuke(computeType(e->e2 ),
                    new /*trm=*/ XtrmExpr()) ))
    return new /*o=*/ XoExpr();
  throw str;
}
```

Fig. 2. C++ custom type computation code for `==`

```
case /*impi=*/ Ximpi_EXPR: {
  /*impi=*/ XimpiExpr *e = (/*impi=*/ XimpiExpr *)_e;
  DefExpr *innervar1 =
          new DefExpr("na",new /*pf=*/ XpfExpr(e->e1),
                       new IdExpr("na"));
  if( (areEqualNuke(computeType(e->e1),
                    new /*o=*/ XoExpr()) &&
       areEqualNuke(computeType(e->e2 ),
                    new /*o=*/ XoExpr()) &&
       areEqualNuke(computeType(e->e3 ),
                    new PiExpr( innervar1,
                                new /*pf=*/ XpfExpr(e->e2))) ))
    return new /*pf=*/ XpfExpr(new /*imp=*/
                               XimpExpr(e->e1,e->e2));
  throw str;
}
```

Fig. 3. C++ custom type computation code for `impi`

references to that same instance. In the C++ checkers, this is achieved using a trie rather than an STL `hash_map`, for performance reasons.

The Signature Compiler inlines the code needed to compute the type of an application of a constant declared or defined in the signature. The expected types of arguments are hard-coded into the emitted checkers, and the substitutions which must normally be performed at run-time to compute the return type of an application of a dependently typed function are performed instead during signature compilation. The emitted checkers thus completely avoid the expensive operation of substitution when computing the return type of an application of a constant declared or defined in the signature.

For example, the custom checker generated by sc produces the code shown in Figures 2 and 3 for cases for `==` and `impi` in a switch statement over all possible expressions. Note that since `==` cannot serve as a C++ or Java identifier, sc encodes this name using decimal ASCII character codes. Comments document the connection to the original name. The function `areEqualNuke` tests convertibility and additionally deletes the memory for the expressions it is given. Since the two subexpressions of any `==` expression must be terms (of type `trm`), the custom code for the `imp` case checks this condition. The type `o` is then returned. The code for `impi` is the result of substitution during signature compilation, and hence directly computes the appropriate substituted types.

The custom checker also has customized code for convertibility checking. For example, consider the case of expanding defined constants of functional type where they are applied. The exact expression resulting from substituting the arguments for the λ-bound variables is known from the signature, and thus code to build it directly is generated for the custom checker by Signature Compiler.

n	size	sc: C++	sc: Java	Twelf	sc (interp.)	flea
100	464 KB	0.2 (0.1)	1.2 (1.0)	4.1 (1.5)	2.0 (0.5)	0.8
150	1.01 MB	0.4 (0.2)	2.4 (2.1)	8.7 (2.6)	4.1 (1.1)	1.6
200	1.77 MB	0.6 (0.3)	4.1 (3.5)	16.2 (5.2)	7.1 (1.9)	2.7
250	2.74 MB	0.9 (0.5)	6.2 (5.4)	26.8 (9.2)	10.9 (3.0)	4.2
300	3.92 MB	1.2 (0.7)	8.8 (7.6)	39.7 (13.1)	15.5 (4.2)	6.0
350	5.30 MB	1.7 (1.0)	11.9 (10.4)	52.3 (16.1)	21.0 (5.7)	8.2

Fig. 4. Runtime for EQ benchmarks (in seconds), explicit form

3 Benchmarks

Results on two families of benchmarks are reported in this section, using both explicit and implicit LF. The first are the EQ benchmarks, a family of proofs of statements of the form "if $f(a) = a$ then $f^n(a) = a$", for various sizes n. The proofs are structured (via deliberate inefficiency) to use both hypothetical and parametric reasoning, central aspects of the LF encoding methodology, as well as β-reduction and defined constants. The second are the QBF benchmarks. To obtain these, a simple Quantified Boolean Formula solver was written. This solver reads benchmarks in the standard QDIMACS format, and emits proof terms showing either that the formula evaluates to true or to false. Easy benchmark formulas, obtained from www.qbflib.org are solved to generate the proof terms.

Results on these two families of benchmarks are obtained using five checkers: the custom C++ and Java checkers generated by sc, Twelf, sc itself, and the flea checker [5]. Twelf version 1.5R1 is included as a widely used interpreting checker. The Signature Compiler itself implements an interpreting checker, using similar infrastructure as the custom checker. Comparing sc with the generated checker thus demonstrates the effect of the specializing optimizations. The flea checker is a highly tuned interpreting LF checker, which additionally implements context-dependent caching of computed types. Such caching is not implemented in sc or the emitted checkers. Note that the flea checker does not support implicit LF, infix directives (thus requiring prefix forms of the benchmarks), or printing of parsing times. These checkers are the only publicly available high-performance LF checkers the authors are aware of.

The results for the EQ benchmarks are shown in Figures 4 and 5, and for the QBF benchmarks in Figures 6 and 7. Parsing times, where available, are shown in parentheses. Experiments are averages of three runs on a 2GHz Pentium 4 with 1.5 GB main memory. The C++ and Java checkers emitted by sc were compiled with g++ and gcj, respectively, version 3.4.5. For the QBF benchmarks, a timeout of 30 minutes was imposed (on the toilet_02_01.2 benchmark, Twelf finished in just under that time on one run, so the average time for three runs is included). Note that the redundancy in the QBF explicit benchmarks explains flea's good performance.

4 Conclusion

The Signature Compiler is the first tool of its kind, supporting compilation of an LF signature to optimized C++ or Java backend checkers specialized for that signature. Results on two families of benchmarks, including one family of proofs of

n	size Twelf	size sc	sc: C++	sc: Java	Twelf
100	80 KB	87 KB	0.06 (0.03)	0.31 (0.26)	1.4 (0.2)
150	166 KB	176 KB	0.11 (0.05)	0.54 (0.45)	3.0 (0.4)
200	281 KB	295 KB	0.17 (0.08)	0.87 (0.68)	5.3 (1.0)
250	426 KB	444 KB	0.23 (0.11)	1.25 (1.03)	7.8 (1.9)
300	602 KB	623 KB	0.31 (0.14)	1.78 (1.34)	11.7 (2.2)
350	807 KB	833 KB	0.41 (0.18)	2.28 (1.80)	16.7 (2.5)

Fig. 5. Runtime for EQ benchmarks (in seconds), implicit form

name	size	sc: C++	sc: Java	Twelf	sc (interp.)	flea
cnt01e	2.2 MB	0.9 (0.6)	6.3 (5.8)	28.6 (7.0)	6.8 (2.2)	2.8
tree-exa2-10	2.7 MB	1.3 (0.8)	7.5 (6.9)	34.4 (8.5)	9.4 (2.8)	2.9
cnt01re	3.9 MB	1.7 (1.1)	10.7 (9.6)	56.7 (12.4)	12.3 (3.9)	5.1
toilet_02_01.2	9.7 MB	4.2 (2.7)	24.5 (22.0)	1809 (35.5)	30.6 (9.5)	10.5
1qbf-160cl.0	16.6 MB	6.4 (4.6)	41.3 (38.2)	timeout	44.5 (16.2)	14.6
tree-exa2-15	32.5 MB	15.9 (9.7)	86.1 (75.9)	timeout	114.1 (33.6)	25.8
toilet_02_01.3	96.4 MB	42.9 (27.8)	277.7 (241.2)	timeout	313.0 (99.0)	105.2

Fig. 6. Runtime on QBF benchmarks (in seconds), explicit form

name	size Twelf	size sc	sc: C++	sc: Java	Twelf
cnt01e	167 KB	184 KB	0.2 (0.1)	1.5 (1.4)	7.2 (0.6)
tree-exa2-10	345 KB	392 KB	0.4 (0.1)	2.1 (1.8)	8.9 (0.7)
cnt01re	250 KB	274 KB	0.3 (0.1)	1.9 (1.6)	12.3 (0.9)
toilet_02_01.2	0.9 MB	1.1 MB	1.0 (0.3)	4.1 (3.3)	38.0 (2.7)
1qbf-160cl.0	1.4 MB	1.5 MB	0.8 (0.4)	4.9 (4.6)	197.7 (4.5)
tree-exa2-15	3.9 MB	4.5 MB	4.7 (1.3)	14.4 (10.5)	timeout
toilet_02_01.3	7.6 MB	8.5 MB	9.4 (2.4)	28.1 (19.3)	timeout

Fig. 7. Runtime on QBF benchmarks (in seconds), implicit form

QBF benchmarks, show order-of-magnitude performance improvements for emitted checkers over Twelf and sc itself, and substantial improvements over the flea checker. A form of implicit arguments is supported by sc, offering further space and performance improvements. Future work includes further support for proofs from decision procedures: the second author is proposing LF, backed by the Signature Compiler, as appropriate technology for a standard proof format for the SMT-LIB (Satisfiability Modulo Theories Library) initiative.

The authors wish to thank the anonymous reviewers for their comments on the paper.

References

[1] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[2] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[3] G. Necula and P. Lee. Efficient representation and validation of proofs. In *13th Annual IEEE Symposium on Logic in Computer Science*, pages 93–104, 1998.

[4] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[5] A. Stump and D. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *18th International Conference on Automated Deduction*, pages 392–407, 2002.

```
o : type.
trm : type.

== : trm -> trm -> o.
imp : o -> o -> o.
all : (trm -> o) -> o.
f: trm -> trm.

%infix left 3 ==.
%infix left 5 imp.

pf : o -> type.

refl : {x:trm} pf (x == x).
symm : {x:trm} {y:trm} pf (x == y) -> pf (y == x).
trans : {x:trm} {y:trm} {z:trm}
        pf (x == y) -> pf (y == z) -> pf (x == z).
congf : {x:trm} {y:trm} pf (x == y) -> pf ((f x) == (f y)).

mp : {p:o} {q:o} pf (p imp q) -> pf p -> pf q.
impi : {p:o} {q:o} (pf p -> pf q) -> pf (p imp q).

alli : {P:trm -> o} ({x:trm} pf (P x)) -> pf (all P).
alle : {P:trm -> o} {t:trm} pf (all P) -> pf (P t).

a : trm.
b : trm.
c : trm.

g : trm -> trm = [x:trm] f x.
```

Fig. A.1. LF signature for the EQ benchmarks

```
pol : type.
pos : pol.
neg : pol.

opp : pol -> pol -> type.
opp1 : opp pos neg.
opp2 : opp neg pos.

o : type.

conn : pol -> o -> o -> o.
not : o -> o.
quant : pol -> (o -> o) -> o.
bval : pol -> o.

Equiv : o -> o -> type.

%infix right 3 Equiv.

refl : {p:o} p Equiv p.
trans : {p:o}{q:o}{r:o} p Equiv q -> q Equiv r -> p Equiv r.

connc : {b:pol} {p1:o} {p2:o} {q1:o} {q2:o}
           p1 Equiv p2 -> q1 Equiv q2 -> conn b p1 q1 Equiv conn b p2 q2.
connz1 : {b:pol} {bb:pol} opp b bb ->
         {q:o} conn b (bval bb) q Equiv (bval bb).
connz2 : {b:pol} {bb:pol} opp b bb ->
         {q:o} conn b q (bval bb) Equiv (bval bb).
connu1 : {b:pol} {q:o} conn b (bval b) q Equiv q.
connu2 : {b:pol} {q:o} conn b q (bval b) Equiv q.

notf : not (bval pos) Equiv (bval neg).
notf : not (bval neg) Equiv (bval pos).

quantz : {b:pol}{bb:pol} opp b bb ->
         {a:pol}{p:o -> o} p (bval a) Equiv (bval bb) ->
         quant b p Equiv (bval bb).
quantu : {b:pol}{p:o -> o}
         p (bval pos) Equiv (bval b) ->
         p (bval neg) Equiv (bval b) ->
         quant b p Equiv (bval b).
quantn : {b:pol} {p1:o} quant b ([x:o]p1) Equiv p1.
quantc : {b:pol}{p1:o -> o}{p2:o -> o}
         ({x:o} (p1 x) Equiv (p2 x)) ->
         quant b p1 Equiv quant b p2.
```

Fig. A.2. LF signature for the QBF benchmarks

Induction on Concurrent Terms

Anders Schack-Nielsen[1]

Programming, Logics and Semantics
IT University of Copenhagen
Denmark

Abstract

This paper considers MiniML equipped with a standard big-step semantics and a destination-passing semantics both represented in concurrent LF (CLF) and prove the two semantics equivalent. The proof is then examined yielding insights into the issues concerning induction on concurrent terms. We conclude by outlining some of the difficulties that one will need to address when designing a meta-logic for CLF.

Keywords: CLF, logical frameworks, induction, destination-passing style.

1 Introduction

CLF [1] is a logical framework with several interesting applications including adequate representations of the π-calculus, protocols and programming languages employing state, concurrency, lazy computations and more. Furthermore, a large subset of these semantic specifications can currently be run with LolliMon [3] which implements parts of CLF. However, CLF currently has no notion of meta-logic and it is therefore not possible to reason about CLF representations within CLF. In this paper we will consider an initial case study in order to shed light on some of the difficulties that one will need to address when designing a meta-logic for CLF.

CLF is a dependently typed lambda calculus extended by linear types and monadic types inhabited by concurrent terms, which makes it a conservative extension of the dependently typed logical framework LF. Therefore CLF supports the same "judgments as types, derivations as terms" methodology as LF. The Twelf system [5] implements LF and provides a meta-logic for reasoning about LF representations. Twelf is well-suited for formalizing functional programming languages, their operational semantics and type systems, as well as classical and intuitionistic logics. However, imperative and concurrent language features are hard to implement and reason about using Twelf since e.g. state has to be modelled and reasoned about explicitly.

[1] Email: anderssn@itu.dk

The presence of concurrent terms in CLF allows for a new representation methodology compared to the way e.g. operational semantics has been represented in Twelf. In Twelf the methodology is a goal-oriented approach focusing on proof-search via backward chaining bearing much resemblance to logic programming, whereas in CLF the canonical representation methodology is context-oriented, employing forward chaining inside the monad. As CLF is a conservative extension to LF it allows both styles of representation to coexist.

The Twelf methodology provides means to represent meta-theory and its proofs as higher-level judgments describing relations between derivations, and these proofs can then be mechanically checked by checking the totality of the relation.

So the important question is whether the methodology of meta-theory representation and proof representation known from Twelf can be conservatively extended to deal with the new CLF representations and how. The CLF extensions over LF are linear and concurrent terms, so a conservative meta-logic for CLF would need to extend Twelf with induction on linear and concurrent terms. The importance of this question is emphasized by the fact that it is a main part of the uncharted CLF-territory and contains valuable insight on the directions in which CLF could be further developed.

The case study that we will consider in this paper is the equivalence proof of two semantics for MiniML. On the one hand we can represent a big-step semantics completely within the LF fragment of CLF, and on the other hand we can also represent the semantics in destination-passing style employing the distinct features of CLF. This style of representation is based on multiset rewriting with names (destinations) representing the holes in evaluation contexts. Furthermore, destination-passing style is a natural way to represent semantics in CLF and it allows for easy extension of the MiniML semantics to include lazy evaluation, futures, mutable references and concurrency [2]. Given these two styles of semantics, the equivalence proof will bridge the two different representation methodologies, and we will use the proof to outline some of the difficulties that one will need to address when designing a meta-logic for CLF.

2 CLF

2.1 Syntax

In the CLF type theory we have objects, types and kinds. In order to simplify the meta-theory all terms are required to be in canonical form (i.e. completely beta-reduced and completely eta-expanded), and this invariant can be maintained by a suitable definition of substitution which performes the necessary reduction steps (hereditary substitution).

The CLF types are the ones known from LF (with $A \to B$ as syntactic sugar for $\Pi x : A.\ B$ as usual) and the linear connectives from LLF, i.e. linear implication ($\multimap$), additive product ($\&$) and top ($\top$). Then there is multiplicative product ($\otimes$), the multiplicative unit (1) and dependent pair ($\exists$) all of which are wrapped in a monadic type constructor $\{S\}$. The complete syntax is given in figure 1. The destinction between normal and atomic objects is simply there to enforce canonical

Kinds

$$K ::= \mathsf{type} \mid \Pi x : A.\ K \qquad \textit{Kinds}$$

Types

$$\begin{array}{rll} A, B ::= & A \multimap B \mid \Pi x : A.\ B \mid A \mathbin{\&} B \mid \top \mid \{S\} \mid P & \textit{Asynchronous types} \\ P ::= & a \mid P\ N & \textit{Atomic type constructors} \\ S ::= & S_1 \otimes S_2 \mid 1 \mid \exists x : A.\ S \mid A & \textit{Synchronous types} \end{array}$$

Objects

$$\begin{array}{rll} N ::= & \widehat{\lambda} x.\ N \mid \lambda x.\ N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R & \textit{Normal objects} \\ R ::= & c \mid x \mid R \mathbin{\widehat{}} N \mid R\ N \mid \pi_1\ R \mid \pi_2\ R & \textit{Atomic objects} \\ E ::= & \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M & \textit{Expressions} \\ M ::= & M_1 \otimes M_2 \mid 1 \mid [N, M] \mid N & \textit{Monadic objects} \\ p ::= & p_1 \otimes p_2 \mid 1 \mid [x, p] \mid x & \textit{Patterns} \end{array}$$

Contexts

$$\begin{array}{rll} \Gamma ::= & \cdot \mid \Gamma, x : A & \textit{Unrestricted contexts} \\ \Delta ::= & \cdot \mid \Delta, x \mathbin{\widehat{?}} A & \textit{Linear contexts} \end{array}$$

Signatures

$$\Sigma ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \qquad \textit{Signatures}$$

Fig. 1. CLF syntax

forms.

Constructing objects inside the monad (i.e. expressions inside curly braces) is supposed to model concurrent computation, and any given term consisting of a sequence of let expressions denotes a trace of that computation. In order to facilitate this interpretation two terms will be considered equivalent if they only differ in the ordering of their let expressions. The equivalence $\equiv$ is defined as the smallest congruence relation satisfying

$$\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E \equiv \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E$$

where the bindings are independent: p_1 and p_2 must bind disjoint sets of variables, no variable bound by p_1 can appear free in R_2 and vice versa.

2.2 Computational interpretation of CLF

The representation of meta-theory in Twelf is based on a computational interpretation of LF signatures as logic programs. With this in place a meta-logic can then be used to state the totality of certain relations, which thereby represent constructive proofs.

The basis of computation is constructing a term of a given type, by the means of proof search. This consists of applying right-rules in the corresponding logic until the goal is reduced to an atomic type, at which point the different constructors of the type is tried one by one by backtracking from unsatisfiable subgoals.

The semantics of CLF is similar (it is implemented as the language LolliMon [2] and described in detail in [3]) except when encountering the monad type. At this point the computation goes from being goal-directed to being context-directed. The context-directed computation consists of a sequence of steps, each of which is a nondeterministic choice between either ending the context-directed mode and constructing the monadic object M directly or nondeterministically choosing a term in the context (or signature) and reduce it to its monadic head with left-rules at which point the context gets augmented with the newly constructed types using a let-binding: $\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E$, where R is the computation step that was just taken, p is the binding of the newly constructed types and E is the rest of the computation.

These steps are considered atomic and are not undone, backtracking is only applied during the construction of the individual steps to make sure that the step can actually be completed before committing to the nondeterministic choice.

2.3 CLF meta-theory

In Twelf, proofs are by structural induction since whatever is represented in Twelf is represented as an inductively defined LF-term. Furthermore the proof objects themselves are inductively defined LF-terms. We expect this meta-level representational methodology to extend to CLF as well, since it is a conservative extension at both the object level and the semantic level. There are however several challenges, and the one we will focus on is how to extend the structural induction known from Twelf to one working with terms with implicit concurrency. [3]

3 MiniML

The primary object of study in this paper will be the semantics of MiniML represented in two different ways. The first representation is a big-step semantics represented entirely in the LF fragment of CLF as it would be done in Twelf. The second representation is done in destination-passing style employing the monad and the linear features of CLF (see [2]). The meta-theorem that we will be examining is the equivalence proof of these semantics.

Note that the destination-passing semantics does everything sequentially, but since it is within the monad the potential for concurrency is still enough to generate intersting observations as we will see below. Furthermore the destination-passing semantics can easily be extended with e.g. concurrency, mutable references, lazy evaluation, etc. (as shown in [2]). In section 5 we will discuss some of the complications of the proof in the context of concurrency.

[2] LolliMon is not exactly CLF since for the monadic and linear types it only includes the corresponding logic and not the terms. But currently LolliMon is as close as one gets to an implementation of CLF and it is sufficient for execution of programs in the destination-passing semantics given in this paper.

[3] As a side note, notice that the monadic terms potentially allows for a "concurrent" proof built in a more algorithmic manner instead of the usual induction proofs. What this means is however still unclear.

3.1 Syntax

The fragment of MiniML that we will be considering include abstractions, applications, fixpoints and natural numbers with zero, successor and case.

$$e ::= x \mid \mathsf{z} \mid \mathsf{s}\ e \mid (\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{z} \Rightarrow e_2 \mid \mathsf{s}\ x \Rightarrow e_3) \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{fix}\ x.e$$

The MiniML syntax is represented in CLF (and LF) as shown in figure 2.

```
exp : type.

z : exp.
s : exp → exp.
case : exp → exp → (exp → exp) → exp.
lam : (exp → exp) → exp.
app : exp → exp → exp.
fix : (exp → exp) → exp.
```

Fig. 2. MiniML syntax in CLF

3.2 Big-step semantics

The first semantics for MiniML is a standard call-by-value big-step semantics (figure 3) and has the standard representation where the type family `ev` E V is inhabited if and only if E evaluates to V. [4]

```
ev : exp → exp → type.

ev_z      : ev z z.
ev_s      : ΠE:exp. ΠV:exp. ev E V → ev (s E) (s V).
ev_case_z : ΠE1:exp. ΠE2:exp. ΠE3:exp → exp. ΠV:exp.
              ev E1 z → ev E2 V → ev (case E1 E2 E3) V.
ev_case_s : ΠE1:exp. ΠE2:exp. ΠE3:exp → exp. ΠV:exp. ΠV':exp.
              ev E1 (s V') → ev (E3 V') V → ev (case E1 E2 E3) V.
ev_lam    : ΠE:exp → exp. ev (lam E) (lam E).
ev_app    : ΠE1:exp. ΠE2:exp. ΠE1':exp → exp. ΠV:exp. ΠV2:exp.
              ev E1 (lam E1') → ev E2 V2 → ev (E1' V2) V
                 → ev (app E1 E2) V.
ev_fix    : ΠE:exp → exp. ΠV:exp. ev (E (fix E)) V → ev (fix E) V.
```

Fig. 3. Big-step semantics in CLF

The given representation is not strictly a CLF signature as defined in [1] since it is not in canonical form. It can however easily be transformed into the equivalent canonical form by eta expansion. In the following I will freely use any form eta equivalent to a canonical form, since the more verbose canonical form can be obtained by mechanical eta expansions.

[4] Note that this semantics as it is given is not suitable for Twelf execution, since Twelf solves subgoals "inside out". If the semantics should be executed in Twelf, one would therefore have to do a simple rewriting, reversing the order of the arguments of `ev_case_z`, `ev_case_s` and `ev_app`.

3.3 Destination-passing semantics

The second semantics for MiniML is a destination-passing semantics. Destination-passing style is based on multi-set rewriting and handles evaluation contexts (a.k.a. continuations) implicitly by naming the context holes. The names of the holes in the evaluation contexts are called destinations [6]. With the logic programming semantics of CLF outlined above in mind, the destination-passing semantics of MiniML is defined as follows. We introduce a type of destinations `dest` [5], a type family `eval` E D representing the instruction to evaluate E and return the result in destination D and a type family `return` V D representing the returned value V in destination D. Now the type $\Pi d : \texttt{dest}.\ \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V\ d\}$ is inhabited if and only if E evaluates to V.

```
dest   : type.
return : exp → dest → type.
eval   : exp → dest → type.

eval_z    : ΠD:dest. eval z D ⊸ {return z D}.
eval_s    : ΠE:exp. ΠD:dest.
              eval (s E) D ⊸ {∃d':dest. eval E d' ⊗
                                ΠV:exp. return V d' ⊸ {return (s V) D}}.
eval_case : ΠE1:exp. ΠE2:exp. ΠE3:exp → exp. ΠD:dest.
              eval (case E1 E2 (λx. E3 x)) D ⊸
                  {∃d':dest. eval E1 d' ⊗
                    ( (return z d' ⊸ {eval E2 D}) &
                      (ΠV':exp. return (s V') d' ⊸ {eval (E3 V') D})
                    ) }.
eval_lam  : ΠE:exp → exp. ΠD:dest.
              eval (lam (λx. E x)) D ⊸ {return (lam (λx. E x)) D}.
eval_app  : ΠE1:exp. ΠE2:exp. ΠD:dest.
              eval (app E1 E2) D ⊸
                  {∃d':dest. eval E1 d' ⊗
                    (ΠE1':exp → exp. return (lam (λx. E1' x)) d' ⊸
                      {∃d'':dest. eval E2 d'' ⊗
                        (ΠV2:exp. return V2 d'' ⊸ {eval (E1' V2) D})
                      }
                    )
                  }.
eval_fix  : ΠE:exp → exp. ΠD:dest.
              eval (fix (λx. E x)) D ⊸ {eval (E (fix (λx. E x))) D}.
```

Fig. 4. Destination-passing semantics

The signature is given in figure 4. Notice how each constructor consumes an `eval` E D to produce either a `return` V D representing the result, or a new `eval` E' d' corresponding to the subexpression to be evaluated next along with a con-

[5] Notice how the type `dest` is empty since we initially have no evaluation context and thus no holes to name, i.e. all we will ever see are variables of type `dest`.

tinuation in the form $\Pi V : \texttt{exp}.\ \texttt{return}\ V\ d' \multimap \{\dots\}$. Take for instance `eval_case`. Assuming that we have an `eval` $E\ D$ in the context with $E = \texttt{case}\ E_1\ E_2\ E_3$ and we aim to construct a `return` $V\ D$ in the monad. Then `eval_case` can be applied to yield a fresh destination d', an `eval` $E_1\ d'$ and a continuation which can only be applied when the result of evaluating E_1 has finished. The continuation is an additive product which means that we can only ever use one of the branches. The `eval` $E_1\ d'$ will trigger further rules and end up with a result in the form of `return` $V_1\ d'$ (assuming termination). If V_1 is `z` we can apply the first projection of the continuation and if V_1 is `s` V' we can apply the second projection. In both cases we end up with a new `eval` $E'\ D$ designating the expression to be evaluated and this will in turn trigger further rules and if this terminates we will end up with the result in the form of a `return` $V\ D$.

4 Equivalence of semantics

We would like to prove the equivalence of the two semantics presented above. More formally, we will prove the following theorem:

Theorem 4.1 *For all closed terms E and V of type* `exp`*, the type* `ev` $E\ V$ *is inhabited if and only if the type* $\Pi d : \texttt{dest}.\ \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V\ d\}$ *is inhabited.*

The proof consists of two parts, each being a translation from one semantics to the other. We will start with the easy one: translating big-step into destination-passing style.

4.1 Translation from big-step to destination-passing style

4.1.1 The paper proof

Lemma 4.2 *Let E and V be closed terms of type* `exp` *and let P be a closed term of type* `ev` $E\ V$*. Then there exists a closed term C of type* $\Pi d : \texttt{dest}.\ \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V\ d\}$*.*

Proof. The proof is a simple structural induction on P.

Case: $P = \texttt{ev\_z}$
We take $C = \texttt{eval\_z}$.

Case: $P = \texttt{ev\_s}\ E'\ V'\ P'$
In this case $E = \texttt{s}\ E'$, $V = \texttt{s}\ V'$ and P' is of type `ev` $E'\ V'$. We can therefore apply the induction hypothesis to P' to get a C'. Now let

$$
\begin{aligned}
C = \lambda d.\widehat{\lambda} u : \texttt{eval}\ (\texttt{s}\ E')\ d.\ \{\mathsf{let}\ \{[d', (p : \texttt{eval}\ E'\ d') \\
\otimes (f : \Pi V.\texttt{return}\ V\ d' \multimap \{\texttt{return}\ (\texttt{s}\ V)\ d\})]\} \\
= \texttt{eval\_s}\ E'\ d\ \widehat{}\ u\ \mathsf{in} \\
\mathsf{let}\ \{r' : \texttt{return}\ V'\ d'\} = C'\ d'\ \widehat{}\ p\ \mathsf{in} \\
\mathsf{let}\ \{r : \texttt{return}\ (\texttt{s}\ V')\ d\} = f\ V'\ \widehat{}\ r'\ \mathsf{in} \\
r\}.
\end{aligned}
$$

Case: $P = \texttt{ev\_case\_z}\ E_1\ E_2\ E_3\ V\ P_1\ P_2$

In this case P_1 is of type $\texttt{ev}\ E_1\ \texttt{z}$ and P_2 is of type $\texttt{ev}\ E_2\ V$. We apply the induction hypothesis to P_1 and P_2 yielding C_1 and C_2. Now let

$$\begin{aligned} C = \lambda d.\widehat{\lambda} u.\{&\mathsf{let}\ \{[d', (p_1 : \texttt{eval}\ E_1\ d') \otimes f_1]\} = \texttt{eval\_case}\ E_1\ E_2\ E_3\ d\ \widehat{}\ u\ \mathsf{in} \\ &\mathsf{let}\ \{r' : \texttt{return z}\ d'\} = C_1\ d'\ \widehat{}\ p_1\ \mathsf{in} \\ &\mathsf{let}\ \{p_2 : \texttt{eval}\ E_2\ d\} = (\pi_1\ f_1)\ \widehat{}\ r'\ \mathsf{in} \\ &\mathsf{let}\ \{r : \texttt{return}\ V\ d\} = C_2\ d\ \widehat{}\ p_2\ \mathsf{in} \\ &r\}. \end{aligned}$$

The remaining cases are similar. The induction hypothesis is applied to all subterms representing subevaluations (i.e. subterms of type $\texttt{eval}\ E\ V$ for some E and V), after which C is easily constructed. □

4.1.2 Representation of the proof in CLF

Since the above proof only relies on straigtforward induction on LF-terms it should be easy to represent in CLF for any conservative extension of the Twelf meta-theory to CLF. This is however still very speculative. More on this below in section 4.2.2.

4.2 Translation from destination-passing style to big-step

This part of the proof is a lot trickier. We cannot simply deconstruct a term of type $\texttt{eval}\ E\ D \multimap \{\texttt{return}\ V\ D\}$ into a constructor and subterms of the same type schema, since this among other things relies on the implicit ordering of the consumption of linear variables.

4.2.1 The paper proof

In order to complete the proof we will need to come up with a much stronger induction hypothesis. We will need to reason about the continuations that can occur in the linear context, and in order to make this precise, we will start with a definition of *normal* linear contexts to be the relevant linear implications from `return` ... into a monadic type:

Definition 4.3 A linear context Δ is called *normal* if it only consists of variables with the following types:

- $\Pi v : \texttt{exp}.\ \texttt{return}\ v\ D' \multimap \{\texttt{return}\ (\texttt{s}\ v)\ D\}$
- $(\texttt{return z}\ D' \multimap \{\texttt{eval}\ E_2\ D\})$
  $\&\ (\Pi v' : \texttt{exp}.\ \texttt{return}\ (\texttt{s}\ v')\ D' \multimap \{\texttt{eval}\ (E_3\ v')\ D\})$
- $\Pi e_1' : \texttt{exp} \to \texttt{exp}.\ \texttt{return}\ (\texttt{lam}\ (\lambda x.\ e_1'\ x))\ D' \multimap \{\exists d'' : \texttt{dest}.\ \texttt{eval}\ E_2\ d'' \otimes$
  $(\Pi v_2 : \texttt{exp}.\ \texttt{return}\ v_2\ d'' \multimap \{\texttt{eval}\ (e_1'\ v_2)\ D\})\}$
- $\Pi v_2 : \texttt{exp}.\ \texttt{return}\ v_2\ D'' \multimap \{\texttt{eval}\ (E_1'\ v_2)\ D\}$

for any instantiations of the free variables (written with capital letters).

Notice that these types correspond exactly to the continuations put in the context by `eval_s`, `eval_case` and `eval_app`. The latter is represented with two

possible types, since the application of the continuation result in yet another continuation.

Now we can state the lemma:

Lemma 4.4 *Let Γ be a context of destinations, $\Gamma = d_1 : \mathtt{dest}, \ldots, d_n : \mathtt{dest}$, and let Δ be a normal linear context. Let E and V' be closed terms of type* `exp`*. Let d and d' be two (not necessarily distinct) destinations in Γ. And let C be a term with a typing $\Gamma; \Delta \vdash C : \mathtt{eval}\ E\ d \multimap \{\mathtt{return}\ V'\ d'\}$. Then there exists a closed term V of type* `exp`*, a closed term P of type* `ev` $E\ V$*, a context Γ' of destinations with $\Gamma \subseteq \Gamma'$ and a subterm R of C with a typing $\Gamma'; \Delta \vdash R : \mathtt{return}\ V\ d \multimap \{\mathtt{return}\ V'\ d'\}$.*

Proof. The proof is by induction on C. First of all C must have the form $\widehat{\lambda}u : \mathtt{eval}\ E\ d.\{\ldots\}$. Secondly, since there is no way to construct a term of type `return` ... directly in the current context and signature, we know that C must consist of at least one computation step (let-term). This first step must be an application of one of the `eval_?`'s from the signature, since everything in the context constructing something monadic requires a term of type `return` ... to be present.

Now we can consider the different possibilities. The cases are very similar so we will only present the zero, successor and the application cases in detail.

Case: $C = \widehat{\lambda}u : \mathtt{eval}\ \mathtt{z}\ d.\{\mathsf{let}\ \{r\} = \mathtt{eval\_z}\ d\ \widehat{}\ u\ \mathsf{in}\ R'\}$
In this case we can take $V = \mathtt{z}$, $P = \mathtt{ev\_z}$ and $R = \widehat{\lambda}r.\{R'\}$.

Case: $C = \widehat{\lambda}u : \mathtt{eval}\ (\mathtt{s}\ E_1)\ d.\{\mathsf{let}\ \{[d_0, p \otimes f]\} = \mathtt{eval\_s}\ E_1\ d\ \widehat{}\ u\ \mathsf{in}\ C'\}$
We apply the induction hypothesis to $\Gamma_0; \Delta_0 \vdash \widehat{\lambda}p.\{C'\} : \mathtt{eval}\ E_1\ d_0 \multimap \{\mathtt{return}\ V'\ d'\}$ where $\Gamma_0 = \Gamma, d_0 : \mathtt{dest}$ and $\Delta_0 = \Delta, f \mathbin{\widehat{?}} \Pi v.\mathtt{return}\ v\ d_0 \multimap \{\mathtt{return}\ (\mathtt{s}\ v)\ d\}$ to get $V_1 : \mathtt{exp}$, $P_1 : \mathtt{ev}\ E_1\ V_1$ and $\Gamma'; \Delta_0 \vdash R' : \mathtt{return}\ V_1\ d_0 \multimap \{\mathtt{return}\ V'\ d'\}$. Now since $d_0 \neq d'$ then R' has to be of the form $\widehat{\lambda}r'.\{\mathsf{let}\ \{r\} = f\ V_1\ \widehat{}\ r'\ \mathsf{in}\ R''\}$. Then we can take $V = \mathtt{s}\ V_1$, $P = \mathtt{ev\_s}\ P_1$ and $R = \widehat{\lambda}r.\{R''\}$.

Case: $C = \widehat{\lambda}u : \mathtt{eval}\ (\mathtt{case}\ E_1\ E_2\ E_3)\ d.\{\mathsf{let}\ \{[d_0, p \otimes f]\} = \mathtt{eval\_case}\ E_1\ E_2\ E_3\ d\ \widehat{}\ u\ \mathsf{in}\ C'\}$
This is similar to the successor case above except that there is now two possible forms for R', each of which yields subcomputations with `eval`'s in the context; i.e. R' can be $\widehat{\lambda}r'.\{\mathsf{let}\ \{p'\} = (\pi_1\ f)\ \widehat{}\ r'\ \mathsf{in}\ C_2'\}$ or $\widehat{\lambda}r'.\{\mathsf{let}\ \{p'\} = (\pi_2\ f)\ V''\ \widehat{}\ r'\ \mathsf{in}\ C_3'\}$. The induction hypothesis can then be applied again on C_2' and C_3' yielding `ev` $E_2\ V$ and `ev` $(E_3\ V'')\ V$ respectively. Together with the big-step term from the first application of the induction hypothesis we can now create a term of type `ev` $(\mathtt{case}\ E_1\ E_2\ E_3)\ V$ with either `ev_case_z` or `ev_case_s`.

Case: $C = \widehat{\lambda}u : \mathtt{eval}\ (\mathtt{lam}\ E')\ d.\{\mathsf{let}\ \{r\} = \mathtt{eval\_lam}\ E'\ d\ \widehat{}\ u\ \mathsf{in}\ R'\}$
This case is similar to the zero case; i.e. we take $V = \mathtt{lam}\ E'$, $P =$ `ev_lam E'` and $R = \widehat{\lambda}r.\{R'\}$.

Case: $C = \widehat{\lambda}u : \mathtt{eval}\ (\mathtt{app}\ E_1\ E_2)\ d.\{\mathsf{let}\ \{[d_0, p_1 \otimes f_1]\} = \mathtt{eval\_app}\ E_1\ E_2\ d\ \widehat{}\ u\ \mathsf{in}\ C_1\}$
We apply the induction hypothesis to $\Gamma_1; \Delta_1 \vdash \widehat{\lambda}p_1.\{C_1\} : \mathtt{eval}\ E_1\ d_0 \multimap \{\mathtt{return}\ V'\ d'\}$. This gives us $P_1 : \mathtt{ev}\ E_1\ V_1$ and $\Gamma_1'; \Delta_1 \vdash R_1 : \mathtt{return}\ V_1\ d_0 \multimap \{\mathtt{return}\ V'\ d'\}$. Now R_1 has to be on the form $\widehat{\lambda}r.\{\mathsf{let}\ \{[d_0', p_2 \otimes f_2]\} = f_1\ E_1'\ \widehat{}\ r\ \mathsf{in}\ C_2\}$. This implies that $V_1 = \mathtt{lam}\ E_1'$. Since C_2 is a subterm of

C we can apply the induction hypothesis on $\Gamma_2; \Delta_2 \vdash \widehat{\lambda} p_2.\{C_2\} : \texttt{eval}\ E_2\ d_0' \multimap \{\texttt{return}\ V'\ d'\}$. This gives $\vdash P_2 : \texttt{ev}\ E_2\ V_2$ and $\Gamma_2'; \Delta_2 \vdash R_2 : \texttt{return}\ V_2\ d_0' \multimap \{\texttt{return}\ V'\ d'\}$. Now R_2 has to be on the form $\widehat{\lambda} r.\{\mathsf{let}\ \{p_3\} = f_2\ V_2\ \widehat{}\ r\ \mathsf{in}\ C_3\}$. Since C_3 is a subterm of C_2 which is a subterm of C we can apply the induction hypothesis on $\Gamma_3; \Delta \vdash \widehat{\lambda} p_3.\{C_3\} : \texttt{eval}\ (E_1'\ V_2)\ d \multimap \{\texttt{return}\ V'\ d'\}$. This gives $\vdash P_3 : \texttt{ev}\ (E_1'\ V_2)\ V_3$ and $\Gamma'; \Delta \vdash R_3 : \texttt{return}\ V_3\ d \multimap \{\texttt{return}\ V'\ d'\}$. Now we can set $V = V_3$, construct P from P_1, P_2 and P_3 using `ev_app` and set $R = R_3$.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ (\texttt{fix}\ E')\ d.\{\mathsf{let}\ \{p\} = \texttt{eval\_fix}\ E'\ d\ \widehat{}\ u\ \mathsf{in}\ C'\}$

This case follows directly from one application of the induction hypothesis.

□

Now we can apply the lemma to $d : \texttt{dest}; \cdot \vdash C : \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V'\ d\}$. This gives $\Gamma'; \cdot \vdash R : \texttt{return}\ V\ d \multimap \{\texttt{return}\ V'\ d\}$, but since Δ is empty and $d = d'$, R has to be equal to $\widehat{\lambda} r.\{r\}$. This in turn implies $V' = V$ which gives us the sought $\vdash P : \texttt{ev}\ E\ V'$ completing the translation from destination-passing style to big-step semantics.

4.2.2 Representation of the proof in CLF

Currently CLF does not have a meta-theory to support the representation of proofs. So even though it is very speculative, it is still interesting to consider the representation of the above proof in CLF, as we will gain insight in some of the unresolved issues regarding the design of a meta-theory for CLF.

One of the main issues is how to adequately state the lemma (or theorems in general). Some of the problems that are related to linearity arise already in the context of linear LF (LLF) and are discussed in [7].

A natural first approach would be: [6]

```
lemma : ΠE:exp. ΠD1:dest. ΠD2:dest. ΠV2:exp. ΠV1:exp.
        (eval E D1 —◦ {return V2 D2})
        → ev E V1
        → (return V1 D1 —◦ {return V2 D2}) → type.
%mode lemma +E +D1 +D2 +V2 -V1 +C -P -R.
```

The line `%mode ...` is the Twelf way of specifying which arguments should be regarded as input (`+`) and which should be regarded as output (`-`). The type should thus be read as "Given E, D_1, D_2, V_2 and C where C has type $\texttt{eval}\ E\ D_1 \multimap \{\texttt{return}\ V_2\ D_2\}$, there exists V_1, P and R such that P has type $\texttt{ev}\ E\ V_1$ and R has type $\texttt{return}\ V_1\ D_1 \multimap \{\texttt{return}\ V_2\ D_2\}$.

The zero case can be encoded without problems:

```
lemma_z : lemma z D1 D2 V2 z
          (λ̂u:eval z D1.{let {r'} = eval_z ^u in let {r} = R^r' in r})
          ev_z R.
```

But we get in trouble with the successor case:

[6] We will disregard the problem with the continuation being a subcomputation of the input, since that is already studied in the context of Twelf and can be solved.

```
lemma_s : lemma (s E) D1 D2 V2 (s V)
            (λ̂u:eval (s E) D1.
                {let {[d',p⊗f]} = eval_s ^u in
                 let {r} = C d' ^f ^p in r})
            (ev_s P) R
    ← Πd'. Πf. lemma E d' D2 V2 V (λ̂p. C d' ^f ^p) P
            (λ̂r'.{let {r''} = f V ^r' in let {r} = R ^r'' in r}).
```

There are two problems. The first is with `f`. One could imagine that a hypothetical CLF coverage checker doing output coverage[7] would not be able to see that `f` cannot occur in `R`. This is because the definition of the type family gives no indication of the relationship between the linear contexts of the given computation trace and the returned continuation, as opposed to the paper formulation in which we are able to state that they should be equal.

The second problem is the newly created destinations. Every time a new destination is created it stays in scope for the entire rest of the computation. This is handled in the paper proof above by stating that the continuation is typed in Γ', even though $\Gamma' \setminus \Gamma$ essentially is superfluous. But we cannot have a type family in which the different arguments are typed in different contexts; and realizing when the different destinations are no longer needed is not trivial by local observations. This problem manifests itself in the same way as the first, namely that a hypothetical coverage checker would not be able rule out the possibility of `d'` occurring in `R`.

Another central issue is that of (input) coverage checking. Once we have all of the cases from the proof, a hypothetical coverage checker would need to figure out that all cases are indeed covered. This implies analyzing the possibilities of pattern matching monadic objects. If we disregard reordering of let-terms then coverage checking should not be much harder than for LF. But this is a very conservative solution and probably not what we want (see section 5 below).

As a side note, notice that the specification of normal contexts resembles the world declarations of Twelf.

5 Handling interleavings of let-bindings

The considered semantics are both sequential. Let us see what happens if we use the features of CLF to make the destination-passing style concurrent. Consider the following alternative, concurrent version of `eval_app`:

```
eval_app' : ΠE1:exp. ΠE2:exp. ΠD:dest.
             eval (app E1 E2) D ⊸
                 {∃d1:dest. ∃d2:dest. eval E1 d1 ⊗ eval E2 d2 ⊗
                     (ΠE1':exp → exp. ΠV2:exp.
                         return (lam (λx. E1' x)) d1 ⊸
                         return V2 d2 ⊸ {eval (E1' V2) D}
                     )
                 }.
```

[7] Output coverage checking is essentially checking the validity of inversion.

This version differs from the previous by adding both $\texttt{eval}\ E_1\ d_1$ and $\texttt{eval}\ E_2\ d_2$ to the context at the same time. This means that the subsequent evaluations of E_1 and E_2 can happen in any order and the individual steps can be arbitrarily interleaved. However, since these two computations are essentially independent, the different traces representing different interleavings must all be equivalent modulo let-floating; but since this fact is not immediate the proof gets more complicated.

Let us see how a proof translating this concurrent version into big-step looks like. First of all, since there can now be multiple `eval`'s in the context we will have to modify the definition of normal contexts to accomodate this, i.e. allow variables with types $\texttt{eval}\ E\ D$ and $\texttt{return}\ V\ D$ to occur in a normal context. With this new definition lemma 4.4 can be reused in its exact same formulation. Notice that this singles out a particular $\texttt{eval}\ E\ d$ to be the focus of the lemma.

Now in order to start the proof and split by cases like we did above we will need to argue that C does indeed begin with the consumption of the $\texttt{eval}\ E\ D$ that the lemma focusses on. This is, however, no longer immediate. The computation trace C can just as well begin with the consumption of any of the other `eval`'s in the context or by the application of a $\Pi \ldots\ \texttt{return} \ldots \multimap \{\ldots\}$ to a corresponding `return`. Therefore we will need a let-floating-lemma to state that any C with the type given is equivalent to a trace in which the particular $\texttt{eval}\ E\ d$ is consumed first:

Lemma 5.1 (let-floating for `eval`'s) *Let Γ be a context of destinations, $\Gamma = d_1 : \texttt{dest}, \ldots, d_n : \texttt{dest}$, and let Δ be a normal linear context. Let E and V' be closed terms of type* `exp`*. Let d and d' be two (not necessarily distinct) destinations in Γ. And let C be a term with a typing $\Gamma; \Delta \vdash C : \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V'\ d'\}$. Then there exists a term $C' \equiv C$, such that $C' = \widehat{\lambda} u.\{\mathsf{let}\ \{\ldots\} = \ldots\ \widehat{}\, u\ \mathsf{in}\ C''\}$.*

The dots in the form for C' covers all the different cases that the main proof subsequently splits into.

The proof of this let-floating-lemma relies on the fact that there can never be introduced anything in the (linear or unrestricted) contexts, which would allow the linear ressource $\texttt{eval}\ E\ d$ to be consumed in any different way.

With this in place we can reuse the cases of the proof for zero, lambda and fixpoint without changes. The other cases will however require their own let-floating-lemmas. Consider for instance the successor case; after the application of the induction hypothesis, we want to apply inversion to conclude that R' begins with the application of f, but this requires a specific let-floating-lemma stating that any R' of the corresponding type is equivalent to a term beginning with the application of f. Similarly for the other cases; each time inversion is used on the R resulting from the induction hypothesis we will need a specific let-floating-lemma.

Here are two of them:

Lemma 5.2 (let-floating for the successor case) *Let Γ be a context of destinations, $\Gamma = d_1 : \texttt{dest}, \ldots, d_n : \texttt{dest}$, and let Δ be a normal linear context. Let V and V' be closed terms of type* `exp`*. Let d and d' be two (not necessarily distinct) destinations in Γ and let d'' be a destination in Γ distinct from the other two. And let R be a term with typing $\Gamma; \Delta, f \mathbin{\widehat{:}} \Pi v.\ \texttt{return}\ v\ d'' \multimap \{\texttt{return}\ (\texttt{s}\ v)\ d\}, r' \mathbin{\widehat{:}} \texttt{return}\ V\ d \vdash R : \{\texttt{return}\ V'\ d'\}$. Then there exists a term*

$R' \equiv R$, *such that* $R' = \{\mathsf{let}\ \{r\} = f\ V\ \widehat{}\ r'\ \mathsf{in}\ R''\}$.

Lemma 5.3 (let-floating for the concurrent app case) *Let* Γ *be a context of destinations,* $\Gamma = d_1 : \texttt{dest}, \ldots, d_n : \texttt{dest}$*, and let* Δ *be a normal linear context. Let* V_1*,* V_2 *and* V' *be closed terms of type* `exp`*. Let* d *and* d' *be two (not necessarily distinct) destinations in* Γ *and let* d_1 *and* d_2 *be two distinct destinations in* Γ *distinct from the other two. And let* R *be a term with typing* $\Gamma; \Delta, f \mathbin{\widehat{:}} \Pi e.\Pi v.\ \texttt{return}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ d_1 \multimap \texttt{return}\ v\ d_2 \multimap \{\texttt{eval}\ (e\ v)\ d\}, r_1 \mathbin{\widehat{:}} \texttt{return}\ V_1\ d_1, r_2 \mathbin{\widehat{:}} \texttt{return}\ V_2\ d_2 \vdash R : \{\texttt{return}\ V'\ d'\}$*. Then* V_1 *is equal to* `lam` E *for some* E *and there exists a term* $R' \equiv R$*, such that* $R' = \{\mathsf{let}\ \{p\} = f\ E\ V_2\ \widehat{}\ r_1\ \widehat{}\ r_2\ \mathsf{in}\ R''\}$.

If let-floating has to be reasoned about explicitly in CLF then we could probably just as well have represented the concurrent features explicitly as it would be done in Twelf. To get actual benefit from CLF it therefore seems likely that we would have to come up with a let-floating aware coverage-checker, such that the let-floating would be handled behind the scenes, much like substitution is handled behind the scenes in Twelf. More specifically, in a trace where A and B can occur in either order, we want to be able to implicitly assume that for instance A occurred first.

6 CLF signatures

All the proofs so far are working with a fixed signature and of course cannot be expected to work with arbitrary extensions to the signature. Extending MiniML in any way will naturally require extensions to the proofs as well. This is all good and reasonable. If we however extend the signature with something completely different i.e. new types and type families, we would expect the proofs to work without any changes. So far these are just the natural expectations coming from the way Twelf works.

In Twelf we know this is how things work, since execution is goal-oriented and adding a new type family does not add any new constructors to the old type families. In CLF execution works differently. When inside the monad, the execution semantics will simply nondeterministically perform any action possible given the current signature and context. And since the proofs at some point have to conclude that there can be no more computation, any signature allowing monadic objects — and thereby computation steps — to be constructed directly will disrupt the proofs.

Therefore I propose a simple restriction on CLF signatures which will hopefully simplify meta-theory representations a bit. Consider the number of terms N of type $\cdot;\cdot \vdash N : \{1\}$ in some signature. Of course we can have $N = \{1\}$. But if there are any other terms $N : \{1\}$ then any computation trace constructing any monadic type can have interjections of completely irrelevant, superfluous steps. The proposed restriction is therefore that there can be only one term N of type $\{1\}$ in the empty context. Adding stuff like

```
junk : type.
junk_intro : junk.
junk_elim : junk ⊸ {1}.
```

would therefore be considered an illegal signature.

A conservative approximation of this restriction which is easy to compute, is to simply start the proof search semantics looking for a term of type $\{1\}$. The first step after entering the monad is a nondeterministic choice depending on the signature. Now if the only option for this nondeterministic choice is to terminate the forward-directed mode and construct 1 directly then we are certain that the signature is legal, otherwise we reject the signature.

7 Conclusion and future work

We have proven a traditional big-step semantics equivalent to a destination-passing semantics by induction on terms with an equivalence relation capable of modelling concurrency. Examination of this proof has identified several problems regarding meta-theory representations in CLF.

First, there is the problem of scoping; during the course of a computation in the monad, every intuitionistically introduced term stays in the context. This means that subcomputations cannot easily be split, since the different parts are typed in increasingly larger contexts. One solution could perhaps be to represent proofs in a forward-directed manner in the monad, since this would allow $\exists$-introductions of variables instead of Π-introductions. In the case of the destinations they did not actually occur; if this is the common case, another solution might be to infer this by an automated analysis.

Second, there is the problem of linear contexts; this could though perhaps be solved at the CLF meta-level with some sort of extended world-declaration stating which terms should be linear in which arguments. Alternatively, the work on hybrid metalogical frameworks [7] might be applicable.

Third, there is the problem regarding coverage in the context of let-floating. There is a lot to be gained if a coverage checker could be devised in such a way that the overhead of let-floating-lemmas described in section 5 could be moved to the correctness proof of the coverage checker.

Furthermore it has been argued that restricting the CLF signatures in some way is necessary for a CLF implementation. Specifically it seems like a good idea to require that the type $\{1\}$ is only inhabited by a single term.

Acknowledgement

I thank Andrzej Filinski and my advisor Carsten Schürmann for helpful discussions.

References

[1] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. “A concurrent logical framework I: Judgments and properties”. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.

[2] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. “A concurrent logical framework II: Examples and applications”. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002.

[3] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. 2005. “Monadic concurrent linear logic programming”. In Proceedings of the 7th ACM SIGPLAN international Conference on Principles and

Practice of Declarative Programming (Lisbon, Portugal, July 11–13, 2005). PPDP '05. ACM Press, New York, NY, 35–46.

[4] Andrew McCreight and Carsten Schürmann. "A Meta-Linear Logical Framework". Proceedings of Logical Frameworks and Meta Languages, July 2004.

[5] Frank Pfenning and Carsten Schürmann. "System description: Twelf — a meta-logical framework for deductive systems". In Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Trento, Italy, June 1999. H. Ganzinger, Ed. Lecture Notes In Computer Science, vol. 1632. Springer-Verlag, London, 202-206.

[6] Frank Pfenning. "Substructural operational semantics and linear destination-passing style". In W.-N. Chin, editor, Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04), page 196, Taipei, Taiwan, Nov. 2004. Springer-Verlag LNCS 3302.

[7] Jason Reed. "A Hybrid Metalogical Framework". Thesis Proposal Working Draft. Jan. 2007. http://www.cs.cmu.edu/∼jcreed/papers/thesprop.pdf

Higher-Order Proof Construction Based on First-Order Narrowing

Fredrik Lindblad[1]

Dept. of Computer Science and Engineering
Chalmers University of Technology / Göteborg University
Gothenburg, Sweden

Abstract

We present the idea of using a proof checking algorithm for the purpose of automated proof construction. This is achieved by applying narrowing search on a proof checker expressed in a functional programming language. We focus on higher-order formalisms, such as logical frameworks, whereas the narrowing techniques we employ are first-order. An obvious advantage of this approach is that a single representation of the semantics can in principle be used for both proof checking and proof construction. The correctness of the search algorithm is consequently more or less trivially provided. The question is whether this representation of the search procedure allows a performance plausible for practical use. In order to achieve this, we add some features to the general narrowing search. We also present some small modifications which can be applied on a proof checker and which further improve the performance. We claim that the resulting proof search procedure is efficient enough for application in an interactive environment, where automation is used mostly on small subproofs.

Keywords: Higher-Order Proof Construction, Narrowing, Logical Frameworks, Type Theory

1 Introduction

Narrowing is the study of efficiently evaluating declarative programs in the presence of unknown data and non-deterministic functions. A narrowing strategy which is complete for inductively sequential term rewrite systems[2] can be used to turn a decidable predicate expressed in a standard functional programming language into a search procedure for its members. Given e.g. a predicate deciding whether a list of natural numbers is sorted,

$$\texttt{sorted} : [\texttt{Nat}] \rightarrow \texttt{Bool},$$

narrowing can be used to construct sorted lists of numbers.

We propose to analogously apply this idea on a proof checking algorithm in order to get a proof search algorithm for free, so as to speak. Given a proof checking algorithm relating propositions and proofs,

$$\texttt{proofCheck} : \texttt{Prop} \rightarrow \texttt{Proof} \rightarrow \texttt{Bool},$$

[1] Email: fredrik.lindblad@cs.chalmers.se

we can fix the proposition and apply narrowing search for an unknown proof. The general search procedure will then try to construct proofs of the given proposition.

One approach to develop a proof construction tool for a given higher-order formalism is to add *meta variables* (or *logical variables*) to the abstract syntax of the proof language. These are used as place holders for unknown data. When searching for a proof, a single meta variable is created at the start, indicating that the entire proof term is initially unknown. The meta variable is then instantiated step-by-step, adding new meta variables representing unknown sub-terms. While instantiating, one keeps track of the semantics of the formalism, back-tracking whenever the current partially instantiated proof turns out to be incorrect.

This approach entails the need of making parts of the functionality, which is essentially shared by the proof checker and proof search, aware of meta variables. E.g. the evaluation of terms has to be able to return a partially evaluated term if it encounters an uninstantiated meta variable. Likewise, the term comparison needs to be able to answer "maybe equal". To give a third example, substitutions must be postponed when encountering a meta variable. Otherwise the instantiation of meta variables and evaluation of terms do not commute. This requires some extra book-keeping, such as introducing *explicit substitutions* to the proof term syntax. Apart from this the proof search algorithm is in a sense a dual representation of the semantics, which can cause inconsistency problems between the checking and search algorithms. Also, the search algorithm is typically larger and more intricate than the checking algorithm. One issue that complicates the implementation of a search algorithm is the need of a refined mechanism for deciding the order in which the sub-terms of a proof are instantiated. In our experience, one ends up craving for a way to control the execution and search branching on the meta level, in order to deal with the fact that a meta variable can be encountered in a large number of different places in the algorithm. Controlling the execution and search branching on the meta level is exactly what narrowing does.

There would be several advantages of the proposed way of attaining proof search from a proof checker. Instead of implementing an often intricate and tedious proof search algorithm, the proof checker, which presumably already exists, is reused. Apart from saving work, this entails that there is little potential for inconsistency. In other words, given that the narrowing search procedure is sound and complete, the same properties are inherited by the proof search. Thus the correctness of the search is in principal directly provided. Another advantage would be that meta variables are handled by the narrowing algorithm. Hence, there is no need to add meta variables to the term syntax, and functions like evaluation and comparison do not need to be aware of them. Also, since the narrowing is first-order, the search algorithm is pretty simple. In this work narrowing is used to construct higher-order proofs. To make this possible the terms are represented in a first-order abstract syntax.

The potential drawback of the approach which could overshadow all these benefits is of course that the resulting search procedure, although working in theory, is not efficient enough for practical use. We have investigated this by implementing a narrowing search algorithm and a proof checker. Our aim has been to achieve a proof construction tool which automates the construction of proofs which are rather

small. It is supposed to be useful in an interactive proof construction environment as an aid for filling in not too complex sub-terms in a proof. It is not meant to compete with advanced algorithms for constructing higher-order proof terms. Instead, the intended point of the approach is to add automation to a formal system for higher-order logic at a low cost.

The formalism we have chosen is a logical framework with dependent types, and recursive data-types and function definitions. Hence the proof checker is in fact a type checker, and will we from now on use the terminology of the Curry-Howard correspondence, i.e. refer to proofs as terms and propositions as types. We have implemented a type checker for the formalism in Haskell. Applying the narrowing search on this type checker indeed does not at first give a proof construction tool which could be used in practise. However, our experiments indicate that, by adding a couple of general features to the narrowing search, as well as introducing some small modifications to the type checker, the performance is substantially improved. We will present the most crucial (in our experience) of these general features and modifications. A central idea of the work is that the same code should in principle be able to serve as a description for both checking and searching. Hence, the modifications introduced to the type checking algorithm should preserve its meaning as a Haskell program.

2 Related Work

Our underlying search procedure is based on narrowing. A survey of various narrowing strategies is found in [2]. A notion of parallel evaluation was presented by Antoy et. al. in [3]. We have used a slightly different notion of parallel evaluation, which is described in [10]. Higher-order term construction using first-order narrowing was investigated by Antoy and Tolmach[5]. Our work is related to this in the sense that we have also looked at using first-order narrowing to construct higher-order terms. The difference is that Antoy and Tolmach focused on constructing terms in the declarative language itself, whereas we encode a new language in a first-order data-type and search for objects of that data-type.

Algorithms for term construction in type theory has been studied by e.g. Dowek[8] and Strecker[14]. Strecker's work is far-reaching, but does not cover systems which have defined recursive constants.

Finding efficient strategies for automatically constructing proof terms is an extensively studied area. The concept of *uniform proofs* largely reduces redundancy in proof search and is implemented in e.g. the formal system Twelf [11,12]. *Focused derivations* is a development of this which further improves performance by detecting chains of construction steps for which the search can proceed deterministically [1]. *Tabling* is another technique for narrowing down search space [13]. It is based on memoizing subproblems in order to avoid searching for the same proof more than once. Our approach cannot compete with these advanced proof construction strategies. Nonetheless we do add some restrictions to the type checker in section 5 which remove certain kinds of redundancy. More refined restrictions could probably be introduced, and a tabling mechanism could possibly be added to the general search procedure. However, such advanced features are outside the scope of this

investigation.

In the area of getting term construction "for free" a couple of contributions should be mentioned. Augustsson's tool Djinn[6] converts a Haskell type to a first-order proposition and sends it off to a theorem prover. The result is then interpreted as a λ-term. Tammet and Smith have presented a set of optimized encodings for a fragment of type theory into first-order logic[15]. Here, too, an external theorem prover is used. The fragment includes inductive proofs, but much information is lost in the translation, so even rather trivial problems are hard for the tool to find. Related is also Cheney's experiment on using logic programming to generate terms for the purpose of testing a type checker[7].

3 Basic Approach

The approach of the work is to apply narrowing on a rewrite system which is possible to use as a type checker by executing it as a Haskell program. This means the program should not contain any non-deterministic functions. Hence, the narrowing search does to begin with not need to handle more general systems than inductively sequential term rewrite systems (TRSs). However, since we will introduce the notion of parallel conjunctions to improve performance, we will consider the slightly larger class of *weakly orthogonal* TRSs[2]. These include programs where function definitions may overlap, but only in such a way that overlapping definitions yield the same result. Instead of using an already existing implementation of functional logic programming, we decided to implement our own narrowing algorithm for weakly orthogonal TRSs. The reason for this was to be able to experiment with a couple of features which are not supported by existing implementations. These features are presented in the section 4. Our implementation is based on a variation of lazy needed narrowing strategy [4], which is the most common strategy and is the basis of e.g. Curry [9]. The term syntax of a logical framework are essentially recursively defined. When applying narrowing for an unknown term, its instantiation can in general continue indefinitely. In order to deal with this, our narrowing search is based on measuring the size of the generated term and exploring the potentially infinite search space by iterated deepening. Our implementation reads the *ghc-core* format, which does not contain pattern matching, only case expressions. Thus the narrowing search need not include a stage which constructs the definitional trees of a rewrite system. This is taken care of by the ghc compiler. The blocking position of an expression is decided by traversing the trees of case expressions which define the functions.

3.1 A simple type checker

Next we will present the core parts of a type checker for a logical framework with dependent types. It will be referred to in this section and later on in conjunction with a few simple examples in order to make clear the basic mechanisms of the approach. The type checker is presented in figure 1. Types and terms are represented by `Type` and `Term` respectively. A term is either an application of a variable on a list of arguments, a λ-abstraction or a dependent function arrow. Later on we will also use a concrete syntax for the terms whenever the thereby obscured details are

```
data Type = El Term
          | Set

data Term = App Var [Term]
          | Lam Var Term
          | Pi Var Type Type

tc :: Ctx -> Type -> Term -> Bool
tc ctx etp trm = case trm of
 App v as -> case tiv ctx v of
  Just ftp -> case tis ctx ftp as of
   Just itp -> eqtp ctx etp itp
   Nothing -> False
  Nothing -> False
 Lam v b -> case hntp ctx etp of
  El (Pi v' itp otp) ->
   tc (extctx v itp ctx)
    (subtp v' (App v []) otp) b
  _ -> False
 Pi v itp otp -> case etp of
  Set -> typ ctx itp &&
         typ (extctx v itp ctx) otp
  _ -> False

tiv :: Ctx -> Var -> Maybe Type

tis :: Ctx -> Type -> [Term] -> Maybe Type
tis ctx tp as = case as of
 [] -> Just tp
 (a:as') -> case hntp ctx tp of
  El (Pi v itp otp) -> if tc ctx itp a then
    tis ctx (subtp v a otp) as'
   else
    Nothing
  otherwise -> Nothing

typ :: Ctx -> Type -> Bool
typ ctx tp = case tp of
 El trm -> tc ctx Set trm
 Set -> True

hntp :: Ctx -> Type -> Type
hntp ctx tp = case tp of
 El trm -> El (hn ctx trm)
 Set -> Set

hn :: Ctx -> Term -> Term
subtp :: Var -> Term -> Type -> Type
eqtp :: Ctx -> Type -> Type -> Bool
empctx :: Ctx
extctx :: Var -> Type -> Ctx -> Ctx
```

Fig. 1. Fragment of a type checker for a logical framework with dependent types

not important. We will write $(x\ t\ \ldots\ t)$ and $\lambda x \to t$ for applications and abstractions respectively. Functions arrows will in general be denoted by $(x : t) \to t$, while non-dependent functions will be written $t \to t$. The `El` construction will be omitted in the concrete syntax, and terms and types will not be explicitly distinguished. The representation of variables, `Var`, and contexts, `Ctx`, are left abstract. A context is assumed to contain type declarations of local variables and global constants, as well as the reduction rules for defined global constants. The function `tc` decides the correctness of a term with respect to a given context and type. When checking applications `tis` is used to traverse the list of arguments. The correctness of a type is decided by `typ` and `hntp` reduces a type to head normal form in case it is a term. For the remaining functions only the type signature is given. The type of a variable is looked up in the context by `tiv`. The function `hn` reduces a term to head normal form, and `subtp` substitutes a variable for a term in a type. The definitional equality between two types is decided by `eqtp`. Finally, `empctx` represents the empty context and `extctx` extends a context with a new local variable type declaration.

3.2 Example of a proof search

In order to illustrate the basic mechanism of applying narrowing search on the presented type checker let us look at the proposition $A \wedge (A \to B) \to B$. This can be encoded as the type $(A : \texttt{Set}) \to (B : \texttt{Set}) \to A \to (A \to B) \to B$, or in our abstract syntax:

$$\begin{aligned}\texttt{goalt} \equiv\ & \texttt{El}\ (\texttt{Pi}\ A\ \texttt{Set}\ (\texttt{El}\ (\texttt{Pi}\ B\ \texttt{Set}\ (\texttt{El}\ (\texttt{Pi}\ x\ (\texttt{El}\ (\texttt{App}\ A\ []))\\ & (\texttt{El}\ (\texttt{Pi}\ y\ (\texttt{El}\ (\texttt{Pi}\ z\ (\texttt{El}\ (\texttt{App}\ A\ []))\ (\texttt{El}\ (\texttt{App}\ B\ [])))))\\ & (\texttt{El}\ (\texttt{App}\ B\ [])))))))))\end{aligned}$$

In order to find a term of this type using the proposed approach we should apply narrowing on the expression

$$\texttt{tc}\ \texttt{empctx}\ \texttt{goalt}\ ?_1,$$

where $?_1$ is a meta variable serving as a placeholder for the yet unknown term.

With a lazy narrowing strategy the expression for which the search is performed is evaluated in a lazy evaluation order, i.e. from the outside and in. Whenever an uninstantiated meta variable is encountered it is said to be in the *blocking position*. The meta variable in the blocking position is chosen for refinement. It is non-deterministically refined to one of the constructors in its type. For each constructor, fresh meta variables are inserted at the argument positions. Then the evaluation of the expression proceeds until a new blocking meta variable is encountered or a value is reached. If the value is `True` a solution has been found. If it is `False` a dead-end has been reached and the search back-tracks.

Proceeding with the example, we start evaluating the given expression lazily. Since `tc` does a case distinction on the third argument, we need to know the head constructor of the term. But the term is $?_1$, which thus blocks further evaluation. There are three possible refinements of $?_1$. The refinement $?_1 := \texttt{App}\ ?_2\ ?_3$ yields an expression with (`tiv empctx` $?_2$) surrounding the potentially blocking position. Assuming that `tiv` returns `Nothing` for the empty context regardless of its second argument, the evaluation proceeds without further refinement with the result being `False`, which means no solution. The refinement $?_1 := \texttt{Pi}\ ?_2\ ?_3\ ?_4$ immediately yields `False` since the type is not `Set`. Finally, the refinement $?_1 := \texttt{Lam}\ ?_2\ ?_3$ results in the expression

$$\texttt{tc}\ (\texttt{extctx}\ ?_2\ \texttt{Set}\ \texttt{empctx})\ (\texttt{subtp}\ A\ (\texttt{App}\ ?_2\ [])\ (\texttt{Pi}\ B\ \ldots))\ ?_3,$$

where $?_3$ is the blocking meta variable.

The search proceeds similarly until all four λ-abstractions have been constructed. The difference is that the context is no longer empty, which means that `App` is not immediately rejected. However, before all λ-abstractions are introduced, the refinement to `App` will eventually fail and the search will back-track. At the resulting state, the term under construction has been instantiated to

$$\texttt{Lam}\ ?_2\ (\texttt{Lam}\ ?_4\ (\texttt{Lam}\ ?_6\ (\texttt{Lam}\ ?_8\ ?_9)))$$

and the expression is essentially

$$\texttt{tc}\ \ldots\ (\texttt{El}\ (\texttt{App}\ B\ []))\ ?_9.$$

Consider the refinement $?_9 := \texttt{App}\ ?_{10}\ ?_{11}$. Then $?_{10}$ is blocking the evaluation. If $?_{10}$ is set to be equal to $?_8$, `tiv` should return

$$\texttt{Just}\ (\texttt{El}\ (\texttt{Pi}\ z\ (\texttt{El}\ (\texttt{App}\ A\ []))\ (\texttt{El}\ (\texttt{App}\ B\ [])))).$$

The values of $?_8$ and $?_{10}$ are not important, as long as they are equal. Let us choose them to be y. Now $?_{11}$ becomes the blocking meta variable. The refinement $?_{11} := []$ results in the comparison between the expected type, B, and the type inferred by `tiv`, namely $A \to B$. These are not equal and hence the expression evaluates to `False`. The refinement $?_{11} := ?_{12} : ?_{13}$ makes $?_{12}$ block. Refining $?_{12}$ to $\texttt{App}\ ?_{14}\ ?_{15}$, setting $?_{14}$ to be equal to $?_6$ and refining $?_{15}$ to [] results in comparing the expected and inferred type for the inner application (which has no arguments). The types are both A, so the search proceeds. The common value of $?_6$ and $?_{14}$ is also arbitrary, as long as it is different from y. Let us choose it to be x. Looking at the definition of `tis` the blocking meta variable is now $?_{13}$. After refining $?_{13}$ to [], the expected and inferred type for the outer application is compared. They are both B, so the

expression finally evaluates to `True`, indicating that we have a solution. The term is composed by all current refinements which expands to

$$\texttt{Lam}\ ?_2\ (\texttt{Lam}\ ?_4\ (\texttt{Lam}\ x\ (\texttt{Lam}\ y\ (\texttt{App}\ y\ ((\texttt{App}\ x\ []) : [])))))).$$

This term represents a proof for the given problem.

3.3 Comments on the suitability of using narrowing

The basic idea of narrowing search is that, by interleaving instantiation and evaluation, and choosing the order of instantiation in a clever way, the data in many cases do not need to be fully instantiated before the predicate is known to return `False`. Every time this happens, all the data instances that are specializations of the current partial instantiation can be skipped. Thereby the search space is reduced. Predicates which are suitable for narrowing are typically to large extent defined by recursion on the structure of the unknown data. Looking at the example above, type checking does seem to be of this kind.

However, the example could very well be formalized in a system without dependent types. In a logical framework with dependent types, argument types and the output type may depend on a previous argument value in applications. This leads to some complications which were not exposed in the example. In the following two sections these complications will be discussed along with suggestions for how to deal with them.

One complication which did appear in the example was the treatment of variables. When constructing variable occurrences, the description of the search was rather irregular, stating that two variables should be the same rather than instantiating a single meta variable. Also, the variables which were never used were left uninstantiated. These problems are however easy to avoid. When representing variable occurrences by de Bruijn indices, no arbitrary choices need to be made, and no uninstantiated variables at binding position will appear. Furthermore, if recursively defined numbers are chosen to represent the indices, the narrowing algorithm will itself limit the search to the set of possible indices for a given context.

4 Features of the General Search Procedure

This section describes the two main non standard features of the general search algorithm we have investigated. As stated above, the search procedure targets weakly orthogonal TRSs. In order to deal this class of rewrite systems efficiently, a concept of *parallel evaluation* has been proposed[3]. In [10] a somewhat different notion of parallel evaluation is presented, which is used in our implementation. Using this feature for the purpose of parallel conjunction is discussed in section 4.1. Section 4.2 introduces the idea of *subproblem separation* which is an attempt to overcome the performance loss which follows from using parallel conjunction in some cases.

4.1 Parallel Conjunction

Let us now look at a simple example which does involve dependent types. Assume that we have the situation

$$?_1 : P\ M,$$

where $P : X \to \texttt{Set}$ is a variable and M is a term of the correct type. Also assume that $h : (x : X) \to P\ x$ is in scope. Constructing a proof by involving h proceeds by the refinement

$$?_1 := h\ ?_2.$$

The type checker presented in the previous section is devised to check the correctness of the arguments in an application from left to right, and at the end check the equality between the expected and the inferred type. In this example that amounts to first checking $?_2 : X$ and then checking $P\ M = P\ ?_2$. This is a natural choice since it means that terms are always type checked before they are used in computations. However, for the purpose of proof search, type checking before equality checking is not desirable, since the latter is in general more restrictive. Type checking the argument, $?_2$, first means that all terms with the correct type are constructed before their equality to M is decided.

To amend this inefficiency there are two rather straightforward ways to go. One is to reverse the order in which the constraints are checked in an application, as discussed in [8]. The other is to check the conditions in parallel. Both these approaches build on the fact that the result of type checking an argument is not needed for type checking the remaining arguments or for the final equality check. However, both of them also introduce the hazard that a term which has not been constructed in a type correct way exposes partiality in the term reduction functions.

In our implementation we chose the second of these approaches with the motivation that checking restrictions in parallel during the narrowing search should in general give a smaller search space than checking them sequentially, regardless of the order. There are also situations where parallel checking is useful, and where the best order is not as clear as in the case of type checking an application. The following example illustrates this:

$$\exists\ X\ (\lambda x \to P\ x \wedge Q\ x)$$

Constructing a proof of this proposition will yield an intermediate state with the constraints

$$?_1 : X, \quad ?_2 : P\ ?_1, \quad ?_3 : Q\ ?_1.$$

The type judgments of $?_2$ and of $?_3$ both put restrictions on $?_1$. By instantiating the meta variables and taking all the type constraints into account in parallel, a more narrow search space can be achieved.

As mentioned, the hazard of giving conjunctions a parallel meaning is that an intentional partiality in the right conjunct is exposed. The implications are different for non-definedness and for non-termination partiality. Non-definedness can be easily handled by treating a undefinedness error in the right conjunct of parallel conjunction as `False`.

Non-termination is more delicate. A term can be non-terminating in two ways. It is either essentially type correct but recursive in a non-terminating way, or it is not type correct and non-terminating, like the Ω-term. In our experiments we have not implemented a termination checker. Instead, we have allowed recursion only via given elimination rules. We have not experienced any problems caused by nontermination. A formal characterization of when and why this is provably safe would be desirable, but we have had to leave this as future work. An informal motivation of why we have not encountered any problems is that nonterminating terms are either not constructible in the syntax or have to be constructed via intermediate steps which are not type correct.

With the needed narrowing strategy, there is always a unique meta variable to branch the search on. However, in the presence of parallel conjunction, or parallel evaluation in general[10], there is no longer a single meta variable blocking the evaluation. The order in which to instantiate meta variables must hence be further specified. A natural choice is to store them in a collection and extract them in either a queue or a stack manner. In our experience the queue is in general, but not always, the better choice regarding performance[10]. The order of instantiating blocking meta variables can also be controlled in more refined ways, which we will come back to in section 5.2.

4.2 Subproblem Separation

Parallel conjunction was introduced in order to check several properties in parallel during the incremental instantiation of a term. This seems to be beneficial in various situations where the properties constrain the same part of the term. However, for problems where there are sub-terms with no dependencies in between, it seems undesirable to interleave the search for their solutions. Consider the situation

$$?_1 : P \wedge Q,$$

where P and Q have no meta variable occurrences. Interleaving the construction of a proof of P and a proof of Q is unnecessary and should lead to a larger search space than if the to subproofs were constructed separately. This is a drawback of switching to parallel conjunction as discussed in sec 4.1. Moreover the situation is very common. It can appear whenever attempting to prove a proposition by case distinction, such as in proofs by induction.

A solution to this drawback could be to add a feature of *subproblem separation* to the general narrowing search. One part of the mechanism would be to detect the presence on independent subproblems, i.e. unconnected graphs where parallel conjuncts and meta variables constitute the vertices and the edges represent meta variable occurrences in the conjuncts. When such a partitioning has been detected, a local search for each subproblem is spawned. When performing the search for several independent subproblems backtracking one of them should not affect the other ones, and when the search space is exhausted for one of them, itself and all of its sibling subproblems are cancelled. We have implemented this feature, but it should be considered a prototype.

A rather artificial example will illustrate the reduction in search space which can be gained using subproblem separation. Let the initial problem be

$$?_1 : P$$

and let the following hypotheses be in scope:

$$g_1 : P_1 \to P_2 \to P$$

$$g_2 : P_3 \to P_4 \to P$$

$$h_{ij} : P_{ij} \to P_i, \quad \text{for } 1 \leq i \leq 4, 1 \leq j \leq k_i$$

There are two alternative ways to prove P, namely by applying either g_1 or g_2 on two arguments. In both cases, the construction of the two arguments are independent of each other. The subproblems are in turn matched by unary applications invoking the hypotheses h_{ij}. A complete proof is not possible to construct in the given context, but that is not important. We will merely measure the size of the search space by counting the number of leaves in the spanned tree. Assuming parallel conjunction is used, the instantiation of the first sub-term will be interleaved with the instantiation of the second. The number of leaves in the search tree thus amounts to $k_1 \cdot k_2 + k_3 \cdot k_4$. Now consider the case where the subproblem separation feature is present. When the term has been instantiated to either $g_1\ ?_2\ ?_3$ or $g_2\ ?_2\ ?_3$, the independence between the construction of $?_2$ and $?_3$ is detected. Assuming that the execution alternates between the two separate subproblems in a fair way, the number of leaves is $2 \cdot min(k_1, k_2) + 2 \cdot min(k_3, k_4)$. Hence the size of the search space is linear in k_i instead of quadratic. Subproblem separation could enable the proof search to scale considerably better.

5 Optimizations of the Type Checker

The features presented in the previous section, parallel conjunction and subproblem separation, do not result in a proof search which is of practical use. In order to improve the performance, we have also experimented with some modifications of the type checker. In this section we will discuss a few such modifications which have proved to be important in our experiments.

5.1 Type Checking Restrictions

Most logical frameworks allow writing essentially the same proof in many different ways. By restricting the type checker to accept fewer terms for a given type, a reduction of the search space can be accomplished. One restriction is to only allow normal terms. It can be easily achieved by adding a side condition checking that no sub-term is reducible. Imposing this restriction does not compromise the completeness of the search. Another possibility could be to add restrictions which allow only uniform proofs.

One can of course also come up with a large number of restrictions, which do limit the completeness. These can be seen as representations of different heuristics. One example is to restrict induction so that no generalizations may take place. This clearly makes the search incomplete, but also contributes a lot to performance.

5.2 Meta Variable Prioritization

As mentioned in section 4.1, the search is based on keeping a queue of blocking meta variables and instantiating them one at a time. By slightly annotating the code of the type checker, one can introduce a notion of priority which refines the scheduling of the meta variable instantiation. Controlling the order of instantiation this way does not affect the completeness, apart from the possibility that an otherwise finite search space could become infinite.

The following simple prioritization has made a great performance improvement in our experiments:

- *high* – equality constraints
- *medium* – type checking constraints for proof terms
- *low* – type checking constraints for non-proof terms and when the type is unknown

By *proof term* we mean sub-terms which correspond to a proof step, i.e. whose inhabitation is not trivial. Non-proof terms are the rest, i.e. terms which are typically trivially inhabited and appear in some equality constraint.

The idea behind this prioritization is that equality constraints are in general more restrictive than type checking constraints. The reason for postponing the type-checking of non-proof terms is that it may be the case that the term does not yet appear in an equality constraint, although it will after further instantiating some proof terms. If the instantiation of the non-proof term is initiated before an equality constraint add further restrictions, the search is quite arbitrary.

In order to be able to prioritize proof and non-proof terms differently, there must be a way to tell them apart. One option is to distinguish between dependent and non-dependent function types. Application arguments which stem from dependent function types are treated as non-proof terms and those from non-dependent function types as proof terms. We have chosen this approach in our implementation.

5.3 Special Treatment of Equality Constraints

Checking equality constraints in a dependently typed system is typically implemented by first reducing the left and right hand sides to head normal form, and then comparing the heads and recursively repeating the procedure for the sub-terms. Assume we apply narrowing on a type checker implemented like this. If one of the compared terms is a meta variable, it is necessary for the search algorithm to guess among all global constants, reduce the term and see if it equals the opposite side. If we have e.g. have the equality constraint

$$?_1 = \texttt{add}\ T\ U,$$

the search must guess $?_1$ to be `add` $?_2$ $?_3$. This is of course a possible solution, but we would like to make better use of the information that is given in the initial type defining the problem. The meta variable on one side should be able to mimic the term on the other side, just like in unification.

The basic approach to achieve this is to compare the terms without first reducing them. Of course, never reducing the terms when comparing for equality is not a real option. That would render the system too weak. A possible option is to mix

reduction with first-order unification, but it is not so clear how this should be done. One way to do this is to check for each term whether it is a meta variable (is currently unknown) and only reduce terms which are not. In order to implement this, a primitive function to test whether a term is a meta variable must be added to the general system.

However, the comparison will not be complete. Assume e.g. we have the global definitions

$$\mathtt{p\ a} \equiv \mathtt{q} \qquad \mathtt{p\ b} \equiv \mathtt{r}$$

for the constant `p`, and the equality constraints

$$?_1[\mathtt{a}/x] = \mathtt{q} \qquad ?_1[\mathtt{b}/x] = \mathtt{r},$$

where the brackets represent postponed variable substitutions. Then the solution, $?_1 := \mathtt{p}\ x$, will not be found using the approach above. However, the definition of `p` is rather artificial. This way of allowing information to migrate from one side to the other in equality constraints seems to be sufficient in most practical cases.

6 Experiments

We have performed some experiments with the presented approach. The implementation consists of a general narrowing search system and a type checker for a logical framework written in Haskell. The narrowing system is constituted by a compiler of Haskell programs and a run-time system implementing the search algorithm. The narrowing algorithm accepts weakly orthogonal TRSs by implementing the parallel evaluation discussed in section 4.1 and presented in more detail in [10]. It also includes a prototype of the subproblem separation feature presented in section 4.2. The type checker is based on the one presented in section 3, but modified to enable parallel conjunction and the performance enhancing features presented in section 5. The implemented logical framework has recursive global function and data definitions.

The examples we have run the resulting proof construction tool on mainly focus on the way instantiation is scheduled when constructing proof terms containing type arguments. Performance-wise the approach cannot compete with more refined proof search methods like focused derivations or higher-order tabling (see section 2). The redundancy quickly becomes overwhelming for propositional problems and problems involving equality reasoning. In the following subsections a couple of examples are presented, illustrating how the presented modifications of the type checker influence the search. In connection with these examples some further remarks on the limitations of the approach are made.

6.1 The scheduling of instantiations

An inductive proof containing some kind of generalization serves well as an illustration of the order of instantiation which is imposed by the prioritization presented in section 5.2. The example is to construct a proof of the proposition stating that a given function, `sort`, always returns a sorted list.

$$?_1 : (xs : \mathtt{List\ Nat}) \rightarrow \mathtt{sorted}\ (\mathtt{sort}\ xs)$$

In order to save space we will omit the definitions of `sorted` and `sort`, but merely state that `sort` implements insertion sort. The reader thus cannot confirm that the proof is correct. Nevertheless, the mechanisms of the proof search should be clear. The function `sort` is defined in terms of `sort′` which, in turn, uses `insert`. The names `List` and `Nat` refer to the standard recursive definitions of lists and natural numbers. The proof search is provided an elimination constant for lists,

$$\texttt{elimList} : (X : \texttt{Set}) \to (xs : \texttt{List}\ X) \to (P : \texttt{List}\ X \to \texttt{Set}) \to$$
$$P\ \texttt{nil} \to ((z : X) \to (zs : \texttt{List}\ X) \to P\ zs \to P\ (\texttt{cons}\ z\ zs)) \to P\ xs,$$

and the lemma

$$\texttt{lem} : (x : \texttt{Nat}) \to (xs : \texttt{List Nat}) \to \texttt{sorted}\ xs \to \texttt{sorted}\ (\texttt{insert}\ x\ xs).$$

The sub-term `sort` xs in the given problem normalizes to `sort′` xs `nil`. A solution to the problem is

$$?_1 := \lambda x \to \texttt{elimList}\ \underbrace{\texttt{Nat}}_{B}\ \underbrace{x}_{A}$$
$$(\lambda y \to (z : \underbrace{\texttt{List Nat}}_{B}) \to \underbrace{\texttt{sorted}\ z}_{E} \to \underbrace{\texttt{sorted}\ (\texttt{sort}'\ y\ z)}_{A})$$
$$\underbrace{(\lambda y \to \lambda z \to z)}_{C}$$
$$\underbrace{(\lambda y \to \lambda z \to \lambda w \to \lambda t \to \lambda u \to w\ (\texttt{insert}\ y\ t)\ (\texttt{lem}\ y\ t\ u))}_{D}$$
$$\underbrace{\texttt{nil}}_{A}\ \underbrace{\texttt{tt}}_{F}$$

where `tt` is the proof of the trivial problem, proving `sorted nil`.

Our implementation constructs the proof term above in the following way. First the λ-abstraction and application of `elimList` with the correct number of arguments are constructed. This also includes constructing the λ-abstraction and the two function arrows in the third argument of the application. A that stage there are a number of type checking constraints and one equality constraint involving the sub-terms marked A. The prioritization presented in section 5.2 makes the instantiation address the equality constraint first. This results in constructing the sub-terms A, followed by the terms marked B. Next, the construction of C and D are interleaved. When the elimination of z has been introduced in C, the resulting equality constraint triggers the construction of the extra hypothesis E. After that the type of F is becomes known, so its construction commences. During the final stage of the search the construction of F and the remaining of D proceeds as two independent subproblems, since they have no uninstantiated meta variables in common.

By employing parallel conjunction combined with the instantiation prioritization, the implementation can find the proof above within a second on a normal desktop computer. Without these features, the construction of this proof term by narrowing search is quite intractable.

6.2 Subproblem separation

To exemplify the improvement gained by introducing subproblem separation, which was discussed in section 4.2, we take the following example:

$$?_1 : (a : \texttt{Nat}) \rightarrow (b : \texttt{Nat}) \rightarrow \texttt{eq}\ a\ b \rightarrow \texttt{eq}\ b\ a$$

It involves a recursively defined equality relation over natural numbers. The solution of the problem includes nested induction, one at the top level and another induction in each of the base and step cases. The problem does not seem very difficult. But in spite of this it is a challenge to our implementation. The example makes it quite clear that more restricted induction is desirable in situations in which no strengthening of the induction hypothesis is required. However, when turning on the subproblem separation feature the base and step cases of the inductions can be solved independently. This makes the search space forty times smaller. The search only becomes a few times faster since our implementation of the feature is rather inefficient. However, we think that the implementation could be considerably improved. We also believe that efficient subproblem separation is essential if one is interested in making the whole approach tractable for more complex problems than those which have been presented here.

7 Conclusions

We have presented the idea of applying first-order narrowing on a type checker for a higher-order formalism in order to achieve proof construction. In order to make the resulting search procedure viable for practical use, we have refined the approach by adding a couple of non-standard features of the narrowing, as well as a number of small and rather general modifications of the type checker. We have made an implementation to get some empirical evidence of the usability of the approach. The preliminary conclusion is that the approach could be useful in situations where low cost is important rather the high performance. The experiments are however too limited to be able to give a more solid conclusion/

For future work we are keen to further investigate the usefulness of the approach by more thouroughly running examples and comparing to other systems. One of the most crucial points in order to improve our system seems to be the subproblem separation feature. We would also like to investigate the addition of more restrictions to the type checker in order to impose heuristics for e.g. equality reasoning. An interesting direction is to look at using the approach for the purpose of generating functions. We have looked into this to some extent, and been able to synthesize e.g. insertion sort. But the main obstacle seems to be that, although we restrict the search to terminating functions, a lot of very inefficient candidates are still generated. This could be amended by adding some notion of function complexity to the type system.

References

[1] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[3] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. Fourteenth International Conference on Logic Programming*, pages 138–152, Leuven, Belgium, July 1997. MIT Press.

[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[5] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722, pages 335–350, Tsukuba, Japan, 11 1999. Springer LNCS.

[6] Lennart Augustsson. Djinn, a theorem prover in haskell, for haskell. `http://www.augustsson.net/Darcs/Djinn/`

[7] James Cheney. `http://homepages.inf.ed.ac.uk/jcheney/publications/wmm06-draft.pdf`

[8] Gilles Dowek. A complete proof synthesis method for the cube of type systems. *J. Logic and Computation*, 3(3):287–315, 1993.

[9] M. Hanus and P. Réty. Demand-driven search in functional logic programs. Research report rr-lifo-98-08, Univ. Orléans, 1998.

[10] Fredrik Lindblad. Property directed generation of first-order test data. Presented at TFP 2007 and submitted for publication.

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

[12] Frank Pfenning and Carsten 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)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[13] Brigitte Pientka. Tabling for higher-order logic programming. In *CADE*, pages 54–68, 2005.

[14] Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Fakultät für Informatik, Universität Ulm, 1999.

[15] Tannel Tammet and Jan Smith. Optimized encodings of fragments of type theory in first-order logic. *Journal of Logic and Computation*, 8, 1998.

The λ-context calculus

Murdoch J. Gabbay

Computer Science Department, Heriot-Watt University, Scotland

Stéphane Lengrand

School of Computer Science, University of St Andrews, Scotland

Abstract

We present a simple but expressive lambda-calculus whose syntax is populated by variables which behave like meta-variables. It can express both capture-avoiding and capturing substitution (instantiation). To do this requires several innovations, including a key insight in the confluence proof and a set of reduction rules which manages the complexity of a calculus of contexts over the 'vanilla' lambda-calculus in a very simple and modular way. This calculus remains extremely close in look and feel to a standard lambda-calculus with explicit substitutions, and good properties of the lambda-calculus are preserved.

Keywords: Lambda-calculus, contexts, meta-variables, capture-avoiding substitution, capturing substitution, instantiation, confluence, nominal techniques, calculus of explicit substitutions.

1 Introduction

This is a paper about a λ-calculus for contexts. A **context** is a term with a 'hole'. The canonical example is probably $C[\text{-}] = \lambda x.\text{-}$ in the λ-calculus. This is not λ-calculus syntax because it has a hole -, but if we fill that hole with a term t then we obtain something, we usually write it $C[t]$, which *is* a λ-calculus term.

For example if $C[\text{-}] = \lambda x.\text{-}$ then $C[x] = \lambda x.x$ and $C[y] = \lambda x.y$. This cannot be modelled by a combination of λ-abstraction and application, because β-reduction avoids capture. Formally: there is no λ-term f such that $ft = C[t]$. The term $\lambda z.\lambda x.z$ is the obvious candidate, but $(\lambda z.\lambda x.z)x =_\alpha \lambda x'.x$. (Here $=_\alpha$ is α-equality.)

Contexts arise often in proofs of meta-properties in functional programming. They have been substantially investigated in papers by Pitts on contextual equivalence between terms in λ-calculi (with global state) [18,20]. This work was about proving programs equivalent in all contexts — **contextual equivalence**. The idea is that two programs, represented by possibly-open λ-terms, are equivalent when one can be exchanged for another in code (without changing whichever notion of observation we prefer to use).

This suggests that we should call holes *context variables* X (say they have 'level 2') distinct from 'normal' variables x (say they have 'level 1') and allow λ-abstraction

over them to obtain a λ-calculus of *contexts*, so that we can study program contexts with the full panoply of vocabulary, and hopefully with many of the theorems, of the λ-calculus. For example $\lambda x.\text{-}$ may be represented by $\lambda X.\lambda x.X$. Substitution for X does not avoid capture with respect to 'ordinary' λ-abstraction, so $(\lambda X.\lambda x.X)x$ reduces to $\lambda x.x$.

The Lambda Context Calculus internalises context variables (as variables of 'level 2', which we write X, Y, Z). X, Y, and Z are now variables which can occur any number of times anywhere in a term — *and* they can be λ-abstracted. The Lambda Context Calculus therefore goes further and internalises another level of contexts (variables of 'level 3', which we write $\mathcal{W}, \mathcal{W}'$) — and so on. There are several possibilities where such a calculus might be applied.

Consider formalising mathematics in a logical framework based on Higher-Order Logic (**HOL**) [28]. Typically we have a goal and some assumptions and we want a derivation of one from the other. This derivation may be represented by a λ-term (the *Curry-Howard correspondence*). But the derivation is arrived at by stages in which it is *in*complete.

To the right are two derivations of $A{\Rightarrow}B{\Rightarrow}C, A{\Rightarrow}B \vdash A{\Rightarrow}C$. The bottom one is complete, the top one is incomplete. [1] An issue arises because the right-most $[A]^i$ in the bottom derivation is discharged, which means that we have to be able to instantiate ? in a sub-derivation for an assumption which will be discharged. Discharge corresponds in the Curry-Howard correspondence precisely to λ-abstraction, and this instantiation corresponds to capturing substitution. Similar issues arise with existential variables [10, Section 2, Example 3].

$$\dfrac{\dfrac{A{\Rightarrow}B{\Rightarrow}C \quad [A]^i}{B{\Rightarrow}C} \qquad \dfrac{?}{B}}{\dfrac{C}{A{\Rightarrow}C}\,{}^i}$$

$$\dfrac{\dfrac{A{\Rightarrow}B{\Rightarrow}C \quad [A]^i}{B{\Rightarrow}C} \qquad \dfrac{A{\Rightarrow}B \quad [A]^i}{B}}{\dfrac{C}{A{\Rightarrow}C}\,{}^i}$$

The central issue for any calculus of contexts is the interaction of context variables with α-equivalence. Let x, y, z be 'ordinary' variables and let X be a context variable. If $\lambda x.X =_\alpha \lambda y.X$ then $(\lambda X.\lambda x.X)x =_\alpha (\lambda X.\lambda y.X)x \rightsquigarrow \lambda y.x$, giving non-confluent reductions. Dropping α-equivalence entirely is too drastic; we need $\lambda y.\lambda x.y$ to be α-convertible with $\lambda z.\lambda x.z$ to reduce a term like $(\lambda y.\lambda x.y)x$.

Solutions include clever control of substitution and evaluation order [23], types to prevent 'bad' α-conversions [21,11,22], explicit labels on meta-variables [10,13], and more [4, Section 2]. More on this in the Conclusions.

We took our technical ideas for handling α-equivalence, not from the literature on context calculi cited above, but from *nominal unification* [27]. This was designed to manage α-equivalence in the presence of holes, in unification — 'unification of contexts of syntax', in other words. Crudely put, we obtained the λ-context calculus (**LCC**) by allowing λ-abstraction over the holes and adding β-reduction.

This work has similar goals to previous work by the first author [6] which presented a calculus called NEWcc. The LCC possesses a significatnly more elementary set of reduction rules; notably, we dispense entirely with the freshness contexts and freshness logic of the NEWcc. Indeed, the LCC has only one single non-obvious side-condition, it is on $(\sigma\mathbf{p})$ in Figure 5.

[1] This example 'borrowed' from [10].

The result is a system with a powerful hierarchy of context variables and which still manages to be clean and, we hope, easy to use and to study.

In Section 2 we present the syntax and reductions of the LCC. The look-and-feel is of a λ-calculus with explicit substitutions, except that each variable has a 'level' which determines how 'strongly' binders by that variable resist capture. We give example reductions and discuss the technical issues which motivated our design. In Section 3 we discuss the λ-free part of the language, prove strong normalisation, and give an algorithm for calculating normal forms. In the usual λ-calculus this normal form is calculated in big-step style and written $s[a{\mapsto}t]$; as is standard for a calculus of explicit substitutions, here this part of evaluation is dissected in detail. In Section 4 we treat confluence, first of the λ-free part of the language, then of the full reduction system. The proof may look elementary but it is not, and we give enough technical detail to show how all the side-conditions interact to ensure confluence. It is not sufficient to give a λ-calculus without *binding*, but the hierarchy of levels means that λ itself is no longer necessarily a binder. We address that issue with a new M in Section 5. We conclude with brief discussions of programming in Section 6, and then discuss related and future work.

2 Syntax and reductions

2.1 Syntax

We suppose a countably infinite set of disjoint infinite **sets of variables** $\mathbb{A}_1$, $\mathbb{A}_2$, ... i, j, k range over levels; we usually maintain a convention that $k \leq i < j$, where we break it we clearly say so. We always use a **permutative convention** that $a_i, b_j, c_k, \ldots$ range *permutatively* over variables of level i; so a_i, b_j, and c_k are always distinct variables. There is no particular connection between a_1 and a_2; we have just given them similar names.

Definition 2.1 *LCC syntax is given by* $s, t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i{\mapsto}t]$.

Application associates to the left, e.g. $tt't''$ is $(tt')t''$. We say that a_i **has level** i. We call b_j **stronger** than a_i, and a_i **weaker** than b_j, when $j > i$. If $i = j$ we say that b_j and a_i have the same strength. We call $s[a_i{\mapsto}t]$ an **explicit substitution** (of level i). We call $\lambda a_i.t$ an **abstraction** (of level i).

By convention $x, y, z, X, Y, Z, \mathcal{W}$ are distinct variables; x, y, z have level 1, X, Y, Z have level 2, and $\mathcal{W}$ has level 3. Note that levels are $1, 2, 3, \ldots$ but our proofs would work as well for levels being integers, reals, or any totally ordered set.

The stronger a variable, the more 'meta' its behaviour. The intuition of $\lambda x.X$ is of the context $\lambda x.$- where - is a hole; this is because, as we shall see, substitution for the relatively strong X does not avoid capture by the relatively weak λx. Strong variables can be abstracted as usual; the intuition of $\lambda X.X$ is of the 'normal' identity function; the intuition of $\lambda X.\lambda x.X$ is of the mapping 't maps to $\lambda x.t$'.

Our syntax has no constant symbols though we shall be lax and use them where convenient, for example $1, 2, 3, \ldots$. This can be accommodated by extending syntax, or by declaring them to be variables of a new level $0 < 1$ which we do not abstract over or substitute for.

$$
\begin{array}{rl@{\qquad}rl}
\mathsf{level}(a_i) & = i & \mathsf{fv}(a_i) & = \{a_i\} \\
\mathsf{level}(ss') & = \mathsf{max}(\mathsf{level}(s), \mathsf{level}(s')) & \mathsf{fv}(\lambda a_i.s) & = \mathsf{fv}(s) \backslash \{a_i\} \\
\mathsf{level}(\lambda a_i.s) & = \mathsf{max}(i, \mathsf{level}(s)) & \mathsf{fv}(s[a_i \mapsto t]) & = (\mathsf{fv}(s) \backslash \{a_i\}) \cup \mathsf{fv}(t) \\
\mathsf{level}(s[a_i \mapsto t]) & = \mathsf{max}(i, \mathsf{level}(s), \mathsf{level}(t)) & \mathsf{fv}(st) & = \mathsf{fv}(s) \cup \mathsf{fv}(t)
\end{array}
$$

Fig. 1. Levels $\mathsf{level}(s)$ and free variables $\mathsf{fv}(s)$

$$
\frac{}{a_i R a_i} \qquad \frac{sRs' \;\; tRt'}{st \;\; R \;\; s't'} \qquad \frac{sRs' \quad tRt'}{s[a_i \mapsto s'] \;\; R \;\; t[a_i \mapsto t']} \qquad \frac{sRs'}{\lambda a_i.s \;\; R \;\; \lambda a_i.s'} \qquad \frac{sRs'}{s'Rs} \qquad \frac{sRs' \;\; s'Rs''}{sRs''}
$$

Fig. 2. Rules for a congruence

$$
\begin{array}{rll}
(a_i \;\; b_i)a_i & = b_i & \\
(a_i \;\; b_i)b_i & = a_i & \\
(a_i \;\; b_i)c & = c & (c \text{ any atom other than } a_i \text{ or } b_i) \\
(a_i \;\; b_i)(ss') & = ((a_i \;\; b_i)s)((a_i \;\; b_i)s') & \\
(a_i \;\; b_i)(\lambda c.s) & = \lambda(a_i \;\; b_i)c.(a_i \;\; b_i)s & (c \text{ any atom}) \\
(a_i \;\; b_i)(s[c \mapsto t]) & = ((a_i \;\; b_i)s)[(a_i \;\; b_i)c \mapsto (a_i \;\; b_i)t] & (c \text{ any atom})
\end{array}
$$

Fig. 3. Rules for swapping

$$
\begin{array}{ll}
\lambda a_i.s =_\alpha \lambda b_i.(b_i \;\; a_i)s & \text{if } \; b_i \# \mathsf{fv}(s) \\
s[a_i \mapsto t] =_\alpha ((b_i \;\; a_i)s)[b_i \mapsto t] & \text{if } \; b_i \# \mathsf{fv}(s)
\end{array}
$$

Fig. 4. Rules for α-equivalence

Definition 2.2 *Define the* **level** $\mathsf{level}(s)$ *and the* **free variables** $\mathsf{fv}(s)$ *by the rules in Figure 1.*

Here $\mathsf{max}(i, j)$ is the greater of i and j, and $\mathsf{max}(i, j, k)$ is the greatest of i, j, and k. Later we shall write '$\mathsf{level}(s_1, \ldots, s_n) \leq i$' as shorthand for '$\mathsf{level}(s_1) \leq i$ and ... and $\mathsf{level}(s_n) \leq i$', similarly for '$\mathsf{level}(s_1, \ldots, s_n) < i$'.

Lemma 2.3 *If* $\mathsf{level}(s) = 1$ *then* $\mathsf{fv}(s)$ *coincides with the usual notion of 'free variables of' for the* λ*-calculus, if we read* $s[a_1 \mapsto t]$ *as* $(\lambda a_1.s)t$.

We shall see that the operational behaviour of such terms is the same as well.

A **congruence** is a binary relation $s \; R \; s'$ satisfying the conditions of Figure 2. Define an **(atoms) swapping** $(a_i \;\; b_i)s$ by the rules in Figure 3. Swapping is characteristic of the underlying 'nominal' method we use in this paper [9,27]. We let swapping $(a_i \;\; b_i)$ act pointwise on sets of variables S: $(a_i \;\; b_i)S = \{(a_i \;\; b_i)c \mid c \in S\}$. Here c ranges over all elements of S, including a_i and b_i (if they are in S).

Lemma 2.4 $\mathsf{fv}((a_i \;\; b_i)s) = (a_i \;\; b_i)\mathsf{fv}(s)$ *and* $\mathsf{level}((a_i \;\; b_i)s) = \mathsf{level}(s)$.

If S is a set of variables write $a_i \# S$ when $a_i \notin S$ and also there exists no variable $b_j \in S$ such that $j > i$.

Definition 2.5 *Call the two rules in Figure 4* α**-conversion** *of* a_i*. Let* α**-equivalence** $=_\alpha$ *be the least congruence relation containing* α*-conversion.*

Note that: a_i may be α-converted in $\lambda a_i.s$ if $\mathsf{level}(s) \leq i$, so $\lambda x.x =_\alpha \lambda y.y$. a_i may be α-converted in $s[a_i \mapsto t]$ if $\mathsf{level}(s) \leq i$, so $x[x \mapsto X] =_\alpha y[y \mapsto X]$. We cannot α-convert a_i in s if $b_j \in \mathsf{fv}(s)$ for $j > i$. For example $\lambda x.X \neq_\alpha \lambda y.X$. This is consistent with a reading of strong variables as unknown terms with respect to weaker variables. We cannot α-convert variables to variables of other levels.

Lemma 2.6 *If* s *mentions only variables of level 1, then* α*-equivalence collapses to the usual* α*-equivalence on untyped* λ*-terms (plus an explicit substitution).*

Theorem 2.7 *If* $s =_\alpha s'$ *then* $\mathsf{fv}(s) = \mathsf{fv}(s')$ *and* $\mathsf{level}(s) = \mathsf{level}(s')$.

Proofs of all results above are by easy inductions.

In the rest of this paper we find it convenient to work on terms up to α-equivalence ($=_\alpha$-equivalence classes of terms). When later we write '$s = t$', the intended reading is that the *α-equivalence classes* of s and t are equal.

2.2 Reductions

Definition 2.8 *Define* **the reduction relation** *by the rules in Figure 5.*

Recall our permutative convention; for example in $(\sigma\lambda')$ a_i and c_i are distinct. Subsection 2.3 shows examples of these rules at work, and Subsection 2.4 discusses their design. We shall use the following notation:

- We write $\rightsquigarrow^*$ for the transitive reflexive closure of $\rightsquigarrow$.
- We write $s \not\rightsquigarrow$ when there exists no t such that $s \rightsquigarrow t$. If $s \not\rightsquigarrow$ we call s a **normal form**, as is standard.
- We write $s \overset{(\mathbf{ruleset})}{\rightsquigarrow} t$ when we can deduce $s \rightsquigarrow t$ using only rules in **(ruleset)** and the rules **(Rapp)** to **(Rσ')**, where **(ruleset)**$\subseteq\{(\beta), (\sigma\mathbf{a}), (\sigma\mathsf{fv}), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}$. (Later in Section 5 we extend reduction with rules for a binder M.)
- Call $\rightsquigarrow$ **terminating** when there is no infinite sequence $t_1 \rightsquigarrow \cdots \rightsquigarrow t_i \rightsquigarrow \cdots$ Similarly for $\overset{(\mathbf{ruleset})}{\rightsquigarrow}$. Call $\rightsquigarrow$ **confluent** when if $s \rightsquigarrow^* t$ and $s \rightsquigarrow^* t'$ then there exists some u such that $t \rightsquigarrow^* u$ and $t' \rightsquigarrow^* u$. Similarly for $\overset{(\mathbf{ruleset})}{\rightsquigarrow}$.

This is all standard [25,1].

We note two easy but important technical properties: reductions does not increase the level of a term or its set of free variables.

Lemma 2.9 *If* $s \rightsquigarrow s'$ *then* $\mathsf{level}(s') \leq \mathsf{level}(s)$.

Lemma 2.10 *If* $s \rightsquigarrow s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$*, and if* $s \rightsquigarrow^* s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$.

2.3 Example reductions

The LCC is a λ-calculus with explicit substitutions [15]. The general form of the σ-rules is familiar from the literature though the conditions, especially those involving

$$
\begin{array}{lll}
(\beta) & (\lambda a_i.s)t \rightsquigarrow s[a_i{\mapsto}t] & \\
(\sigma\mathbf{a}) & a_i[a_i{\mapsto}t] \rightsquigarrow t & \\
(\sigma\mathsf{fv}) & s[a_i{\mapsto}t] \rightsquigarrow s & a_i\#\mathsf{fv}(s) \\
(\sigma\mathbf{p}) & (ss')[a_i{\mapsto}t] \rightsquigarrow (s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]) & \mathsf{level}(s,s',t) \leq i \\
(\sigma\sigma) & s[a_i{\mapsto}t][b_j{\mapsto}u] \rightsquigarrow s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]] & i < j \\
(\sigma\lambda) & (\lambda a_i.s)[b_j{\mapsto}u] \rightsquigarrow \lambda a_i.(s[b_j{\mapsto}u]) & i < j \\
(\sigma\lambda') & (\lambda a_i.s)[c_i{\mapsto}u] \rightsquigarrow \lambda a_i.(s[c_i{\mapsto}u]) & a_i\#\mathsf{fv}(u)
\end{array}
$$

$$
\frac{s \rightsquigarrow s'}{st \rightsquigarrow s't}\,(\mathbf{Rapp}) \qquad \frac{t \rightsquigarrow t'}{st \rightsquigarrow st'}\,(\mathbf{Rapp'}) \qquad \frac{s \rightsquigarrow s'}{\lambda a_i.s \rightsquigarrow \lambda a_i.s'}\,(\mathbf{R}\lambda)
$$

$$
\frac{s \rightsquigarrow s'}{s[a_i{\mapsto}t] \rightsquigarrow s'[a_i{\mapsto}t]}\,(\mathbf{R}\sigma) \qquad \frac{t \rightsquigarrow t'}{s[a_i{\mapsto}t] \rightsquigarrow s[a_i{\mapsto}t']}\,(\mathbf{R}\sigma')
$$

Fig. 5. Reduction rules of the LCC

levels, are not; we discuss them in Subsection 2.4 below. First, we consider some example reductions. Recall our convention that we write x, y, z for variables of level 1, and X, Y, Z for variables of level 2.

- (β) is standard for a calculus with explicit substitutions.
- The behaviour of a substitution on a variable depends on strengths:

$$
x[X{\mapsto}t] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} x \qquad x[x'{\mapsto}t] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} x \qquad x[x{\mapsto}t] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t \qquad X[x{\mapsto}t] \not\rightsquigarrow
$$

  The term $X[x{\mapsto}t]$ will not reduce until a suitable strong substitution $[X{\mapsto}t]$ arrives from the surrounding context, if any.

- Substitutions for relatively strong variables may distribute using $(\sigma\sigma)$ or $(\sigma\lambda)$ under substitutions or λ-abstractions for relatively weaker variables:

$$
\begin{array}{l}
X[x{\mapsto}t][X{\mapsto}x] \overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}x][x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x] \\
(\lambda x.X)[X{\mapsto}x] \rightsquigarrow \lambda x.(X[X{\mapsto}x]) \rightsquigarrow \lambda x.x
\end{array}
$$

  This makes strong variables behave like 'holes'. Instantiation of holes is compatible with β-reduction; here is a typical example:

$$
\begin{array}{ll}
((\lambda x.X)t)[X{\mapsto}x] \overset{(\sigma\mathbf{p})}{\rightsquigarrow} (\lambda x.X)[X{\mapsto}x](t[X{\mapsto}x]) & ((\lambda x.X)t)[X{\mapsto}x] \overset{(\beta)}{\rightsquigarrow} X[x{\mapsto}t][X{\mapsto}x] \\
\quad \overset{(\sigma\lambda)}{\rightsquigarrow} (\lambda x.(X[X{\mapsto}x]))(t[X{\mapsto}x]) & \quad \overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}x][x{\mapsto}t[X{\mapsto}x]] \\
\quad \overset{(\sigma\mathbf{a})}{\rightsquigarrow} (\lambda x.x)(t[X{\mapsto}x]) & \quad \overset{(\sigma\mathbf{a})}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x] \\
\quad \overset{(\beta)}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x] &
\end{array}
$$

- There is no restriction in $s[a_i{\mapsto}t]$ that $\mathsf{level}(t) < i$; for example the terms $X[x{\mapsto}Y]$ and $X[x{\mapsto}\mathcal{W}]$ are legal.
- $[a_i{\mapsto}t]$ is not a term, but the term $\lambda b_j.b_j[a_i{\mapsto}t]$ where $j > i$ and $j > \mathsf{level}(t)$ will achieve the effect of 'the substitution $[a_i{\mapsto}t]$ as a term':

$$
(\lambda b_j.b_j[a_i{\mapsto}t])s \overset{(\beta)}{\rightsquigarrow} b_j[a_i{\mapsto}t][b_j{\mapsto}s] \overset{(\sigma\sigma)}{\rightsquigarrow} b_j[b_j{\mapsto}s][a_i{\mapsto}t[b_j{\mapsto}s]] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} b_j[b_j{\mapsto}s][a_i{\mapsto}t] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} s[a_i{\mapsto}t].
$$

2.4 Comments on the side-conditions

- $(\sigma\mathsf{fv})$ is a form of garbage-collection. We do not want to garbage-collect $[x{\mapsto}2]$ in $X[x{\mapsto}2]$ because $(\sigma\sigma)$ could turn X into something with x free — for example x itself; this is why the side-condition is not $a_i \notin \mathsf{fv}(s)$ but $a_i\#\mathsf{fv}(s)$.

  It is unusual for a garbage collection rule to appear in a calculus of explicit substitutions; we might hope to 'push substitutions into a term until they reach variables' and so make do with a rule of the form $c_k[a_i{\mapsto}t] \rightsquigarrow c_k$ (for $k \leq i$). In the LCC this will not do because side-conditions (such as that of $(\sigma\mathbf{p})$) can stop a substitution going deep into a term. Without $(\sigma\mathsf{fv})$ we lose confluence (see the second case of Theorem 4.10). A version of $(\sigma\mathsf{fv})$ appears in the literature as 'garbage collection' [3].
- Recall that the level of a term is the level of the strongest variable it contains, free *or* bound. The side-condition $\mathsf{level}(s, s', t) \leq i$ in $(\sigma\mathbf{p})$ seems to be fundamental for confluence to work; we have not been able to sensibly weaken it, even if we also change other rules to fix what goes wrong when we do. Here is what happens if we drop the side-condition entirely:

$$\begin{array}{lll} X[x{\mapsto}y][y{\mapsto}x] & \overset{(\beta)}{\leftsquigarrow} ((\lambda x.X)y)[y{\mapsto}x] & \overset{(\sigma\mathbf{pFALSE})}{\rightsquigarrow} ((\lambda x.X)[y{\mapsto}x])(y[y{\mapsto}x]) \\ & & \overset{(\sigma\mathsf{a})}{\rightsquigarrow} ((\lambda x.X)[y{\mapsto}x])x \end{array}$$

- The side-conditions on $(\sigma\sigma)$, $(\sigma\lambda)$, and $(\sigma\lambda')$ implement that a strong substitution can capture. There is no $(\sigma\sigma')$ since that would destroy termination of the part of the LCC without λ — and we have managed to get confluence without it.
- There is no rule permitting a weak substitution to propagate under a stronger abstraction, *even if* we avoid capture:

$$(\sigma\lambda'\mathbf{FALSE}) \qquad (\lambda a_i.s)[c_k{\mapsto}u] \rightsquigarrow \lambda a_i.(s[c_k{\mapsto}u]) \qquad a_i\#\mathsf{fv}(u),\ k \leq i$$

  Such a rule causes the following problem for confluence:

$$\begin{array}{lll} (\lambda Y.(xZ))[x{\mapsto}3][Z{\mapsto}\mathcal{W}] & \overset{(\sigma\lambda'\mathbf{FALSE})}{\rightsquigarrow} & (\lambda Y.(xZ)[x{\mapsto}3])[Z{\mapsto}\mathcal{W}] \\ (\lambda Y.(xZ))[x{\mapsto}3][Z{\mapsto}\mathcal{W}] & \overset{(\sigma\sigma)}{\rightsquigarrow} & (\lambda Y.(xZ))[Z{\mapsto}\mathcal{W}][x{\mapsto}3[Z{\mapsto}\mathcal{W}]] \\ & \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} & (\lambda Y.(xZ))[Z{\mapsto}\mathcal{W}][x{\mapsto}3] \end{array}$$

  As is the case for the side-condition of $(\sigma\mathbf{p})$, any stronger form of $(\sigma\lambda')$ than what we admit in the LCC seems to provoke a cascade of changes which make the calculus more complex.

  Investigation of these side-conditions is linked to strengthening the theory of freshness and α-equivalence, and possibly to developing a good semantic theory to guide us. This is future work and some details are mentioned in the Conclusions.

3 The substitution action

Define $(\mathbf{sigma}) = \{(\sigma\mathbf{a}), (\sigma\mathsf{fv}), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}$ (so (**sigma**) is 'everything except for (β)'). It would be good if this is is terminating [3,15]. Do we sacrifice this property because of the hierarchy of variables? No. To prove it we translate LCC syntax to first-order terms (terms without binding [1,25]) in the signature

$$\Sigma = \{\star, \mathsf{Abs}, \mathsf{App}\} \cup \{\mathsf{Sub}^i \mid i\}$$

as follows:

$$\overline{x} = \star \quad \overline{\lambda a_i.s} = \mathsf{Abs}(\overline{s}) \quad \overline{s\ t} = \mathsf{App}(\overline{s}, \overline{t}) \quad \overline{s[a_i{\mapsto}t]} = \mathsf{Sub}^i(\overline{s}, \overline{t})$$

Here $\star$ has arity 0, Abs has arity 1, App has arity 2, and Sub^i has arity 2 for all i (i ranges over levels). Give symbols precedence (lowest precedence on the right)

$$\ldots, \mathsf{Sub}^j, \ldots, \mathsf{Sub}^i, \ldots, \mathsf{App}, \mathsf{Abs}, \star \qquad\qquad (j > i).$$

Define the **lexicographic path ordering** [14,1] by:

$$\frac{}{t_i \ll f(t_1, \ldots, t_n)} \qquad \frac{s \ll t_i}{s \ll f(t_1, \ldots, t_n)}$$

$$\frac{(t'_1, \ldots, t'_n) \ll (t_1, \ldots, t_n)}{f(t'_1, \ldots, t'_n) \ll f(t_1, \ldots, t_n)} \qquad \frac{u_i \ll f(t_1, \ldots, t_n) \text{ for } 1 \le i \le m}{g(u_1, \ldots, u_m) \ll f(t_1, \ldots, t_n)}$$

Here g and f are first-order symbols, g has strictly lower precedence than f, and $t_1, \ldots, t_n, t'_1, \ldots, t'_n, u_1, \ldots, u_m, s$ are first-order terms. It is a fact [14,1] that $\ll$ is a well-founded order on first-order terms satisfying the *subterm property*, i.e. if s is a subterm of t then $s \ll t$.

Theorem 3.1 *If* $t \overset{(\mathbf{sigma})}{\rightsquigarrow} u$ *then* $\overline{t} \gg \overline{u}$. *Thus* (**sigma**)*-reduction terminates.*

The proof is by checking that a (**sigma**)-reduction strictly reduces the lexicographic path order of the associated first-order term; this is not hard.

Let x have level 1. $(\lambda x.xx)(\lambda x.xx)$ has an infinite series of reductions in the LCC. It follows that — even with a hierarchy of variables — (β) strictly adds power to the LCC.

Call s (**sigma**)**-normal** when $s \overset{(\mathbf{sigma})}{\not\rightsquigarrow}$. What does a (**sigma**)-normal form look like? Define a **substitution action** $s[a_i{:=}t]$ and using it define s^*, by the rules in Figure 6. Rules are listed in order of precedence so that a later rule is only used if no earlier rule is applicable. We apply the rule $(\lambda c_i.s)[a_i{:=}t]$ renaming where possible to ensure $c_i \# \mathsf{fv}(t)$.

Lemma 3.2 $s[a_i{\mapsto}t] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s[a_i{:=}t]$.

Proof. Each clause in the definition of $s[a_i{:=}t]$ is simulated by a (**sigma**)-rule. □

Theorem 3.3 $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$ *and* s^* *is a* (**sigma**)*-normal form.*

Proof. The first part is by an easy induction on the definition of s^*; the case of $(s[a_i{\mapsto}t])^*$ uses Lemma 3.2. The second part is by a routine induction on s. □

$$
\begin{array}{rll}
s[a_i{:=}t] = s & a_i\#\mathsf{fv}(s), \text{ and } \textit{otherwise} & a_i^* = a_i \\
a_i[a_i{:=}t] = t & & (\lambda a_i.s)^* = \lambda a_i.(s^*) \\
(ss')[a_i{:=}t] = (s[a_i{:=}t])(s'[a_i{:=}t]) & \mathsf{level}(s,s',t) \le i & (s[a_i{\mapsto}t])^* = s^*[a_i{:=}t^*] \\
s[c_k{\mapsto}u][a_i{:=}t] = s[a_i{:=}t][c_k{:=}u[a_i{:=}t]] & k < i & (st)^* = (s^*)(t^*) \\
(\lambda c_k.s)[a_i{:=}t] = \lambda c_k.(s[a_i{:=}t]) & k < i & \\
(\lambda c_i.s)[a_i{:=}t] = \lambda c_i.(s[a_i{:=}t]) & c_i\#\mathsf{fv}(t) & \\
s[a_i{:=}t] = s[a_i{\mapsto}t] & &
\end{array}
$$

Fig. 6. Substitution $s[a_i{:=}t]$ and (**sigma**)-normal form s^*

4 Confluence

Let (**beta**) be the set $\{(\beta),(\sigma\lambda),(\sigma\lambda'),(\sigma\mathsf{fv})\}$. (**sigma**) $\cap$ (**beta**) is non-empty; we discuss why at the end of Subsection 4.3.

Theorem 4.1 $\rightsquigarrow$ *is confluent.*

The proof of Theorem 4.1 occupies this section. Two standard proof-methods are: *(1)* Use a *parallel reduction relation* $\Rightarrow$, and *(2)* for all s define a $s^{\downarrow}$ such that $s \rightsquigarrow^* s^{\downarrow}$ and if $s \rightsquigarrow s'$ then $s' \rightsquigarrow^* s^{\downarrow}$. Both methods are standard [25]. Which to use for the LCC? It seems that λ 'wants' method 1 — but σ 'wants' method 2. Confluence is (relatively) easy to prove if we split the reduction relation into (**sigma**) and (**beta**) and apply different methods to each — and then join them together.

4.1 Confluence of (**sigma**)

Note there is no capture-avoidance condition in Lemma 4.2, because $i < j$. The full proofs also contain another version where $i = j$ and $a_i\#\mathsf{fv}(u)$.

Lemma 4.2 *If* $i < j$ *then* $s[a_i{:=}t][b_j{:=}u] = s[b_j{:=}u][a_i{:=}t[b_j{:=}u]]$.

Proof. By induction on i, then on s. We illustrate the induction with two cases.

- Suppose $i < j < k$. Note that usually we take $k \le i$; this is an exception. Then:

$$
\begin{array}{rl}
c_k[a_i{:=}t][b_j{:=}u] & = c_k[a_i{\mapsto}t][b_j{:=}u] \\
& = c_k[b_j{:=}u][a_i{\mapsto}t[b_j{:=}u]] \\
& = c_k[b_j{\mapsto}u][a_i{\mapsto}t[b_j{:=}u]]
\end{array}
\qquad
\begin{array}{rl}
c_k[b_j{:=}u][a_i{:=}t[b_j{:=}u]] & = c_k[b_j{\mapsto}u][a_i{:=}t[b_j{:=}u]] \\
& = c_k[b_j{\mapsto}u][a_i{\mapsto}t[b_j{:=}u]]
\end{array}
$$

- Suppose that $\mathsf{level}(s,s',t) < j$. By Lemma 3.2 we have $(ss')[a_i{\mapsto}t] \rightsquigarrow^* (ss')[a_i{:=}t]$. By Lemma 2.9 we have $\mathsf{level}((ss')[a_i{:=}t]) \le \mathsf{level}((ss')[a_i{\mapsto}t]) = \mathsf{level}(s,s',t) < j$. Then by our assumptions on levels,

$$(ss')[a_i{:=}t][b_j{:=}u] = (ss')[a_i{:=}t] = (ss')[b_j{:=}u][a_i{:=}t[b_j{:=}u]].$$

□

Lemma 4.3 (i) $(a_i[a_i{\mapsto}t])^* = t^*$.

(ii) $(c_k[a_i{\mapsto}t])^* = c_k$ *where* $k \le i$.

(iii) $((ss')[a_i{\mapsto}t])^* = ((s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]))^*$ *where* $\mathsf{level}(s,s',t) \le i$.

(iv) $(s[a_i{\mapsto}t][b_j{\mapsto}u])^* = (s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]])^*$ *if* $i < j$.

(v) $((\lambda a_i.s)[b_j{\mapsto}u])^* = (\lambda a_i.(s[b_j{\mapsto}u]))^*$ *if* $i < j$.

$$\frac{}{a_i \Longrightarrow a_i}\ (\mathbf{Pa}) \qquad \frac{s \Longrightarrow s' \quad t \Longrightarrow t'}{s[a_i{\mapsto}t] \Longrightarrow s'[a_i{\mapsto}t']}\ (\mathbf{P}\sigma) \qquad \frac{s \Longrightarrow s' \quad t \Longrightarrow t'}{st \Longrightarrow s't'}\ (\mathbf{Papp}) \qquad \frac{s \Longrightarrow s'}{\lambda a_i.s \Longrightarrow \lambda a_i.s'}\ (\mathbf{P}\lambda)$$

$$\frac{s \Longrightarrow s' \quad t \Longrightarrow t' \quad s'[a_i{\mapsto}t'] \overset{R_\epsilon}{\rightsquigarrow} u}{s[a_i{\mapsto}t] \Longrightarrow u}\ (\mathbf{P}\sigma\epsilon) \qquad \frac{s \Longrightarrow s' \quad t \Longrightarrow t' \quad s't' \overset{R_\epsilon}{\rightsquigarrow} u}{st \Longrightarrow u}\ (\mathbf{Papp}\epsilon) \qquad (R \in (\mathbf{beta}))$$

Fig. 7. Parallel reduction relation for the LCC

(vi) $((\lambda a_i.s)[c_i{\mapsto}u])^* = (\lambda a_i.(s[c_i{\mapsto}u]))^*$ *if (renaming where possible)* $a_i \# \mathsf{fv}(u)$.

Proof. Most cases are easy; we consider only the fourth one. Recall that we assume $i < j$. Using Lemma 4.2

$$\begin{aligned}(s[a_i{\mapsto}t][b_j{\mapsto}u])^* = s^*[a_i{:=}t^*][b_j{:=}u^*] = s^*[b_j{:=}u^*][a_i{:=}t^*[b_j{:=}u^*]] \\ = (s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]])^*.\end{aligned}$$

□

Lemma 4.4 *If* $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ *then* $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$.

Proof. By induction on the derivation of $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$, using Lemma 4.3. □

Theorem 4.5 $\overset{(\mathbf{sigma})}{\rightsquigarrow}$ *is confluent.*

Proof. By an easy inductive argument using Lemma 4.4. □

4.2 (**beta**)*-reduction*

Define the **parallel reduction relation** $\Longrightarrow$ by the rules in Figure 7.

In rules $(\mathbf{P}\sigma\epsilon)$ and $(\mathbf{Papp}\epsilon)$, $s't' \overset{R_\epsilon}{\rightsquigarrow} u$ and $s'[a_i{\mapsto}t'] \overset{R_\epsilon}{\rightsquigarrow} u$ indicate a rewrite with $R \in (\mathbf{beta})$ derivable without using $(\mathbf{Rapp})$, $(\mathbf{Rapp}')$, $(\mathbf{R}\lambda)$, $(\mathbf{R}\sigma)$, or $(\mathbf{R}\sigma')$.

Lemma 4.6 $s \Longrightarrow^* s'$ *if and only if* $s \overset{(\mathbf{beta})}{\rightsquigarrow^*} s'$.

Corollary 4.7 *If* $s \Longrightarrow s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$ *and* $\mathsf{level}(s') \leq \mathsf{level}(s)$.

Proof. From Lemma 4.6 and Lemma 2.10. □

Lemma 4.8 $\Longrightarrow$ *satisfies the diamond property. That is, if* $s' \Longleftarrow s \Longrightarrow s''$ *then there is some* s''' *such that* $s' \Longrightarrow s''' \Longleftarrow s''$.

Proof. We work by induction on the depth of the derivation of $s \Longrightarrow s'$ proving $\forall s''.\ s \Longrightarrow s'' \Rightarrow \exists s'''.\ (s' \Longrightarrow s''' \wedge s'' \Longrightarrow s''')$. We consider possible pairs of rules which could derive $s \Longrightarrow s_1$ and $s \Longrightarrow s_2$. All cases are very easy, we only sketch that of $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$, which is the least trivial.

Suppose $s \Longrightarrow s'$ and $u \Longrightarrow u'$ and also $s \Longrightarrow s''$ and $u \Longrightarrow u''$. Suppose also that (renaming where necessary) $a_i \# u''$ so that by $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$

$$(\lambda a_i.s')[c_i{\mapsto}u'] \Longleftarrow (\lambda a_i.s)[c_i{\mapsto}u] \Longrightarrow \lambda a_i.(s''[c_i{\mapsto}u'']).$$

By inductive hypothesis there are s''' and u''' such that $s' \Longrightarrow s''' \Longleftarrow s''$ and

$u' \Longrightarrow u''' \Longleftarrow u''$. By Corollary 4.7 $a_i \# u'''$. Using $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$ and $(\mathbf{P}\sigma)$

$$(\lambda a_i.s')[c_i \mapsto u'] \Longrightarrow \lambda a_i.(s'''[c_i \mapsto u''']) \Longleftarrow \lambda a_i.(s''[c_i \mapsto u'']).$$

□

Theorem 4.9 $\overset{(\mathbf{beta})}{\rightsquigarrow}$ *is confluent.*

Proof. By Lemmas 4.6 and 4.8 and a standard argument [2]. □

4.3 Combining (**sigma**) *and* (**beta**)

Theorem 4.10 *If* $s \Longrightarrow s'$ *and* $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s''$ *then there is some* s''' *such that* $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$ *and* $s'' \Longrightarrow s'''$.

Proof. We work by induction on the derivation of $s \Longrightarrow s'$. For brevity we merely indicate the non-trivial parts. We always assume that $s \Longrightarrow s'$, $t \Longrightarrow t'$, and $u \Longrightarrow u'$, where appropriate.

- (β) has a divergence with $(\sigma\mathbf{p})$ in the case that $i < j$ and $\mathsf{level}(s,t,u) \leq j$. This can be closed using a $\Longrightarrow$-rewrite which uses $(\sigma\lambda)$:

$$(\lambda a_i.s)[b_j \mapsto u](t[b_j \mapsto u]) \overset{(\sigma\mathbf{p})}{\rightsquigarrow} ((\lambda a_i.s)t)[b_j \mapsto u] \Longrightarrow s'[a_i \mapsto t'][b_j \mapsto u']$$

$$(\lambda a_i.s)[b_j \mapsto u](t[b_j \mapsto u]) \Longrightarrow s'[b_j \mapsto u'][a_i \mapsto t'[b_j \mapsto u']] \overset{(\sigma\sigma)}{\rightsquigarrow} s'[a_i \mapsto t'][b_j \mapsto u']$$

- $(\sigma\sigma)$ has a divergence with $(\sigma\lambda')$. Suppose $i<j$ and (renaming if necessary) $c_i \# \mathsf{fv}(t)$:

$$(\lambda c_i.s)[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]] \overset{(\sigma\sigma)}{\rightsquigarrow} (\lambda c_i.s)[a_i \mapsto t][b_j \mapsto u] \Longrightarrow (\lambda c_i.(s'[a_i \mapsto t']))[b_j \mapsto u']$$

  We know $b_j \# \mathsf{fv}(t)$ because $c_i \# \mathsf{fv}(t)$ and $i < j$. We deduce $b_j \# \mathsf{fv}(t')$ using Corollary 4.7. This justifies the $\Longrightarrow$-rewrite below, which uses $(\sigma\mathsf{fv})$:

$$\begin{aligned} \lambda c_i.(s'[a_i \mapsto t'])[b_j \mapsto u'] &\overset{(\sigma\lambda')}{\rightsquigarrow} \lambda c_i.(s'[a_i \mapsto t'][b_j \mapsto u']) \\ &\overset{(\sigma\sigma)}{\rightsquigarrow} \lambda c_i.(s'[b_j \mapsto u'][a_i \mapsto t'[b_j \mapsto u']]) \\ &\overset{(\sigma\mathsf{fv})}{\rightsquigarrow} \lambda c_i.(s'[b_j \mapsto u'][a_i \mapsto t']) \Longleftarrow (\lambda c_i.s)[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]] \end{aligned}$$

□

$(\sigma\lambda)$ is in (**sigma**) ∩ (**beta**) to make the case of $(\sigma\mathbf{p})$ with (β) work. $(\sigma\lambda')$ is in (**sigma**) ∩ (**beta**) to make a similar divergence of $(\sigma\mathbf{p})$ with (β) work. $(\sigma\mathsf{fv})$ is in (**sigma**) ∩ (**beta**) to make the case of $(\sigma\sigma)$ with $(\sigma\lambda')$ work.

Theorem 4.1 now follows by an easy diagrammatic argument using Theorem 4.10, Theorem 4.5, and Lemma 4.8.

5 A NEW part for the LCC

x is not α-convertible in $\lambda x.X$. Suppose we really do want to bind x; we can do so with И. We extend syntax: $s,t ::= \ldots \mid И a_i.t$. We extend the notions of level,

fv, congruence, and swapping with cases for $И$ which are identical to those for λ (except that we write $И$ instead). For example $\mathsf{fv}(Иa_i.s) = \mathsf{fv}(s) \setminus \{a_i\}$.

The difference is in the α-equivalence: $Иa_i.s =_\alpha Иb_i.(b_i\ a_i)s$ if $b_i \notin \mathsf{fv}(s)$.

Note the $b_i \notin \mathsf{fv}(s)$ instead of $b_i\#\mathsf{fv}(s)$ as in the clause for λ. This lets variables bound by $И$ α-convert regardless of whether stronger variables are present. For example $\lambda x.X \neq_\alpha \lambda y.X$ but $Иx.\lambda x.X =_\alpha Иy.\lambda y.X$. We add reduction rules:

$$\begin{array}{lll} (И\mathbf{p}) & (Иa_i.s)t \rightsquigarrow Иa_i.(st) & a_i \notin \mathsf{fv}(t) \\ (И\sigma) & (Иc_k.s)[a_i{\mapsto}t] \rightsquigarrow Иc_k.(s[a_i{\mapsto}t]) & k \leq i,\ c_k \notin \mathsf{fv}(t) \\ (И \notin) & Иa_i.s \rightsquigarrow s & a_i \notin \mathsf{fv}(s) \end{array} \qquad \frac{s \rightsquigarrow s'}{Иa_i.s \rightsquigarrow Иa_i.s'}\ (\mathbf{R}И)$$

x is not bound in $\lambda y.s$ if s mentions a strong variable, for example in $\lambda y.(Xy)$ substitution for X can capture y. We may want y to be *really* local and avoid capture by substitutions for X. We can increase the level of y; $\lambda Y.(XY)$ will do in this case. This has a hidden cost because side-conditions (especially on $(\sigma\mathbf{p})$) look at strengths of variables, so having strong variables can block reductions in the context. $И$ avoids this, for example $Иy.\lambda y.(Xy)$ has the behaviour we need:

$$(Иy.\lambda y.(Xy))[X{\mapsto}y] \overset{(и\sigma)}{\rightsquigarrow} Иy'.((\lambda y'.(Xy'))[X{\mapsto}y]) \overset{(и\lambda)}{\rightsquigarrow} Иy'.\lambda y'.(Xy')[X{\mapsto}y]) \rightsquigarrow^* Иy'.\lambda y'.yy'$$

$И$ is reminiscent of π-calculus restriction [16]. $(И\mathbf{p})$ and $(И\sigma)$ are reminiscent of scope-extrusion. $(И \notin)$ is reminiscent of 'garbage-collection'.

We do not admit a rule '$s(Иa.t) \rightsquigarrow Иa.(st)$ if $a \notin \mathsf{fv}(s)$':

$$Иy.Иy'.(yy') \ {}^*\!\!\leftsquigarrow\ (Иy.y)Иy'.y' \ {}^*\!\!\leftsquigarrow\ (\lambda x.xx)Иy.y \rightsquigarrow Иy.(\lambda x.xx)y \rightsquigarrow^* Иy.yy$$

For similar reasons we do not admit a rule '$s[b{\mapsto}Иa.t] \rightsquigarrow Иa.(s[b{\mapsto}t])$ if $a \notin \mathsf{fv}(s)$'.

Why the side-conditions on $(И\sigma)$? $c_k \notin \mathsf{fv}(t)$ comes from the intuition of $И$ as defining a scope. We need $k \leq i$ for confluence:

$$ИX.(X[x{\mapsto}2]) \leftsquigarrow (ИX.X)[x{\mapsto}2] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} ИX.X$$

The proof of termination of **(sigma)** extends smoothly if we add the rules for $И$ to **(sigma)** (to make a set **(sigmanew)**). The proof of confluence for the system as a whole also extends smoothly. We see some examples of the use of $И$ in a moment.

6 Programming in the calculus

Call t **single-leveled** of level i when all variables in it (free or bound) have level i. Then it is easy to prove that notions of free variable and substitution coincide with the 'traditional' definitions and we have:

Theorem 6.1 *For any i the single-leveled terms of level i, with their reductions, form an isomorphic calculus to λx with garbage collection [3].*

As a corollary, the trivial mapping from the untyped λ-calculus to single-leveled terms of level 1 (say), preserves normal forms and strong normalisation.

We can exploit the hierarchy to do some nice things. Here is one example: $R = X[x{\mapsto}2][y{\mapsto}3]$ can be viewed as a record with 'handle' X and with 2 stored at x and 3 at y. Then $\lambda\mathcal{W}.(\mathcal{W}[X{\mapsto}x])$ applied to R looks up the data stored at x, and $\lambda\mathcal{W}.(\mathcal{W}[X{\mapsto}X[x{\mapsto}3]])$ updates it. In fact these terms do a little more than this, because their effect is the same when applied to a term in which a record with 'handle' X is buried deep in the term, perhaps as part of a β-redex or substitution. $\lambda\mathcal{W}.(\mathcal{W}[X \mapsto (\mathcal{W}[X{\mapsto}x])+1])$ increments the value stored at x.

Here is an example reduction:

$$\begin{array}{lll}
(\lambda\mathcal{W}.\mathcal{W}[X{\mapsto}X[x{\mapsto}3]])\ R & \overset{(\beta)}{\rightsquigarrow} & \mathcal{W}[X{\mapsto}X[x{\mapsto}3]][\mathcal{W}{\mapsto}R] \\
& \overset{(\sigma\sigma),(\sigma\mathbf{a}),(\sigma\mathsf{fv})}{\rightsquigarrow^*} & R[X{\mapsto}X[x{\mapsto}3]] = X[x{\mapsto}2][y{\mapsto}3][X{\mapsto}X[x{\mapsto}3]] \\
& \overset{(\sigma\sigma)}{\rightsquigarrow^*} & X[X{\mapsto}X[x{\mapsto}3]][x{\mapsto}2[X{\mapsto}X[x{\mapsto}3]]][y{\mapsto}3[X{\mapsto}X[x{\mapsto}3]]] \\
& \overset{(\sigma\mathbf{a}),(\sigma\mathbf{b})}{\rightsquigarrow^*} & X[x{\mapsto}3][x{\mapsto}2][y{\mapsto}3].
\end{array}$$

There is some garbage here, but a later look-up on x returns 3, not 2:

$$(\lambda\mathcal{W}.\mathcal{W}[X{\mapsto}x])(X[x{\mapsto}3][x{\mapsto}2][y{\mapsto}3]) \rightsquigarrow^* x[x{\mapsto}3][x{\mapsto}2][y{\mapsto}3] \rightsquigarrow^* 3$$

We can use $\mathsf{И}$ to assign fresh storage. The following program, if applied to a value and R, extends R with a fresh location and returns the new record together with a lookup function for the new location:

$$\lambda Z.\mathsf{И}x.\lambda Y.(Y[x{\mapsto}Z], \lambda\mathcal{W}.\mathcal{W}[X{\mapsto}x]).$$

Here we use a pairing constructor (-,-) just for convenience.

Note that we access data in R by applying a substitution for X; in this sense the 'handle' X in R is externally visible. We can hide it by λ-abstracting X to obtain $\lambda X.(X[x{\mapsto}2][y{\mapsto}3])$. Then lookup at x becomes $\lambda\mathcal{W}.(\mathcal{W}x)$ and update becomes $\lambda\mathcal{W}.\lambda X.(\mathcal{W}[X{\mapsto}X[x{\mapsto}1]])$.

We can parameterise over the data stored in the record: $\lambda X'.(X'[x{\mapsto}X][y{\mapsto}Y])$. Furthermore a term of the form $\lambda X.(X[x{\mapsto}X][y{\mapsto}X])$ can capture a form of self-reference within the record. Finally, $\lambda X.(X[x{\mapsto}\mathcal{W}][y{\mapsto}\mathcal{W}'])$ makes no committment about the data stored.

7 Related work, conclusions, and future work

The LCC of this paper is simpler than the NEWcc [6]. Compare the side-condition of $(\sigma\mathbf{a})$ (there is none) with that of $(\sigma\mathbf{a})$ from [6]. The notion of freshness is simpler and intuitive; we no longer require a logic of freshness, or the 'freshness context with sufficient freshnesses', see most of page 4 in [6]. A key innovation in attaining this simplicity is our use of conditions involving $\mathsf{level}(s)$ the level of s, which includes information about the levels of free *and* bound variables.

But there is a price: this calculus has fewer reductions. Notably $(\sigma\lambda')$ will not reduce $(\lambda a_i.s)[c_k{\mapsto}u]$ where $k < i$; a rule $(\sigma\lambda')$ in [6] does. That stronger version seems to be a major source of complexity.

Still, the LCC is part of something larger yet to be constructed. Other papers on nominal techniques contain elements of the developments we have in mind when we imagine such a system. So for example:

In this paper we cannot α-convert x in $\lambda x.X$. Nominal terms can: swappings are in the syntax (here swappings are in the meta-level) and also freshness contexts [27]. A problem is that we do not yet understand the theory of swappings for strong variables; the underlying Fraenkel-Mostowski sets model [9] only has (in the terminology of this paper) one level of variable. A semantic model of the hierarchy of variables would be useful and this is current work.

In this paper we cannot deduce $x\#\mathsf{fv}(\lambda x.X)$ even though for every instance this does hold (for example $x\#\lambda x.x$ and $x\#\lambda x.y$). Hierarchical nominal rewriting [7] has a more powerful notion of freshness which can prove the equivalent of $x\#\mathsf{fv}(\lambda x.X)$. Note that hierarchical nominal rewriting does not have the conditions on *levels* which we use to good effect in this paper.

We cannot reduce $(\lambda x.y)[y{\mapsto}Y]$ because there is no z such that $z\#Y$. We can allow programs to dynamically generate fresh variables in the style of FreshML [19] or the style of a sequent calculus for Nominal Logic by Cheney [5].

We cannot reduce $X[x{\mapsto}2][y{\mapsto}3]$ to $X[y{\mapsto}3][x{\mapsto}2]$. Other work [8] gives an equational system which can do this, and more.

There is no denotational semantics for the LCC. This is current work.

More related work (not using nominal techniques). The calculi of contexts λm and $\lambda\mathcal{M}$ [23] also have a hierarchy of variables. They use carefully-crafted scoping conventions to manage problems with α-conversion. Other work [21,11,22] uses a type system; connections with this work are unclear. λc of Bognar's thesis contains [4, Section 2] an extensive literature survey on the topic of context calculi.

A separation of abstraction λ and binding $\mathsf{И}$ appears in one other work we know of [24], where they are called q and ν. In this vein there is [12], which manages scope explicitly in a completely different way, just for the fun. Finally, the reduction rules of $\mathsf{И}$ look remarkably similar to π-calculus restriction [16], and it is probably quite accurate to think of $\mathsf{И}$ as a 'restriction in the λ-calculus'.

Ours is a calculus with explicit substitutions. See [15] for a survey. Our treatment of substitution is simple-minded but still quite subtle because of interactions with the rest of the language. We note that the translation of possibly open terms of the untyped λ-calculus into the LCC preserves strong normalisation. One reduction rule, $(\sigma\mathsf{fv})$, is a little unusual amongst such calculi, though it appears as 'garbage collection' of λx [3].

The look and feel of the LCC is squarely that of a λ-calculus with explicit substitutions. All the real cleverness has been isolated in the side-condition of $(\sigma\mathbf{p})$; other side-conditions are obvious given an intuition that strong variables can cause capturing substitution (in the NEWcc [6] complexity spilled over into other rules and into a logic for freshness). $\mathsf{И}$ is only necessary when variables of different strengths occur, and the hierarchy of variables only plays a rôle to trigger side-conditions.

Further work. Desirable and nontrivial meta-properties of the λ-calculus survive in the LCC including confluence, and preservation of strong normalisation for a natural encoding of the untyped λ-calculus into the LCC. It is possible, in principle

at least, to envisage an extension of ML or Haskell [17,26] with meta-variables based on the LCC's notion of strong and weak variables.

We can go in the direction of logic, treating equality instead of reduction and imitating higher-order logic, which is based on the simply-typed λ-terms enriched with constants such as $\forall : (o \to o) \to o$ and $\Rightarrow\colon o \to o \to o$ where o is a type of truth-values [29], along with suitable equalities and/or derivation rules. There should be no problem with imposing a simple type system on LCC and writing down a 'context higher-order logic'. This takes the LCC in the direction of calculi of contexts for incomplete proofs [13,10]. The non-trivial work (in no particular order) is to investigate cut-elimination, develop a suitable theory of models/denotations, and possibly to apply it to model incomplete proofs.

An implementation is current work.

The LCC is simple, clear, and it has good properties. It seems to hit a technical sweet spot: every extension of it which we have considered, provokes significant non-local changes. Often in computer science the trick is to find a useful balance between simplicity and expressivity. Perhaps the LCC does that.

References

[1] Franz Baader and Tobias Nipkow, *Term rewriting and all that*, Cambridge University Press, 1998.

[2] H. P. Barendregt, *The lambda calculus: its syntax and semantics (revised ed.)*, North-Holland, 1984.

[3] Roel Bloo and Kristoffer Høgsbro Rose, *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*, CSN-95: Computer Science in the Netherlands, 1995.

[4] Mirna Bognar, *Contexts in lambda calculus*, Ph.D. thesis, Vrije Universiteit Amsterdam, 2002.

[5] James Cheney, *A simpler proof theory for nominal logic*, FOSSACS, Springer, 2005, pp. 379–394.

[6] Murdoch J. Gabbay, *A new calculus of contexts*, PPDP '05: Proc. of the 7th ACM SIGPLAN int'l conf. on Principles and Practice of Declarative Programming, ACM Press, 2005, pp. 94–105.

[7] ———, *Hierarchical nominal rewriting*, LFMTP'06: Logical Frameworks and Meta-Languages: Theory and Practice, 2006, pp. 32–47.

[8] Murdoch J. Gabbay and Aad Mathijssen, *Capture-avoiding substitution as a nominal algebra*, ICTAC'2006: 3rd Int'l Colloquium on Theoretical Aspects of Computing, 2006, pp. 198–212.

[9] Murdoch J. Gabbay and A. M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing **13** (2001), no. 3–5, 341–363.

[10] Herman Geuvers and Gueorgui I. Jojgov, *Open proofs and open terms: A basis for interactive logic*, CSL, Springer, 2002, pp. 537–552.

[11] Masatomo Hashimoto and Atsushi Ohori, *A typed context calculus*, Theor. Comput. Sci. **266** (2001), no. 1-2, 249–272.

[12] Dimitri Hendriks and Vincent van Oostrom, *Adbmal*, CADE, 2003, pp. 136–150.

[13] Gueorgui I. Jojgov, *Holes with binding power.*, TYPES, LNCS, vol. 2646, Springer, 2002, pp. 162–181.

[14] Samuel Kamin and Jean-Jacques Lévy, *Attempts for generalizing the recursive path orderings*, Handwritten paper, University of Illinois, 1980.

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

[16] Robin Milner, Joachim Parrow, and David Walker, *A calculus of mobile processes, II*, Information and Computation **100** (1992), no. 1, 41–77.

[17] Lawrence C. Paulson, *ML for the working programmer (2nd ed.)*, Cambridge University Press, 1996.

[18] A. M. Pitts, *Operationally-based theories of program equivalence*, Semantics and Logics of Computation (P. Dybjer and A. M. Pitts, eds.), Publications of the Newton Institute, Cambridge University Press, 1997, pp. 241–298.

[19] A. M. Pitts and Murdoch J. Gabbay, *A metalanguage for programming with bound names modulo renaming*, Mathematics of Program Construction. 5th Int'l Conf. , MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings (R. Backhouse and J. N. Oliveira, eds.), LNCS, vol. 1837, Springer-Verlag, 2000, pp. 230–255.

[20] A. M. Pitts and I. D. B. Stark, *Operational reasoning for functions with local state*, Higher Order Operational Techniques in Semantics (A. D. Gordon and A. M. Pitts, eds.), Publications of the Newton Institute, Cambridge University Press, 1998, pp. 227–273.

[21] Masahiko Sato, Takafumi Sakurai, and Rod Burstall, *Explicit environments*, Fundamenta Informaticae **45:1-2** (2001), 79–115.

[22] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama, *A simply typed context calculus with first-class environments*, Journal of Functional and Logic Programming **2002** (2002), no. 4, 359 – 374.

[23] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama, and Atsushi Igarashi, *Calculi of meta-variables*, Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings (M. Baaz, ed.), LNCS, vol. 2803, 2003, pp. 484–497.

[24] Francois Maurel Sylvain Baro, *The qnu and qnuk calculi : name capture and control*, Tech. report, Université Paris VII, 2003, Extended Abstract, Prépublication PPS//03/11//n16.

[25] Terese, *Term rewriting systems*, Cambridge Tracts in Theoretical Computer Science, no. 55, Cambridge University Press, 2003.

[26] Simon Thompson, *Haskell: The Craft of Functional Programming*, Addison Wesley, 1996.

[27] C. Urban, A. M. Pitts, and Murdoch J. Gabbay, *Nominal unification*, Theoretical Computer Science **323** (2004), no. 1–3, 473–497.

[28] Johan van Benthem, *Modal foundations for predicate logic*, Logic Journal of the IGPL **5** (1997), no. 2, 259–286.

[29] ———, *Higher-order logic*, Handbook of Philosophical Logic, 2nd Edition (D.M. Gabbay and F. Guenthner, eds.), vol. 1, Kluwer, 2001, pp. 189–244.