

# Structural Recursion over Contextual Objects

Brigitte Pientka and Sherry Shanshan Ruan

School of Computer Science

McGill University

Montreal, Canada

bpientka@cs.mcgill.ca, shanshan.ruan@mail.mcgill.ca

Andreas Abel

Department of Computer Science and Engineering

Chalmers and Gothenburg University

Gothenburg, Sweden

andreas.abel@gu.se

**Abstract**—A core programming language is presented that allows structural recursion over open LF objects and contexts. The main technical tool is a coverage checking algorithm that also generates valid recursive calls. Termination of call-by-value reduction is proven using a reducibility semantics. This establishes consistency and allows the implementation of proofs about LF specifications as well-founded recursive functions using simultaneous pattern matching.

## I. INTRODUCTION

The logical framework LF [Harper et al., 1993] supports concise and elegant specifications of formal systems and proofs based on higher-order abstract syntax (HOAS) where we model binders in the object language using binders in LF. This provides a general treatment of syntax, rules and proofs where both variables and assumptions in the object language are represented uniformly using LF’s function space and allows for a smooth treatment of variables together with discharging and replacing them.

While the elegance of higher-order abstract syntax encodings is widely acknowledged, it has been challenging to reason inductively about LF specifications and formulate well-founded recursion principles. HOAS specifications are not inductive in the standard sense, since they violate the positivity restriction. Despeyroux et al. [1997], Schürmann et al. [2001], and Despeyroux and Leleu [2001] propose to separate the LF function space used for representing HOAS from the function space of computations used for writing proofs about HOAS representations using the modal necessity  $\Box$ . Their calculus, a modal lambda-calculus with a primitive recursive iterator, allows the representation of proofs as primitive recursive functions. Hofmann [1999] investigated a categorical explanation for the proposed reasoning principles. These calculi only handle closed LF objects. As we recursively traverse higher-order abstract syntax trees, we however extend our context of assumptions and our LF object does not remain closed. To tackle this problem, Pientka and collaborators [Pientka, 2008, Cave and Pientka, 2012] propose to pair LF objects together with the context in which they are meaningful. This notion is then internalized as a contextual type  $[\Psi.A]$  which is inhabited by terms  $M$  of type  $A$  in the context  $\Psi$  [Nanevski et al., 2008]. Contextual objects are then embedded into a computation language which supports pattern matching on contexts and contextual objects and general recursion. Beluga, a programming environment based on these ideas [Pientka and

Dunfield, 2010], has been used for a wide range of applications such as encoding normalization-by-evaluation [Cave and Pientka, 2013] and a type-preserving compiler including closure conversion and hoisting [Belanger et al., 2013]. However, Beluga lacks guarantees that a given program is total.

In this paper, we present a dependently typed core language for reasoning about contexts and contextual objects which instead of general recursion uses a structural recursion principle for contexts and contextual objects based on a simultaneous well-founded pattern matching construct. It provides a type-theoretic foundation for Beluga restricted to well-founded recursive programs.

Unlike the general matching construct which simply splits a given object into different cases, it in addition introduces and assumes valid recursive calls based on a specified invariant. This dynamic generation of structural recursive calls is in contrast to a generic induction principle which is statically derived from the inductive definition as for example in Coq. Since obviously impossible cases are not even generated, this reduces the number of necessary cases which need to be considered in practice. Our type system will not only guarantee that we are manipulating well-typed objects but also that a given set of cases is covering and recursive calls are well-founded. Our coverage and termination analysis builds on [Schürmann and Pfenning, 2003, Dunfield and Pientka, 2009] and provides the basis of an interactive theorem prover which uses the splitting and recursive call generation.

Closely related to our approach is the work by Schürmann and Pfenning [1998] and Schürmann [2000] where the authors present a meta-logic  $\mathcal{M}^2$  for reasoning about LF specifications and describe the generation of splits and structural recursive calls. However,  $\mathcal{M}^2$  does not support higher-order computations and lacks first-class contexts, i.e., all assumptions live in an ambient context. This makes it less direct to justify reasoning with assumptions and less expressive compared to our core language which naturally supports higher-order functions and provides a proof language for first-order logic with contextual LF as a domain.

To establish consistency, we define a call-by-value small-step semantics for our core language and prove that every well-typed program terminates. This justifies the interpretation of well-founded recursive programs in our core language as inductive proofs. Our proof of normalization follows Tait’s method of logical relations. Such a proof is missing in the

work by Schürmann.

The remainder of the paper is organized as follows. We review and summarize contextual LF [Cave and Pientka, 2012] in Section II. We then present the core language in Section III which includes well-founded recursion principle and simultaneous pattern matching and illustrate how to write programs in this language. The operational semantics is given in Section IV together with the normalization proof are given in Section V. The rest of the paper is concerned with some related work, current status and future research directions.

## II. BACKGROUND

We review contextual LF which is based on contextual modal types [Nanevski et al., 2008] and its extension with context variables in [Pientka, 2008, Cave and Pientka, 2012]. We subsequently embed contextual LF objects into a (dependently-typed) functional language which supports simultaneous pattern matching and well-founded recursion over LF objects.

LF Constants	$c$
LF Variables	$x, y, z$
Parameter Variables	$p$
Meta Variables	$u$
Context Variables	$\psi, \phi$
LF Base Types	$P, Q ::= c \cdot S$
LF Types	$A, B ::= P \mid \Pi x:A.B$
Heads	$H ::= c \mid x \mid p[\sigma]$
Neutral Terms	$R ::= H \cdot S \mid u[\sigma]$
Spines	$S ::= \text{nil} \mid M S$
Normal Terms	$M, N ::= R \mid \lambda x.M$
Substitutions	$\sigma ::= \cdot \mid \text{id}_\psi \mid \sigma, M \mid \sigma; H$
Variable Substitutions	$\pi ::= \cdot \mid \text{id}_\psi \mid \sigma; x$
LF Contexts	$\Psi, \Phi ::= \cdot \mid \psi \mid \Psi, x:A$

Fig. 1. Grammar for Contextual LF

### A. Contextual LF

Contextual LF extends the logical framework LF [Harper et al., 1993] with contextual objects  $\hat{\Psi}.M$  of type  $[\Psi.A]$ . Expression  $M$  denotes an object which may refer to the bound variables listed in  $\hat{\Psi}$  and has type  $A$  in the context  $\Psi$ . The variable list  $\hat{\Psi}$  can be obtained from the context  $\Psi$  by simply dropping the type annotations and keeping only the declared variable names. We consider only objects in  $\eta$ -long  $\beta$ -normal form, since these are the only meaningful objects in LF. Furthermore, we concentrate here on characterizing well-typed terms; spelling out kinds and kinding rules for types is straightforward.

$\Delta; \Psi \vdash h \Rightarrow A$  Synthesize type  $A$  for head  $h$

$$\frac{\Psi(x) = A}{\Delta; \Psi \vdash x \Rightarrow A} \quad \frac{\Sigma(c) = A}{\Delta; \Psi \vdash c \Rightarrow A}$$

$$\frac{\Delta(p) = \#\Phi.A \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash p[\sigma] \Rightarrow [\sigma]A}$$

$\Delta; \Psi \vdash S : A > P$  Check spine  $S$  against  $A$  with target  $P$

$$\overline{\Delta; \Psi \vdash \text{nil} : P > P}$$

$$\frac{\Delta; \Psi \vdash M \Leftarrow A \quad \Delta; \Psi \vdash S : [M/x]B > P}{\Delta; \Psi \vdash M S : \Pi x:A.B > P}$$

$\Delta; \Psi \vdash M \Leftarrow A$  Check normal object  $M$  against type  $A$

$$\frac{\Delta; \Psi, x:A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.M \Leftarrow A \rightarrow B}$$

$$\frac{\Delta(u) = \Phi.P \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad Q = [\sigma]P}{\Delta; \Psi \vdash u[\sigma] \Leftarrow Q}$$

$$\frac{\Delta; \Psi \vdash h \Rightarrow A \quad \Delta; \Psi \vdash S : A > P}{\Delta; \Psi \vdash h \cdot S \Leftarrow P}$$

$\Delta; \Psi \vdash \sigma \Leftarrow \Phi$  Check substitution  $\sigma$  against domain  $\Phi$

$$\overline{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \quad \overline{\Delta; (\psi, \Psi^0) \vdash \text{id}_\psi \Leftarrow \psi}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]A}{\Delta; \Psi \vdash (\sigma, M) \Leftarrow (\Phi, x:A)}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash H \Rightarrow B \quad B = [\sigma]A}{\Delta; \Psi \vdash (\sigma; H) \Leftarrow (\Phi, x:A)}$$

Fig. 2. Bi-directional typing for contextual LF

Normal terms are either lambda-abstractions or neutral terms which are defined using a spine representation to give us direct access to the head of a neutral term. Normal objects may contain *ordinary bound variables*  $x$  which are used to represent object-level binders and are bound by  $\lambda$ -abstraction or in a context  $\Psi$ . They may also contain *contextual variables*; these are meta-variables  $u[\sigma]$  which are placeholders for LF terms and parameter variables  $p[\sigma]$  which are placeholders for LF variables. Contextual variables are associated with a postponed substitution  $\sigma$  which is applied as soon as we instantiate it. More precisely, a meta-variable  $u$  stands for a contextual object  $\hat{\Psi}.R$  where  $\hat{\Psi}$  describes the ordinary bound variables which may occur in  $R$ . This allows us to rename the free variables occurring in  $R$  when necessary. The parameter variable  $p$  stands for a contextual object  $\hat{\Psi}.H$  where  $H$  must be either an ordinary bound variable from  $\hat{\Psi}$  or another parameter variable.

In the simultaneous substitutions  $\sigma$ , we do not make its domain explicit. Rather we think of a substitution together

with its domain  $\Psi$  and the  $i$ -th element in  $\sigma$  corresponds to the  $i$ -th declaration in  $\Psi$ . We have two different ways of building a substitution entry: either by using a normal term  $M$  or a variable  $x$ . Note that a variable  $x$  is only a normal term  $M$  if it is of base type. However, as we push a substitution  $\sigma$  through a  $\lambda$ -abstraction  $\lambda x.M$ , we need to extend  $\sigma$  with  $x$ . The resulting substitution  $\sigma, x$  may not be well-formed, since  $x$  may not be of base type and in fact we do not know its type. Hence, we allow substitutions not only to be extended with normal terms  $M$  but also with variables  $x$ ; in the latter case we write  $\sigma; x$ . Expression  $\text{id}_\psi$  denotes the identity substitution with domain  $\psi$  while  $\cdot$  describes the empty substitution.

As is common, we rely on hereditary substitutions [Watkins et al., 2003], written as  $[N/x]B$  (or  $[\sigma]B$ ) to guarantee that when we substitute a term  $N$  for the variable  $x$  in the type  $B$ , we obtain a type  $B'$  which is in normal form. Hereditary substitutions continue to substitute, if a redex is created; for example, when replacing naively  $x$  by  $\lambda y.c y$  in the object  $x z$ , we would obtain  $(\lambda y.c y) z$  which is not in normal form and hence not a valid term in our grammar. Hereditary substitutions continue to substitute  $z$  for  $y$  in  $c y$  to obtain  $c z$  as a final result. For a more detailed description of hereditary substitution, we refer the reader to Nanevski et al. [2008].

An LF context  $\Psi$  is either a list of bound variable declarations  $x : A$  or a context variable  $\psi$  followed by such a list. We write  $\Psi^0$  for contexts that do not start with a context variable. We write  $\Psi, \Phi^0$  or sometimes  $\Psi, \Phi$  for the extension of context  $\Psi$  by the variable declarations of  $\Phi^0$  or  $\Phi$ , resp. The identity substitution  $\text{id}(\Psi)$  for a given context  $\Psi$  is defined inductively as follows:

$$\text{id}(\cdot) = \cdot \quad \text{id}(\psi) = \text{id}_\psi \quad \text{id}(\Psi, x:A) = \text{id}(\Psi); x$$

We require the usual conditions on bound variables, tacitly apply  $\alpha$ -renaming and maintain that contexts declare no variable more than once. Note that substitutions  $\sigma$  are defined only on ordinary variables  $x$  and not on contextual variables. We use a special symbol  $\#$  to indicate the type of a parameter variable.

We summarize the bi-directional type system for contextual LF in Figure 2. LF objects may depend on variables declared in the context  $\Psi$  and a fixed meta-context  $\Delta$  which contains contextual variables such as meta-variables  $u$ , parameter variables  $p$ , and context variables  $\psi$ . We introduce  $\Delta$  more formally in the next section. All typing judgments have access to both contexts and a fixed well-typed signature  $\Sigma$  where we store constants  $c$  together with their types and kinds.

A remark on equality checking: When checking  $A = B$  we must take into account  $\eta$ -contraction, because we have two ways to build substitutions. If  $x$  has type  $\Pi y.A.B$  then we may have written  $\sigma; x$  or  $\sigma, \lambda y.x y$ .

### B. Meta-level terms and typing rules

We lift contextual LF objects to meta-objects to have a uniform definition of all meta-objects. Meta-objects (both contextual objects  $\hat{\Psi}.R$  and contexts  $\Psi$ ) can be used to index computation-level types  $\tau$  (see next section). Just as

$\boxed{\Delta \vdash C : U}$  Check meta-object  $C$  against meta-type  $U$

$$\frac{}{\Delta \vdash \cdot : G} \quad \frac{\Delta(\psi) = G}{\Delta \vdash \psi : G}$$

$$\frac{\Delta \vdash \Psi : G \quad \exists \Phi^0. B \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi^0 \quad [\sigma]B = B'}{\Delta \vdash \Psi, x:B' : G}$$

$$\frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \hat{\Psi}.R : \Psi.P}$$

$$\frac{\Psi(x) = A}{\Delta \vdash \hat{\Psi}.x : \#\Psi.A} \quad \frac{\Delta(p) = \#\Phi.A \quad \Delta; \Psi \vdash \pi \Leftarrow \Phi}{\Delta \vdash \hat{\Psi}.p[\pi] : \#\Psi.A}$$

$\boxed{\Delta \vdash \theta : \Delta'}$  Check meta-substitution  $\theta$  against domain  $\Delta'$

$$\frac{}{\Delta \vdash \cdot : \cdot} \quad \frac{\Delta \vdash \theta : \Delta' \quad \Delta \vdash C : \llbracket \theta \rrbracket U}{\Delta \vdash \theta, C/X : \Delta', X:U}$$

Fig. 3. Typing for meta-objects

types classify terms, context schemas  $G$  classify contexts. For simplicity, we restrict schemas to classify only contexts containing LF types.<sup>1</sup>

Context Schemas	$G ::= \exists \Phi^0. B \mid G + \exists \Phi^0. B$
Meta Types	$U ::= \Psi.P \mid \#\Psi.A \mid G$
Meta Objects	$C, D ::= \hat{\Psi}.R \mid \Psi$
Meta Substitutions	$\rho, \theta ::= \cdot \mid \theta, C/X$
Meta Contexts	$\Delta ::= \cdot \mid \Delta, X:U$

A consequence of the uniform treatment of meta-terms is that the design of the computation language is modular and parametrized over meta-terms and meta-types. This has two main advantages: First, we can in principle easily extend meta-terms and meta-types without affecting the computation language; Second, it will be key to a modular, clean design of our computation language.

The above definition gives rise to a compact treatment of meta-context  $\Delta$ . A meta-variable  $X$  can denote a meta-variable  $u$ , a parameter variable  $p$ , or a context variable  $\psi$ . Meta substitution  $C/X$  can represent  $\hat{\Psi}.R/u$ , or  $\Psi/\psi$ , or  $\hat{\Psi}.x/p$ , or  $\hat{\Psi}.p'[\pi]/p$  (where  $\pi$  is a variable substitution so that  $p[\pi]$  always produces a variable). Meta declaration  $X:U$  can stand for  $u : \Psi.P$ , or  $p : \#\Psi.A$ , or  $\psi : G$ . Intuitively, as soon as we replace  $u$  with  $\hat{\Psi}.R$  in  $u[\sigma]$ , we apply the substitution  $\sigma$  to  $R$  hereditarily. The simultaneous meta-substitution, written as  $\llbracket \theta \rrbracket$ , is a straightforward extension of the single substitution. For a full definition of meta-substitutions, we refer the reader to Nanevski et al. [2008], Cave and Pientka [2012]. Find the typing rules for meta-objects summarized in Figure 3.

*Theorem 1 (Meta-substitution property):*

If  $\Delta' \vdash \theta : \Delta$  and  $\Delta; \Psi \vdash J$  then  $\Delta'; \llbracket \theta \rrbracket \Psi \vdash \llbracket \theta \rrbracket J$ .

<sup>1</sup>In practice, we support a limited notion of  $\Sigma$ -types.

$\Lambda\psi:(\text{ctx}).\Lambda A:(\text{tp}).\text{fn } x:[\psi.\text{tm } A].\text{let } X = x \text{ in } \text{rec}^T(\psi.X \dots)$ with $  \phi:(\text{ctx}), B:(\text{tp}), p:(\#\phi.B)$ $.f [\phi] [.B] [\phi.p \dots]$ $  \phi:(\text{ctx}), B_1:(\text{tp}), B_2:(\text{tp}), M:(\phi, x:\text{tm } B_1.\text{tm } B_2)$ $.f [\phi] [\phi.\text{arr } B_1 B_2] [\phi.\text{lam } \lambda x. M \dots x]$  $  \phi:(\text{ctx}), B_1:(\text{tp}), B_2:(\text{tp}), M:(\phi.\text{tm } (\text{arr } B_1 B_2)), N:(\phi.\text{tm } B_1)$ $.f [\phi] [.B_2] [\phi.\text{app } (M \dots) (N \dots)]$	$; \cdot$ $\mapsto [.z]$ $; f [\phi, x:\text{tm } B_1] [.B_2] [\phi, x:\text{tm } B_1.M \dots x]$ $\mapsto \text{let } N = f [\phi, x:\text{tm } B_1] [.B_2] [\phi, x:\text{tm } B_1.M \dots x]$ $\text{in } [\text{suc } N]$ $; f [\phi] [\phi.\text{arr } B_1 B_2] [\phi.M \dots], f [\phi] [.B_1] [\phi.N \dots]$ $\mapsto \text{let } X = f [\phi] [\phi.\text{arr } B_1 B_2] [\phi.M \dots] \text{ in}$ $\text{let } Y = f [\phi] [.B_1] [\phi.N \dots] \text{ in}$ $\text{let } Z = \text{plus } [X] [Y] \text{ in}$ $[\text{suc } Z]$
--	---

Fig. 4. Counting abstractions and applications in a lambda-term

### III. CORE LANGUAGE WITH WELL-FOUNDED RECURSION

In this section, we present the core of Beluga’s computational language which allows the manipulation contextual LF objects by means of higher-order functions and primitive recursion `rec` over such objects. In terms of proof-theoretical strength, the language is comparable to Gödel’s T or Heyting Arithmetic, only that the objects of study are not natural numbers, but HOAS terms.

Types	$\tau ::= [U] \mid \tau_1 \rightarrow \tau_2 \mid \Pi X:U.\tau$
Expressions	$e ::= y \mid [C] \mid \text{fn } y:\tau. e \mid e_1 e_2 \mid \Lambda X:U.e \mid e [C]$ $\mid \text{let } X = e_1 \text{ in } e_2 \mid \text{rec}^{\Pi\Delta.\tau} C \text{ with } \vec{b}$
Branches	$b ::= \Delta; \vec{r}. r \mapsto e$
Pattern	$r ::= f [\vec{C}] [C]$
Contexts	$\Gamma ::= \cdot \mid \Gamma, y:\tau$

There are three forms of computation-level types  $\tau$ . The base types  $[U]$  are introduced by wrapped contextual objects  $[C]$ ; the non-dependent function space  $\tau_1 \rightarrow \tau_2$  is introduced by function abstraction `fn  $y:\tau_1.e$`  and eliminated by application  $e_1 e_2$ ; finally, the dependent function type  $\Pi X:U.\tau$  which corresponds to universal quantification  $\forall$  in predicate logic or Heyting Arithmetic is introduced by abstraction  $\Lambda X:U.e$  over meta object variables  $X$  and eliminated by application  $e [C]$  to meta objects  $C$ . The type annotations on both abstractions ensure that every expression has a unique type. Note that we can only index computation-level types  $\tau$  by meta objects (but this includes LF contexts!), not by arbitrary computation-level objects. Thus, the resulting logic is just first-order, although the proofs we can write correspond to higher-order functional programs manipulating HOAS objects.

Our language supports simultaneous pattern matching on meta-objects  $C$  using `rec`-expressions. Note that one cannot match on a computational object  $e$  directly; instead one can bind it to a meta variable  $X$  using `let` and then match on  $X$ . We annotate the recursor `rec` with the type of the inductive invariant  $\Pi\Delta.\tau$  which the recursion satisfies. Since we are working in a dependently-typed setting, it is not sufficient to simply state the type  $U$  of the scrutinee. Instead, we generalize over the index variables occurring in the scrutinee, since they may be refined during pattern matching. Hence,  $\Delta = \Delta', X:U$

where  $\Delta'$  exactly describes the free meta-variables occurring in  $U$ . We also give the return type  $\tau$  of the recursor, since it might be refined during pattern matching.<sup>2</sup>

A branch  $b_i$  is expressed as  $\Delta_i.\vec{r}_i.r_{i0} \mapsto e_i$ . We explicitly list all meta-variables occurring in a branch in  $\Delta_i$ . In practice, they often can be inferred (see for example [Pientka, 2013]). We also list all valid well-founded recursive calls  $\vec{r}_i$ , i.e.  $r_{i0}, \dots, r_{i1}$ , for pattern  $r_{i0}$ . In practice, they can be derived dynamically as we check that a given pattern  $r_{i0}$  is covering. We list them here to emphasize these recursive assumptions.

In larger examples, we use the following layout for each of the branches  $\Delta; \vec{r}. r \mapsto e$ :

$\mid \Delta$ (bound metas)	; $\vec{r}$ (recursive calls)
. $r$ (considered case)	$\mapsto e$ (body)

The identifier  $f$  in call patterns  $r$  denotes the local function that is essentially introduced by `rec`; this notation is inspired by primitive recursion in *Tutch* [Abel, 2002]. Currently, it just improves the readability of call patterns; however, it is vital for extensions to nested recursion.

#### A. Examples

In this section, we give several examples to illustrate how to program recursively in this language. For better readability, we write capital letters for meta-variables and adopt Beluga’s syntax for identity substitutions writing `...` instead of `[id $_{\psi}$ ]`. All our examples show total programs over intrinsically well-typed terms which are defined in the logical framework LF by indexing terms with their corresponding type. We also assume that we have defined an LF type `nat` of natural numbers with the constructors `z` and `suc`. Below is a signature, written in Beluga or Twelf syntax.

<code>tp</code> : type.	<code>tm</code> : <code>tp</code> $\rightarrow$ type.
<code>bool</code> : <code>tp</code> .	<code>app</code> : <code>tm</code> ( $A \rightarrow B$ ) $\rightarrow$ <code>tm</code> $A \rightarrow$ <code>tm</code> $B$ .
<code>arr</code> : <code>tp</code> $\rightarrow$ <code>tp</code> $\rightarrow$ <code>tp</code> .	<code>lam</code> : ( <code>tm</code> $A \rightarrow$ <code>tm</code> $B$ ) $\rightarrow$ <code>tm</code> ( <code>arr</code> $A B$ ).

Free variables,  $A$  and  $B$  are implicitly quantified at the outside. Subsequently, we will omit passing instantiations for them when using constructors such as `app` and `lam`, since they can be reconstructed [Pientka, 2013].

<sup>2</sup>This is analogous to Coq’s `match_as_in_return_with_end` construct.

a) *Counting constructors*: In Figure 4, we present a function which counts the number of constructors (abstractions and applications) in a given term. It naturally has type

$$T = \Pi\psi:(\text{ctx}).\Pi A:(\text{tp}).[\psi.\text{tm } A] \rightarrow [\text{nat}]$$

where  $\text{ctx}$  describes the schema  $\exists A:\text{tp}.\text{tm } A$  for contexts containing declarations  $x : \text{tm } A$  for some  $A$ . After introducing the context variable  $\psi$  and the meta-variable  $A$  using  $\Lambda$ -abstraction, we introduce  $x$  of type  $[\psi.\text{tm } A]$  via a computation-level abstraction. To recurse over  $x$ , we first unbox it using  $\text{let}$  and bind it to the contextual variable  $X$ . We then split and recurse over  $(\psi.X \dots)$ . There are three cases we must consider: either  $X$  denotes a variable, written as  $f [\phi] [.B] [\phi.p \dots]$ , or it stands for a lambda-abstraction, written as  $f [\phi] [.\text{arr } B_1 B_2] [\phi.\text{lam } \lambda x. M \dots x]$ , or it describes an application, written as  $f [\phi] [.B_2] [\phi.\text{app } (M \dots) (N \dots)]$ .

In the case for lambda-abstraction, we recurse on  $M$ , written as  $f [\phi, x:\text{tm } B_1] [.B_2] [\phi, x:\text{tm } B_1.M \dots x]$ . This recursive call is justified as  $\phi, x:\text{tm } B_1.M \dots x$  is considered smaller than  $\phi.\text{lam } \lambda x. M \dots x$ . We define a subterm ordering on contextual types later. Intuitively, the size of a contextual object is the size of the context in which it makes sense and the size of the term which is defined by how many  $\lambda$ -abstractions and applications it contains.

In the case for application, we have two recursive calls, one on  $M$ , written as  $f [\phi] [.\text{arr } B_1 B_2] [\phi.M \dots]$ , and the other on  $N$ , written as  $f [\phi] [.B_1] [\phi.N \dots]$ .

May we also write a program which is defined recursively over  $\psi.\text{tm } \text{bool}$  instead of  $\psi.\text{tm } T$ ? - The answer is yes. In such a case, we only generate cases whose patterns have type  $\psi.\text{tm } \text{bool}$ , i.e., the variable case and the case for applications. What well-founded recursive calls are generated? In fact none, since given  $[\phi.\text{app } (M \dots) (N \dots)]$  we have that  $M$  has type  $\phi.\text{tm } (\text{arr } B_1 \text{bool})$  and  $N$  has type  $\phi.\text{tm } B_1$ . Neither of which is a possible instance of  $\phi.\text{tm } \text{bool}$ , the type required by the invariant.

b) *Computing the length of a context*: We illustrate recursion over the context to compute its length by writing a function of type  $T = \Pi\psi:(\text{ctx}).[\text{nat}]$ . There are two possible cases to consider: either  $\psi$  is empty or  $\psi = \phi, x:\text{tm } B$ . In the latter case,  $\phi$  is smaller than  $\psi$  and hence we assume  $f [\phi]$  as a well-founded recursive call.

$$\begin{array}{l} \Lambda\psi:(\text{ctx}).\text{rec}^T(\psi) \text{ with} \\ | \cdot \quad ; \quad \cdot \\ | f [\cdot] \quad \mapsto \quad [z] \\ | \phi:(\text{ctx}), B:(\text{tp}) \quad ; \quad f [\phi] \\ | f [\phi, x:\text{tm } B] \quad \mapsto \quad \text{let } N = f [\phi] \text{ in } [.\text{suc } N] \end{array}$$

c) *Conversion to de Bruijn*: The function in Figure 5 converts well-typed lambda-terms to de Bruijn terms of the same type. The central difficulty lies in returning the position of a variable in a given context, which is its de Bruijn index. The example illustrates recursion over elements of type  $\#\psi.\text{tm } A$ . Note that simultaneously matching on  $\psi$  and  $\#\psi.\text{tm } A$  allows us to split first  $\psi$  followed by a split on

$\#\psi.\text{tm } A$ . The case  $\psi = \cdot$  (i.e.,  $\psi$  is empty) and  $\# \cdot.\text{tm } A$  is impossible, since there is no variable.

d) *Impossible cases*: A brief note regarding impossible cases. If a type is not inhabited, i.e. we cannot generate a covering set, then this case is impossible, i.e. there is nowhere to step. Effectively such case is irrelevant during run-time, but it is proof-relevant. This amounts to writing a recursive simultaneous match without a body.

## B. Well-founded structural subterm order

There are two key ingredients to guarantee that a given function is total: we need to ensure that all the recursive calls are on smaller arguments according to a well-founded order and the function covers all possible cases. We define here a well-founded structural subterm order on contexts and contextual objects similar to the subterm relations for LF objects [Rohwedder and Pfenning, 1996, Pientka, 2005].

To consider also mutual recursive type families, we define the notion of subordination. Let  $\mathbf{hd}(A)$  denote the head of a clause  $A$ , i.e. the overall return type. A type family  $a$  is a subordinate to a type family  $a'$  ( $a \triangleleft^* a'$ ) whenever a canonical term  $M:A$  with  $\mathbf{hd}(A) = a$  may be used in constructing a canonical term  $N:B$  with  $\mathbf{hd}(B) = a'$ . If additionally  $a' \triangleleft^* a$ , we say that  $a, a'$  are mutually recursive. We write  $a \triangleleft^* a'$ , denoting strict subordination, if  $a$  is a subordinate to  $a'$ , but not mutually recursive. Subordination of type families is the transitive closure of the immediate subordination relation ( $a \triangleleft^* a'$ ) which can be directly read off the signature. If the type family  $a$  ( $\mathbf{hd}(A)$ ) is a strict subordinate of the type family  $a'$  ( $\mathbf{hd}(A')$ ), then a canonical subterm of type  $A$  can never contain a subterm of type  $A'$ . Therefore, a term  $M$  is considered smaller than a  $\lambda$ -term  $(\lambda x.N)$  if there exists an arbitrary instantiation  $T$  for  $x$  s.t.  $M$  is smaller than  $[T/x]N$  and the type of  $T$  is a strict subordinate to  $N$ . An example of this strict subordination can be found in the representation of first-order logic, where the objects of type  $\mathfrak{i}$  (individuals) are a strict subordinate of the objects of type  $\circ$  (propositions).

If the type family  $a$  ( $\mathbf{hd}(A)$ ) is *not* a strict subordinate of the type family  $a'$  ( $\mathbf{hd}(A')$ ), then  $M$  is only considered smaller than  $\lambda x.N$  if there exists a parameter  $y$  such that  $[y/x]N$  is smaller than  $M$ . For a more detailed development of subordination we refer to R. Virga's PhD thesis Virga [1999].

Type subordination plays a role in handling the subterm ordering for  $\lambda$ -abstractions. In a comparison, if the type of the variable bound by an abstraction is not subordinate to the type of the other expression, then we can simply substitute the bound variable with any variable of the same type declared in the context. We first define an ordering on contexts:  $\boxed{\Psi \preceq \Phi}$ , read as "context  $\Psi$  is a subcontext of  $\Phi$ ", shall hold if all declarations of  $\Psi$  are also present in the context  $\Phi$ , i.e.,  $\Psi \subseteq \Phi$ . The strict relation  $\boxed{\Psi \prec \Phi}$ , read as "context  $\Psi$  is strictly smaller than context  $\Phi$ " holds if  $\Psi \preceq \Phi$  but  $\Psi$  is strictly

$$\begin{array}{l}
\Lambda\psi:(\text{ctx}).\Lambda A:(\text{tp}).\Lambda p:(\#\psi.\text{tm } A).\text{rec}^{\Pi\psi:(\text{ctx}).\Pi A:(\text{tp}).\Pi p:(\#\psi.\text{tm } A).[\text{nat}]}(\psi.p\dots) \text{ with} \\
| \phi:(\text{ctx}), B:(\text{tp}) \quad ; \quad \cdot \\
.f [\phi, x:\text{tm } B] [.B] [\phi, x:\text{tm } B. x] \quad \mapsto \quad [. \text{suc } z] \\
| \phi:(\text{ctx}), B:(\text{tp}), A:(\text{tp}), p:(\#\phi.\text{tm } A) \quad ; \quad f [\phi] [.A] [\phi.p\dots] \\
.f [\phi, x:\text{tm } B] [.A] [\phi, x:\text{tm } B.p\dots] \quad \mapsto \quad \text{let } N = f [\phi] [.A] [\phi.p\dots] \text{ in } [. \text{shift } N]
\end{array}$$

Fig. 5. Conversion to de Bruijn

shorter than  $\Phi$ .

$\boxed{\Psi \prec \Phi}$  Context  $\Psi$  is strictly smaller than  $\Phi$ .

$$\frac{\Psi \prec \Phi}{\Psi \prec \Phi, x:A}$$

$\boxed{\Psi \prec \Phi}$  Context  $\Psi$  is a subcontext of  $\Phi$ .

$$\frac{\Psi \prec \Phi}{\Psi \prec \Phi} \quad \cdot \prec \cdot \quad \frac{\Psi \prec \Phi}{\psi \prec \psi} \quad \frac{\Psi \prec \Phi}{\Psi, x:A \prec \Phi, x:A}$$

Further, we define three relations on contextual objects  $\hat{\Psi}.M$ : a strict subterm relation  $\prec$ , an equivalence relation  $\equiv$ , and an auxiliary relation  $\preceq$ . Our subterm relation is in fact simpler than the subterm relation employed in the Twelf system. To allow mutual recursive definitions and richer subterm relationships, we can as in Twelf incorporate subordination information defining the the subterm relations on simple types in a type-directed manner.

$\boxed{\hat{\Psi}.M \equiv \hat{\Phi}.N}$  Equivalence on contextual objects

$$\frac{\hat{\Psi} \subseteq \hat{\Phi} \quad \hat{\Phi} \subseteq \hat{\Psi} \quad \pi \text{ is a permutation subst. s.t. } M = [\pi]N}{\hat{\Psi}.M \equiv \hat{\Phi}.N}$$

$\boxed{\hat{\Psi}.M \prec \hat{\Phi}.N}$  Strict subterm relation on contextual objects

$$\frac{\hat{\Psi}.M \preceq \hat{\Phi}.N_i \text{ for some } 1 \leq i \leq n}{\hat{\Psi}.M \prec \hat{\Phi}.h \cdot N_1 \dots N_n \text{ nil}}$$

$$\frac{\hat{\Psi} \subseteq \hat{\Phi} \quad \pi \text{ is a strengthening substitution } \pi \text{ s.t. } M = [\pi]N}{\hat{\Psi}.M \prec \hat{\Phi}.N}$$

$\boxed{\hat{\Psi}.M \preceq \hat{\Phi}.N}$  Subterm relation on contextual objects

$$\frac{\hat{\Psi}.M \prec \hat{\Phi}.N \quad \hat{\Psi}.M \equiv \hat{\Phi}.N \quad \hat{\Psi}.M \preceq \hat{\Phi}.x.N}{\hat{\Psi}.M \preceq \hat{\Phi}.N \quad \hat{\Psi}.M \preceq \hat{\Phi}.N \quad \hat{\Psi}.M \preceq \hat{\Phi}.\lambda x.N}$$

Two terms  $\hat{\Psi}.M$  and  $\hat{\Phi}.N$  are structurally equivalent, if they describe the same term modulo  $\alpha$ -renaming. We in addition require that  $\hat{\Psi}$  and  $\hat{\Phi}$  describe the same set of variables and context variables, i.e.  $\hat{\Psi} \subseteq \hat{\Phi}$  and  $\hat{\Phi} \subseteq \hat{\Psi}$ .  $\hat{\Psi}.M$  is a strict subterm of  $\hat{\Phi}.M$  if either  $\hat{\Psi}$  is a subset of  $\hat{\Phi}$  and there exists a strengthening substitution  $\pi$  to guarantee that  $M$  is syntactically equal to  $[\pi]N$ ; in other words  $N$  depends only on  $\hat{\Psi}$  or if  $N = h \cdot N_1 \dots N_n \text{ nil}$  and  $\hat{\Psi}.M$  is smaller than  $\hat{\Phi}.N_i$  for some  $1 \leq i \leq n$ .

We say  $\hat{\Psi}.M \preceq \hat{\Phi}.N$ , if either  $\hat{\Psi}.M$  is strictly smaller than  $\hat{\Phi}.N$ , or if they are equivalent, or if  $N = \lambda x.N'$ , we move  $x$  to the context  $\hat{\Phi}$  and compare  $\hat{\Psi}.N$  with  $\hat{\Phi}, x.N'$ . To check our intuition, consider following example which arose previously:

$$\frac{|\psi, x| = |\psi, x|}{\psi, x.M \dots x \prec \psi, x.M \dots x}$$

$$\frac{\psi, x.M \dots x \prec \psi, x.M \dots x}{\psi, x.M \dots x \prec \psi.\lambda x.M \dots x}$$

$$\psi, x.M \dots x \prec \psi.\lambda x.M \dots x$$

We also note that  $\psi.p\dots \prec \psi, x.p\dots$  is immediately justified by noting that  $\psi \subseteq \psi, x$ . Using the defined subterm order, we can easily verify that the recursive calls in the examples are structurally smaller.

The given subterm relation is well-founded. We define the measure  $\|\Psi\|$  of a ground context  $\Psi^0$  or its erasure  $\hat{\Psi}^0$  as its length  $|\Psi|$ . The measure  $\|\hat{\Psi}.M\|$  of a contextual object  $\hat{\Psi}.M$ , is the measure of  $\hat{\Psi}$  plus the measure  $\|M\|$  of  $M$ .

Measure for erased contexts

$$\begin{aligned} \|\cdot\| &= 0 \\ \|\hat{\Psi}, x\| &= 1 + \|\hat{\Psi}\| \end{aligned}$$

Measure for normal and neutral terms

$$\begin{aligned} \|h \cdot M_1 \dots M_n \text{ nil}\| &= 1 + \max(\|M_1\|, \dots, \|M_n\|) \\ \|\lambda x.M\| &= 1 + \|M\| \end{aligned}$$

*Theorem 2 (Order on contextual objects is well-founded):* Let  $\theta$  be a grounding meta-substitution.

- 1) If  $C \prec C'$  then  $\|\llbracket \theta \rrbracket C\| < \|\llbracket \theta \rrbracket C'\|$ .
- 2) If  $C \equiv C'$  then  $\|\llbracket \theta \rrbracket C\| = \|\llbracket \theta \rrbracket C'\|$ .
- 3) If  $C \preceq C'$  then  $\|\llbracket \theta \rrbracket C\| \leq \|\llbracket \theta \rrbracket C'\|$ .

*Proof.* By mutual structural induction on the relation  $C \prec C'$ ,  $C \equiv C'$ , and  $C \preceq C'$ .  $\square$

*e) Side remark:* Our subterm order for contexts and contextual objects is very similar to first-order subterm ordering and is in fact simpler than the structural subterm ordering on LF terms employed in the Twelf system. In their system, to establish that  $[y/x]M \preceq \lambda x:A.M$ , we need to be able to compare  $[y/x]M \preceq [y/x]M$ , i.e. we must instantiate  $x$  in  $M$  with an existing parameter. This is due to the fact that the order is defined in a shared variable context. In our case, each contextual object has their own surrounding context and we compare two LF objects modulo  $\alpha$ -renaming. As a consequence, we believe our order is more straightforward and natural.

### C. Coverage

For well-formed recursors ( $\text{rec}^{\Delta, X:U, \tau} C$  with  $\vec{b}$ ), branches  $\vec{b}$  need to cover all different cases for the argument  $C$  of type  $U$ . We only take the shape of  $U$  into account and generate the unique complete set  $\mathcal{U}_{\Delta \vdash U}$  of non-overlapping shallow patterns by splitting meta-variable  $X$  of type  $U$ . If  $U = \Psi.P$  is a base type, then the set  $\mathcal{U}_{\Delta \vdash U}$  contains all neutral terms  $R = H \cdot P$  type  $P$  in context  $\Psi$  where  $H$  is a constructor  $c$ , a variable  $x$  from  $\Psi$  or a valid parameter  $p[\sigma]$ , and  $S$  is a spine of fresh meta variables. Such terms  $R$  correspond to the “patterns” in functional programming, but there only  $H = c$  needs to be considered. If  $U$  denotes a context schema  $G$ , we generate all shallow context patterns of type  $G$ . If  $U = \# \Psi.A$  is a parameter type, we take the variables of type  $A$  in  $\Psi$ .

From  $\mathcal{U}_{\Delta \vdash U}$  we generate the complete minimal set  $\mathcal{C} = \{\Delta_i; r_{ik}, \dots, r_{i1}.r_{i0} \mid 1 \leq i \leq n\}$  of possible, non-overlapping cases where the  $i$ -th branch shall have the well-founded recursive calls  $r_{ik}, \dots, r_{i1}$  for the case  $r_{i0}$ . For the given branches  $\vec{b}$  to be covering, each element in  $\mathcal{C}$  must correspond to one branch  $b_i$ . Our algorithm generalizes previous work on coverage of LF objects [Dunfield and Pientka, 2009, Schürmann and Pfenning, 2003]. In the following, we detail the cases for split type  $U$ .

*a) Splitting on a contextual type:* The patterns  $R$  of type  $\Psi.P$  are computed by brute force: We first synthesize a set  $\mathcal{H}_{\Delta; \Psi}$  of all possible heads together with their type: constants  $c \in \Sigma$ , variables  $x \in \Psi$ , and parameter variables if  $\Psi$  starts with a context variable  $\psi$ .

$$\begin{aligned} \mathcal{H}_{\Delta; \Psi} = & \{(\Delta; \Psi \vdash c : A) \mid (c:A) \in \Sigma\} \\ & \cup \{(\Delta; \Psi \vdash x : A) \mid (x:A) \in \Psi\} \\ & \cup \{(\Delta, \overrightarrow{X:\vec{U}}, p:\#(\psi.B'); \Psi \vdash p[\text{id}_\psi] : B') \mid \\ & \quad \Psi = \psi, \Psi^0 \text{ and } \psi:G \in \Delta \text{ and } \exists x : \overrightarrow{A}.B \in G \text{ and} \\ & \quad \text{lower } (\psi.A_i) = (X_i:U_i, M_i) \text{ for all } i, \text{ and} \\ & \quad B' = [\overrightarrow{M/x}]B \} \end{aligned}$$

See Figure 6. Using a head  $H$  of type  $A$  from the set  $\mathcal{H}_{\Delta; \Psi}$ , we then generate, if possible, the most general pattern  $H \cdot S$  whose target type is unifiable with  $P$  in the context  $\Psi$ . We describe unification using the judgement  $\boxed{\Delta; \Psi \vdash Q \doteq P / (\Delta', \theta)}$ . If unification succeeds then  $\llbracket \theta \rrbracket Q = \llbracket \theta \rrbracket P$  and  $\Delta' \vdash \theta : \Delta$ .

The generation of a neutral pattern is accomplished by the judgement  $\boxed{\Delta; \Psi \vdash R : A \Leftarrow P / (\Delta', \theta, R_0)}$  where all the elements on the left side of  $/$  are inputs and the right side is the output, which satisfies  $\Delta' \vdash \theta : \Delta$  and  $\Delta'; \llbracket \theta \rrbracket \Psi \vdash R \Rightarrow \llbracket \theta \rrbracket A$  and  $\Delta'; \llbracket \theta \rrbracket \Psi \vdash R_0 \Leftarrow \llbracket \theta \rrbracket P$ . Moreover,  $R_0 = \llbracket \theta \rrbracket R \vec{M}$  for a list of eta-expanded meta variables  $\vec{M}$  that bring  $R$  down to base type. To generate a neutral term  $R_0$  of the expected base type, we start with head  $H : A$ . As we recursively analyze  $A$ , we generate all the arguments  $H$  is applied to until we reach an atomic type  $Q$ . If  $Q$  unifies with the expected type  $P$ , then generating a most

general neutral term with head  $H$  succeeds.

$$\begin{aligned} \mathcal{U}_{\Delta \vdash \Psi.P} = & \{(\Delta'' \vdash \hat{\Phi}.R : \Phi.P') \mid (\Delta'; \Psi \vdash H : A) \in \mathcal{H}_{\Delta; \Psi} \\ & \text{and } \Delta'; \Psi \vdash H : A \Leftarrow P / (\Delta'', \theta, R) \\ & \text{and } \Phi = \llbracket \theta \rrbracket \Psi \text{ and } P' = \llbracket \theta \rrbracket P \} \end{aligned}$$

*b) Splitting on a context schema:* Spitting a context variable of schema  $G$  generates the empty context and the non-empty contexts  $(\phi, x:B')$  for each possible form of context entry  $\exists \Phi^0.B \in G$ .

$$\begin{aligned} \mathcal{U}_{\Delta \vdash G} = & \{(\Delta \vdash \cdot : G) \} \\ & \cup \{(\Delta, \phi:G, \overrightarrow{X:\vec{U}} \vdash (\phi, x:\overrightarrow{M/x})B) : G) \mid \\ & \quad \exists x : \overrightarrow{A}.B \in G \text{ and} \\ & \quad \text{lower } (\psi.A_i) = (X_i:U_i, M_i) \text{ for all } i \text{ and} \\ & \quad \phi \text{ a fresh context variable } \} \end{aligned}$$

*c) Splitting on a parameter type:* We show how to generate all variables of type  $\# \Psi.A$ . Intuitively, only bound variables  $x:B$  from  $\Psi$  whose type is unifiable with  $A$  inhabit this type; if the context  $\Psi$  contains a context variable  $\psi : G$  we also include all parameter variables of the appropriate type synthesized from  $G$ .

*Definition 3 (Generating variable objects):*

The set  $\mathcal{U}_{\Delta \vdash \# \Psi.A}$  contains variables  $x$  from  $\Psi$  of matching type plus variables and parameters  $\mathcal{U}$  spanning from the context variable  $\psi$  in  $\Psi$ , if any.

$$\begin{aligned} \mathcal{U}_{\Delta \vdash \# \Psi.A} = & \{(\Delta'; \llbracket \theta \rrbracket \Psi \vdash x : \llbracket \theta \rrbracket A) \\ & \mid x:B \in \Psi \text{ and } \Delta; \Psi \vdash B \doteq A / (\Delta', \theta) \} \\ & \cup \mathcal{U} \end{aligned}$$

For  $\mathcal{U}$ , we distinguish two cases.

- $\Psi = \psi$  where  $\psi:G \in \Delta$ :

$$\mathcal{U} = \bigcup_{(\Delta' \vdash \Phi:G) \in \mathcal{U}_{\vdash G}} \mathcal{U}_{\Delta', [\Phi/\psi] \Delta \vdash \# \Phi. [\Phi/\psi] A}$$

- $\Psi = \psi, \Psi^0$  where  $\Psi^0$  is non-empty and  $\psi:G \in \Delta$ .

$$\begin{aligned} \mathcal{U} = & \{(\Delta', p:\#(\psi.\llbracket \theta \rrbracket A); \llbracket \theta \rrbracket \Psi \vdash p[\text{id}_\psi] : \llbracket \theta \rrbracket A) \mid \\ & \quad \Psi = \psi, \Psi^0 \text{ and } \psi:G \in \Delta \text{ and } \exists x : \overrightarrow{A}.B \in G \text{ and} \\ & \quad \text{lower } (\psi.A_i) = (X_i:U_i, M_i) \text{ for all } i, \text{ and} \\ & \quad B' = [\overrightarrow{M/x}]B \text{ and } \Delta; \Psi \vdash B' \doteq A / (\Delta', \theta) \} \end{aligned}$$

Note that unifying  $B'$  and  $A$  will not change the context variable  $\psi$ , i.e.  $\theta$  maps  $\psi$  to itself in  $\Delta'$ . We use this fact in the definition above and simply write  $\psi$  instead of  $\llbracket \theta \rrbracket \psi$  in declaring the type of  $p$  and describing the identity substitution it is associated with. Note that if  $\Psi$  is empty, the type  $\# \Psi.A$  is not inhabited and the set  $\mathcal{U}_{\Delta \vdash \# \Psi.A}$  is empty.

To illustrate, we generate  $\mathcal{U}_{\psi:\text{ctx}, A:(\text{tp}) \vdash \# \psi.\text{tm } A}$ . Since there are no bound variables in the context, we split  $\psi$ , i.e.  $\Psi = \cdot$  and  $\Psi, x:\text{tm } B$ . In the first case, we note that  $\mathcal{U}_{A:(\text{tp}) \vdash \# (\text{tm } A)}$  is empty. In the other case,  $\mathcal{U}_{\phi:\text{ctx}, B:(\text{tp}), A:(\text{tp}) \vdash \# (\phi, x:\text{tm } B.\text{tm } A)}$  contains two elements:

$$\begin{aligned} & \phi:\text{ctx}, A:(\text{tp}) \quad ; \phi, x:\text{tm } A; \vdash x:\text{tm } A \\ & \phi:\text{ctx}, B:(\text{tp}), A:(\text{tp}), p:\#(\phi.\text{tm } A); \phi, x:\text{tm } B; \vdash p[\text{id}_\phi] : \text{tm } A \end{aligned}$$

$\boxed{\text{lower}(\Psi.A) = (X:U, M)}$  Generation of a lowered meta variable

$\text{lower}(\Psi.\overrightarrow{\Pi x:A}.B) = (u : (\Psi, \overrightarrow{x:A}.P), \lambda \vec{x}.u[\text{id}(\Psi)])$  for a fresh meta variable  $u$

$\boxed{\Delta; \Psi \vdash R : A \Leftarrow P / (\Delta', \theta, R_0)}$  Lowering of neutral object  $R : A$  to  $R_0 : \llbracket \theta \rrbracket P$  by appending fresh meta variables.

$$\frac{\Delta; \Psi \vdash Q \doteq P / (\Delta_0, \theta)}{\Delta; \Psi \vdash R : Q \Leftarrow P / (\Delta_0, \theta, \llbracket \theta \rrbracket R)} \quad \frac{\text{lower}(\Psi.A) = (X:U, M) \quad \Delta, X:U; \Psi \vdash R M : [M/x]B \Leftarrow P / (\Delta_0, \theta, R')}{\Delta; \Psi \vdash R : \Pi x:A.B \Leftarrow P / (\Delta_0, \theta, R')}$$

$\boxed{\vdash_{C:U} r : \tau / r'}$  Generation of call pattern  $r'$  through extending  $r$  by eta-expanded meta variables.

$$\frac{\vdash_{C:U} r [C_n] : \llbracket [C_n/X_n] \rrbracket (\Pi \Delta_0.\tau_0) / r' \quad \text{where } C_n = \eta(X_n:U_n)}{\vdash_{C:U} r : \Pi(X_n:U_n, \Delta_0).\tau_0 / r'} \quad \frac{U = U_0[\theta]}{\vdash_{C:U} r : \Pi X_0:U_0.\tau_0 / \llbracket \theta \rrbracket r [C]}$$

Fig. 6. Lowering of neutral object and generation of call patterns

#### D. Generation of call patterns

We introduce the operation  $\eta(X:U) = C$  which returns a proper contextual term  $C$  from a meta variable  $X : U$ .

$$\begin{aligned} \eta(u : \Psi.A) &= \hat{\Psi}.u[\text{id}(\Psi)] \\ \eta(p : \# \Psi.A) &= \hat{\Psi}.p[\text{id}(\Psi)] \\ \eta(\psi : G) &= \psi \end{aligned}$$

*Definition 4 (Generation of call patterns and recursive calls):* Given  $\Delta_0 = \Delta, X_0:U_0$ , the set  $\mathcal{C}$  of call patterns is generated as follows: For each meta-object  $\Delta_i \vdash C_{i0} : U_{i0}$  in  $\mathcal{U}_{\Delta \vdash U_0}$ , we generate, if possible, a call pattern  $r_{i0}$  using

$$\vdash_{C_{i0}:U_{i0}} f : \Pi \Delta_0.\tau_0 / r_{i0}$$

This may fail if  $U_{i0}$  is not an instance of the scrutinee type  $U_0$ ; then, the case  $C_{i0}$  is impossible. Further, for all  $1 \leq j \leq k$ ,  $\Delta_i = X_n:U_n, \dots, X_1:U_1$ , we generate a recursive call

$$\vdash_{X_j:U_j} f : \Pi \Delta_0.\tau_0 / r_{ij}$$

if  $\eta(X_j:U_j) \prec C_{i0}$ . Then  $\boxed{\Delta_i ; r_{ik}, \dots, r_{i1}.r_{i0}}$  is in  $\mathcal{C}$ .

*Definition 5 (Coverage):* We say  $\vec{b}$  covers  $\Delta_0$  iff for every  $\Delta_i ; \vec{r}_i.r_{i0} \in \mathcal{C}$  where  $\mathcal{C}$  is the set of patterns and recursive calls given  $\Delta_0$ , we have one corresponding  $b_i = \Delta_i ; \vec{r}_i.r_{i0} \mapsto e_i \in \vec{b}$  and vice versa.

#### E. Properties of Splitting and Coverage

*Theorem 6 (Splitting on meta-types):* The set  $\mathcal{U}_{\Delta \vdash U}$  of meta-objects generated is non-redundant and complete.

*Proof.*  $\mathcal{U}_{\Delta \vdash G}$  is obviously non-redundant.  $\mathcal{U}_{\Delta \vdash \Psi.A}$  is non-redundant since all generated neutral terms have distinct heads.  $\mathcal{U}_{\Delta \vdash \# \Psi.A}$  is non-redundant, since all declarations in a context  $\Psi$  are distinct. Completeness is proven by cases. We consider here the three cases of meta-objects.

Case Splitting on contextual type  $\Psi.P$ . We need to show that all closed canonical objects  $C$  of type  $\Psi.P$  are covered by the generated splits. Since  $C$  is normal, we know  $C = \hat{\Psi}.R$  and  $R = h M_1 \dots M_n$  s.t.  $\cdot \vdash \theta : \Delta$  and  $\cdot; \llbracket \theta \rrbracket \Psi \vdash R \Leftarrow \llbracket \theta \rrbracket P$ . The set  $\mathcal{H}_{\Delta; \Psi}$  is complete and for all heads  $h$  we have  $h:A \in \mathcal{H}_{\Delta; \Psi}$ .

Moreover, by the properties of unification,  $\Delta; \Psi \vdash h : A \Leftarrow P / (\Delta', \theta', R')$  generates the most general  $R'$  s.t.  $\Delta'; \llbracket \theta' \rrbracket \Psi \vdash R' \Leftarrow \llbracket \theta' \rrbracket P$ . Therefore there exists a meta-substitution  $\rho$  s.t.  $\cdot \vdash \rho : \Delta'$  and  $\llbracket \rho \rrbracket (\llbracket \theta' \rrbracket (\Psi.P)) = \llbracket \theta \rrbracket (\Psi.P)$  and  $\llbracket \rho \rrbracket R' = R$ .

Case Splitting on a context schema  $G$ . We need to show that all closed canonical objects  $C$  of type  $G$  are covered by the generated splits. Since  $C$  is normal, it stands for a concrete context which is either empty or  $\Psi = x_1:B_1, \dots, x_n:B_n$  s.t.  $\cdot \vdash \Psi : G$ . Our splitting definition generates the most general declarations which are instances of the schema  $G$ , i.e. for all  $\exists x:\vec{A}.B \in G$ , we generate  $\Psi' = \psi, x : [u[\text{id}_\psi]/\vec{x}]B$  s.t.  $\psi:G, \overline{u:\psi.A} \vdash \Psi' : G$ . Since it is most general, there exists a meta-substitution  $\rho$  s.t.  $\cdot \vdash \rho : \psi:G, \overline{u:\psi.A}$  s.t.  $\llbracket \rho \rrbracket \Psi' = \Psi$ .

Case Splitting on a parameter type  $\# \Psi.A$ . We need to show that all closed canonical objects  $C$  of type  $\# \llbracket \theta \rrbracket (\Psi.A)$  where  $\cdot \vdash \theta : \Delta$  are covered by the generated splits. Since  $C$  is canonical, it must be of the form  $C = x_1, \dots, x_n.x_i$  where  $1 \leq i \leq n$  and  $\cdot; \llbracket \theta \rrbracket \Psi \vdash x_i \Rightarrow A'$  s.t.  $\llbracket \theta \rrbracket A_i = A'$ . We distinguish two cases. If  $\Psi = x_1:A_1, \dots, x_n:A_n$ , then  $x_i$  has type  $A_i$  and our splitting algorithm guarantees that there exists a most general meta-substitution  $\Delta' \vdash \theta' : \Delta$  s.t.  $\llbracket \theta' \rrbracket A_i = \llbracket \theta' \rrbracket A$ . Since  $\theta'$  is most general, there exists a grounding meta-substitution  $\rho$  s.t.  $\cdot \vdash \rho : \Delta'$  and  $\llbracket \rho \rrbracket (\llbracket \theta' \rrbracket A_i) = A' = \llbracket \theta \rrbracket A$ .

If  $\Psi = \psi, x_i:A_i, \dots, x_n:A_n$ , we also need to consider the case where our algorithm generates for all  $\exists x:\vec{A}.B \in G$ ,  $\psi:G, \overline{u:\psi.A}, p:(\# \psi.B) \vdash \psi.p[\text{id}_\psi] \Rightarrow B$ , if  $B$  unifies with  $A$ . As a consequence there is a most general meta-substitution  $\Delta' \vdash \theta : \Delta, \psi:G, \overline{u:\psi.A}, \# p:\psi.B$ . To generate closed instances of the form  $x_1, \dots, x_n.x_k$  where  $1 \leq k < i$ , we instantiate  $\psi$  with  $x_1:A_1, \dots, x_k:A_k$  and  $p$  with  $x_1, \dots, x_{i-1}.x_k$ .

□



*Theorem 7 (Pattern generation):* The set  $\mathcal{C}$  of call patterns and recursive calls generated is non-redundant and complete and the recursive calls are well-founded.

*Proof.* Using Theorem 6 and the properties of unification.  $\square$

*Theorem 8 (Recursive calls are decreasing):* Given a set  $\mathcal{C}$  of patterns, for each  $\Delta_{ik}; r_{ik}, \dots, r_{i1}.r_{i0}$  in  $\mathcal{C}$ , we have that each recursive calls  $r_{ij}$  ( $1 \leq j \leq k$ ) is smaller than  $r_{i0}$ .

*Proof.* This is true by construction.  $\square$

### F. Computation-level Type System

Our type system for computations performs coverage checking and verifies well-formed primitive recursions in the typing rule for  $\text{rec}$  expressions. We annotate the  $\text{rec}$ -expression with the inductive invariant  $\Pi\Delta_0.\tau_0$  which is a closed computation-level type. The intention is that given  $\Delta_0 = X_n:U_n, \dots, X_0:U_0$ , we induct on objects of type  $U_0$  which may depend on  $X_n, \dots, X_1$ .  $\Delta_0$  must therefore contain at least one declaration. The result of the inductive invariant is  $\tau$  which also might depend on  $\Delta_0$ . One might ask whether this form of inductive invariant is too restrictive, since it seems not to capture, e.g.,  $\Pi\Delta.(\tau \rightarrow \Pi X:U.\tau')$ . While allowing more general invariant and supporting pattern matching on computation types do not pose any fundamental issues, we simply note here that above type is isomorphic to  $\Pi(\Delta, X:U).\tau \rightarrow \tau'$  which is treated by our calculus.

In the typing judgement (Figure 7), we distinguish between the context  $\Delta$  for contextual variables from our index domain and the context  $\Gamma$  which includes declarations of computation-level variables. Contextual variables will be introduced via  $\Lambda$ -abstraction. The contextual variables in  $\Delta$  are also introduced in the branch of a  $\text{rec}$ -expression. Computation-level variables in  $\Gamma$  are introduced by non-dependent function abstraction.

We will tacitly rename bound variables, and maintain that contexts declare no variable more than once. Moreover, we require the usual conditions on bound variables. For example in the rule for  $\Lambda$ -abstraction the contextual variable  $X$  must be new and cannot already occur in the context  $\Delta$ . This can be always achieved via  $\alpha$ -renaming. Similarly, in the rule for function abstraction, the variable  $y$  must be new and cannot already occur in  $\Gamma$ .

Most typing rules should look familiar. We only draw attention to a few rules; a  $\Lambda$ -abstraction has type  $\Pi X:U.\tau$  if the body of the abstraction has type  $\tau$  in the extended meta-context  $\Delta, X:U$ . A function  $\text{fn } y:\tau. e$  has type  $\tau_1 \rightarrow \tau_2$ , if the body  $e$  has type  $\tau_2$  in the extended computation context  $\Gamma, y:\tau_1$ . We have two rules for applications: a non-dependent application  $(e_1 e_2)$  has type  $\tau$ , if  $e_1$  has type  $\tau_2 \rightarrow \tau$  and  $e_2$  has type  $\tau_2$ . For the dependent application  $e [C]$  to be well-typed,  $e$  must have type  $\Pi X:U.\tau$  and  $c$  must be a well-typed meta-object of type  $U$ . Note that we drop the computation context  $\Gamma$  when we transition to type check a meta-object, since meta-objects cannot refer to computations. The final type for  $e [C]$  is  $\llbracket C/X \rrbracket \tau$ .

Interesting is the rule for recursion: the expression  $\text{rec}^{\Pi\Delta_0.\tau_0} C$  with  $\vec{b}$  is well-typed under three conditions: First, the meta-object  $C$  we are recursing over has some type  $U$  and moreover,  $U$  is an instance of the type specified in the invariant, i.e.  $\Delta_0 = \Delta_1, X_0:U_0$  and  $U = \llbracket \theta \rrbracket U_0$  for some meta-substitution  $\theta$  with domain  $\Delta_1$ . Secondly, all branches  $b_i$  are well-typed with respect to the given invariant  $\Pi\Delta_0.\tau_0$ . Finally,  $\vec{b}$  must cover the meta-context  $\Delta_0$ , i.e., it must be a complete, non-redundant set of patterns covering  $\Delta_0$ .

Note that we drop the meta-context  $\Delta$  and the computation context  $\Gamma$  when we proceed to check that all branches satisfy the specified invariant. Dropping  $\Delta$  is fine, since we require the invariant  $\Pi\Delta_0.\tau_0$  to be closed. One might object to dropping  $\Gamma$ ; indeed this could be generalized to keeping those assumptions from  $\Gamma$  which do not depend on  $\Delta$  and generalizing the allowed type of inductive invariant (see our earlier remark).

For a branch  $b = \Delta'; \vec{r}.r_0 \mapsto e$  to be well-typed with respect to a given invariant  $\Pi\Delta.\tau$ , we check the call pattern  $r_0$  and each recursive call  $r_j$  against the invariant and synthesize target types  $\tau_j$  ( $j \geq 0$ ). We then continue checking the body  $e$  against  $\tau_0$ , i.e., the target type of the call pattern  $r_0$ , populating the computation context with the recursive calls  $\vec{r}$  at their types  $\vec{\tau}$ . As members of a context  $\Gamma$ , the  $\vec{r}$  are simply fancy names for variables.

A pattern  $l$  recursive call  $r_{ij} = f \overrightarrow{[C_j]}$  intuitively corresponds to the given inductive invariant  $\Pi\Delta_0.\tau_0$ , if the spine  $\overrightarrow{[C_j]}$  matches the specified types in  $\Delta_0$  and it has intuitively the type  $\llbracket C_{jn}/X_n, \dots, C_{j0}/X_0 \rrbracket \tau_0$  which we denote with  $\tau'_{ij}$ .

*Lemma 9 (Substitution Lemma):*

- 1) If  $\Delta \vdash C : U$  and  $\Delta' \vdash \theta : \Delta$ , then  $\Delta' \vdash \llbracket \theta \rrbracket C : \llbracket \theta \rrbracket U$ .
- 2) If  $\Delta; \Gamma \vdash e : \tau$  and  $\Delta' \vdash \theta : \Delta$ , then  $\Delta'; \llbracket \theta \rrbracket \Gamma \vdash \llbracket \theta \rrbracket e : \llbracket \theta \rrbracket \tau$ .
- 3) If  $\Delta; \Gamma \vdash e : \tau$  and  $\Delta; \Gamma' \vdash \eta : \Gamma$ , then  $\Delta; \Gamma' \vdash [\eta]e : \tau$ .

*Proof.* By induction on the first typing derivation.  $\square$

## IV. OPERATIONAL SEMANTICS

Figure 8 specifies the call-by-value (cbv) one-step reduction relation  $e \longrightarrow e'$ ; we have omitted the usual congruence rules for cbv. Reduction is deterministic and does not get stuck on closed terms, due to completeness of pattern matching in  $\text{rec}$ . To reduce  $(\text{rec}^\tau C \text{ with } \vec{b})$  we find the branch  $(\Delta.r_k, \dots, r_1.r_0 \mapsto e) \in \vec{b}$  such that the principal argument  $C_0$  of its clause head  $r_0 = f \overrightarrow{[C_0]} [C_0]$  matches  $C$  under meta substitution  $\theta$ . The reduct is body  $e$  under  $\theta$  where we additionally replace each place holder  $r_j$  for a recursive call by the actual recursive invocation  $(\text{rec}^\tau \llbracket \theta \rrbracket C_j \text{ with } \vec{b})$ .

Values  $v$  shall be boxed meta objects  $[C]$  and functions  $\text{fn } x:\tau. e$  and  $\Lambda X:U.e$ .

*Theorem 10 (Subject reduction):*

If  $\cdot; \cdot \vdash e : \tau$  and  $e \longrightarrow e'$ , then  $\cdot; \cdot \vdash e' : \tau$ .

*Proof.* By induction on  $e \longrightarrow e'$ . **Case:**  $\text{rec}^\tau C \text{ with } \vec{b} \longrightarrow \llbracket \text{rec}^\tau C'_k \text{ with } \vec{b}/r_k, \dots, \text{rec}^\tau C'_1 \text{ with } \vec{b}/r_1 \rrbracket \llbracket \theta_i \rrbracket e_i$

$\Delta; \Gamma \vdash e : \tau$  : Computation  $e$  has type  $\tau$

$$\frac{\Gamma(y) = \tau}{\Delta; \Gamma \vdash y : \tau} \quad \frac{\Delta \vdash C : U}{\Delta; \Gamma \vdash [C] : [U]} \quad \frac{\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1 e_2 : \tau} \quad \frac{\Delta; \Gamma \vdash e : \Pi X:U.\tau \quad \Delta \vdash C : U}{\Delta; \Gamma \vdash e [C] : [[C/X]]\tau}$$

$$\frac{\Delta; \Gamma, y:\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \text{fn } y:\tau_1. e : \tau_1 \rightarrow \tau_2} \quad \frac{\Delta, X:U; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda X:U. e : \Pi X:U.\tau} \quad \frac{\Delta; \Gamma \vdash e_1 : [U] \quad \Delta, X:U; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash \text{let } X = e_1 \text{ in } e_2 : \tau}$$

$$\frac{\Delta_0 = \Delta_1, X:U_0 \quad \Delta \vdash C : U \quad U = [[\theta]]U_0 \quad \Delta \vdash \theta : \Delta_1 \quad b_i : \Pi \Delta_0.\tau_0 \text{ (for all } i) \quad \vec{b} \text{ covers } \Delta_0}{\Delta; \Gamma \vdash \text{rec}^{\Pi \Delta_0.\tau_0} C \text{ with } \vec{b} : [[\theta, C/X]]\tau_0}$$

$b : \Pi \Delta.\tau$  : Branch  $b$  has type  $\Pi \Delta.\tau$

$$\frac{\text{for all } 0 \leq j \leq k. \Delta' \vdash r_j : \Pi \Delta.\tau > \tau_j \quad \Delta' ; r_k:\tau_k, \dots, r_1:\tau_1 \vdash e : \tau_0}{\Delta' ; r_k \dots r_1 . r_0 \mapsto e : \Pi \Delta.\tau}$$

$\Delta' \vdash r : \Pi \Delta.\tau > \tau'$  and  $\Delta' \vdash \vec{C} : \Pi \Delta.\tau > \tau'$  : Recursive call  $r$  / pattern spine  $\vec{C}$  has type  $\Pi \Delta.\tau$  and target type  $\tau'$

$$\frac{\Delta' \vdash \vec{C} : \Pi \Delta.\tau > \tau'}{\Delta' \vdash f \vec{C} : \Pi \Delta.\tau > \tau'} \quad \frac{\Delta' \vdash C : U \quad \Delta' \vdash \vec{C} : [[C/X]](\Pi \Delta.\tau) > \tau'}{\Delta' \vdash [C] \vec{C} : \Pi(X:U, \Delta).\tau > \tau'} \quad \frac{}{\Delta' \vdash \text{nil} : \tau > \tau'}$$

Fig. 7. Type system for dependently-typed functional computation language

$$\overline{(\text{fn } x:\tau. e) v \longrightarrow [v/x]e} \quad \overline{(\Lambda X:U. e) [C] \longrightarrow [[C/X]]e} \quad \overline{\text{let } X = [C] \text{ in } e \longrightarrow [[C/X]]e}$$

$$\frac{\exists \text{ unique } (\Delta. r_k, \dots, r_1. r_0 \mapsto e) \in \vec{b} \text{ where } r_j = f \vec{C}_j [C_j] \text{ such that } \Delta \vdash C \doteq C_0/\theta}{\text{rec}^\tau C \text{ with } \vec{b} \longrightarrow [(\text{rec}^\tau [[\theta]] C_k \text{ with } \vec{b})/r_k, \dots, (\text{rec}^\tau [[\theta]] C_1 \text{ with } \vec{b})/r_1][[\theta]]e}$$

Fig. 8. Small-step semantics  $e \longrightarrow e'$

$\exists \text{ unique } b_i = \Delta_i. r_k, \dots, r_1. r_0 \mapsto e_i$   
 where  $r_j = f [C_{jn}] \dots [C_{j1}] [C_{j0}]$   
 $\Delta_i \vdash C \doteq C_{j0}/\theta_i$   
 $C'_{j0} = [[\theta_i]]C_{j0}$   
 $\cdot; \vdash \text{rec}^{\Pi \Delta_0.\tau_0} C \text{ with } \vec{b} : \tau'$  by inversion on rule Emrec  
 $\vec{b} \text{ covers } \Delta_0$  by assumption  
 $\cdot \vdash C : U$   
 $U' = [[\theta]]U_0$  and  $\tau' = [[\theta]]\tau_0$   
 for all  $i$   $b_i : \Pi \Delta_0.\tau_0$  by inversion on Trec  
 $b_i = \Delta_i; r_k \dots r_1 . r_0 \mapsto e_i$  by definition  
 for all  $0 \leq j \leq k. \Delta_i \vdash r_j : \Pi \Delta_0.\tau_0 > \tau'_j$   
 $\Delta_i; r_k:\tau'_k, \dots, r_1:\tau'_1 \vdash e_i : \tau'_0$  by inversion on Tb  
 for  $0 \leq j \leq k, r_j = f [C_{jn}] \dots [C_{j1}] [C_{j0}]$   
 where  $\Delta_0 = X_n:U_n, \dots, X_0:U_0$   
 and  $U'_{j0} = [[C_{jn}/X_n, \dots, C_{j1}/X_1]]U_{j0}$   
 $\Delta_i \vdash C_{j0} : U'_{j0}$  by typing  
 $\cdot \vdash \theta_i : \Delta_i$  by soundness of matching  
 $U' = [[\theta]]U_0 = [[\theta_i]]U'_{j0}$  by definition  
 $= [[\theta_i]] [[C_{jn}/X_n, \dots, C_{j1}/X_1]]U_{j0}$  by typing  
 $\theta = [[\theta_i]] (C_{jn}/X_n, \dots, C_{j1}/X_1)$  by previous lines  
 $\cdot; \vdash \text{rec}^\tau C'_j \text{ with } \vec{b} : \tau'_j$  by typing  
 $\cdot; \vdash \text{rec}^\tau C'_k \text{ with } \vec{b}/r_k, \dots, \text{rec}^\tau C'_1 \text{ with } \vec{b}/r_1 : r_k:\tau'_k, \dots, r_1:\tau'_1$

by typing  
 $\cdot; \vdash e_i : [[\theta_i]]\tau'_0$   
 by substitution lemma  
 $[[\theta_i]]\tau'_0 = [[\theta_i]] [[C_{1i}/X_1, \dots, C_{ni}/X_n]]\tau_0 = [[\theta]]\tau_0 = \tau'$   
 $\cdot; \vdash e_i : \tau'$  by previous lines  $\square$

**Lemma 11 (Canonical forms):** Let  $v$  a value.

- 1) If  $\cdot; \vdash v : [U]$  then  $v = [C]$ .
- 2) If  $\cdot; \vdash v : \Pi X:U.\tau$  then  $v = \Lambda X:U. e$ .
- 3) If  $\cdot; \vdash v : \tau \rightarrow \tau'$  then  $v = \text{fn } x:\tau. e$ .

**Theorem 12 (Progress):**

If  $\cdot; \vdash e : \tau$  then either  $e$  is a value or  $e \longrightarrow e'$ .

*Proof.* By induction on  $\cdot; \vdash e : \tau$ .

$$\text{Case } \mathcal{D} = \frac{\cdot; \vdash e_1 : [U] \quad X:U; \cdot \vdash e_2 : \tau}{\cdot; \vdash \text{let } X = e_1 \text{ in } e_2 : \tau}$$

$\cdot; \vdash e_1 : [U]$  by inversion  
 Either  $e_1$  is a value or  $e_1 \longrightarrow e'_1$  by i.h.  
 If  $e_1 \longrightarrow e'_1$   
 $\text{let } X = e_1 \text{ in } e_2 \longrightarrow \text{let } X = e'_1 \text{ in } e_2$  by reduction  
 If  $e_1$  is a value  
 $e_1 = [C]$  by canonical forms lemma  
 $\text{let } X = [C] \text{ in } e_2 \longrightarrow [[C/X]]e_2$  by reduction

**Case } \mathcal{D} = \cdot; \vdash \text{rec}^{\Pi \Delta\_0.\tau\_0} C \text{ with } \vec{b} : \tau**

$\vec{b}$  covers  $\Delta_0$ ,  $\cdot \vdash C : U$ ,  $\Delta_0 = (X_n:U_n, \dots, X_1:U_1, X_0:U_0)$ ,  
 $\cdot \vdash \theta_0 : (X_n:U_n, \dots, X_1:U_1)$ ,  $U = \llbracket \theta_0 \rrbracket U_0$ ,  $\tau = \llbracket \theta_0 \rrbracket \tau_0$ ,  
for all  $i$   $b_i : \Pi \Delta_0. \tau_0$  by inversion  
 $\exists$  a unique branch  $b_i = \Delta_i; r_{ik} \dots r_{i1}.r_{i0} \mapsto e_i$  by coverage  
with  $r_{i0} = f [C_n^{i0}] \dots [C_1^{i0}] [C_0^{i0}]$  s.t.  $C = \llbracket \rho \rrbracket C_0^{i0}$  and  $\cdot \vdash \rho : \Delta_i$   
 $\Delta_i \vdash C \doteq C_0^{i0} / \rho$  by unification  
for each  $r_{ij} = f [C_n^{ij}] \dots [C_1^{ij}] [C_0^{ij}]$  where  $1 \leq j \leq k$ ,  
let  $C'_j = \llbracket \rho \rrbracket C_0^{ij}$   
let  $\eta' = [\text{rec}^{\Pi \Delta_0. \tau_0} C'_k \text{ with } \vec{b} / r_k, \dots, \text{rec}^{\Pi \Delta_0. \tau_0} C'_1 \text{ with } \vec{b} / r_1]$   
 $\text{rec}^{\Pi \Delta_0. \tau_0} C \text{ with } \vec{b} \longrightarrow [\eta'] \llbracket \rho \rrbracket e_i$  by reduction.  $\square$

## V. TERMINATION

In this section, we prove that every well-typed closed program  $e$  terminates (halts) under cbv reduction. Note that since reduction  $e \longrightarrow e'$  is deterministic,  $e$  halts if and only if  $e'$  halts. Termination is proven by a standard reducibility argument; closely related is [Xi \[2002\]](#). The set  $\mathcal{R}_\tau$  of reducible closed programs  $\cdot; \cdot \vdash e : \tau$  is defined by induction on the size of  $\tau$  in [Figure 9](#). For the size of  $\tau$  all meta types  $U$  shall be disregarded, thus, the size is invariant under meta substitution  $C/X$ .

*Lemma 13 (Expansion closure):*

If  $\cdot; \cdot \vdash e : \tau$  and  $e \longrightarrow e'$  and  $e' \in \mathcal{R}_\tau$ , then  $e \in \mathcal{R}_\tau$ .

*Proof.* By induction on the size of type  $\tau$ .

### Case Contextual Type $[U]$

$\cdot; \cdot \vdash e : [U]$  by assumption  
 $e' \in \mathcal{R}_{[U]}$  by assumption  
 $e'$  halts by def. of  $\mathcal{R}_{[U]}$   
 $e$  halts by lemma  
 $e \in \mathcal{R}_{[U]}$  by definition of  $\mathcal{R}_{[U]}$

### Case Meta Abstraction Type $\Pi X:U.\tau$

$\cdot; \cdot \vdash e : \Pi X:U.\tau$  by assumption  
 $e' \in \mathcal{R}_{\Pi X:U.\tau}$  by assumption  
 $e'$  halts by def. of  $\mathcal{R}_{\Pi X:U.\tau}$   
 $e$  halts by lemma  
 $\forall C \in \mathcal{R}_U. e' [C] \in \mathcal{R}_{[C/X]\tau}$  by def. of  $\mathcal{R}_{\Pi X:U.\tau}$   
 $e \longrightarrow e'$  by assumption  
 $e [C] \longrightarrow e' [C]$  by evaluation rule  
 $\cdot \vdash C : U$  by def. of  $\mathcal{R}_C$   
 $\cdot; \cdot \vdash e [C] : \llbracket C/X \rrbracket \tau$  by typing rule  
 $e [C] \in \mathcal{R}_{\llbracket C/X \rrbracket \tau}$  by i.h. on  $\llbracket C/X \rrbracket \tau$   
 $e \in \mathcal{R}_{\Pi X:U.\tau}$  by definition of  $\mathcal{R}_{\Pi X:U.\tau}$

### Case Function Type

$\tau_1 \rightarrow \tau_2$   
classical proof  $\square$

*Lemma 14 (Fundamental Lemma):* Assume  $\Delta; \Gamma \vdash e : \tau$ .

If  $\cdot \vdash \theta : \Delta$  and  $\eta \in \mathcal{R}_{[\theta]\Gamma}$  then  $[\eta] \llbracket \theta \rrbracket e \in \mathcal{R}_{[\theta]\tau}$ .

*Proof.* By induction on  $\Delta; \Gamma \vdash e : \tau$ .

**Case  $D = \frac{\Delta, X:U; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda X.e : \Pi X:U.\tau}$  Tmabs**

$\theta \in \mathcal{R}_\Delta$  and  $\cdot \vdash \theta : \Delta$  by assumption and definition of  $\mathcal{R}_\Delta$

$\cdot; \llbracket \theta \rrbracket \Gamma \vdash \llbracket \theta \rrbracket (\Lambda X.e) : \llbracket \theta \rrbracket (\Pi X:U.\tau)$  by substitution lemma  
 $\eta \in \mathcal{R}_{[\theta]\Gamma}$  and  $\Delta; \cdot \vdash \eta : \llbracket \theta \rrbracket \Gamma$  by ass. and def. of  $\mathcal{R}_{[\theta]\Gamma}$   
 $\cdot; \cdot \vdash [\eta] \llbracket \theta \rrbracket (\Lambda X.e) : \llbracket \theta \rrbracket (\Pi X:U.\tau)$  by substitution lemma  
 $\cdot; \cdot \vdash \Lambda X. [\eta] \llbracket \theta, X/X \rrbracket e : \Pi X: \llbracket \theta \rrbracket U. \llbracket \theta, X/X \rrbracket \tau$  by subs. prop.  
 $\Lambda X. [\eta] \llbracket \theta, X/X \rrbracket e$  halts by def. of halts

let  $C \in \mathcal{R}_{[\theta]U}$  be arbitrary

$\cdot \vdash C : \llbracket \theta \rrbracket U$  by def. of  $\mathcal{R}_{[\theta]U}$   
 $\cdot \vdash \theta, C/X : \Delta, X:U$  by typing rule  
 $\theta, C/X \in \mathcal{R}_{\Delta, X:U}$  by definition of  $\mathcal{R}_{\Delta, X:U}$   
 $\eta \in \mathcal{R}_{[\theta, C/X]\Gamma}$  since  $\Gamma$  is independent of  $X$   
 $[\eta] \llbracket \theta, C/X \rrbracket e \in \mathcal{R}_{[\theta, C/X]\tau}$  by i.h.  
 $[\eta] \llbracket C/X \rrbracket (\llbracket \theta, X/X \rrbracket e) \in \mathcal{R}_{\llbracket C/X \rrbracket (\llbracket \theta, X/X \rrbracket \tau)}$  by subst. prop.  
 $(\Lambda X. [\eta] \llbracket \theta, X/X \rrbracket e) [C] \longrightarrow \llbracket C/X \rrbracket ([\eta] \llbracket \theta, X/X \rrbracket e)$   
by evaluation rule  
 $\cdot; \cdot \vdash (\Lambda X. [\eta] \llbracket \theta, X/X \rrbracket e) [C] : \llbracket C/X \rrbracket (\llbracket \theta, X/X \rrbracket \tau)$  by typ.  
 $(\Lambda X. [\eta] \llbracket \theta, X/X \rrbracket e) [C] \in \mathcal{R}_{\llbracket C/X \rrbracket (\llbracket \theta, X/X \rrbracket \tau)}$  by back. closed  
 $\Lambda X. [\eta] \llbracket \theta, X/X \rrbracket e \in \mathcal{R}_{\Pi X: \llbracket \theta \rrbracket U. \llbracket \theta, X/X \rrbracket \tau}$   
by def. of  $\mathcal{R}_{\Pi X: \llbracket \theta \rrbracket U. \llbracket \theta, X/X \rrbracket \tau}$   
 $[\eta] \llbracket \theta \rrbracket (\Lambda X.e) \in \mathcal{R}_{[\theta] (\Pi X:U.\tau)}$  by substitution property

**Case  $D = \frac{\Delta; \Gamma \vdash e : \Pi X:U.\tau \quad \Delta \vdash C : U}{\Delta; \Gamma \vdash e [C] : \llbracket C/X \rrbracket \tau}$  Tmapp**

$\theta \in \mathcal{R}_\Delta$  and  $\eta \in \mathcal{R}_{[\theta]\Gamma}$  by assumption  
 $[\eta] \llbracket \theta \rrbracket e \in \mathcal{R}_{[\theta] (\Pi X:U.\tau)}$  by i.h.  
 $[\eta] \llbracket \theta \rrbracket e \in \mathcal{R}_{\Pi X: [\theta] U. [\theta, X/X] \tau}$  by substitution property  
 $\cdot \vdash \theta : \Delta$  by definition of  $\mathcal{R}_\Delta$   
 $\cdot \vdash \llbracket \theta \rrbracket C : \llbracket \theta \rrbracket U$  by substitution lemma  
 $\llbracket \theta \rrbracket C \in \mathcal{R}_{[\theta]U}$  by def. of  $\mathcal{R}_{[\theta]U}$   
 $([\eta] \llbracket \theta \rrbracket e) \llbracket \llbracket \theta \rrbracket C \rrbracket \in \mathcal{R}_{\llbracket [\theta] C/X \rrbracket (\llbracket \theta, X/X \rrbracket \tau)}$  by def. of  $\mathcal{R}_{\Pi X: [\theta] U. [\theta, X/X] \tau}$   
 $[\eta] \llbracket \theta \rrbracket (e [C]) \in \mathcal{R}_{[\theta] (\llbracket C/X \rrbracket \tau)}$  by substitution property

**Case  $D = \frac{\Delta \vdash C : U}{\Delta; \Gamma \vdash [C] : [U]}$  Tmeta**

$\theta \in \mathcal{R}_\Delta$  and  $\cdot \vdash \theta : \Delta$  by assumption and definition of  $\mathcal{R}_\Delta$   
 $\cdot; \cdot \vdash \llbracket \theta \rrbracket C : \llbracket \theta \rrbracket U$  by substitution  
 $\llbracket \theta \rrbracket C$  halts since  $[C]$  is a value  
 $\llbracket \theta \rrbracket C \in \mathcal{R}_{\llbracket \theta \rrbracket U}$  by definition  
 $[\eta] \llbracket \theta \rrbracket [C] \in \mathcal{R}_{[\theta]U}$  by substitution property

**Case  $D = \frac{\Gamma(y) = \tau}{\Delta; \Gamma \vdash y : \tau}$  Tvar**

$\theta \in \mathcal{R}_\Delta$  and  $\cdot \vdash \theta : \Delta$  by assumption and definition of  $\mathcal{R}_\Delta$   
 $\cdot; \llbracket \theta \rrbracket \Gamma \vdash \llbracket \theta \rrbracket y : \llbracket \theta \rrbracket \tau$  by substitution lemma  
 $\cdot; \llbracket \theta \rrbracket \Gamma \vdash y : \llbracket \theta \rrbracket \tau$  by substitution property  
 $\eta \in \mathcal{R}_{[\theta]\Gamma}$  and  $[\eta] y \in \mathcal{R}_{[\theta]\tau}$  by assumption and def. of  $\mathcal{R}_{[\theta]\Gamma}$   
 $[\eta] \llbracket \theta \rrbracket y \in \mathcal{R}_{[\theta]\tau}$  by substitution property

Contextual Type	$\mathcal{R}_{[U]}$	=	$\{e \mid \cdot; \cdot \vdash e : [U] \text{ and } e \text{ halts}\}$
Function Type	$\mathcal{R}_{\tau' \rightarrow \tau}$	=	$\{e \mid \cdot; \cdot \vdash e : \tau' \rightarrow \tau \text{ and } e \text{ halts and } \forall e' \in \mathcal{R}_{\tau'}. e e' \in \mathcal{R}_{\tau}\}$
Dependent Type	$\mathcal{R}_{\Pi X:U.\tau}$	=	$\{e \mid \cdot; \cdot \vdash e : \Pi X:U.\tau \text{ and } e \text{ halts and } \forall C \in \mathcal{R}_U. e [C] \in \mathcal{R}_{[C/X]\tau}\}$
Computation-Level Context	$\mathcal{R}_{\Gamma}$	=	$\{\eta \mid \cdot; \cdot \vdash \eta : \Gamma \text{ and } \eta(x) \in \mathcal{R}_{\tau} \text{ for all } (x:\tau) \in \Gamma\}$

Fig. 9. Reducibility

**Case  $\mathcal{D} = \frac{\Delta; \Gamma \vdash e_1 : [U] \quad \Delta, X:U; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash \text{let } X = e_1 \text{ in } e_2 : \tau}$  Tlet**

$\theta \in \mathcal{R}_{\Delta}$  and  $\eta \in \mathcal{R}_{[\theta]\Gamma}$  by assumption  
 $[\eta][\theta]e_1 \in \mathcal{R}_{[\theta][U]}$  by i.h.  
 $\cdot; \cdot \vdash [\eta][\theta]e_1 : [\theta][U]$  by def. of  $\mathcal{R}_{[\theta][U]}$   
 $[\eta][\theta]e_1$  halts by def. of  $\mathcal{R}_{[\theta][U]}$

$[\eta][\theta]e_1 \longrightarrow^* v$  by progress  
 $\cdot; \cdot \vdash v : [\theta][U]$  by type preservation  
 $\cdot; \cdot \vdash v : [[\theta]U]$  by substitution property

let  
 $v = [C']$  by canonical forms lemma  
 $\cdot; \cdot \vdash C' : [\theta]U$  by typing inversion  
 $\cdot \vdash \theta : \Delta$  by def. of  $\mathcal{R}_{\Delta}$   
 $\cdot \vdash \theta, C'/X : \Delta, X:U$  by typing rule  
 $\theta, C'/X \in \mathcal{R}_{\Delta, X:U}$  by def. of  $\mathcal{R}_{\Delta, X:U}$   
 $\eta \in \mathcal{R}_{[\theta, C'/X]\Gamma}$   $\Gamma$  is independent of  $X$   
 $[\eta][\theta, C'/X]e_2 \in \mathcal{R}_{[\theta, C'/X]\tau}$  by i.h.  
 $\cdot; \cdot \vdash [\eta][\theta](\text{let } X = e_1 \text{ in } e_2) : [\theta]\tau$  by substitution lemma  
let  $X = [\eta][\theta]e_1$  in  $[\eta][\theta, X/X]e_2 \longrightarrow^* [\eta][\theta, C'/X]e_2$   
by evaluation rule  
let  $X = [\eta][\theta]e_1$  in  $[\eta][\theta, X/X]e_2 \in \mathcal{R}_{[\theta, C'/X]\tau}$   
by backward closed  
 $[\eta][\theta](\text{let } X = e_1 \text{ in } e_2) \in \mathcal{R}_{[\theta]\tau}$  by substitution property

**Case  $\mathcal{D} = \Delta; \Gamma \vdash \text{rec}^{\Pi\Delta_0.\tau_0} C_s$  with  $\vec{b} : \tau'$**

let  $\tau = \Pi\Delta_0.\tau_0$

$\vec{b}$  covers  $\Delta_0$ ,  $\Delta \vdash C_s : U_s$ ,  $U_s = [[\theta_0]U_0]$  and  $\tau' = [[\theta_0]\tau_0]$   
 $\cdot \vdash [[\theta]C_s] : [[\theta]U_s]$  by subst. lemma  
 $[[\theta]U_s = [[\theta][\theta_0]U_0]$  by previous lines

for all  $i$   $b_i : \Pi\Delta_0.\tau$  by inversion on Trec

$\exists$  a unique branch  $b_i = \Delta_i; r_{ik} \dots r_{i1}.r_{i0} \mapsto e_i$  by coverage  
with  $r_{i0} = f \overrightarrow{[C_0]} [C_0]$  s.t.  $[[\theta]C_s = [[\rho]C_0]$  and  $\cdot \vdash \rho : \Delta_i$ .

let  $\overrightarrow{[C]} = [C_{0k}] \dots [C_{01}]$  and  $\Delta_0 = X_k:U_k, \dots, X_1:U_1, X_0:U_0$

$\delta_{i0} = C_{0k}/X_k, \dots, C_{01}/X_1$  s.t.  $\Delta_i \vdash C : [[\delta_{i0}]U_0]$  by typing  
 $[[\rho][\delta_{i0}]U_0 = [[\theta]U_s = [[\theta][\theta_0]U_0]$  since  $[[\theta]C_s = [[\rho]C_0]$

and therefore  $[[\rho][\delta_{i0}] = [[\theta][\theta_0]$

for each  $r_{ij} = f \overrightarrow{[C_j]} [C_j]$  where  $1 \leq j \leq k$ ,  $C_j \prec C_0$   
by size decreasing thm

$\Delta_i \vdash C_j : U'_j$  where  $\exists \delta_{ij} = C_{jk}/X_k, \dots, C_{j1}/X_1$  and  
 $U'_j = [[\delta_{ij}]U_0]$  by typing

for each  $1 \leq j \leq k$ ,  $\Delta_i; \cdot \vdash \text{rec}^{\tau} C_j$  with  $\vec{b} : [[\delta_{ij}]\tau_0$  by typing  
 $C_j \prec C$  by size decreasing thm.

for any grounding meta-substitution  $\theta' \mid \|\llbracket \theta' \rrbracket C_j\| < \|\llbracket \theta \rrbracket C_0\|$   
by well-foundedness of  $\prec$

$\|\llbracket \rho \rrbracket C_j\| < \|\llbracket \rho \rrbracket C_0\|$  by choosing  $\theta' = \rho$   
 $\|\llbracket \rho \rrbracket C_j\| < \|\llbracket \theta \rrbracket C\|$  since  $[[\rho]C_0 = [[\theta]C_2]$  by previous lines

$\text{rec}^{\tau} [[\rho]C_j$  with  $\vec{b} \in \mathcal{R}_{[[\rho][\delta_{ij}]\tau_0}$  by inner i.h.  
with  $\rho \in \mathcal{R}_{\Delta_i}$ ,  $\cdot \in \mathcal{R}$ . since  $\|\llbracket \rho \rrbracket C_j\| < \|\llbracket \rho \rrbracket C_0\|$

let  $C'_j = [[\rho]C_j$  and  $\Gamma' = r_j: [[\delta_{ij}]\tau_0$   
 $\eta' = [\text{rec}^{\tau} C'_k \text{ with } \vec{b}/r_k, \dots, \text{rec}^{\tau} C'_1 \text{ with } \vec{b}/r_1]$

$\eta' \in \mathcal{R}_{[[\rho]\Gamma'}$  by definition

$\Delta_i; \Gamma' \vdash e_i: [[\delta_{i0}]\tau_0$  by typing

$[\eta'][[\rho]e_i \in \mathcal{R}_{[[\rho][\delta_{ij}]\tau_0}$  by i.h. with  $\rho \in \mathcal{R}_{\Delta_i}$ ,  $\eta' \in \mathcal{R}_{[[\rho]\Gamma'}$

$\text{rec}^{\tau} [[\theta]C_s$  with  $\vec{b} \longrightarrow [\eta][\rho]e_i$  by reduction

$\text{rec}^{\tau} [[\theta]C_s$  with  $\vec{b} \in \mathcal{R}_{[[\rho][\delta_{ij}]\tau_0}$  by backwards closed

$\text{rec}^{\tau} [[\theta]C_s$  with  $\vec{b} \in \mathcal{R}_{[[\theta][\theta_0]\tau_0}$  since  $[[\rho][\delta_{i0}] = [[\theta][\theta_0]$

□

*Theorem 15 (Termination):* If  $\cdot; \cdot \vdash e : \tau$  then  $e$  halts.

*Proof.* Taking both empty meta-context  $\Delta$  and empty computation-level context  $\Gamma$ , we obtain  $e \in \mathcal{R}_{\tau}$  by the fundamental lemma, which implies that  $e$  halts by definition of reducibility candidates. □

## VI. RELATED WORK

Establishing well-founded induction principles to support reasoning about higher-order abstract syntax specifications has been challenging. Due to these difficulties, [Gabbay and Pitts \[2002\]](#) proposed nominal logic which provides first-class names and  $\alpha$ -renaming together with structural recursion principles. This approach is appealing because it gives us direct access to names of bound variables, however capture-avoiding substitution is implemented separately. The generation of a new name and binding names are separate operations and fresh name generation is an observable effect. As a consequence, languages such as FreshML [\[Shinwell et al., 2003\]](#) allowed

the generation of data which contains accidentally unbound names. While early work [Gabbay and Pitts, 2002] justified the structural recursion principles on Fraenkel-Mostowski set theory, more recently Pitts [2011] describes a calculus of total, higher-order functions with a structural recursion modulo  $\alpha$ -renaming based on nominal sets [Pitts, 2003].

The key difference between our work and work on nominal calculi lies in the status of names. While in nominal calculi names have a global status, in our language based on contextual types we pair every type with its surrounding contexts giving the system a more fine-grained nature. This allows us to abstract over contexts and distinguish between different contexts. As in nominal systems, our bound names are first-class citizens that can be tested for equality, passed to functions as arguments and returned as results—albeit for us they always must be associated with their surrounding context. Further, this line of work mostly concentrates on simple types and as such is not suitable to represent proofs about formal systems by recursive functions. In contrast to the simply typed foundational nominal calculi, we developed a core calculus with indexed types, simultaneous pattern matching and recursion.

Approaches which support higher-order abstract syntax (HOAS) encodings of formal systems together with proofs about them fall into two categories: one grounded in proof theory [McDowell and Miller, 2002, Momigliano et al., 2008, Gacek et al., 2008, Gacek, 2008] and the other grounded in type theory [Schürmann and Pfenning, 1998, Pfenning and Schürmann, 1999, Schürmann, 2000, Schürmann et al., 2001, 2005, Poswolsky and Schürmann, 2008, Licata et al., 2008, Pientka, 2008, Pientka and Dunfield, 2008, 2010, Cave and Pientka, 2012].

In the proof-theoretic approaches, we adopt a two-level system where we implement a specification logic (similar to LF) inside either a (higher-order) reasoning logic—the approach taken in Abella [Gacek, 2008, Gacek et al., 2012]—or type theory—the approach taken in Hybrid [Momigliano et al., 2008]. Hypothetical judgments of object logics are modeled using implication in the specification logic (SL) and parametric judgments are handled via (generic) universal quantification. Substituting for an assumption is then justified by appealing to the cut-admissibility lemma of the SL. To distinguish in the reasoning logic between quantification over variables and quantification over terms, Gacek et al. [2008] introduce a new quantifier,  $\nabla$ , to describe nominal abstraction logically. Induction in these systems is typically supported by reasoning about the height of a proof tree; this reduces reasoning to induction over natural numbers. Baelde and Nadathur [2012] propose a uniform approach with least and greatest fixed points to support inductive reasoning. The cited work however lacks generic quantification and as such is not powerful enough yet to support reasoning about HOAS specifications.

In general, the proof-theoretic approach of encoding the SL inside a reasoning logic is less direct. Although much of this complexity and indirectness can be hidden in implementations as demonstrated in Abella, the programs we would obtain

would bear little resemblance to the functional programs we would expect. Moreover, although  $\nabla$  allows the distinction between generic and universal quantification, the proof-theory lacks intrinsic support for contexts; contexts are typically represented inductively as lists. As a consequence, properties such as the uniqueness of declarations in a context must be established separately. Our work pushes the boundaries of the provided infrastructure by treating contexts as first-class citizens and eliminating the burden on users to manage and maintain contexts together with their properties explicitly. More importantly, our reasoning logic, a first-order modal logic, supports reasoning about HOAS specifications without introducing new logical connectives. The complexity of working with HOAS specifications is pushed and encapsulated on the level of contextual objects, i.e., the objects we reason about. Finally, our logical foundation gives directly rise to a functional programming language supporting pattern matching and structural recursion.

From the type-theoretic foundations, our calculus is closely related to Despeyroux et al. [1997], Schürmann et al. [2001], Despeyroux and Leleu [2001] where the authors propose a modal lambda-calculus with iteration to reason about closed HOAS objects. The key insight in their work is to separate LF objects from computations using the modal type  $\Box$  to describe closed LF objects. Hofmann [1999] investigated a categorical explanation for the proposed reasoning principles. However, this line of work is also restricted to closed objects. Our work builds on this work allowing open data-objects and defining simultaneous pattern matching together with structural recursion.

Closely related to our approach is the work by Schürmann and Pfenning [1998], Schürmann [2000] where the author describes a meta-logic  $\mathcal{M}^2$  for reasoning about LF specifications and describes the generation of splits and well-formed recursive calls. However,  $\mathcal{M}^2$  does not support higher-order computations. Moreover, the foundation lacks first-class contexts, but all assumptions live in an ambient context. This makes it less direct to justify reasoning with assumptions. Finally, we are not aware of a normalization proof for this line of work.

## VII. CONCLUSION

We developed a core language with structural recursion for implementing total functions about LF specification. We describe a sound coverage algorithm which in addition to verifying that there exists a branch for all possible contexts and contextual objects, also generates and verifies valid primitive recursive calls. To establish consistency of our core language we prove termination using reducibility semantics.

Our framework can be extended to handle mutual recursive functions: By annotating a given rec-expression with a list of invariants using the subordination relation, we can generate well-founded recursive calls matching each of the invariants. Moreover we believe that adding reasoning principles for inductive types [Cave and Pientka, 2012] follows well-trodden

paths; we must ensure that our inductive type satisfies the positivity restriction and define generation of patterns for them.

Our language not only serves as a core programming language but can be interpreted by the Curry-Howard isomorphism as a proof language for interactively developing proofs about LF specifications. In the future, we plan to implement and design such a proof engine. We also intend to generalize the valid recursive calls; in this paper we concentrate on generating primitive recursive calls which are structurally immediately smaller. This notion will be extended to support also lexicographic orderings and in general well-founded recursion.

## REFERENCES

- Andreas Abel. *Tutch User's Guide*. Carnegie-Mellon University, Pittsburg, PA, 2002. Section 7.1: Proof terms for structural recursion.
- David Baelde and Gopalan Nadathur. Combining deduction modulo and logics of fixed-point definitions. In *LICS'12*, pages 105–114. IEEE CS Press, 2012.
- Olivier Savary Belanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In *CPP'13*, volume 8307 of *LNCS*, pages 243–258. Springer, 2013.
- Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *LFMTP'13*, pages 15–24. ACM, 2013.
- Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *POPL'12*, pages 413–424. ACM, 2012.
- Joëlle Despeyroux and Pierre Leleu. Recursion over objects of functional type. *MSCS*, 11(4):555–572, 2001.
- Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163. Springer, 1997. Extended version available as Technical Report CMU-CS-96-172, Carnegie Mellon University.
- Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. *ENTCS*, 228:69–84, 2009.
- Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *FAC*, 13:341–363, 2002.
- Andrew Gacek. The abella interactive theorem prover (system description). In *IJCAR'08*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.
- Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *LICS'08*, pages 33–44. IEEE CS Press, 2008.
- Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *JAR*, 49(2):241–273, 2012.
- Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.
- Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *LICS'99*, pages 204–213. IEEE CS Press, 1999.
- Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *23rd Symposium on Logic in Computer Science*, pages 241–252. IEEE Computer Society Press, 2008.
- Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002.
- Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-Level Hybrid: A system for reasoning using higher-order abstract syntax. In *LFMTP'07*, volume 196 of *ENTCS*, pages 85–93. Elsevier, 2008.
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM TOCL*, 9(3):1–49, 2008.
- Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), pages 202–206. Springer, 1999.
- Brigitte Pientka. Verifying termination and reduction properties about higher-order logic programs. *JAR*, 34(2):179–207, 2005.
- Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *JFP*, 1(1–37), 2013.
- Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL'08*, pages 371–382. ACM, 2008.
- Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM, 2008.
- Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR'10*, volume 6173 of *LNCS*, pages 15–21. Springer, 2010.
- Andrew Pitts. Structural recursion with locally scoped names. *JFP*, 21(3):235–286, 2011.
- Andrew Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.
- Adam B. Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *17th European Symposium on Programming (ESOP '08)*, volume 4960, pages 93–107. Springer, 2008.
- Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In *ESOP'96*, volume 1058 of *LNCS*, pages 296–310. Springer, 1996.
- Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.
- Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In *TPHOLS'03*, volume 2758 of *LNCS*, pages 120–135, Rome, Italy, 2003. Springer.
- Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *CADE'98*, volume 1421 of *LNCS*, pages 286–300. Springer, 1998.

- Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *TCS*, 266(1-2):1–57, 2001.
- Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The  $\nabla$ -calculus. Functional programming with higher-order encodings. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.
- Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In *8th International Conference on Functional Programming (ICFP'03)*, pages 263–274. ACM, 2003.
- Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. CMU-CS-99-167.
- Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.
- Hongwei Xi. Dependent types for program termination verification. *HOSC*, 15(1):91–131, 2002.