Structural Recursion over Contextual Objects (Extended Abstract)

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.

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. 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 Ψ [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 [2000] which presents a meta-logic $\mathcal{M}^2$ for reasoning about LF specifications and describes 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.

Due to space constraints, proofs have been omitted from this extended abstract. The full version of this article is available online [Pientka et al., 2014].

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].

LF Constants	c
LF Variables	x, y, z
Parameter Variables	p
Meta Variables	u
Context Variables	ψ, ϕ
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	$S ::= \mathsf{nil} \mid M\ S$
Normal Terms	$M, N ::= R \mid \lambda x.\, M$
Substitutions	$\sigma ::= \cdot \mid \mathsf{id}_\psi \mid \sigma, M \mid \sigma; H$
Variable Substitutions	$\pi ::= \cdot \mid \mathsf{id}_\psi \mid \sigma; x$
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 Ψ. The variable list $\hat{\Psi}$ can be obtained from the context Ψ by simply dropping the type annotations. We consider only objects in η-long β-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.

Normal terms are either lambda-abstractions or neutral terms which are defined using a spine representation to give

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

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

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

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

$$\frac{}{\Delta; \Psi \vdash \mathsf{nil} : P > P}$$

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

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

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

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

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

$\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Phi}$ Check substitution σ against domain Φ

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{}{\Delta; (\psi, \Psi^0) \vdash \mathsf{id}_\psi \Leftarrow \psi}$$

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

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

Fig. 2. Bi-directional typing for contextual LF

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 λ-abstraction or in a context Ψ. 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 σ 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 σ, we do not make its domain explicit. Rather we think of a substitution together with its domain Ψ and the i-th element in σ corresponds to the i-th declaration in Ψ. 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 σ through a λ-abstraction $\lambda x.M$, we need to extend σ with x. The resulting substitution σ, 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 id_ψ denotes the identity substitution with domain ψ while $\cdot$ describes the empty substitution. Application of a substitution σ to an LF normal form B, written as $[\sigma]B$, is *hereditary* [Watkins et al., 2003] and produces in turn a normal form by removing generated redexes on the fly, possibly triggering further hereditary substitutions.

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

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

We summarize the bi-directional type system for contextual LF in Figure 2. LF objects may depend on variables declared in the context Ψ and a fixed meta-context Δ which contains contextual variables such as meta-variables u, parameter variables p, and context variables ψ. All typing judgments have access to both contexts and a fixed well-typed signature Σ where we store constants c together with their types and kinds.

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 Ψ) can be used to index computation-level types τ (see next section). Just as types classify terms, context schemas G classify contexts.

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 Δ. A meta-variable X can denote a meta-variable u, a parameter variable p, or a context variable ψ. Meta substitution C/X can represent $\hat{\Psi}.R/u$, or Ψ/ψ, or $\hat{\Psi}.x/p$, or $\hat{\Psi}.p'[\pi]/p$ (where π 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 σ to R hereditarily. The simultaneous meta-substitution, written as $[\![\theta]\!]$, 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 $\Delta'; [\![\theta]\!]\Psi \vdash [\![\theta]\!]J$.

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

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

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

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

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

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

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \Delta \vdash C : [\![\theta]\!]U}{\Delta \vdash \theta, C/X : \Delta', X{:}U}$$

Fig. 3. Typing for meta-objects

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

There are three forms of computation-level types τ. The base types $[U]$ are introduced by wrapped contextual objects $[C]$; the non-dependent function space $\tau_1 \to \tau_2$ is introduced by function abstraction $\mathsf{fn}\, y{:}\tau_1.\, e$ 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 $e\ [C]$

$\Lambda\psi{:}(\texttt{ctx}).\Lambda A{:}(.\texttt{tp}).\mathsf{fn}\ x{:}[\psi.\texttt{tm}\,A].\mathsf{let}\ X = x\ \mathsf{in}\ \mathsf{rec}^{T}(\psi.X\ldots)\ \mathsf{with}$

$\mid \phi{:}(\texttt{ctx}),\ B{:}(.\texttt{tp}),\ p{:}(\#\phi.B)$; $\cdot$

$.f\ [\phi]\ [.B]\ [\phi.p\ldots]$ $\mapsto$ $[.\texttt{z}]$

$\mid \phi{:}(\texttt{ctx}),\ B_1{:}(.\texttt{tp}),\ B_2{:}(.\texttt{tp}),\ M{:}(\phi, x{:}\texttt{tm}\,B_1.\texttt{tm}\,B_2)$; $f\ [\phi, x{:}\texttt{tm}\,B_1]\ [.B_2]\ [\phi, x{:}\texttt{tm}\,B_1.M\ldots x]$

$.f\ [\phi]\ [\phi.\texttt{arr}\ B_1\ B_2]\ [\phi.\texttt{lam}\,\lambda x.\,M\ldots x]$ $\mapsto$ $\mathsf{let}\,N = f\ [\phi, x{:}\texttt{tm}\,B_1]\ [.B_2]\ [\phi, x{:}\texttt{tm}\,B_1.M\ldots x]$ $\mathsf{in}\ [.\texttt{suc}\,N]$

$\mid \phi{:}(\texttt{ctx}),\ B_1{:}(.\texttt{tp}),\ B_2{:}(.\texttt{tp}),\ M{:}(\phi.\texttt{tm}\,(\texttt{arr}\ B_1\ B_2)),\ N{:}(\phi.\texttt{tm}\,B_1)$; $f\ [\phi]\ [\phi.\texttt{arr}\ B_1\ B_2]\ [\phi.M\ldots],\ f\ [\phi]\ [.B_1]\ [\phi.N\ldots]$

$.f\ [\phi]\ [.B_2]\ [\phi.\texttt{app}\ (M\ldots)\ (N\ldots)]$ $\mapsto$ $\mathsf{let}\,X = f\ [\phi]\ [.\texttt{arr}\ B_1\ B_2]\ [\phi.M\ldots]\ \mathsf{in}$ $\mathsf{let}\,Y = f\ [\phi]\ [.B_1]\ [\phi.N\ldots]\ \mathsf{in}$ $\mathsf{let}\,Z = \texttt{plus}\ [.X]\ [.Y]\ \mathsf{in}$ $[.\texttt{suc}\ Z]$

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

to meta objects C. Note that we can only index computation-level types τ 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 on 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 Δ' exactly describes the free meta-variables occurring in U. We also give the return type τ of the recursor, since it might be refined during pattern matching.[1]

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

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 $\ldots$ instead of $[\mathsf{id}_\psi]$. 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.

```
tp   : type.                 tm : tp → type.
bool: tp.                    app: tm (A → B) → tm A → tm B.
arr  : tp → tp → tp.         lam: (tm A → tm B) → tm (arr 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].

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

$$T = \Pi\psi{:}(\texttt{ctx}).\Pi A{:}(.\texttt{tp}).[\psi.\texttt{tm}\,A] \to [.\texttt{nat}]$$

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

In the case for lambda-abstraction, we recurse on M, written as $f\ \ [\phi, x{:}\texttt{tm}\,B_1]\ \ [.B_2]\ \ [\phi, x{:}\texttt{tm}\,B_1.M\ldots x]$. This recursive call is justified as $\phi, x{:}\texttt{tm}\,B_1.M\ldots x$ is considered smaller than $\phi.\texttt{lam}\,\lambda x.\,M\ldots x$ (details later). In the case for application, we have two recursive calls, one on M, written as $f\ [\phi]\ [.\texttt{arr}\ B_1\ B_2]\ [\phi.M\ldots]$, and the other on N, written as $f\ [\phi]\ [.B_1]\ [\phi.N\ldots]$.

May we also write a program which is defined recursively over $\psi.\texttt{tm}\,\texttt{bool}$ instead of $\psi.\texttt{tm}\,T$? - The answer is yes. In

[1] This is analogous to Coq's `match_as_in_return_with_end` construct.

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

Fig. 5. Conversion to de Bruijn

such a case, we only generate cases whose patterns have type $\psi.\texttt{tm}\,\texttt{bool}$, i.e., the variable case and the case for applications. What well-founded recursive calls are generated? In fact none, since given $[\phi.\texttt{app}\ (M\,\ldots)\ (N\,\ldots)]$ we have that M has type $\phi.\texttt{tm}\ (\texttt{arr}\ B_1\ \texttt{bool})$ and N has type $\phi.\texttt{tm}\ B_1$. Neither of which is a possible instance of $\phi.\texttt{tm}\ \texttt{bool}$, 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 $T = \Pi\psi{:}(\texttt{ctx}).[.\texttt{nat}]$. There are two possible cases to consider: either ψ is empty or $\psi = \phi, x{:}\texttt{tm}\,B$. In the latter case, ϕ is smaller than ψ and hence we assume $f\ [\phi]$ as a well-founded recursive call.

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

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 $\#\psi.\texttt{tm}\,A$. Note that simultaneously matching on ψ and $\#\psi.\texttt{tm}\,A$ allows us to split first ψ followed by a split on $\#\psi.\texttt{tm}\,A$. The case $\psi = \cdot$ (i.e., ψ is empty) and $\#\cdot.\texttt{tm}\,A$ is impossible, since there is no variable.

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 [Pientka, 2005].

For simplicity, we only consider here non-mutual recursive type families; those can be incorporated using the notion of subordination Virga [1999]. We first define an ordering on contexts: $\boxed{\Psi \preceq \Phi}$, read as "context Ψ is a subcontext of Φ", shall hold if all declarations of Ψ are also present in the context Φ, i.e., $\Psi \subseteq \Phi$. The strict relation $\boxed{\Psi \prec \Phi}$, read as "context Ψ is strictly smaller than context Φ" holds if $\Psi \preceq \Phi$ but Ψ is strictly shorter than Φ.

Further, we define three relations on contextual objects $\hat{\Psi}.M$: a strict subterm relation $\prec$, an equivalence relation $\equiv$, and an auxiliary relation $\preceq$.

$\boxed{\hat{\Psi}.M \equiv \hat{\Phi}.N}$ 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

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

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

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

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

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 Ψ^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. The latter is defined inductively by:

$$
\begin{array}{lcl}
||h \cdot M_1 \ldots M_n\ \mathsf{nil}|| & = & 1 + \max(||M_1||, \ldots, ||M_n||) \\
||\lambda x.M|| & = & 1 + ||M||
\end{array}
$$

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

1) If $C \prec C'$ then $||[\![\theta]\!]C|| \ < \ ||[\![\theta]\!]C'||$.
2) If $C \equiv C'$ then $||[\![\theta]\!]C|| \ = \ ||[\![\theta]\!]C'||$.
3) If $C \preceq C'$ then $||[\![\theta]\!]C|| \ \le \ ||[\![\theta]\!]C'||$.

C. Coverage

For well-formed recursors $(\mathsf{rec}^{\Delta, X:U.\tau}\,C\ \mathsf{with}\ \vec{b})$, 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 Ψ where H is a constructor c, a variable x from Ψ 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 Ψ.

From $\mathcal{U}_{\Delta \vdash U}$ we generate the complete minimal set $\mathcal{C} = \{\Delta_i; r_{ik}, \ldots, r_{i1}.r_{i0} \mid 1 \leq i \leq n\}$ of possible, non-overlapping cases where the i-th branch shall have the well-founded recursive calls $r_{ik}, \ldots, r_{i1}$ 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 Ψ starts with a context variable ψ.

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

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 Ψ. We describe unification using the judgment $\boxed{\Delta; \Psi \vdash Q \doteqdot P \ /\ (\Delta'\ ,\ \theta)}$. If unification succeeds then $[\![\theta]\!]Q = [\![\theta]\!]P$ 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 $\Delta'; [\![\theta]\!]\Psi \vdash R \Rightarrow [\![\theta]\!]A$ and $\Delta'; [\![\theta]\!]\Psi \vdash R_0 \Leftarrow [\![\theta]\!]P$. Moreover, $R_0 = [\![\theta]\!]R\ \vec{M}$ for a list of eta-expanded meta variables $\vec{M}$ that bring R down to base type.

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

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 $\exists \Phi^0.B \in G$.

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

c) Splitting on a parameter type: We show how to generate all variables of type $\#\Psi.A$. Intuitively, only bound variables $x{:}B$ from Ψ whose type is unifiable with A inhabit this type; if the context Ψ 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 Ψ of matching type plus variables and parameters $\mathcal{U}$ spanning from the context variable ψ in Ψ, if any.

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

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 Ψ^0 is non-empty and $\psi{:}G \in \Delta$.

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

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

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

$$\begin{array}{ll}\phi{:}\mathtt{ctx}, A{:}(\mathtt{.tp}) & ;\ \phi, x{:}\mathtt{tm}\ A; \vdash x{:}\mathtt{tm}\ A\\ \phi{:}\mathtt{ctx}, B{:}(\mathtt{.tp}), A{:}(\mathtt{.tp}), p{:}\#(\phi.\mathtt{tm}\ A); & \phi, x{:}\mathtt{tm}\ B; \vdash p[\mathsf{id}_\phi]{:}\mathtt{tm}\ A\end{array}$$

D. Generation of call patterns

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

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

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 $1 \leq j \leq k$, $\Delta_i = X_n{:}U_n, \ldots, X_1{:}U_1$, we generate a recursive call

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

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

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

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

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

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

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

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

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

Definition 5 (Coverage): We say $\vec{b}$ covers Δ_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 Δ_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 Ψ are distinct. Completeness is proven by cases. □

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. □

F. Computation-level Type System

Our type system for computations performs coverage checking and verifies well-formed primitive recursions in the typing rule for rec expressions. We annotate the rec-expression with the inductive invariant $\Pi\Delta_0.\tau_0$ which is a closed computation-level type. The intention is that given $\Delta_0 = X_n{:}U_n, \ldots, X_0{:}U_0$, we induct on objects of type U_0 which may depend on $X_n, \ldots, X_1$. Δ_0 must therefore contain at least one declaration. The result of the inductive invariant is τ which also might depend on Δ_0. One might ask whether this form of inductive invariant is too restrictive, since it seems not to capture, e.g., $\Pi\Delta.(\tau \to \Pi X{:}U.\tau')$. 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 $\Pi(\Delta, X{:}U).\,\tau \to \tau'$ which is treated by our calculus.

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

Most typing rules should look familiar. Interesting is the rule for recursion: the expression $\mathsf{rec}^{\Pi\Delta_0.\tau_0}\ C\ \text{with}\ \vec{b}$ 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 $U = [\![\theta]\!]U_0$ for some meta-substitution θ with domain Δ_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 Δ_0, i.e., it must be a complete, non-redundant set of patterns covering Δ_0.

Note that we drop the meta-context Δ and the computation context Γ when we proceed to check that all branches satisfy the specified invariant. Dropping Δ is fine, since we require the invariant $\Pi\Delta_0.\tau_0$ to be closed. One might object to dropping Γ; indeed this could be generalized to keeping those assumptions from Γ which do not depend on Δ 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 τ_j $(j \geq 0)$. We then continue checking the body e against τ_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 Γ, the $\vec{r}$ are simply fancy names for variables.

A pattern / 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 Δ_0 and it has intuitively the type $[\![C_{jn}/X_n, \ldots, C_{j0}/X_0]\!]\tau_0$ which we denote with τ'_{ij}.

Lemma 8 (Substitution Lemma):

1) If $\Delta \vdash C : U$ and $\Delta' \vdash \theta : \Delta$, then $\Delta' \vdash [\![\theta]\!]C : [\![\theta]\!]U$.

$\boxed{\Delta;\Gamma \vdash e : \tau}$: Computation e has type τ

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

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

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

$\boxed{b : \Pi\Delta.\tau}$: Branch b has type $\Pi\Delta.\tau$

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

$\boxed{\Delta' \vdash r : \Pi\Delta.\tau > \tau'}$ and $\boxed{\Delta' \vdash \overrightarrow{[C]} : \Pi\Delta.\tau > \tau'}$: Recursive call r / pattern spine $\overrightarrow{[C]}$ has type $\Pi\Delta.\tau$ and target type τ'

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

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

2) If $\Delta;\Gamma \vdash e : \tau$ and $\Delta' \vdash \theta : \Delta$,
   then $\Delta'; [\![\theta]\!]\Gamma \vdash [\![\theta]\!]e : [\![\theta]\!]\tau$.
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. □

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 rec. To reduce $(\mathsf{rec}^{\tau} C\ \mathsf{with}\ \vec{b})$ we find the branch $(\Delta.r_k, \ldots, r_1.r_0 \mapsto e) \in \vec{b}$ such that the principal argument C_0 of its clause head $r_0 = f\ \overrightarrow{[C_0]}\ [C_0]$ matches C under meta substitution θ. The reduct is body e under θ where we additionally replace each place holder r_j for a recursive call by the actual recursive invocation $(\mathsf{rec}^{\tau} [\![\theta]\!]C_j\ \mathsf{with}\ \vec{b})$.

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

Theorem 9 (Subject reduction):
If $\cdot;\cdot \vdash e : \tau$ and $e \longrightarrow e'$, then $\cdot;\cdot \vdash e' : \tau$.

Proof. By induction on $e \longrightarrow e'$. □

Lemma 10 (Canonical forms): Let v a value.

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

Theorem 11 (Progress):
If $\cdot;\cdot \vdash e : \tau$ then either e is a value or $e \longrightarrow e'$.

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

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 τ in Figure 9. For the size of τ all meta types U shall be disregarded, thus, the size is invariant under meta substitution C/X.

Lemma 12 (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 τ. □

Lemma 13 (Fundamental Lemma): Assume $\Delta;\Gamma \vdash e : \tau$.
If $\cdot \vdash \theta : \Delta$ and $\eta \in \mathcal{R}_{[\![\theta]\!]\Gamma}$ then $[\eta][\![\theta]\!]e \in \mathcal{R}_{[\![\theta]\!]\tau}$.

Proof. By induction on $\Delta;\Gamma \vdash e : \tau$. In the interesting case of recursion rec, we make essential use of coverage and structural descent in the recursive calls. Due to lack of space, the proof can only be found in the full version of this article. □

Theorem 14 (Termination): If $\cdot;\cdot \vdash e : \tau$ then e halts.

Proof. Taking both empty meta-context Δ and empty computation-level context Γ, 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 α-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. While early work [Gabbay and Pitts, 2002] justified the structural recursion

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

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

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

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

Fig. 9. Reducibility

principles on Fraenkel-Mostowski set theory, more recently Pitts [2011] describes a calculus of total, higher-order functions with a structural recursion modulo α-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: proof-theoretic and type-theoretic. Since we discussed the relationship to other type-theoretic foundations in the introduction, we concentrate here on the former.

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, ∇, 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 ∇ 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.

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.

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.

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.

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. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR'10*, volume 6173 of *LNCS*, pages 15–21. Springer, 2010.

Brigitte Pientka, Sherry Shanshan Ruan, and Andreas Abel. Structural recursion over contextual objects. Full version, January 2014. URL http://www.tcs.ifi.lmu.de/~abel/wfrec-dep-long.pdf.

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.

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, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *TCS*, 266(1-2):1–57, 2001.

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.