Well-founded Recursion over Contextual Objects

Brigitte Pientka[1] **and Andreas Abel**[2]

1 **School of Computer Science**
McGill University, Montreal, Canada
`bpientka@cs.mcgill.ca`

2 **Department of Computer Science and Engineering, Gothenburg University**
Göteborg, Sweden
`andreas.abel@gu.se`

Abstract

We present a core programming language that supports writing well-founded structurally recursive functions using simultaneous pattern matching on contextual LF objects and contexts. The main technical tool is a coverage checking algorithm that also generates valid recursive calls. To establish consistency, we define a call-by-value small-step semantics and prove that every well-typed program terminates using a reducibility semantics. Based on the presented methodology we have implemented a totality checker as part of the programming and proof environment Beluga where it can be used to establish that a total Beluga program corresponds to a proof.

1998 ACM Subject Classification D.3.1[Programming Languages]: Formal Definitions and Languages. F.3.1[Logics and Meaning of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords and phrases Type systems, Dependent Types, Logical Frameworks

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.x

1 Introduction

Mechanizing formal systems and their proofs play an important role in establishing trust in formal developments. A key question in this endeavor is how to represent variables and assumptions to which the logical framework LF [8], a dependently typed lambda-calculus, provides an elegant and simple answer: both can be represented uniformly using LF's function space, modelling binders in the object language using binders in LF. This kind of encoding is typically referred to as *higher-order abstract syntax* (HOAS) and provides a general uniform treatment of syntax, rules and proofs.

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. As we recursively traverse higher-order abstract syntax trees, we extend our context of assumptions, and our LF object does not remain closed. To tackle this problem, Pientka and collaborators [11, 4] 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 [9]. Contextual objects are then embedded into a computation language which supports general recursion and pattern matching on contexts and contextual objects. Beluga, a programming environment based on these ideas [13], facilitates the use of HOAS for non-trivial applications such as normalization-by-evaluation [4] and a type-preserving compiler including closure conversion and hoisting [3]. However, nothing in this work enforces or guarantees that a given program is total.

In this paper, we develop a core functional language for reasoning inductively about context and contextual objects. One can think of this core language as the target of a Beluga program: elaboration may use type reconstruction to infer implicit indices [6] and generate valid well-founded recursive calls that can be made in the body of the function. Type checking will guarantee that we are manipulating well-typed objects and, in addition, that a given set of cases is covering and the given recursive calls are well-founded. To establish consistency, we define a call-by-value small-step semantics for our core language and prove that every well-typed program terminates, using Tait's method of logical relations. Thus, we justify the interpretation of well-founded recursive programs in our core language as inductive proofs. Based on our theoretical work, we have implemented a totality checker for Beluga.

Our approach is however more general: our core language can be viewed as a language for first-order logic proofs by structural induction over a given domain. The domain must only provide answers to three domain-specific questions: (1) how to unify objects in the domain, (2) how to split on a domain object and (3) how to justify that a domain object is smaller according to some measure. The answer to the first and second question allows us to justify that the given program is covering, while the third allows us to guarantee termination. For the domain of contextual LF presented in this paper, we rely on higher-order unification [2] for (1), and our splitting algorithm (2) and subterm ordering (3) builds on previous work [5, 10]. As a consequence, our work highlights that reasoning about HOAS representations via contextual types can be easily accommodated in a first-order theory. In fact, it is a rather straightforward extension of how we reason inductively about simple domains such as natural numbers or lists.

The remainder of the paper is organized as follows. We first present the general idea of writing and verifying programs to be total in Sec. 2 and then describe in more detail the foundation of our core programming language which includes well-founded recursion principles and simultaneous pattern matching in Sec. 3. The operational semantics together with basic properties such as type safety is given in Sec. 4. In Sec. 5, we review contextual LF [4], define a well-founded measure on contextual objects and contexts, and define the splitting algorithm. Subsequently we describe the generation of valid well-founded recursive calls generically, and prove normalization (Sec. 7). We conclude with a discussion of related work, current status and future research directions. Due to space constraints, proofs have been omitted.

2 General Idea

2.1 Example 1: Equality on Natural Numbers

To explain the basic idea of how we write inductive proofs as recursive programs, we consider first a very simple example: reasoning about structural equality on natural numbers (see Listing 1). We encode natural numbers and equality on them in the logical framework LF.

Listing 1 Encoding of an Inductive Proof as a Recursive Function

```
nat : type.                                     eq   : nat → nat → type.
z   : nat.                                      eq_z : eq z z.
s   : nat → nat.                                eq_s : eq M N → eq (s M) (s N).
.
ref : Π M:nat. [eq M M] = Λ M ⇒ rec-case M of
|                             ref  z     ⇒ [eq_z]
|  M’:nat ; ref M’:[eq M’ M’]. ref (s M’) ⇒ let  D = ref M’ in [eq_s M’ M’ D];
```

The free variables `M` and `N` in the definition of `eq_s` are implicitly quantified at the outside. Program `ref` proves reflexivity of `eq`: for all `M:nat` we can derive `eq M M`. Following type-theoretic notation, we write Π for universal quantification; we embed LF objects which denote base predicates via `[ ]`. Abstraction over LF object `M` is written Λ `M` $\Rightarrow$ in our language. Using `rec-case`, we prove inductively that for all `M` there is a derivation for `[eq M M]`. There are two cases to consider: `ref z` describes the base case where `M` is zero and the goal refines to `[eq z z]`. In this case, the proof is simply `[eq_z]`. In the step case, written as `ref (s M')`, we also list explicitly the other assumptions: the type of `M'` and the induction hypothesis written as `ref M':[eq M' M']`. To establish that `[eq (s M') (s M')]`, we first obtain a derivation `D` of `eq M' M'` by induction hypothesis and then extend it to a derivation `[eq_s M' M' D]` of `[eq (s M') (s M')]`. We highlight in green redundant information which can be inferred automatically. In the pattern, it is the typing (here: `M':nat`) of the pattern variables [12, 6] and the listing of the induction hypotheses. The dot "`.`" separates these assumptions from the main pattern. For clarity, we choose to write the pattern as a simultaneous pattern match and make the name of the function explicit; in practice, we only write the main pattern which is left in black, and all other arguments are inferred.

2.2 Example 2: Intrinsically Typed Terms

Next, we encode intrinsically typed λ-terms. This example does exploit the power of LF.

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

We define base types such as `bool` and function types, written as `arr T S`, and represent simply-typed lambda-terms using the constructors `lam` and `app`. In particular, we model the binding in the lambda-calculus (our object language) via HOAS, using the LF function space. For example, the identity function is represented as `lam λx.x` and function composition as `lam λg. lam λf. lam λx. app (f (app g x))`. As we traverse λ-abstractions we record the variables we are encountering in a context ϕ `: cxt`. Its shape is given by a *schema declaration* `schema ctx = tm A` stating that it contains only variable bindings of type `tm A` for some `A`. To reason about typing derivations, we package the term (or type) together with its context, forming a contextual object (or contextual type, resp.). For example, we write $\phi \vdash$ `tm A` for an object of type `tm A` in the context ϕ. Such *contextual types* are embedded into logical statements as `[`$\phi \vdash$ `tm A]`. When the context ϕ is empty, we may drop the turnstile and simply write `[tm A]`.

Counting constructors: Induction on (contextual) LF object

As an example, we consider counting constructors in a term. This corresponds to defining the overall size of a typing derivation. We recursively analyze terms `M` of type `tm A` in the context ϕ. In the variable case, written as `count` ϕ `B (`$\phi \vdash$ `p…)`, we simply return zero. The pattern variable `p` stands for a variable from the context ϕ. We explicitly associate it with the identity substitution, written as `…`, to use `p` which has declared type $\phi \vdash$ `tm B` in the context ϕ. Not writing the identity substitution would enforce that the pattern variable does not depend on ϕ and forces the type of `p` to be `[`$\vdash$ `tm B]`. While it is certainly legitimate to use `p` in the context ϕ, since the empty substitution maps variables from the empty context to ϕ, the type of `p` is empty; since the context is empty, there are no variables of the type `[`$\vdash$ `tm B]`. Hence writing `(`$\phi \vdash$ `p)` would describe an empty pattern. In contrast, types described by

Listing 2 Counting constructors

```
count: Π ϕ:ctx. Π A:tp. Π M:(ϕ ⊢ tm A) . [ nat] =
Λ ϕ ⇒ Λ A ⇒ Λ M ⇒ rec-case M of
| B:tp, p:(ϕ ⊢ tm B); . count ϕ B (ϕ ⊢ p … ) ⇒ [ z ]              % Variable Case
| B:tp,C:tp,M:(ϕ,x:tm B ⊢ tm C) ;                                   % Abstraction Case
  count (ϕ,x:tm B) C (ϕ,x:tm B ⊢ M  …  x) : [nat].                  % IH
  count ϕ (arr B C) (ϕ ⊢ lam  B C λx. M …  x) ⇒
  let  X = count (ϕ,x:tm B) C (ϕ,x:tm B ⊢ M …  x) in [ s X ]
| B:tp,C:tp,M:(ϕ ⊢ tm (arr B C)), N:(ϕ ⊢ tm B) ;                    % Application Case
  count ϕ (arr B C) (ϕ ⊢ M … ):[nat],                               % IH1
  count ϕ         B  (ϕ ⊢ N … ):[nat].                              % IH2
  count ϕ C (ϕ ⊢ app  B C  (M … ) (N … )) ⇒
  let  X = count ϕ (arr B C) (ϕ ⊢ M … ) in
  let  Y = count ϕ         B  (ϕ ⊢ N … ) in add (s X) Y
```

Listing 3 Computing length of a context

```
length = Π ϕ:ctx. [nat] =
Λ ϕ ⇒ rec-case ϕ of
|                                . count ∅              ⇒ [ z ]
|  ψ:ctx, A:tp ; count ψ: [nat] . count (ψ, x:tm A) ⇒ let X = count ψ in [ s X ]
```

meta-variables `A` or `B`, for example, are always closed and can be instantiated with any closed object of type `tp` and we do not associate them with an identity substitution.

In the case for lambda-abstractions, `count ϕ (arr B C) (ϕ ⊢ lam λx.M … x)`, we not only list the type of each of the variables occurring in the pattern, but also the induction hypothesis, `count (ϕ,x:tm B) C (ϕ,x:tm B ⊢ M …x):[nat]`. Although the context grows, the term itself is smaller. In the body of the case, we use the induction hypothesis to determine the size `X` of `M … x` in the context `ϕ, x:tm B` and then increment it.

The case for application, `count ϕ C (ϕ ⊢ app B C (M … )(N … ))`, is similar. We again list all the types of variables occurring in the pattern as well as the two induction hypotheses. In the body, we determine the size `X` of `(ϕ ⊢M … )` and the size `Y` of `(ϕ ⊢N … )` and then add them.

Computing the length of a context: Induction on the context

As we have the power to abstract and manipulate contexts as first-class objects, we also can reason inductively about them. Contexts are similar to lists and we distinguish between the empty context, written here as ∅, and a context consisting of at least one element, written as `ψ, x:tm A`. In the latter case, we can appeal to the induction hypothesis on ψ (see Listing 3).

3 Core language with well-founded recursion

In this section, we present the core of Beluga's computational language which allows the manipulation of contextual LF objects by means of higher-order functions and primitive recursion. In our presentation of the computation language we keep however our domain abstract simply referring to U, the type of a domain object, and C, the object of a given domain. In fact, our computational language is parametric in the actual domain. To guarantee totality of a program, the domain needs to provide answers for two main questions: 1) how to split on a domain type U and 2) how to determine whether a domain object C is smaller according to some domain-specific measure. We also need to know how to unify two terms and determine when two terms in our domain are equal. In terms of proof-theoretical strength, the language is comparable to Gödel's T or Heyting Arithmetic where the objects of study are natural numbers. However in our case, U will stand for a (contextual) LF type and C describes a (contextual) LF object.

Types	$\mathcal{I}, \tau$	$::= [U] \mid \tau_1 \to \tau_2 \mid \Pi X{:}U.\tau$
Expressions	e	$::= y \mid [C] \mid \mathsf{fn}\, y{:}\tau \Rightarrow e \mid e_1\ e_2 \mid \Lambda X{:}U \Rightarrow e \mid e\ C$
		$\mid\ \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \mid \mathsf{rec{-}case}^{\mathcal{I}} C\ \mathsf{of}\ \overrightarrow{b}$
Branches	b	$::= \Delta; \overrightarrow{r}\ .\ r \Rightarrow e$
Assumptions	r	$::= f\ \overrightarrow{C}\ C$
Contexts	Γ	$::= \cdot \mid \Gamma, y{:}\tau \mid \Gamma, r : \tau$
Meta Context	Δ	$::= \cdot \mid \Delta, X{:}U$

We distinguish between computation variables, simply referred to as variables and written using lower-case letter y; variables that are bound by Π-types and Λ-abstraction are referred to as meta-variables and written using upper-case letter X. Meta-variables occur inside a domain object. For example we saw earlier the object `(`$\psi \vdash$ `app B C (M…)(N…))`. Here, ψ, `B`, `C`, `M`, and `N` are referred to as meta-variables.

There are three forms of computation-level types τ. The base type $[U]$ is introduced by wrapping a contextual object C inside a box; an object of type $[U]$ is eliminated by a let-expression effectively unboxing a domain object. The non-dependent function space $\tau_1 \to \tau_2$ is introduced by function abstraction $\mathsf{fn}\, y{:}\tau_1 \Rightarrow e$ and eliminated by application $e_1\, e_2$; finally, the dependent function type $\Pi X{:}U.\tau$ which corresponds to universal quantification in predicate logic is introduced by abstraction $\Lambda X{:}U \Rightarrow e$ over meta-variables X and eliminated by application to a meta objects C written as $e\, C$. The type annotations on both abstractions ensure that every expression has a unique type. Note that we can index computation-level types τ only 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 pattern matching on a meta-object C using rec−case-expressions. Note that one cannot match on a computational object e directly; instead one can bind an expression of type $[U]$ to a meta variable X using let and then match on X. We annotate the recursor rec−case with the type of the inductive invariant $\Pi\Delta_0.\tau_0$ 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, Δ_0 is $\Delta_1, X_0{:}U_0$ where Δ_1 exactly describes the free meta-variables occurring in U_0. The intention is that we induct on objects of type U_0 which may depend on Δ_1. Δ_0 must therefore contain at least one declaration. We also give the return type τ_0 of the recursor, since it might also depend on Δ_0 and might be refined during pattern matching. This is analogous to Coq's `match_as_in_return_with_end` construct.

One might ask whether this form of inductive invariant is too restrictive, since it seems not to capture, e.g., $\Pi\Delta_0.(\tau \to \Pi X{:}U_0.\tau')$. While allowing more general invariants does not pose any fundamental issues, we simply note here that the above type is isomorphic to $\Pi(\Delta_0, X{:}U_0).\, \tau \to \tau'$ which is treated by our calculus. Forcing all quantifiers at the outside simplifies our theoretical development; however, our implementation is more flexible.

A branch b_i is expressed as $\Delta_i; \overrightarrow{r_i}\ .\ r_{i0} \Rightarrow e_i$. As shown in the examples, we explicitly list all pattern variables (i.e. meta-variables) occurring in the pattern in Δ_i. In practice, they often can be inferred (see for example [12]). 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 also derived dynamically while we check that a given pattern r_{i0} is covering and we give an algorithm in Section 6.

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

$$\frac{\Gamma(y) = \tau}{\Delta;\Gamma \vdash y : \tau} \qquad \frac{\Gamma(r) = \tau \quad r = r'}{\Delta;\Gamma \vdash r' : \tau}$$

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

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

$$\frac{\mathcal{I} = \Pi\Delta_1.\Pi X_0{:}U_0.\tau_0 \quad \Delta \vdash C : [\![\theta]\!]U_0 \quad \Delta \vdash \theta : \Delta_1 \quad b_i : \mathcal{I}\ (\text{for all } i) \quad \vec{b}\ \mathtt{covers}\ \mathcal{I}}{\Delta;\Gamma \vdash \mathsf{rec{-}case}^{\mathcal{I}}\ C\ \mathsf{of}\ \vec{b} : [\![\theta, C/X]\!]\tau_0}$$

$\boxed{b : \mathcal{I}}$: Branch b satisfies the invariant $\mathcal{I}$

$$\frac{\text{for all } 0 \leq j \leq k\ .\ \Delta \vdash_{\mathcal{I}} r_j : \tau_j \qquad \Delta\ ;\ r_k{:}\tau_k, \ldots, r_1{:}\tau_1 \vdash e : \tau_0}{(\Delta; r_k\ \ldots\ r_1\ .\ r_0 \Rightarrow e) : \mathcal{I}}$$

$\boxed{\Delta \vdash_{\mathcal{I}} r : \tau'}$ and $\boxed{\Delta \vdash \vec{C} : \mathcal{I} > \tau'}$: Assumption r/pattern spine $\vec{C}$ has type τ' given $\mathcal{I}$

$$\frac{\Delta \vdash \vec{C} : \mathcal{I} > \tau'}{\Delta \vdash_{\mathcal{I}} f\ \vec{C} : \tau'} \qquad \frac{\Delta \vdash C : U \quad \Delta \vdash \vec{C} : [\![C/X]\!]\tau > \tau'}{\Delta \vdash C\ \vec{C} : \Pi X{:}U.\tau > \tau'} \qquad \frac{\Delta \vdash C : U}{\Delta \vdash C : \Pi Y{:}U.\tau > [\![C/Y]\!]\tau}$$

Figure 1 Type system for dependently-typed functional computation language

The identifier f in assumptions r denotes the local function that is essentially introduced by rec−case; this notation is inspired by primitive recursion in *Tutch* [1]. Currently, it just improves the readability of call patterns; however, it is vital for extensions to nested recursion.

3.1 Computation-level Type System

In the typing judgement (Fig. 1), we distinguish between the context Δ for meta-variables from our index domain and the context Γ which includes declarations of computation-level variables. 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 Λ-abstraction the meta-variable X must be new and cannot already occur in the context Δ. This can always be achieved via α-renaming. Similarly, in the rule for function abstraction, the variable y must be new and cannot already occur in Γ. We have two variable rules to look up a computation-level variable y and an induction hypothesis r. To verify that the induction hypothesis r' has type τ and its use is valid, we simply check whether there exists $r : \tau$ in Γ where $r = r'$. For now it suffices to think of $=$ as syntactically equivalent.

The most interesting rule is the one for recursion: given the invariant $\mathcal{I} = \Pi\Delta_1.\Pi X{:}U_0.\tau_0$ the expression $\mathsf{rec{-}case}^{\mathcal{I}}\ C$ of $\vec{b}$ 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, and all recursive calls are well-founded. Since the coverage check is domain specific, we leave it abstract for now and return to it when we consider (contextual)

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

$$\frac{\exists\,\text{unique}\,(\Delta.r_k,\ldots,r_1.r_0 \Rightarrow e) \in \vec{b} \text{ where } r_j = f\ \overrightarrow{C_j}\ C_{j0} \text{ such that } \vdash C \doteq C_{00}/\theta}{\mathsf{rec{-}case}^{\mathcal{I}}\ C\ \ \mathsf{of}\ \vec{b} \longrightarrow [\![\theta]\!][\,(\mathsf{rec{-}case}^{\mathcal{I}}\ C_{k0}\ \ \mathsf{of}\ \vec{b})/r_k,\ldots,(\mathsf{rec{-}case}^{\mathcal{I}}\ C_{10}\ \ \mathsf{of}\ \vec{b})/r_1\,]e}$$

Figure 2 Small-step semantics $e \longrightarrow e'$

LF as one possible domain (see Sec. 5).

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 $b = \Delta; \vec{r}.r_0 \Rightarrow e$ to be well-typed with respect to a given invariant $\mathcal{I}$, 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 $\vec{\tau}$. A pattern / recursive call $r_j = f\ \overrightarrow{C_j}$ intuitively corresponds to the given inductive invariant $\mathcal{I} = \Pi\Delta_1.\Pi X_0{:}U_0.\tau_0$, if the spine $\overrightarrow{C}$ matches the specified types in $\Delta_1, X_0{:}U_0$ and it has intuitively the type $[\![C_{jn}/X_n,\ldots,C_{j0}/X_0]\!]\tau_0$ which we denote with τ_j'.

More generally, we write $\Delta \vdash \theta : \Delta_0$ for a well-typed simultaneous substitution where Δ_0 is the domain and Δ is the range of the substitution. It can be inductively defined (see below) and the standard substitution lemmas hold (see for example [4]).

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \theta : \Delta_0 \qquad \Delta \vdash C : [\![\theta]\!]U}{\Delta \vdash \theta, C/X : \Delta_0, X : U}$$

4 Operational Semantics

Fig. 2 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−case. To reduce $(\mathsf{rec{-}case}^{\mathcal{I}}\ C\ \ \mathsf{of}\ \vec{b})$ we find the branch $(\Delta.r_k,\ldots,r_1.r_0 \Rightarrow e) \in \vec{b}$ such that the principal argument C_{00} of its clause head $r_0 = f\ \overrightarrow{C_0}\ C_{00}$ matches C under meta substitution θ. The reduct is the body e under θ where we additionally replace each place holder r_j of a recursive call by the actual recursive invocation $(\mathsf{rec{-}case}^{\mathcal{T}}\ [\![\theta]\!]C_{j0}\ \ \mathsf{of}\ \vec{b})$. The object C_{j0} in fact just denotes the meta-variable on which we are recursing. We also apply θ to the body e. In the rule, we have lifted out θ. *Values* v in our language are boxed meta objects $[C]$, functions $\mathsf{fn}\, x{:}\tau \Rightarrow e$, and $\Lambda X{:}U.e$.

▶ **Theorem 1** (Subject reduction). *If* $\cdot;\cdot \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\cdot;\cdot \vdash e' : \tau$.

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

▶ **Theorem 2** (Progress). *If* $\cdot;\cdot \vdash e : \tau$ *then either* e *is a value or* $e \longrightarrow e'$.

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

5 Contextual LF: Background, Measure, Splitting

If we choose as our domain natural numbers or lists, it may be obvious how to define splitting together with a measure that describes when an object is smaller. Our interest however is to use the contextual logical framework LF [9] as a general domain language. Contextual LF extends the logical framework LF [8] by packaging an LF objects M of type A together with the context Ψ in which it is meaningful. This allows us to represent rich syntactic structures such as open terms and derivation trees that depend on hypotheses. The core language introduced in Sec. 3 then allows us to implement well-founded recursive programs over these rich abstract syntax trees that correspond to proofs by structural induction.

5.1 Contextual LF

We briefly review contextual LF here. As usual we consider only objects in η-long β-normal form, since these are the only meaningful objects in LF. Further, we concentrate on characterizing well-typed terms; spelling out kinds and kinding rules for types is straightforward.

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	σ	$::= \cdot \mid \mathsf{id}_\psi \mid \sigma, M \mid \sigma; H$
Variable Substitutions	π	$::= \cdot \mid \mathsf{id}_\psi \mid \sigma; x$
LF Contexts	Ψ, Φ	$::= \cdot \mid \psi \mid \Psi, x{:}A$

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 λ-abstraction or in a context Ψ. Contextual LF extends LF by allowing two kinds of *contextual variables*: the meta-variable u has type $(\Psi.P)$ and stands for a general LF object that has type P and may use the variables declared in Ψ; the parameter variable p has type $\#(\Psi.A)$ and stands for an LF variable object of type A in the context Ψ.

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 the 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

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

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

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

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

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

$$\frac{\Delta; \Psi, x{:}A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.\, M \Leftarrow A \to B} \qquad \frac{\Delta(u) = \Phi.P \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad Q = [\sigma]P}{\Delta; \Psi \vdash u[\sigma] \Leftarrow Q}$$

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

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

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

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

Figure 3 Bi-directional typing for contextual LF

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* [20] 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 operation $\mathsf{id}(\Psi)$ that generates an identity substitution for a given context Ψ is defined inductively as follows: $\mathsf{id}(\cdot) = \cdot$, $\mathsf{id}(\psi) = \mathsf{id}_\psi$, and $\mathsf{id}(\Psi, x{:}A) = \mathsf{id}(\Psi); x$.

We summarize the bi-directional type system for contextual LF in Fig. 3. 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.

5.2 Meta-level Terms and Typing Rules

We lift contextual LF objects to meta-objects to have a uniform definition of all meta-objects. We also define context schemas G that classify contexts.

$$\begin{array}{llll} \text{Context Schemas} & G & ::= \exists\Phi^0.B \mid G + \exists\Phi^0.B & \\ \text{Meta Types} & U, V & ::= \Psi.P \mid G \mid \#\Psi.A & \qquad \text{Meta Objects} \quad C, D \ ::= \hat{\Psi}.R \mid \Psi \end{array}$$

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.

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). A 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, see [9, 4]. We summarize the typing rules for meta-objects below.

$$\boxed{\Delta \vdash C : U} \text{ Check meta-object } C \text{ against meta-type } U \qquad \frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \hat{\Psi}.R : \Psi.P}$$

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

We write $\hat{\Psi}$ for a list of variables obtained by erasing the types from the context Ψ. We have omitted the rules for parameter types $\#\Psi.A$ because they are not important for the further development. Intuitively an object R has type $\#\Psi.A$ if R is either a concrete variable x of type A in the context Ψ or a parameter variable p of type A in the context Ψ. This can be generalized to account for re-ordering of variables allowing the parameter variable p to have some type A' in the context Ψ' s.t. there exists a permutation substitution π on the variables such that $\Psi \vdash \pi : \Psi'$ and $A = [\pi]A'$.

5.3 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[10]. For simplicity, we only consider here non-mutual recursive type families; those can be incorporated using the notion of subordination [19].

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 $\preceq$.

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

$$\frac{\hat{\Psi}.M \preceq \hat{\Phi}.N_i \quad \text{for some } 1 \leq i \leq n}{\hat{\Psi}.M \prec \hat{\Phi}.\ h \cdot N_1 \ \ldots \ N_n \ \mathsf{nil}} \qquad \frac{\hat{\Psi}.M \prec \hat{\Phi}.N}{\hat{\Psi}.M \preceq \hat{\Phi}.N} \qquad \frac{\hat{\Psi}.M \equiv \hat{\Phi}.N}{\hat{\Psi}.M \preceq \hat{\Phi}.N} \qquad \frac{\hat{\Psi}.M \preceq \hat{\Phi}, x.N}{\hat{\Psi}.M \preceq \hat{\Phi}.\lambda x.N}$$

$\hat{\Psi}.M$ is a strict subterm of $\hat{\Phi}.N$ if M is a proper subterm of N modulo α-renaming and weakening. Two terms $\hat{\Psi}.M$ and $\hat{\Phi}.N$ are structurally equivalent, if they describe the same term modulo α-renaming and possible weakening. To allow mutual recursive definitions and richer subterm relationships, we can incorporate subordination information and generalize the variable substitution π (see for example [10] for such a generalization). 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 $||M||$ of M. The latter is defined inductively by:

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

▶ **Theorem 3** (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 $||[\![\theta]\!]C|| \ \leq \ ||[\![\theta]\!]C'||$.*

5.4 Case Splitting

Our language allows pattern matching and recursion over contextual objects. For well-formed recursors ($\mathsf{rec{-}case}^{\mathcal{I}}\ C$ of $\vec{b}$) with invariant $\mathcal{I} = \Pi\Delta.\Pi X{:}U.\tau$, 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 intuitively the set $\mathcal{U}_{\Delta \vdash U}$ contains all neutral terms $R = H \cdot S$ where H is a constructor c, a concrete variable x from Ψ or a parameter variable $p[\mathsf{id}_\psi]$ denoting a variable from the context variable ψ, and S is a most general spine s.t. the type of R is an instance of P in the context Ψ. We note that when considering only closed terms it suffices to consider only terms with $H = c$. However, when considering terms with respect to a context Ψ, we must generate additional cases covering the scenario where H is a variable—either a concrete variable x if $x{:}A$ is in Ψ or a parameter variable if the context is described abstractly using a context variable ψ.

If U denotes a context schema G, we generate all shallow context patterns of type G. This includes the empty context and a context extended with a declaration formed by $\Psi, x{:}A$.

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}, …, 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.

Splitting on a Contextual Type

Following [5, 17], 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.

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

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

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

$\boxed{\Delta; \Psi \vdash R : A \Leftarrow P/(\Delta', \theta, R_0)}$ Extending $R{:}A$ to most general normal term $R_0 : [\![\theta]\!]P$.

$$\frac{\Delta; \Psi \vdash Q \doteqdot P\ /\ (\Delta_0,\ \theta)}{\Delta; \Psi \vdash R : Q \Leftarrow P\ /\ (\Delta_0\ ,\ \theta\ ,\ [\![\theta]\!]R)}$$

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

Figure 4 Generation of most general normal objects and call patterns

See Fig. 4. 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 $\Delta' \vdash \theta : \Delta$.

$\boxed{\Delta; \Psi \vdash R : A \Leftarrow P\ /\ (\Delta', \theta, R_0)}$ describes the generation of a normal pattern 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 $\Delta'; [\![\theta]\!]\Psi \vdash R_0 \Leftarrow [\![\theta]\!]P$. To generate a normal 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.

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

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

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

▶ **Theorem 4** (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. $\mathcal{U}_{\Delta\vdash\Psi.P}$ is non-redundant since all generated neutral terms have distinct heads. Completeness is proven by cases. ◀

6 Generation of Call Patterns and Coverage

Next, we explain the generation of call patterns, i.e. well-founded recursive calls as well as the actual call pattern being considered.

▶ **Definition 5** (Generation of call patterns). Given the invariant $\mathcal{I} = \Pi(\Delta_1, X_0{:}U_0).\tau_0$ where $\Delta_1 = X_n{:}U_n, \dots, X_1{:}U_1$, the set $\mathcal{C}$ of call patterns $(\Delta_i \ ; \ r_{ik}{:}\tau_{ik}, \dots, r_{i1}{:}\tau_{i1} \ . \ r_{i0})$ is generated as follows:

- For each meta-object $\Delta_i \vdash C_{i0} : V_i$ in $\mathcal{U}_{\Delta_0 \vdash U_0}$, we generate using unification, if possible, a call pattern $r_{i0} = f \ C_{in} \dots C_{i1} \ C_{i0}$ s.t. $\tau_{i0} = [\![C_{in}/X_n, \dots, C_{i1}/X_1, C_{i0}/X_0]\!]\tau_0$ and $\Delta_i \vdash r_{i0} : \tau_{i0}$. This may fail if V_i is not an instance of the scrutinee type U_0; then, the case C_{i0} is impossible.
- Further, for all $1 \leq j \leq k$, $\Delta_i = Y_k{:}V_k, \dots, Y_1{:}V_1$, we generate a recursive call $r_{ij} = f \ C_{jn} \dots C_{j1} \ Y_j$ s.t. $\tau_{ij} = [\![C_{jn}/X_n, \dots, C_{j1}/X_1, Y_j/X_0]\!]\tau_0$ and $\Delta_i \vdash r_{ij} : \tau_{ij}$, if $Y_j \prec C_{i0}$. This may also fail, if V_i is not an instance with U_0; in this case V_i does not give rise to recursive call.

▶ **Theorem 6** (Pattern generation). *The set $\mathcal{C}$ of call patterns generated is non-redundant and complete and the recursive calls are well-founded.*

Proof. Using Theorem 4 and the properties of unification. ◀

▶ **Definition 7** (Coverage). We say $\boxed{\vec{b} \text{ covers } \mathcal{I}}$ iff for every $\Delta_i \ ; \ \overrightarrow{r_i{:}\tau_i} \, . \, r_{i0} \in \mathcal{C}$ where $\mathcal{C}$ is the set of call patterns given $\mathcal{I}$, we have one corresponding $\Delta_i \ ; \ \overrightarrow{r_i{:}\tau_i} \, . \, r_{i0} \Rightarrow e_i \in \vec{b}$ and vice versa.

7 Termination

We now prove that every well-typed closed program e terminates (halts) by a standard reducibility argument; closely related is [21]. The set $\mathcal{R}_\tau$ of reducible closed programs $\cdot ; \cdot \vdash e : \tau$ is defined by induction on the size of τ.

$$
\begin{array}{lll}
\text{Contextual Type} & \mathcal{R}_{[U]} & = \{e \mid \cdot;\cdot \vdash e : [U] \text{ and } e \text{ halts}\} \\
\text{Function Type} & \mathcal{R}_{\tau' \to \tau} & = \{e \mid \cdot;\cdot \vdash e : \tau' \to \tau \text{ and } e \text{ halts and } \forall e' \in \mathcal{R}_{\tau'}.\, e\ e' \in \mathcal{R}_\tau\} \\
\text{Dependent Type} & \mathcal{R}_{\Pi X:U.\tau} & = \{e \mid \cdot;\cdot \vdash e : \Pi X{:}U.\tau \text{ and} \\
 & & \quad e \text{ halts and } \forall C s.t. \cdot \vdash C : U.\, e\ C \in \mathcal{R}_{[\![C/X]\!]\tau}\} \\
\text{Context} & \mathcal{R}_\Gamma & = \{\eta \mid \cdot;\cdot \vdash \eta : \Gamma \text{ and } \eta(x) \in \mathcal{R}_\tau \text{ for all } (x{:}\tau) \in \Gamma\}
\end{array}
$$

For the size of τ all meta types U shall be disregarded, thus, the size is invariant under meta substitution C/X. We also note that since reduction $e \longrightarrow e'$ is deterministic, e halts if and only if e' halts.

▶ **Lemma 8** (Expansion closure).

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

Proof. The first statement, by induction on the size of type τ. The second statement, inductively on $\longrightarrow^*$. ◀

▶ **Lemma 9** (Fundamental Lemma).
If $\Delta;\Gamma \vdash e : \tau$ and grounding substitution θ s.t. $\cdot \vdash \theta : \Delta$ 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−case, we make essential use of coverage and structural descent in the recursive calls. ◀

▶ **Theorem 10** (Termination). *If $\cdot;\cdot \vdash e : \tau$ then e halts.*

Proof. Taking the 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 τ. ◀

8 Related Work

Our work is most closely related to [16] where the authors propose a modal lambda-calculus with iteration to reason about closed HOAS objects. In their work the modal type $\Box$ describes closed LF objects. Our work extends this line to allow open LF objects and define functions by pattern matching and well-founded recursion.

Similar to our approach, Schürmann [15] presents 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, but maybe more importantly complicates establishing meta-theoretic results such as proving normalization.

Establishing well-founded induction principles to support reasoning about higher-order abstract syntax specifications has been challenging and a number of alternative approaches have been proposed. These approaches have led to new reasoning logics—either based on nominal set theory [14] or on nominal proof theory [7]. Proving consistency of these theories in the presence of induction is often significantly more complicated [18]. Our work shows that reasoning about HOAS representations can be supported using first-order logic by modelling HOAS objects as contextual objects. As a consequence, we can directly take advantage of established proof and mechanization techniques. This also opens up the possibility of supporting contextual reasoning as a domain in other systems.

9 Conclusion

We have 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−case-expression with a list of invariants using the subordination relation, we can generate well-founded recursive calls matching each of the invariants. Based on these ideas, we have implemented a totality checker in Beluga. We also added reasoning principles for inductive types [4] that follow 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 and to generalize our work to allow lexicographic orderings and general well-founded recursion.

Acknowledgements We thank Sherry Shanshan Ruan for her work during her Summer Undergraduate Research Internship in 2013 at the beginning of this project.

References

1 Andreas Abel. *Tutch User's Guide.* Carnegie-Mellon University, Pittsburgh, PA, 2002. Section 7.1: Proof terms for structural recursion.

2 Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In TLCA'11, volume 6690 of *LNCS*, pages 10–26. Springer, 2011.

3 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.

4 Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *POPL'12*, pages 413–424. ACM, 2012.

5 Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. *ENTCS*, 228:69–84, 2009.

6 Francisco Ferreira and Brigitte Pientka. Bidirectional elaboration of dependently typed languages. In *PPDP'14*. ACM, 2014.

7 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *LICS'08*, pages 33–44. IEEE CS Press, 2008.

8 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.

9 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM TOCL*, 9(3):1–49, 2008.

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

11 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.

12 Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *JFP*, 1(1–37), 2013.

13 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.

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

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

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

17 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.

18 Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *J. Applied Logic*, 10(4):330–367, 2012.

19 Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. CMU-CS-99-167.

20 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.

21 Hongwei Xi. Dependent types for program termination verification. *HOSC*, 15(1):91–131, 2002.