Under consideration for publication in Math. Struct. in Comp. Science

Programming proofs: a novel approach based on contextual types

Brigitte Pientka

Received February 2011

We present an overview of Beluga, a dependently-typed programming and proof development environment. Beluga uses a two-level approach: it supports specifying formal systems within the logical framework LF and on top of LF, it provides a dependently-typed functional language that supports manipulating and analyzing LF objects via pattern matching. A distinct feature of Beluga is the explicit support for contexts and contextual objects, which concisely characterize objects depending on assumptions. The design of the dependently-typed functional language is generic and independent of the concrete specification language which in Beluga's case is LF. Moreover, it is unique in its treatment of pattern matching in the presence of dependent types: it is centered around refinement substitutions which refine the types of patterns in case-expressions. This makes type checking decidable although constraint solving itself could be undecidable. In addition to a bi-directional decidable type system for Beluga we give an environment-based operational semantics and show that types are preserved. Beluga together with type reconstruction is implemented in OCaml and has been used on a wide variety of examples such as type uniqueness, proofs about compiler transformations, and preservation and progress for various ML-like languages.

1. Introduction

Formal systems, for example type systems or authorization logics, play a central role in describing and statically verifying guarantees about the runtime behaviour of programs and are commonly used nowadays. Unfortunately, we lack a common infrastructure to integrate the formal specification of the behaviour and the reasoning about these properties in actual programs. Dependently typed systems allow us to track rich properties statically and provide one solution to this problem. However, to elegantly and compactly encode formal systems and program with proofs, we need also good answers to the following questions: how do we represent binders and, more generally, how to represent proof objects which depend on assumptions.

This paper presents an overview of the programming and reasoning framework called Beluga [Pientka 2008; Pientka and Dunfield 2008]. Beluga uses a two-level approach: it supports specifying formal systems within the logical framework LF [Harper et al. 1993] and programming with LF (proof) objects using a dependently-typed functional language with pattern matching. The LF methodology has been successfully used to define logics and represent derivations and proofs. Its strength and elegance comes from supporting encodings based on higher-order abstract syntax (HOAS), in which binders in the object language are represented as binders in the LF meta-language. Several advantages are well-known: HOAS obviates the cumbersome, low-level bureaucracy of α-renaming and, most notably, capture-avoiding substitutions. The combination of HOAS and dependent types makes LF encodings one of the most advanced technologies for prototyping formal systems, leading to concise and elegant encodings and providing the most support for such an endeavour. In Beluga, we generalize LF to support contextual objects and first-class contexts continuing work by Nanevski et al. [2008]. This allows us to describe and manipulate (proof) objects which depend on a context of assumptions. The type $A[\Psi]$ describes a contextual object which has type A in the context Ψ and hence may refer to the variables and assumptions declared in the context Ψ. Moreover, we support context variables to abstract over concrete contexts.

On top of contextual LF, we develop a dependently typed functional language that supports analyzing and manipulating contextual objects via pattern matching. Its design is generic and independent of the concrete specification language which in Beluga's case is LF. Moreover, we describe a novel treatment of pattern matching in the presence of dependent types: it is centered around refinement substitutions which refine the types of patterns in case-expressions. This makes type checking decidable and easy to trust, although constraint solving itself could be undecidable. This work generalizes our previous presentations [Pientka 2008; Pientka and Dunfield 2008] and forms the internal core language of Beluga. In addition to the static semantics, we describe a novel environment-based operational semantics and prove types are preserved.

We have implemented the programming and reasoning environment Beluga [Pientka and Dunfield 2010b] in OCaml. It reimplements LF [Harper et al. 1993] including LF type reconstruction [Pientka 2010], constraint-based higher-order unification and type checking. In addition we also provide an interpreter to execute programs according to the environment-based semantics and a coverage checker [Dunfield and Pientka 2009]. We have tested Beluga on all examples from the Twelf [Pfenning and Schürmann 1999] repository in addition to new problems such as the ones described by Felty and Pientka [2010]. Beluga has three main application domains:

Programming with names and binders A wide range of programs, in particular program optimizers, type checkers, automated reasoning engines etc., transform data structures that contain binders. Several approaches to this problem exist [Shinwell et al. 2003; Pottier 2007; Pouillard and Pottier 2010; Licata and Harper 2009; Schürmann et al. 2005]. Contextual types and contextual objects are one solution to the puzzles surrounding programming with binders.

Programming as proving. Due to its support for dependent types, Beluga's main application at the moment is to prototype formal systems together with their meta-theory. Formal systems are specified in the logical framework LF, and proofs about them can be represented as recursive programs in Beluga, i.e. recursive programs about LF objects. It is most closely related to logical frameworks such as Delphin [Schürmann et al. 2005] and Twelf [Pfenning and Schürmann 1999].

Programming with proofs. Beluga also provides an experimental framework for programming with proof objects. Its powerful type system not only allows the programmer to enforce strong invariants about programs statically, but to create, manipulate, and analyze certificates (proofs) that guarantee that a program satisfies a user-defined safety property. Therefore, Beluga is ideally suited for applications such as certified programming and proof-carrying code [Necula 1997]. Most recently, it has also inspired the design of novel, strongly typed tactic languages such as VeriML [Stampoulis and Shao 2010] which support modular proof development.

Overview

The paper begins with a discussion of implementing a recursive program which normalizes typed lambda-terms in Beluga. Using this example we introduce Beluga's surface language and at the same time illustrate some of the key ideas underlying Beluga's design (Section 2). We then introduce the theoretical foundation for contextual LF (Section 3) and the computation language (Section 4). In particular, we present the static and operational semantics together with the type preservation proof in Section 4.2. Finally, we discuss implementation (Section 5) and survey related work (Section 6).

Acknowledgement

The author thanks Andreas Abel. His insightful remarks and his desire to understand the core ideas helped shape this presentation of this paper. The author would also like to thank Ali Assaf for his initial work on the lazy operational semantics for Beluga.

2. Background: Programming in Beluga

We briefly illustrate the main idea behind Beluga by implementing a program which normalizes λ-terms. Dependent types ensure that we are only manipulating well-typed terms and the final normal form preserves the type of the original term.

Representation of simply-typed lambda-terms in LF

We first represent well-typed lambda-terms in the logical framework LF and subsequently show how to implement a normalizer for simply-typed lambda-terms as a recursive function. The definition for types in LF is straightforward and since several excellent tutorials and notes exist already [Pfenning 1997; Twelf Wiki], we will keep this short.

We introduce an LF type `tp` and define type constructors `nat` and `arr`. Next, we represent terms with the goal to only characterize well-typed lambda-terms. We will achieve this by indexing the type of expressions. In addition, we will employ higher-order abstract syntax to encode the binder in the object-language by binders in the meta-language, namely LF. Hence, the constructor `lam` takes in a meta-level abstraction of type `(exp T1 → exp T2)`. To illustrate, consider the object-level term $\lambda x.\lambda y.x\ y$. It is represented as `lam λx. lam λy. app x y` in LF.

```
tp: type.                         exp: tp → type.
nat: tp.                          lam : (exp T1 → exp T2) → exp (arr T1 T2).
arr: tp → tp → tp.                app : exp (arr T1 T2) → exp T1 → exp T2.
```

Following Twelf's methodology, type reconstruction for LF will infer the type of the free variables `T1` and `T2`. The general recipe is that if a variable was free in the declared type of a constant, then we must omit passing a concrete instantiation for it, when we use the constant. For example, the user must write `lam` λ`x. lam` λ`y. app x y`, although the constant `lam` and the constant `app` both take in two additional arguments for `T1` and `T2` respectively. We will not discuss type reconstruction for LF signatures in this paper, but refer the interested reader to Pientka [2010] which provides an algorithmic description for LF type reconstruction and discusses the challenges in detail.

Implementation of normalization in Beluga

We now discuss the implementation of the normalization algorithm. Intuitively, we will implement a function which when given a lambda-term M in a context Ψ, it produces a lambda-term N in the same context Ψ which will be in normal form. We interpret every lambda-term M within a context Ψ, i.e. M is closed with respect to the context Ψ, since we want to make reductions inside λ-abstractions. This allows us to ensure we are working with well-scoped lambda-terms, i.e. variables do not escape their scope. Since the context Γ keeps track of variables, it will grow as we traverse a lambda-abstraction and we define its shape using *context schemas* in Beluga as follows:

```
schema ctx = some [t:tp] exp t;
```

Schemas classify contexts just as types classify terms. The schema `ctx` describes a context which contains assumptions `x:exp t` for some type `t`. In other words, all declarations occurring in a context of schema `ctx` are instances of `exp t` for some `t`.

Next, we define the type of the function `norm`. Since we index expressions with their types, our statement will naturally enforce that types are preserved. The type will state that "for all contexts Ψ, given an expression M of type T in the context Ψ, we return an expression N of type T in the context Ψ". In Beluga's concrete syntax, this is written as `{`ψ`:ctx} (exp T)[`ψ`]` $\rightarrow$ `(exp T)[`ψ`]` where we use `{`ψ`:ctx}` to denote quantification over contexts.

Writing `{`ψ`:ctx}` in concrete syntax corresponds to quantifying over concrete contexts Ψ; the context variable ψ has schema `ctx`. While we quantify over contexts explicitly in the type, the user can omit passing the context to the function `norm` and type reconstruction will infer the appropriate context. At this point, we do not infer the schema for a context, and hence this information must always be provided.

The contextual type `(exp T)[`ψ`]` directly describes an expression M with type `T` in the context ψ. The element inhabiting the computation-level type `(exp T)[g]` is called a contextual object, since they may refer to variables listed in the context ψ and hence only make sense within the context ψ. For example, the contextual object `[x:exp nat] lam` λ`y.app y x` has type `(exp (arr (arr nat nat) nat))[x:exp nat]` and describes an expression which may refer to the bound variable `x`. Weakening is built-in; for example the object `lam` λ`y.y` can be used in a context `[x:exp]` or it can be used in the empty context `[]`. Accordingly, `[x:exp nat] lam` λ`y. y` has type `(exp (arr nat nat))[x:exp nat]` and `[] lam` λ`y.y` has type `exp (arr nat nat)[]`.

The variable `T` which is free in the specified computation-level type is implicitly quantified at the outside and has type `tp[]` denoting a closed object of type `tp`. Type reconstruction will infer the type of `T` and abstract over it. Since in general free variables occurring in computation-level types may depend on the context in which they occur, we always will create a prefix which first lists context variables and then the types of free variables. This leads to the following reconstructed type for `norm`:

```
{ψ:(ctx)}{T:tp[]} (exp T)[ψ] → (exp T)[ψ]
```

We highlight the prefix which denotes the implicit arguments of the function `norm` while we keep the explicit arguments in black. When using the function `norm` we must omit passing instantiations for the context ψ and the type `T`. Type reconstruction will infer the concrete instantiation to be passed. We will now show the recursive function which implements the normalization algorithm. The function proceeds by pattern matching on elements of type `(exp T)[ψ]`. For better readability, we write all contextual objects in its η-expanded form writing `lam λx.M … x` instead of simply `lam (M … )`.

```
rec norm : {ψ:ctx}(exp T)[ψ] → (exp T)[ψ] =
fn e ⇒ case e of
| [ψ] #p …  ⇒ [ψ] #p …                                       % Variable

| [ψ] lam (λx. M …  x) ⇒                                      % Abstraction
  let [ψ,x:exp _ ] N …  x = norm ([ψ, x:exp _ ] M …  x) in
    [ψ] lam λx. N …  x

| [ψ] app (M1 … ) (M2 … ) ⇒                                   % Application
  (case norm ([ψ] M1 … ) of
    [ψ] lam (λx. M' …  x) ⇒ norm ([ψ] M' …  (M2 … ) )
  | [ψ] N1 …              ⇒
    let [ψ] N2 …  = norm ([ψ] M2 …  ) in [ψ] app (N1 … ) (N2 … )
   );
```

The Beluga syntax follows ideas from ML-like languages with a few extensions. In particular, we split on an object `e` which has contextual type `(exp T)[ψ]`. There are three cases to consider for `e`: it is either a variable from the context, it is a lambda-abstraction, or it is an application. Each pattern is written as a contextual object, i.e. the object itself together with its context. For the variable case, we use a *parameter variable*, written as `#p …` .Operationally, `#p …` in the pattern `[ψ] #p …` will match any declaration from the context ψ once ψ is concrete. The parameter variable `#p` is associated with the identity substitution (written in concrete syntax with `…`) to explicitly state its dependency on the context ψ.

The pattern `[ψ] lam λx. M … x` describes the case where the object `e` is a lambda-abstraction. We write `M … x` for the body of the lambda-abstraction which may refer to all the variables from the context ψ (written as `…`) and the variable `x`. Technically, `… x` describes the identity substitution which maps all the variables from `ψ, x:exp T` to themselves. We now recursively normalize the contextual object `[ψ,x: exp _ ] M … x`. We write an underscore for the type of `x` in the context `ψ, x:exp _` and let type reconstruction determine it. Note, that we cannot write `x:exp T1` since `T1` would be free. Hence, supporting holes is crucial to be able to write the program compactly and avoid unnecessary type annotations. The result of the recursive call is a contextual object `[ψ ,x:exp _ ] N … x` which we will use to assemble the result. In the case for applications, we recursively normalize the contextual object `[ψ] M1 …` and then pattern match on its result. If it returned a lambda-abstraction `lam λx. M' … x`, we simply replace `x` with `M2 …`. Substitution is primitive in Beluga and `… (M2 … )` describes the substitution which maps all variables in ψ to themselves (written as `…`) and `x` to `M2 …`. In the case where normalizing `[ψ] M1 …` does not return a lambda-abstraction, we continue normalizing `[ψ] M2 …` and reassemble the final result. In conclusion, our implementation yields a natural, elegant, and very direct encoding of the formal description of normalization.

3. Contextual LF

Beluga's specification language is an extension of the logical framework LF [Harper et al. 1993] where we also allow meta-variables, parameter variables and context variables. This continues our work on contextual types [Nanevski et al. 2008] and was previously described in [Pientka 2008; Pientka and Dunfield 2008]. We give here a more compact formulation merging kinds, types and terms. Following Watkins et al. [2002] we concentrate on normal forms, since these are the only objects of interest in the logical framework. While our grammar only enforces that objects are β-normal, our typing rules in Figure 1 will also ensure objects are in η-long form.

Sorts	s	$::=$	$\mathsf{type} \mid \mathsf{kind}$
Atomic types	P	$::=$	$\mathbf{a}\,\vec{M}$
Types/kinds	A, B, K	$::=$	$\mathsf{type} \mid P \mid \Pi x{:}A.B$
Heads	H	$::=$	$x \mid \mathbf{c} \mid p[\sigma] \mid \mathbf{a}$
Neutral Terms	R	$::=$	$H \mid R\,N \mid u[\sigma]$
Normal Terms	M, N	$::=$	$R \mid \lambda x.M$
Substitutions	σ	$::=$	$\cdot \mid \mathsf{id}_\psi \mid \sigma, M \mid \sigma; x$
Contexts	Ψ	$::=$	$\cdot \mid \psi \mid \Psi, x{:}A$
Signature	Σ	$::=$	$\cdot \mid \Sigma, \mathbf{a}{:}K \mid \Sigma, \mathbf{c}{:}A$

Normal objects may contain *ordinary bound variables* which are used to represent object-level binders and are bound by λ-abstraction. They may also contain meta-variables $u[\sigma]$ and parameter variables $p[\sigma]$ which we call *contextual variables.* Contextual variables are associated with a post-poned substitution σ. The 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 $\hat{\Psi}.R$ where R 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 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-typed, since x may not be of base type and in fact we do not know its type. Hence, we allow substitutions not only be extended with normal terms M but also with variables x.

Without loss of generality we require that meta-variables have base type; this can always be achieved using lowering.

A bound variable context Ψ contains bound variable declarations in addition to context variables. A context may only contain at most one context variable and it must occur at the left. This will make it easier to ensure bound variable dependencies are satisfied in the dependently typed setting.

3.1. *Meta-terms and Meta-types*

We also introduce a new class of meta-types and meta-terms to treat abstraction over meta-terms uniformly. Meta-terms are either contextual objects written as $\hat{\Psi}.R$ or contexts Ψ. These are the data-objects which computations manipulate and analyze. There are three different meta-types: $P[\Psi]$ denotes the type of a meta-variable u, $\#A[\Psi]$ denotes the type of a parameter variable p, and G describes the schema (i.e. type) of a context. The tag # on the type of parameter variables is a simple syntactic device to distinguish between the type of meta-variables and parameter variables. It does not introduce a subtyping relationship between the type $\#A[\Psi]$ and the type $A[\Psi]$. The meta-context in which an LF object appears uniquely determines if X denotes a meta-variable, parameter variable or context variable. We use the following convention: if X denotes a meta-variable we usually write u, or v; if it stands for a parameter-variable, we write p and for context variables we use ψ.

Context schemas	G	$::=$	$\exists\overrightarrow{(x{:}A)}.B \mid G + \exists\overrightarrow{(x{:}A)}.B$
Meta Terms	C	$::=$	$\hat{\Psi}.R \mid \Psi$
Meta Types	U	$::=$	$P[\Psi] \mid \#A[\Psi] \mid G$
Meta substitutions	θ	$::=$	$\cdot \mid \theta, C/X$
Meta-context	Δ	$::=$	$\cdot \mid \Delta, X{:}U$

Context schemas consist of different schema elements $\exists\overrightarrow{(x{:}A)}.B$ which are built using $+$. Intuitively, this means a concrete declaration in a context must be an instance of one of the elements specified in the schema.

The uniform treatment of meta-terms, called C, and meta-types, called U, allows us to give a compact definition of meta-substitutions θ and meta-contexts Δ.

A consequence of the uniform treatment of meta-terms is that the design of the computation language is modular and parameterized 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; in particular, it is straightforward to add substitution variables which were present in Pientka [2008] or allow for richer context schemas which are in fact supported in our implementation, but are here omitted for simplicity. Second, it will streamline the design of computations in Section 4.

Neutral Terms/Types $\boxed{\Delta; \Psi \vdash R \Rightarrow A}$

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

$$\frac{\Delta; \Psi \vdash R \Rightarrow \Pi x{:}A.B \quad \Delta; \Psi \vdash M \Leftarrow A}{\Delta; \Psi \vdash R\,M \Rightarrow [M/x]_A B} \quad \frac{u{:}P[\Phi] \in \Delta \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]_\Phi P}$$

Normal Terms $\boxed{\Delta; \Psi \vdash M \Leftarrow A}$

$$\frac{\Delta; \Psi \vdash R \Rightarrow P \quad \Delta; \Psi \vdash P = Q : \mathsf{type}}{\Delta; \Psi \vdash R \Leftarrow Q} \quad \frac{\Delta; \Psi, x{:}A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B}$$

Substitutions $\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Psi'}$

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \quad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Psi' \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]_{\Psi'} A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Psi', x{:}A}$$

$$\frac{}{\Delta; \psi, \Psi \vdash \mathsf{id}_\psi \Leftarrow \psi} \quad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Psi' \quad \Psi(x) = [\sigma]_{\Psi'} A}{\Delta; \Psi \vdash \sigma; x \Leftarrow \Psi', x{:}A}$$

LF Types and Kinds $\boxed{\Delta; \Psi \vdash A \Leftarrow s}$

$$\frac{}{\Delta; \Psi \vdash \mathsf{type} \Leftarrow \mathsf{kind}} \quad \frac{\Delta; \Psi \vdash P \Rightarrow \mathsf{type}}{\Delta; \Psi \vdash P \Leftarrow \mathsf{type}} \quad \frac{\Delta; \Psi \vdash A \Leftarrow \mathsf{type} \quad \Delta; \Psi, x{:}A \vdash B \Leftarrow s}{\Delta; \Psi \vdash \Pi x{:}A.B \Leftarrow s}$$

Fig. 1. Typing rules for LF with contextual variables and context variables

3.2. *Typing rules for contextual LF*

We use a bi-directional type system where we check normal terms against a type and synthesize a type for neutral terms. LF objects may depend on variables declared in the context Ψ and variables declared in the meta-context Δ, and hence all typing judgements for LF objects have access to both contexts. Finally, all typing judgments have access to a well-typed signature Σ where we store constants together with their types and kinds.

$\Delta; \Psi \vdash M \Leftarrow A$	Normal term M checks against type A
$\Delta; \Psi \vdash R \Rightarrow A$	Neutral term R synthesizes type A
$\Delta; \Psi \vdash \sigma \Leftarrow \Psi'$	Substitution σ has domain Ψ' and range Ψ.
$\Delta; \Psi \vdash A \Leftarrow s$	LF types and kinds are well-formed

The bi-directional typing rules given in Figure 1 are mostly straightforward. We will tacitly rename bound variables, and maintain that contexts and substitutions declare no variable more than once. Note that substitutions σ are defined only on ordinary variables x and not contextual variables. Moreover, we require the usual conditions on bound variables. For example in the rule for λ-abstraction the bound variable x must be new and cannot already occur in the context Ψ. This can be always achieved via α-renaming. Similarly, in meta-terms we tacitly apply α-renaming.

Moreover, we rely on hereditary substitutions, written as $[N/x]_A(B)$ to guarantee that when we substitute a term N which has type A for the variable x in the type B, we obtain a type B' which is in normal form. Hereditary substitutions continue to substitute, if a redex is created; for example, when replacing naively x by $\lambda y.c\ y$ in the object $x\ z$, we would obtain $(\lambda y.c\ y)\ z$ which is not in normal form and hence not a valid term in our grammar. Hereditary substitutions continue to substitute z for y in $c\ y$ to obtain $c\ z$ as a final result.

Hereditary substitution can be defined structurally considering the term to which the substitution operation is applied and the type of the object which is being substituted. We define the hereditary substitution operations for normal object, neutral objects and substitutions. The hereditary substitution operations will be defined by nested induction, first on the structure of the type A and second on the structure of the objects N, R, and σ. In other words, we either go to a smaller type, in which case the objects themselves can become larger, or the type remains the same and the objects become smaller. We write $A \leq B$ and $A < B$ if A occurs in B (as a proper subexpression in the latter case)[†]. For an in depth discussion, we refer the reader to Nanevski et al. [2008]. Hereditary substitution is defined in Figure 2.

If the original term is not well-typed, a hereditary substitution, though terminating, cannot always return a meaningful term. We formalize this as failure to return a result. However, on well-typed terms, hereditary substitution will always return well-typed terms. The definition for single hereditary substitutions can be easily extended to simultanous substitutions substitution written as $[\sigma]_\Psi(A)$. We annotate the substitution with the sub-script Ψ for two reasons. First, σ itself does not carry its domain and hence we will look up the instantiation for a variable x in σ/Ψ. Second, we rely on the type of x in the context Ψ to guarantee that applying σ to an object terminates. Either we apply σ to sub-expressions or the type of the object we substitute will be smaller. We state here termination property for single hereditary substitutions; it can be appropriately generalized to simultanous substitutions.

Lemma 3.1 (Termination).

1 If $[M/x]_A(R) = M' : A'$ then $A' \leq A$

2 $[M/x]_A(\_)$ terminates, either by returning a result or failing after a finite number of steps.

We have the following substitution properties.

Theorem 3.1 (Substitution property).

1 If $\Delta; \Psi, x{:}A \vdash J$ and $\Delta; \Psi \vdash M \Leftarrow A$ then $\Delta; \Psi \vdash [M/x]_A J$.

2 If $\Delta; \Psi \vdash J$ and $\Delta; \Psi' \vdash \sigma \Leftarrow \Psi$ then $\Delta; \Psi' \vdash [\sigma]_\Psi J$.

Proof. Each statement is proven by simultaneous induction on the typing derivation $\Delta; \Psi \vdash J$. □

[†] To ensure termination, it suffices to rely on type approximations of the dependent type; we leave this out from the discussion.

Normal Terms / Types			
$[M/x]_A(\Pi y{:}B_1.B_2)$	$=$	$\Pi y{:}B'_1.B'_2$	where $B'_1 = [M/x]_A(B_1)$ and $B'_2 = [M/x]_A(B_2)$, $y \notin \mathsf{FV}(M)$, and $y \neq x$
$[M/x]_A(\mathsf{type})$	$=$	type	
$[M/x]_A(\lambda y.N)$	$=$	$\lambda y.N'$	where $[M/x]_A(N) = N'$, $y \notin \mathsf{FV}(M)$, and $y \neq x$
$[M/x]_A(R)$	$=$	M'	if $[M/x]_A(R) = M' : A'$
$[M/x]_A(R)$	$=$	R'	if $[M/x]_A(R) = R'$
$[M/x]_A(N)$		fails	otherwise
Neutral terms			
$[M/x]_A(x)$	$=$	$M : A$	
$[M/x]_A(y)$	$=$	y	if $y \neq x$
$[M/x]_A(p[\sigma])$	$=$	$p[\sigma']$	where $[M/x]_A(\sigma) = \sigma'$
$[M/x]_A(u[\sigma])$	$=$	$u[\sigma']$	where $[M/x]_A(\sigma) = \sigma'$
$[M/x]_A(R\ N)$	$=$	$R'\,N'$	where $[M/x]_A(R) = R'$ and $[M/x]_A(N) = N'$
$[M/x]_A(R\ N)$	$=$	$M'' : B$	if $[M/x]_A(R) = \lambda y.M' : \Pi y{:}A_1.B$ where $\Pi x{:}A_1.B \leq A$ and $[M/x]_A(N) = N'$ and $[N'/y]_{A_1}(M') = M''$
$[M/x]_A(R)$		fails	otherwise
Substitution			
$[M/x]_A(\cdot)$	$=$	$\cdot$	
$[M/x]_A(\mathsf{id}_\psi)$	$=$	id_ψ	
$[M/x]_A(\sigma\ ,\ N)$	$=$	$(\sigma'\ ,\ N')$	where $[M/x]_A(\sigma) = \sigma'$ and $[M/x]_A(N) = N'$
$[M/x]_A(\sigma\ ;\ y)$	$=$	$(\sigma'\ ;\ y)$	if $y \neq x$ and $[M/x]_A(\sigma) = \sigma'$
$[M/x]_A(\sigma\ ;\ x)$	$=$	$(\sigma'\ ,\ M)$	if $[M/x]_A(\sigma) = \sigma'$
$[M/x]_A(\sigma)$		fails	otherwise

Fig. 2. Hereditary substitutions for LF objects with contextual variables

Before showing decidability of type checking, we remark on equality used in the type checking rules. Because all terms are in normal form, equality between two LF objects reduces to syntactic equality with one small caveat: Because we can build simultanous substitutions with norma terms as in σ, M and also with simply extending it with a variable $\sigma; x$, substitutions are β-normal, but may not be η-long. In other words, the substitution $\sigma, \lambda x.yx$ and $\sigma; y$ are equivalent. Syntactic equality must take into account η-contraction on substitutions whenever necessary. We are not in a position to state and prove decidability of type checking for contextual LF.

Meta Terms $\boxed{\Delta \vdash C \Leftarrow U}$

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

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \exists\overrightarrow{(x : B')}.B \in G \quad A = [\sigma]_{\overrightarrow{(x:B')}}B \quad \Delta; \Psi \vdash \sigma \Leftarrow \overrightarrow{(x{:}B')}}{\Delta \vdash \Psi, x{:}A \Leftarrow G}$$

$$\frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \hat{\Psi}.R \Leftarrow P[\Psi]} \qquad \frac{\Psi(x) = A}{\Delta \vdash \hat{\Psi}.x \Leftarrow \#A[\Psi]}$$

$$\frac{\Delta(p) = \#A[\Phi] \quad \text{where } \pi \text{ is a pattern substitution} \quad \Delta; \Psi \vdash \pi \Leftarrow \Phi \quad [\pi]_\Phi(A) = B}{\Delta \vdash \hat{\Psi}.p[\pi] \Leftarrow \#B[\Psi]}$$

Meta-Substitutions $\boxed{\Delta \vdash \theta \Leftarrow \Delta'}$

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

Meta-Types $\boxed{\Delta \vdash U \ \mathsf{mtype}}$

$$\frac{\Delta \vdash \Psi \ \mathsf{ctx} \quad \Delta; \Psi \vdash P \Leftarrow \mathsf{type}}{\Delta \vdash P[\Psi] \ \mathsf{mtype}} \qquad \frac{\Delta \vdash \Psi \ \mathsf{ctx} \quad \Delta; \Psi \vdash A \Leftarrow \mathsf{type}}{\Delta \vdash \#A[\Psi] \ \mathsf{mtype}}$$

$$\frac{\text{for all } \exists\overrightarrow{(x{:}A)}.B \in G. \ \vdash \overrightarrow{(x{:}A)} \ \mathsf{ctx} \ \text{ and } \ \cdot; \overrightarrow{(x{:}A)} \vdash B \Leftarrow \mathsf{type}}{\Delta \vdash G \ \mathsf{mtype}}$$

Meta-Context $\boxed{\vdash \Delta \ \mathsf{mctx}}$

$$\frac{}{\vdash \cdot \ \mathsf{mctx}} \qquad \frac{\vdash \Delta \ \mathsf{mctx} \quad \Delta \vdash U \ \mathsf{mtype}}{\vdash \Delta, X : U \ \mathsf{mctx}}$$

Fig. 3. Typing rules for meta-terms and meta-types

Theorem 3.2 (Decidability of type checking for contextual LF).
All typing judgments for LF terms, LF types, LF substitutions, and LF contexts are decidable.

Proof. The typing judgments are syntax-directed and hereditary substitution is terminating; therefore the typing rules are clearly decidable. □

3.3. *Typing rules for meta-terms and meta-types*

On top of LF we define meta-terms and meta-types which provide contextual objects and context a first-class status. Meta-terms are accessed and manipulated by computations. We use the following judgments for type checking meta-level objects. Meta-level objects can depend on meta-context Δ.

$\vdash \Delta$ mctx	Check that meta-context Δ is well-formed
$\Delta \vdash U$ mtype	Check meta-type U is well-kinded in meta-context Δ
$\Delta \vdash C \Leftarrow U$	Check meta-term C against meta-type U in meta-context Δ
$\Delta \vdash \theta \Leftarrow \Delta'$	Check that meta-substitution θ has domain Δ' and range Δ

The typing rules for meta-terms and meta-types in Figure 3 are mostly straightforward. To check that a meta-term has a valid meta-type, we revert to LF type checking (see the first rule for checking $\hat{\Psi}.R$ has type $P[\Psi]$). Similarly, to verify that a meta-type is well-kinded, we revert to LF kind checking.

To type check meta-substitutions, we must check that a meta-term C has meta-type U and we add two more rules which allow us to verify that we have a valid instantiation for a parameter variable. There are two possibilities: first, we may instantiate a parameter variable of type $\#A[\Psi]$ with an ordinary bound variable from Ψ. This is written as $\hat{\Psi}.x$. Second, we may instantiate a parameter variable of type $\#B[\Psi]$ with another parameter $\hat{\Psi}.p[\pi_\Psi]$; note, we appropriately restrict the substitution associated with the parameter variable p to ensure that the term $p[\pi_\Psi]$ is itself only a variable and not an arbitrary term. To achieve this, we restrict the substitution which is associated with the parameter substitution p to be a pattern substitution, i.e. a substitution which maps distinct variables to distinct variables and is denoted with π.

Contexts must not only be well-formed but also check against a context schema. This follows similar ideas as in Schürmann [2000]. Intuitively, a context inhabits a context schema $G = \exists\overrightarrow{(x{:}A)}.B'$, if every declaration $x_i{:}B_i$ in the context is an instance of the schema element s.t. $[\sigma_i]_{\overrightarrow{(x:A)}} B' = B_i$.

3.4. *Meta-substitution*

The two classes of variables, ordinary variables declared in the context Ψ and variables declared in the meta-context Δ, give rise to two different substitution operations. The single meta-substitution operation, written as $[\![C/X]\!]_U(M)$ (and $[\![C/X]\!]_U(R)$ etc.) and the simultanous meta-substitution written as $[\![\theta]\!]_\Delta(M)$ (or $[\![\theta]\!]_\Delta(R)$ etc.). Subsequently, we define the application of the single meta-substitution to a given term and type, but the simultanous meta-substitution definition can be easily derived from it.

When we apply $[\![C/X]\!]_U$ to $u[\sigma]$, we first apply $[\![C/x]\!]_U$ to σ to obtain σ'. Subsequently, we distinguish two cases: if $[\![C/X]\!]_U = [\![\hat{\Psi}.R/u]\!]$, then continue to apply σ' to R appropriately annotating σ' with its domain Ψ. Annotating meta-substitution with their domain allows us to subsequently annotate the operation $[\sigma']R$ appropriately; otherwise, we simply return $u[\sigma']$. The typing rules ensure that the type of the instantiation $\hat{\Psi}.R$ and the type of u agree, i.e. we can replace u which has type $P[\Psi]$ with a neutral term R if R has type P in the context Ψ. Because of α-conversion, the variables that are substituted at different occurrences of u may be different, and we write $\hat{\Psi}.R$ where $\hat{\Psi}$ binds all the free variables in R. We can always appropriately rename the bound variable in $\hat{\Psi}$ such that they match the domain of the post-poned substitution σ'. This complication can be eliminated in an implementation of the calculus based on de Bruijn indexes.

Applying the meta-substitution C/X to the parameter variable $p[\sigma]$ is similar. First, we substitute C for X in σ to obtain σ'. If $[\![C/X]\!]_U \neq [\![\hat{\Psi}.R/p]\!]_{\#A[\Psi]}$ then we simply return

$$\begin{array}{lcll}
\llbracket C/X \rrbracket_U(\Pi x{:}A.B) & = & \Pi x{:}A'.B' & \text{where } \llbracket C/X \rrbracket_U(A) = A' \text{ and } \llbracket C/X \rrbracket_U(B) = B' \\
\llbracket C/X \rrbracket_U(\lambda x.M) & = & \lambda x.M' & \text{where } \llbracket C/X \rrbracket_U(M) = M' \\
\llbracket C/X \rrbracket_U(u[\sigma]) & = & R' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & & \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.R/u \rrbracket_{P[\Psi]} \text{ and } [\sigma']_\Psi(R) = R' \\
\llbracket C/X \rrbracket_U(u[\sigma]) & = & u[\sigma']' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & & \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.R/u \rrbracket_{P[\Psi]} \\
\llbracket C/X \rrbracket_U(R\,N) & = & R\,'N' & \text{where } \llbracket C/X \rrbracket_U(R) = R' \text{ and } \llbracket C/X \rrbracket_U(N) = N' \\
\llbracket C/X \rrbracket_U(R\,N) & = & M'' : B & \text{if } \llbracket C/X \rrbracket_U(R) = \lambda y.M' : \Pi x{:}A_1.B \text{ where} \\
 & & & \Pi x{:}A_1.B \leq U \text{ and } N' = \llbracket C/X \rrbracket_U(N) \\
 & & & \text{and } M'' = [N'/y]_{A_1}(M') \\
 & & & \\
\llbracket C/X \rrbracket_U(x) & = & x & \\
\llbracket C/X \rrbracket_U(\mathbf{c}) & = & \mathbf{c} & \\
\llbracket C/X \rrbracket_U(\mathbf{a}) & = & \mathbf{a} & \\
\llbracket C/X \rrbracket_U(p[\sigma]) & = & R' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & & \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.R/p \rrbracket_{\#A[\Psi]} \text{ and} [\sigma']_\Psi(R) = R' \\
\llbracket C/X \rrbracket_U(p[\sigma]) & = & M' : A & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & & \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.R/p \rrbracket_{\#A[\Psi]} \text{ and } [\sigma']_\Psi(R) = M' : A \\
\llbracket C/X \rrbracket_U(p[\sigma]) & = & p[\sigma'] & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & & \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.R/p \rrbracket_{\#A[\Psi]} \\
 & & & \\
\llbracket C/X \rrbracket_U(\cdot) & = & \cdot & \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi) & = & \sigma & \text{where } \llbracket \mathcal{C}/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G \text{ and } \mathsf{id}(\Psi) = \sigma \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi) & = & \mathsf{id}_\psi & \text{where } \llbracket \mathcal{C}/X \rrbracket_U \neq \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\sigma, M) & = & \sigma', M' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and } \llbracket C/X \rrbracket_U(M) = M' \\
\llbracket C/X \rrbracket_U(\sigma; x) & = & \sigma'; x & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & & \\
\llbracket C/X \rrbracket_U(\cdot) & = & \cdot & \\
\llbracket C/X \rrbracket_U(\psi) & = & \Psi & \text{where } \llbracket C/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\psi) & = & \psi & \text{where } \llbracket C/X \rrbracket_U \neq \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\Psi, x{:}A) & = & \Psi', x{:}A' & \text{where } \llbracket C/X \rrbracket_U(\Psi) = \Psi' \text{ and } \llbracket C/X \rrbracket_U(A) = A'
\end{array}$$

Fig. 4. Meta-substitution

$p[\sigma']$; otherwise we must replace p by $[\sigma']_\Psi(R)$. Note that the term R must be a variable, i.e. either an ordinary bound variable or another parameter variable $q[\pi_\Phi]$. The result of $[\sigma']_\Psi(R)$ may either be a neutral term R or a normal term $M : A$, since σ' may map a bound variable x to a normal term M. Because this last step may not always yield a normal term, we need to hereditarily substitute. Consequently, the operation $\llbracket C/X \rrbracket_U(R)$ may either return a neutral term R or a normal term $M : A$, and we re-normalize the term in the case for applications.

When applying $\llbracket C/X \rrbracket_U$ to a context variable ψ in a context and $\llbracket C/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G$, we simply replace ψ with the context Ψ. When applying the meta-substitution $\llbracket C/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G$ to the context $\Psi, x{:}A$, we apply it to the context Ψ to yield some new context Ψ' and return $\Psi', x{:}A$. This will produce a meaningful context since we can always appropriately rename bound variable occurrences prior to applying the substitution to avoid name conflicts and overshadowing.

When we apply $[\![C/X]\!]_U$ to an ordinary substitution σ, we apply it to its constituents. The main issue arises when we apply it to the identity substitution id_ψ. If $[\![C/X]\!]_U = [\![\Psi/\psi]\!]_G$, then we replace the sub-script ψ with a context Ψ and we subsequently expand Ψ to an identity substitution using the definition $\mathsf{id}(\Psi)$. This is defined as follows:

$$\begin{array}{lcl} \mathsf{id}(\cdot) & = & \cdot \\ \mathsf{id}(\Psi, x{:}A) & = & \mathsf{id}(\Psi)\,;\,x \\ \mathsf{id}(\psi) & = & \mathsf{id}_\psi \end{array}$$

Unfolding the context Ψ to an identity substitution yields a well-typed substitution σ.

Lemma 3.2. [Unfolding identity substitution]
If $\mathsf{id}(\Psi) = \sigma$ then $\Delta; \Psi, \Psi' \vdash \sigma \Leftarrow \Psi$.

Proof. Induction on the structure of Ψ. □

Applying the meta-substitution to an LF object will terminate for the same reasons as the ordinary substitution operation terminates; either we apply the substitution to a sub-expression or the objects we substitute are smaller. The following meta-substitution property holds for meta-terms and meta-types; it is a straightforward extension of the meta-substitution property for LF objects.

Lemma 3.3 (Termination).

1 If $[\![C/X]\!]_U(R) = M' : A'$ then $A' \leq U$.
2 $[\![C/X]\!]_U(\_)$ terminates, either by returning a result or failing after a finite number of steps.

Lemma 3.4 (Meta-substitution property).

1 If $\Delta \vdash C \Leftarrow U$ and $\Delta, X{:}U \vdash J$ then $\Delta' \vdash [\![C/X]\!]_U(J)$
2 If $\Delta' \vdash \theta \Leftarrow \Delta$ and $\Delta \vdash J$ then $\Delta' \vdash [\![\theta]\!]_\Delta(J)$

Proof. Induction on the structure of $\Delta \vdash J$. □

We finally state that type checking for LF objects and meta-objects is decidable.

Theorem 3.3 (Decidability of type checking for meta-level).
All typing judgments for meta-terms, meta-types, meta-substitutions, and meta-contexts are decidable.

Proof. The typing judgments are syntax-directed and therefore clearly decidable. □

4. Computation language

4.1. *Static semantics for computations*

We present here a uniform view of computations and their types where the computation language is parameterized over meta-terms C and meta-types U. The grammar for our dependently-typed language supports two different function types: one for computations and a dependent-function type which allows us to abstract over the meta-objects which can occur in meta-types. Corresponding to the ordinary function type we can create functions using $\mathsf{fn}\ x.E$ and apply them using $I\ \ E$. Corresponding to the dependent function type, we support abstractions over meta-objects using $\lambda^\Box X.\,E$ and application to a meta-object written as $I\ \lceil C \rceil$. This distinction is justified by its underlying logical interpretation and the Curry-Howard isomorphism; logical implication corresponds to the ordinary function type, while the dependent function type corresponds to universal quantification in first-order logic. In our case, we quantify over meta-terms. Meta-types not only classify meta-terms which we may quantify over, but at this point they also form the base types on the computation-level. There is a clear distinction between meta-types and computation-level types – in particular, computation-level expressions cannot appear in meta-objects. Following common logic terminology, we call assumptions, introduced when we traverse a function $\mathsf{fn}\ x.E$ of type $T_1 \to T_2$ hypothetical and assumptions, introduced when we traverse a dependent function $\lambda^\Box X.\,E$ of type $\Pi^\Box X{:}U.T$, parametric. Parametric assumptions denote the fact that our computation is parameterized over meta-terms X and they are kept in the meta-context Δ. Hypothetical assumptions are stored in the computation-level context Γ.

Our grammar for a branch is noteworthy: $\Pi^\Box\Delta.C : \theta \mapsto E$. Here, the meta-context Δ describes all the free variables occurring in the meta-term C. The meta-substitution θ is a refinement of the variables occurring in the type U of the scrutinee. The idea is as follows: If the scrutinee I has type U in an outer meta-context Δ', then θ refines the variables in Δ' to Δ. Moreover, our typing rules will guarantee that E only depends on the meta-context Δ – not the meta-context Δ'. This already foreshadows one key aspect of our typing rules for computations: instead of keeping track of a list of constraints and solve the constraints, when we check whether two types are equal, we will work with refinement substitutions. Refinement substitutions are essentially constraints in solved form; by concentrating on solved constraints, we eliminate the need to keep track and solve constraints during type checking.

Our typing rules generalize previous formulation in Pientka and Dunfield [2008], in two ways: First, we treat all meta-objects, i.e. context and meta-terms, uniformly. Second, we attach to patterns in the branches of case-expressions a refinement substitution θ and unlike previous formulations, also support context refinement.

Types	$T ::= U \mid T_1 \to T_2 \mid \Pi^\Box X{:}U.T$
Expressions (synth.)	$I ::= y \mid I\ E \mid I\ \lceil C \rceil \mid (E : T)$
Expressions (checked)	$E ::= I \mid C \mid \mathsf{fn}\ y.E \mid \lambda^\Box X.\,E \mid \mathsf{rec}\ f.E \mid \mathsf{case}\ I\ \mathsf{of}\ \vec{B}$
Branch	$B ::= \Pi^\Box\Delta.C : \theta \mapsto E$
Branches	$\vec{B} ::= \cdot \mid (B \mid \vec{B})$
Program contexts	$\Gamma ::= \cdot \mid \Gamma, y{:}T$

Next, we present bi-directional typing rules for programs which will minimize the amount of typing annotations in Figure 5. We distinguish here between typing of expressions and branches. In the typing judgment, we will distinguish between the meta-context Δ and the context Γ which includes declarations of computation-level variables.

$\boxed{\Delta;\Gamma \vdash I \Rightarrow T}$ Expression I synthesizes type T

$$\frac{y{:}T \in \Gamma}{\Delta;\Gamma \vdash y \Rightarrow T} \qquad \frac{\Delta;\Gamma \vdash I \Rightarrow T_2 \to T \qquad \Delta;\Gamma \vdash E \Leftarrow T_2}{\Delta;\Gamma \vdash I\ E \Rightarrow T}$$

$$\frac{\Delta;\Gamma \vdash I \Rightarrow \Pi^{\Box} X{:}U.T \qquad \Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash I\ \lceil C \rceil \Rightarrow [\![C/X]\!]_U T} \qquad \frac{\Delta;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash (E : T) \Rightarrow T}$$

$\boxed{\Delta;\Gamma \vdash E \Leftarrow T}$ Expression E checks against type T

$$\frac{\Delta;\Gamma \vdash I \Rightarrow T \qquad \Delta \vdash T = T'}{\Delta;\Gamma \vdash I \Leftarrow T'} \qquad \frac{\Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash C \Leftarrow U} \qquad \frac{\Delta;\Gamma, y{:}T_1 \vdash E \Leftarrow T_2}{\Delta;\Gamma \vdash \mathsf{fn}\ y.E \Leftarrow T_1 \to T_2}$$

$$\frac{\Delta, X{:}U;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash \lambda^{\Box} X.\,E \Leftarrow \Pi^{\Box} X{:}U.T} \qquad \frac{\Delta;\Gamma \vdash I \Rightarrow U \qquad \text{for all } k,\ \Delta;\Gamma \vdash B_k \Leftarrow U \to T}{\Delta;\Gamma \vdash \mathsf{case}\ I\ \mathsf{of}\ B_1 \mid \ldots \mid B_n \Leftarrow T}$$

$$\frac{\Delta;\Gamma,\ f : T\ \vdash\ E\ \Leftarrow\ T}{\Delta;\Gamma\ \vdash\ \mathsf{rec}\ f.E\ \Leftarrow\ T} \qquad \frac{\Delta \vdash C \Rightarrow U \qquad \text{for all } k,\ \Delta;\Gamma \vdash B_k \Leftarrow U \to T}{\Delta;\Gamma \vdash \mathsf{case}\ C\ \mathsf{of}\ B_1 \mid \ldots \mid B_n \Leftarrow T}$$

$\boxed{\Delta;\Gamma \vdash B \Leftarrow U \to T}$ Branch B with pattern of type U checks against T

$$\frac{\Delta_i \vdash C \Leftarrow [\![\theta_i]\!]_{\Delta}(U) \quad \Delta_i \vdash \theta_i \Leftarrow \Delta \quad \Delta_i; [\![\theta_i]\!]_{\Delta}(\Gamma) \vdash E \Leftarrow [\![\theta_i]\!]_{\Delta}(T)}{\Delta;\Gamma \vdash \Pi\Delta_i.C : \theta_i \mapsto E \Leftarrow U \to T}$$

Fig. 5. Typing for computations

We observe the usual bound variable renaming conditions in the rule for function abstraction, recursion, and dependent abstraction. There are a few interesting issues we briefly highlight: The typing rule for meta-terms checks the meta-term C against a meta-type U in the meta-context Δ and computation-level context Γ. Since meta-terms cannot depend on computations, we simply revert to type-checking C in the meta-context Δ thereby ensuring it is a pure LF object.

To type-check a case-expression against a type T, we first synthesize the type U of the scrutinee and then continue to check that each branch maps patterns of type U to expressions E of type T. A branch $B = \Pi^{\Box}\Delta_i.C_i : \theta_i \mapsto E_i$ is checked against $U \to T$, by verifying that the pattern C_i has type $[\![\theta_i]\!]_{\Delta}(U)$ in the meta-context Δ_i. To ensure we are working with a meaningful meta-substitution θ_i, we first check that it has domain Δ and co-domain Δ_i, i.e. it refines all meta-variables from the outer meta-context Δ. Next, we verify that the body of the branch, E_i has type $[\![\theta_i]\!]_{\Delta}(T)$ in the meta-context Δ_i and refined computation-level context $[\![\theta_i]\!]_{\Delta}(\Gamma)$. Our type system is elegant and compact and easily shown to be decidable. Unlike other foundations for type-checking dependently typed programs, we do not collect and subsequently solve constraints. Hence, the type system is easily trusted.

Decidability of type checking is easily established and only relies on the decidability of LF type and equality checking. Since LF objects are in canonical form, equality checking reduces to checking syntactic equality. Although constraint solving for LF objects is in general undecidable, since it relies on higher-order unification, type checking for computations remains decidable because we manage refinement substitutions in branches which track constraints in solved form.

Theorem 4.1 (Decidability of type checking for computations).
All typing judgments for computations are decidable.

Proof. The typing judgments are syntax-directed and because LF type and equality checking is decidable, clearly decidable. □

4.2. *Operational semantics for computations*

Next, we define the operational semantics for computations. Since we keep refinement substitutions in branches, an eager operational semantics where we propagate the instantiations for meta-variables eagerly is possible, but not elegant. The main difficulty is that refinement substitutions in branches would need to be refined. This can be done by pushing constraint solving into the meta-substitution definition for computation-level expressions, but we will follow a more elegant path here.

Instead of an eager operational semantics, we adopt an environment-based approach. Recall that we distinguish between meta-variables in Δ and program variables in Γ. This is also closer to an actual implementation. We hence define two environments: θ denotes the instantiation for the meta-context Δ; ρ provides instantiations for the program context Γ.

$$
\begin{array}{lrcl}
\text{Values} & V & ::= & C \mid (\mathsf{fn}\ y.E)[\theta;\rho] \mid (\lambda^{\Box} X.\, E)[\theta;\rho] \\
\text{Extended Values} & W & ::= & V \mid (\mathsf{rec}\ f.E)[\theta;\rho] \\
\text{Closures} & L & ::= & C \mid E\ [\theta\ ;\ \rho] \\
\text{Environments} & \rho & ::= & \cdot \mid \rho, W/y
\end{array}
$$

Closures are snapshots of computations inside an environment. The environment is represented by the two suspended substitutions θ and ρ for each of the two contexts Δ and Γ respectively.

We write $E[\theta;\rho]$ for a closure consisting of the expression E and the suspended meta-substitution θ and the program environment ρ. The intended meaning is that first meta-substitution θ is applied to E and then ordinary substitution ρ to the result. For clarification, we show the typing for well-typed environments and values. Unlike our previous typing rules which were algorithmic, we present them in a type assignment style simply describing the relationship between computation-level types and expressions.

Definition 4.2 (Closure typing). We define $L : T$ and $\rho : \Gamma$ simultaneously by the rules:

$$\frac{\cdot \vdash \theta \Leftarrow \Delta \qquad \rho : [\![\theta]\!]_\Delta(\Gamma) \qquad \Delta;\Gamma \vdash E \Leftarrow T \text{ or } \Delta;\Gamma \vdash E \Rightarrow T}{E\ [\theta\ ;\ \rho] : [\![\theta]\!]_\Delta(T)} \qquad \frac{\cdot \vdash C \Leftarrow U}{C : U}$$

$$\frac{}{\cdot : \cdot} \qquad \frac{\rho : \Gamma \qquad W : T}{(\rho, W/y) : \Gamma, y{:}T}$$

We present the big-step semantics in Figure 6. Values evaluate to themselves as defined in the first rule. For an expression E which is annotated with its type, we simply ignore the type annotation. For computation-level variables, we look up their extended value W in the environment ρ and continue to evaluate W. This is necessary, since not all computation-level variables are bound to values, but we also store closures consisting of a recursive function together with its corresponding meta-substitution and environment.

To evaluate a recursive function $\mathsf{rec}\ f.E$ under the meta-substitution θ and the environment ρ, we evaluate the body of the function and extend the environment with $(\mathsf{rec}\ f.E)\ [\theta\ ;\ \rho]/f$.

In the rule for applications, we evaluate $(I_1\ E_2)\ [\theta\ ;\ \rho]$ by first evaluating the closure $I_1\ [\theta\ ;\ \rho]$ to a function $(\mathsf{fn}\ y.E)\ [\theta_1\ ;\ \rho_1]$; next, we evaluate the second argument E_2 under the meta-substitution θ and the environment ρ to some value V_2; finally, we continue evaluating the body E where the meta-substitution θ_1 remains unchanged, but the environment ρ_1 is extended with the value V_2 for y. The rule for meta-application is dual. To evaluate $I\ \lceil C \rceil\ [\theta\ ;\ \rho]$ we evaluate first the expression I under the meta-substitution θ and the environment ρ to a value $(\lambda^\square X. E)\ [\theta_1\ ;\ \rho_1]$. We then continue to evaluate E where we extend the meta-substitution θ_1 with the closed meta-object $[\![\theta]\!]C$, but the environment ρ_1 remains unchanged.

Technically, we need to annotate the meta-substitution in $[\![\theta]\!]C$ with the domain of θ to prove that applying θ to the meta-object C terminates. For well-typed closures, we always know that there is a meta-context Δ s.t. $\cdot \vdash \theta \Leftarrow \Delta$, and hence meta-substitution will terminate.

Finally the rules for evaluating a case-expression. They rely on checking that two meta-substitutions (or two meta-objects) are compatible. We say that a meta-substitution θ is compatible with a meta-substitution θ_k if there exists an instantiation θ', s.t. $[\![\theta']\!]\theta_k = \theta$. Similarly, we say a meta-object C is compatible with a meta-object C_k, if there exists an instantiation θ' s.t. $[\![\theta']\!]C_k = C$.

There are three possibilities when we evaluate a case-expression: first, the current meta-substitution is incompatible with the meta-substitution in a given branch. Hence, this branch is not applicable; the scrutinee has some type $[\![\theta]\!]U$ while the branch has type $[\![\theta_i]\!]U$ and because θ_i and θ are not compatible, the type of the pattern is incompatible with the type of the scrutinee. We therefore simply continue to look for an applicable branch.

The second option is that the meta-substitution θ is compatible with the meta-substitution of a given branch. Hence, the type of the scrutinee and the type of the pattern are compatible. We therefore continue to check whether the scrutinee which evaluated to the

$\boxed{L \Downarrow V}$ Closure L evaluates to value V

$$\frac{}{V \Downarrow V} \qquad \frac{E\ [\theta\ ;\ \rho] \Downarrow V}{(E : T)\ [\theta\ ;\ \rho] \Downarrow V} \qquad \frac{\rho(y) \Downarrow V}{y\ [\theta\ ;\ \rho] \Downarrow V} \qquad \frac{E\ [\theta\ ;\ \rho, (\mathsf{rec}\ f.E)\ [\theta\ ;\ \rho]/f] \Downarrow V}{(\mathsf{rec}\ f.E)\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\frac{I_1\ [\theta\ ;\ \rho] \Downarrow (\mathsf{fn}\ y.E)\ [\theta_1\ ;\ \rho_1] \quad E_2\ [\theta\ ;\ \rho] \Downarrow V_2 \quad E\ [\theta_1\ ;\ \rho_1, V_2/y] \Downarrow V}{(I_1\ E_2)\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\frac{I\ [\theta\ ;\ \rho] \Downarrow (\lambda^{\Box} X.\, E)\ [\theta_1\ ;\ \rho_1] \quad E\ [\theta_1, [\![\theta]\!]C/X\ ;\ \rho_1] \Downarrow V}{I\ \lceil C \rceil\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\frac{\Delta_k \vdash \theta \not\doteq \theta_k \quad \mathsf{case}\ I\ \mathsf{of}\ \vec{B}\ [\theta\ ;\ \rho] \Downarrow V}{\mathsf{case}\ I\ \mathsf{of}\ (\Pi^{\Box}\Delta_k.C_k : \theta_k \mapsto E_k \mid \vec{B})\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\frac{\Delta_k \vdash \theta \doteq \theta_k/(\theta';\Delta'_k) \quad I\ [\theta\ ;\ \rho] \Downarrow C \quad \Delta'_k \vdash C \not\doteq [\![\theta']\!]C_k \qquad \mathsf{case}\ I\ \mathsf{of}\ \vec{B}\ [\theta\ ;\ \rho] \Downarrow V}{\mathsf{case}\ I\ \mathsf{of}\ (\Pi^{\Box}\Delta_k.C_k : \theta_k \mapsto E_k \mid \vec{B})\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\frac{\Delta_k \vdash \theta \doteq \theta_k/(\theta';\Delta'_k) \quad I\ [\theta\ ;\ \rho] \Downarrow C \quad \Delta'_k \vdash C \doteq [\![\theta']\!]C_k/(\theta'';\cdot) \quad E_k\ [[\![\theta'']\!]\theta'\ ;\ \rho] \Downarrow V}{\mathsf{case}\ I\ \mathsf{of}\ (\Pi^{\Box}\Delta_k.C_k : \theta_k \mapsto E_k \mid \vec{B})\ [\theta\ ;\ \rho] \Downarrow V}$$

Fig. 6. Big-step semantics

value C is compatible with the pattern C_k in a given branch. If this fails, we again look for an applicable branch.

We reach an applicable branch, if the meta-substitution θ is compatible with the meta-substitution θ_k of the given branch and in addition the pattern C_k is compatible with the meta-object C which is the result of evaluating the scrutinee $I\ [\theta\ ;\ \rho]$. In this case, we have a partial instantiation θ' for the meta-context Δ_k, i.e. θ' maps the variables from the meta-context Δ_k to some meta-context Δ'_k. Matching C against the pattern $[\![\theta']\!]_{\Delta_k} C_k$ will then return a ground instantiation θ'' for the meta-context Δ'_k. The meta-substitution $[\![\theta'']\!]\theta'$ hence denotes the combined meta-substitution under which the value of the scrutinee is equal to the pattern, i.e. $C = [\![\theta'']\!][\![\theta']\!]C_k$. We hence continue to evaluate the body E_k of the applicable branch under the meta-substitution $[\![\theta'']\!]\theta'$, but we keep the environment ρ for computation-level variables unchanged because patterns do not contain any computation-level variables in our language. We show next that types are preserved during evaluation.

Theorem 4.3 (Subject reduction). Let $L : T$. If $L \Downarrow V$ then $V : T$.

Proof. Structural induction on $L \Downarrow V$. □

5. Implementation of Beluga

Beluga is implemented in OCaml. It provides a complete reimplementation of the logical framework LF. Similarly to the Twelf core, Beluga supports type reconstruction for LF signatures based on higher-order pattern unification with constraints. Building on the presented foundation, we also designed a palatable source language for Beluga programs and implemented a type reconstruction algorithm for dependently-typed Beluga programs.

Type reconstruction for LF is in general undecidable [Dowek 1993]. Our algorithm reports a principal type, a type error, or that the source term needs more type information. As in Twelf, it is always possible to make typing unambiguous by adding more annotations. We tested our implementation of LF type reconstruction on many examples from the Twelf repository [Pfenning and Schürmann 1999].

Type reconstruction for Beluga programs is also undecidable. In our implementation, we check functions against a given type and either succeed, report a type error, or fail by asking for more type information.

An efficient implementation of higher-order unification is central to Beluga; it plays a crucial role in type reconstruction and in executing Beluga programs. We implemented a higher-order dynamic unification algorithm, solving higher-order patterns [Miller 1991] eagerly and delay working on some subterms which fall outside the decidable fragment until more information has been gathered. In our implementation, we support a limited form of Σ-types which we omitted from this presentation and we follow ideas described in [Abel and Pientka 2011] to apply type isomorphisms to translate objects of Σ-type back into the pattern fragment.

We also implemented a broad range of proofs as recursive Beluga functions, including proofs of the Church-Rosser theorem, proofs about compiler transformations, subject reduction, and translation from natural deduction to Hilbert style. To illustrate the expressive power of Beluga, our test suite includes simple theorems about structural relationships between expressions and proofs about the paths in expressions. These latter theorems have nested quantifiers and implications, placing them outside the fragment of propositions directly expressible in systems such as Twelf (see also Felty and Pientka [2010]). Our experience with coverage checking Beluga programs shows that problems and difficulties sometimes encountered in systems such as Twelf and Delphin are avoided (see Pientka and Dunfield [2010a]) and Beluga programs require fewer lemmas to work around existing limitations.

Finally, Beluga provides an interpreter, based on the described lazy environment-based semantics, to execute computation-level programs.

The Beluga system, including source code, examples, and an Emacs mode, is available from `http://complogic.cs.mcgill.ca/beluga/`.

6. Related Work

6.1. *Programming with HOAS*

One of the first proposals for functional programming with support for binders and higher-order abstract syntax was presented by Miller [1990]. Later, Despeyroux et al. [1997] developed a type-theoretic foundation for programming which supports primitive recursion. To separate data from computation, they introduce modal types $\Box A$ which can be injected into computation. However, data in their work must always be closed and hence pattern matching on objects using higher-order abstract syntax is not supported. Our work essentially continues the path set out in Despeyroux et al. [1997], and generalizes their work to allow for open data-objects and first-class contexts building on contextual modal type theory [Nanevski et al. 2008].

Closely related to our approach is the work by Schürmann et al. [2005]; Poswolsky and Schürmann [2008] where the authors present the ∇-calculus which provides a foundation for programming with higher-order abstract syntax which underlies the programming and reasoning environment Delphin [Poswolsky and Schürmann 2009]. The main difference between the theoretical foundation for Delphin and Beluga is in the treatment of contexts and meta-terms. In Beluga, we employ the contextual type $A[\Psi]$ to denote LF objects of type A which may refer to the variables declared in the context Ψ, and contexts are treated explicit in our foundation. Contexts are meta-data; contexts can be passed around explicitly and the programmer can analyzed and manipulated them using pattern matching. For example, our test-suite includes a program which translates terms in HOAS representation to de Bruijn representation by pattern matching on the context directly. Moreover, contextual objects allow us to specify and enforce fine-grained invariants and allows us distinguish between closed objects which have type $A[\cdot]$ and objects which depend on assumptions.

In Delphin, the context containing binding occurrences is global and left implicit. Instead of associating the context with the LF object, it is associated with the computation. A function is executed within a bound variable context and case-expressions can introduce assumptions and extend the context. When a new assumption is introduced into the context, the computation moves from the present world to a world where the context is extended. As a consequence, the programmer does not have direct access to the context and it is common practice to employ a variable-carrying continuation as an extra argument to keep track of and easily retrieve assumptions.

The primary application of Beluga and Delphin is in prototyping the meta-theory of formal systems, i.e. representing proofs as recursive functions. Beluga and Delphin separate the LF specifications and LF data from computations. One important consequence is that we cannot mix computation-level functions which support pattern matching and recursion with LF-abstractions which are used to model binders. More recently, Licata and Harper [2009] have proposed a system based on contextual type theory where we can mix computation functions and binding abstractions; this builds on their earlier ideas in Licata et al. [2008]. Their notion of a contextual type $A[\Psi]$ is similar, i.e. it describes an object of type A in a context Ψ. There is however one main differences: their framework does not support context variables as first-class objects; hence, a function runs within an outer context Ψ and we interpret a contextual object of type $A[\Psi']$ as an object of type A in a context Ψ, Ψ'. Note, that the contexts Ψ and Ψ' are combined. Licata and Harper [2009] have implemented a prototype for their framework in Agda thereby supporting programming with HOAS within Martin Löf type theory. Unlike Beluga or Delphin, structural properties such as weakening or substitution do however not hold in general, but they can be implemented generically. While [Licata and Harper 2009] demonstrate convincingly that their library within Agda elegantly supports programming with binders, it is less clear whether their prototype will scale to support meta-reasoning.

6.2. *Nominal programming approaches*

The nominal approach [Gabbay and Pitts 1999] to allow manipulation of binding structures which serve as a foundation of for example FreshML [Shinwell et al. 2003] is more pragmatic. In this approach, names and α-renaming are supported but implementing substitution is left to the user. The type system distinguishes at the type-level between expressions and names, and provides a special type `atom` which is inhabited by all names. Generation of a new name and binding names are separate operations which means it is possible to generate data which contains accidentally unbound names since fresh name generation is an observable side effect. To address this problem, Pottier [2007] describes pure FreshML where we can reason about the set of names occurring in an expression via a Hoare-style proof system. The system relies on assertions written by the programmer to reason about the scope of names. This static-analysis approach is quite expressive since the language of constraints includes subset relations, equality, intersection etc. In contrast, our work aims to provide a type-theoretic understanding of open data, binders, and substitutions. This has various benefits: For example, it is possible to provide precise error messages on where names escape their scope. Moreover, the programmer can directly analyze and manipulate contexts and it scales elegantly to dependent types.

More recently, Pouillard and Pottier [2010] provide a fresh look at programming with names and binders. Similar to Licata and Harper [2009] they provide a library within Agda to support programming with names and binders. Their proposal may also be viewed as an extension of a standard calculus such as system F^ω, with new primitive types and operations. To control the use of names, their framework introduces the abstract notion of *world*; the type system associates a world with each name, and allows two names to be compared for equality only if they inhabit a common world. Similar to [Licata and Harper 2009; Poswolsky and Schürmann 2008], we move to a new world whenever a new bound variable is introduced. Unlike the existing work however they give worlds names and keep track of how worlds relate to a previous world using links. At this point, it is unclear how the proposed framework scales to support dependent types and prototyping the meta-reasoning about systems.

7. Conclusion

We have presented a type-theoretic foundation for programming with higher-order abstract syntax and first-class contexts which is implemented in the Beluga framework. Our framework consists of two levels: The logical framework LF is used to specify formal systems; to describe and manipulate (proof) objects which depend on a context of assumptions, we use contextual type $A[\Psi]$. To abstract over a context of assumptions we support context variables. We give a decidable bi-directional type system and discuss various substitution properties.

On top of LF, we designed a dependently typed functional language that supports analyzing and manipulating contextual objects and contexts. Its design is kept generic; to keep track of type refinement in pattern matching, we attach to each pattern a refinement substitutions. This makes type checking decidable and easy to trust. We show type checking is decidable and our lazy operational semantics preserves types.

We have implemented the given theoretical foundation in the Beluga language [Pientka and Dunfield 2010b] and used it on a wide variety of examples. In the future, we plan to address two significant issues.

Totality.

Type-checking guarantees local consistency and partial correctness, but does not guarantee that functions are total. Thus, while we can implement, partially verify, and execute functions about derivations in deductive systems, Beluga does not currently guarantee the validity of a meta-proof. The two missing pieces are coverage and termination. We formulated a coverage algorithm [Dunfield and Pientka 2009] to ensure that all cases are covered, and plan to implement it over the next few months. Verifying termination will follow ideas in Twelf [Rohwedder and Pfenning 1996; Pientka 2005] for checking that arguments in recursive calls are indeed smaller.

Automation.

Currently, the recursive functions that implement induction proofs must be written by hand. We plan to explore how to enable the user to interactively develop functions in collaboration with theorem provers that can fill in parts of functions (that is, proofs) automatically.

References

Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. Technical report, McGill University, February 2011.

Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163. Springer, 1997. Extended version available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *International Conference on Typed Lambda Calculi and Applications(TLCA '93)*, pages 139–145, London, UK, 1993. Springer-Verlag. ISBN 3-540-56517-5.

Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.

Amy P. Felty and Brigitte Pientka. Reasoning with higher-order abstract syntax and contexts: A comparison. In Matt Kaufmann and Lawrence C. Paulson, editors, *International Conference on Interactive Theorem Proving*, LNCS. Springer, 2010.

Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224. IEEE Computer Society Press, 1999. URL `citeseer.ist.psu.edu/gabbay99new.html`.

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

Daniel R. Licata and Robert Harper. A universe of binding and computation. In Graham Hutton and Andrew P. Tolmach, editors, *14th ACM SIGPLAN International Conference on Functional Programming*, pages 123–134. ACM Press, 2009.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*, pages 241–252. IEEE Computer Society Press, 2008.

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

Dale Miller. An extension to ML to handle bound variables in data structures. In *Proceedings of the First Workshop on Logical Frameworks*, pages 323–335, 1990.

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

George C. Necula. Proof-carrying code. In *24th Annual Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, January 1997.

Frank Pfenning. Computation and deduction, 1997.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer, 1999.

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

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. Technical report, McGill University, August 2010.

Brigitte Pientka and Joshua Dunfield. Covering all bases: design and implementation of case analysis for contextual objects. Technical report, McGill University, 2010a.

Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI), 2010b.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, July 2008.

Adam Poswolsky and Carsten Schürmann. System description: Delphin—a functional programming language for deductive systems. In *International Workshop on Logical*

Frameworks and Meta-Languages: Theory and Practice (LFMTP'08), volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 135–141. Elsevier, 2009.

Adam B. Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, volume 4960, page 93. Springer, 2008.

François Pottier. Static name control for FreshML. In *22nd IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 356–365. IEEE Computer Society, July 2007.

Nicolas Pouillard and Franois Pottier. A fresh look at programming with names and binders. In *Proceedings of the fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 217–228, September 2010.

Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag Lecture Notes in Computer Science (LNCS) 1058.

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, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In Pawel Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In *8th International Conference on Functional Programming (ICFP'03)*, pages 263–274, New York, NY, USA, 2003. ACM Press. .

Antonis Stampoulis and Zhong Shao. Veriml: typed computation of logical terms inside a language with effects. In Paul Hudak and Stephanie Weirich, editors, *15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010), Baltimore, USA*, pages 333–344. ACM, 2010.

Twelf Wiki. Twelf wiki, 2009. `http://twelf.plparty.org/wiki/Main_Page`.

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