First-class Substitutions in Contextual Type Theory

Andrew Cave   Brigitte Pientka

School of Computer Science,
McGill University, Montreal, Canada
{acave1,bpientka}@cs.mcgill.ca

Abstract

In this paper, we revisit the theory of first-class substitution in contextual type theory (CTT); in particular, we focus on the abstract notion of substitution variables. This forms the basis for extending Beluga, a dependently typed proof and programming language which already supports first-class contexts and contextual objects, with first-class substitutions. To illustrate the elegance and power of first-class substitution variables, we describe the implementation of a weak normalization proof for the simply-typed lambda-calculus in Beluga.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Logical frameworks; higher-order abstract syntax; dependent types

1. Introduction

Substitutions are a common key concept in describing and reasoning about logics and formal systems in general. In this setting, a substitution describes the mapping between variables of some type A to objects of the same type. In programming languages, substitutions are used to keep track of the binding of variables to values and are often referred to as environments. This naturally gives rise to an environment-based operational semantics where we pair an open expression M together with the environment which provides the appropriate value bindings for the free variables in M. Environments also play a key role when describing program transformations such as closure conversion. Because the concept of environments is pervasive when talking about programming languages, some programming languages even provide them as first-class objects giving programmers direct access to inspect and manipulate the environment. Gelernter et al. [1987] were one of the first to propose first-class environments in the setting of LISP where environments can be passed as arguments, analyzed, and returned as results. Subsequently, first-class environments have been for example incorporated into Scheme and proposed as a means for code sharing (see for example [Queinnec and Roure 1996]).

From a more theoretical perspective, first-class environments have been studied by Nishizaki [2000], Sato et al. [2001], and subsequently by Sato et al. [2002] in the simply typed setting.

In this paper, we take a different view and revisit first-class substitutions in contextual type theory [Nanevski et al. 2008] which were first proposed by Pientka [2008]. Fundamentally, first-class substitutions relate two contexts Γ and Ψ by mapping variables of Γ to object of the same type in the context Ψ. We therefore introduce the type $\Gamma[\Psi]$ which classifies substitutions with domain Γ and range Ψ. One of the simplest forms of first-class substitutions arises when we want to relate abstractly two contexts via weakening or exchange. For example, we might have a lemma which establishes a property P in a context Γ which contains natural numbers. We now want to use the result of the lemma in a context Ψ which contains natural numbers and equality assumptions. To use P in the context Ψ, we must weaken it.

When reasoning about formal systems, substitutions also naturally arise for example in logical relations proofs such as normalization proofs. In the fundamental lemma of a normalization proof, we typically prove that given a well-typed term M in a context Γ and an environment ρ which provides reducible terms for all variables in Γ, that $[\rho]M$ is reducible.

As mentioned, first-class substitutions were introduced by Pientka [2008]; however there were several issues left open and unsatisfactory: what role can substitution variables play in the practice of mechanizing the meta-theory of formal systems? How do substitution variables compose with identity substitutions? How do substitution variables compose with variables? What is the canonical form of a substitution? How do we instantiate and replace substitution variables?

In this work, we develop a full theory of substitution variables within contextual type theory CTT$^{\mathsf{s}}$. Our theory takes a fresh look at the definition of substitutions and enforces a canonical structure for substitutions, i.e. there is exactly one way to write a substitution. We prove various substitution properties and give a decidable, bi-directional type system for our theory. We also have mechanized these properties restricted to the simply-typed setting in Agda which increases our confidence in the correctness.

CTT extended with substitution variables can also be plugged into Beluga [Pientka and Dunfield 2010], a dependently-typed programming and proof environment which already supports abstracting over and pattern matching on contexts and contextual objects thereby allowing programmers to abstract over substitutions, to pass them as arguments and return them as results. We illustrate the elegance of first-class substitutions by presenting the implementation of a weak normalization proof for the simply-typed lambda-calculus. It directly mirrors the theoretical development on paper which we find a remarkably elegant solution. More generally, any logical relations proof (parametricity, completeness proofs, adequacy and full abstraction) benefits from our direct, first-class treatment of substitutions.

2. Example: Weak Normalization

In this section we present an example illustrating the application of substitution variables in a weak normalization proof.

2.1 Hand written proof

We begin by illustrating a typical logical relations proof of weak normalization for a simply typed lambda calculus to recap the encoding of formal systems and proofs in Beluga, and to emphasize the applications of a first-class notion of substitution.

The grammar is straightforward: it includes lambda abstractions, variables, application, and a single constant of a constant type **i**. Lambda abstractions and the constant **c** are considered values.

$$\begin{array}{lrcl}
\text{Types} & A, B & ::= & \mathbf{i} \mid A \to B \\
\text{Terms} & M, N & ::= & \mathbf{c} \mid x \mid \lambda x.M \mid M\ N \\
\text{Values} & V & ::= & \mathbf{c} \mid \lambda x.M
\end{array}$$

The typing rules are standard and use the judgment $\Gamma \vdash M : A$ where Γ describes a typing context.

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A} \quad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

$$\frac{}{\Gamma \vdash \mathbf{c} : \mathbf{i}} \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

We then define a form of weak head reduction with transitivity and reflexivity built in, simply because it is compact and convenient for this proof.

$$\frac{}{M \longrightarrow M}\ \mathsf{refl} \quad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \mathsf{trans}$$

$$\frac{}{(\lambda x.M)N \longrightarrow [N/x]M}\ \mathsf{beta} \quad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}\ \mathsf{app}$$

We say that a term M halts if there exists a value V such that $M \longrightarrow V$.

We can now define the notion of reducibility which is typical for logical relations proofs. A term is reducible at base type precisely when it halts. A term M is reducible at arrow type $A \to B$ when it halts, and for every reducible N at type A, the application of M to N is reducible at type B.

$$\begin{array}{l}
\mathcal{R}_{\mathbf{i}} = \{M \mid M \text{ halts}\} \\
\mathcal{R}_{A \to B} = \{M \mid M \text{ halts and } \forall N \in \mathcal{R}_A, (M\ N) \in \mathcal{R}_B\}
\end{array}$$

It is trivial from the definition that if a term M is reducible at some type A, then it halts. One can easily prove that $\mathcal{R}_A$ is closed under expansion by induction on the type:

Lemma 2.1 (Closure under expansion)**.** *If $M' \in \mathcal{R}_A$ and $M \longrightarrow M'$ then $M \in \mathcal{R}_A$*

Our aim is to show that all well-typed closed terms are reducible, but as usual we must generalize to showing that all closed instantiations of a well-typed open term are reducible. For this we need the notion of a simultaneous substitution, and to define reducibility of substitutions. If $\sigma = M_1/x_1, M_2/x_2, ..., M_n/x_n$, we say σ is reducible at context $\Gamma = x_1{:}A_1, ..., x_n{:}A_n$ if each M_i is reducible at A_i, i.e. $M_i \in \mathcal{R}_{A_i}$. We write this as $\sigma \in \mathcal{R}_\Gamma$. We can now state the main lemma:

Lemma 2.2 (Main lemma)**.** *If $\Gamma \vdash M : A$ and $\sigma \in \mathcal{R}_\Gamma$ then $[\sigma]M \in \mathcal{R}_A$.*

Proof. By induction on the typing derivation. We show only the interesting case:

Case $\dfrac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$:

First, $[\sigma](\lambda x.M) = \lambda x.([\sigma, x/x]M)$ halts, since it is a value. Suppose then that we are given $N \in \mathcal{R}_A$.

1. $[\sigma, N/x]M \in \mathcal{R}_B$ (by I.H.)
2. $[N/x][\sigma, x/x]M \in \mathcal{R}_B$ (properties of substitution)
3. $(\lambda x.([\sigma, x/x]M))\ N \in \mathcal{R}_B$ (by closure under expansion)

Hence $[\sigma](\lambda x.M) \in \mathcal{R}_{A \to B}$ (by definition) □

Corollary 2.3 (Weak normalization)**.** *If $\vdash M : A$ then M halts.*

Proof. By the main lemma, we have that $[\cdot]M \in \mathcal{R}_A$ and hence that $[\cdot]M (= M)$ halts. □

We wish to draw attention to the use of properties of simultaneous substitutions in the main lemma and its corollary. These properties are typically the source of the most overhead when formalizing results using various low level representations of variables and variable binding. However, we will illustrate that these properties are immediate in our framework, as the equational theory of (simultaneous) substitutions is built into our type theory.

2.2 Encoding in Beluga

Here we demonstrate the particularly elegant encoding of this proof in Beluga using first class (simultaneous) substitutions. By hand we defined grammar and typing separately, but here it is actually more convenient to define intrinsically typed terms directly. Below, `tm` defines our family of simply-typed lambda terms indexed by their type as an LF signature. In typical higher-order abstract syntax fashion, lambda abstraction takes a *function* representing the abstraction of a term over a variable. There is no case for variables, as they are treated implicitly. We remind the reader that this is a weak, representational function space – there is no case analysis or recursion, and hence only genuine lambda terms can be represented.

```
tp : type.
i : tp.
arr : tp → tp → tp.

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

We can then encode our step relation in a similar fashion below. Notice in particular we use LF's notion of substitution to encode the object-level substitution in the `s/beta` case.

```
mstep : tm A → tm A → type.
s/beta : mstep (app (lam M) N) (M N).
s/app : mstep M M’ → mstep (app M N) (app M’ N).
s/refl : mstep M M.
s/trans: mstep M M’ → mstep M’ N → mstep M N.
```

We define a predicate on terms expressing what it means to be a value: again, the constant `c` and lambda abstractions are our values.

```
val : tm A → type.
val/c : val c.
val/lam : val (lam M).
```

A term halts if it reduces to a value. In the following, `M’` is implicitly universally quantified by the constructor `h/val`, and so this has the effect of encoding the existential statement "There exists a value M' that M steps to". As in Twelf [Pfenning and Schürmann 1999], type reconstruction infers types for any free variable in a given type or kind declaration and implicitly quantifies over them. Programmers subsequently do not need to supply arguments for implicitly quantified variables.

```
halts : tm A → type.
h/val : mstep M M’ → val M’ → halts M.
```

Reducibility cannot be directly encoded at the LF layer, since it involves a strong, computational function space. Hence we move to the computation layer of Beluga, and employ an indexed recursive type, as defined by Cave and Pientka [2012]. Contextual LF objects and contexts which are embedded into computation-level types and programs are written inside `[ ]`.

Once again, a term of type `i` is reducible if it halts, and a term `M` of type `arr A B` is reducible if it halts, and moreover for every reducible `N` of type `A`, the application `app M N` is reducible. We write `{N:[.tm A]}` for explicit Π-quantification over `N`, a closed term of type `A`. To the left of the dot in `[.tm A]` is where one writes the context the term is defined in – in this case, it is empty.

```
datatype Reduce : {A:[.tp]} {M:[.tm A]} ctype =
| I : [. halts M] → Reduce [. i] [. M]
| Arr : [. halts M] →
       ({N:[.tm A]} Reduce [. A] [. N]
           → Reduce [. B] [. app M N])
     → Reduce [. arr A B] [. M];
```

In this definition, the arrows represent the usual computational function space, not the weak function space of LF. We note that this definition is not (strictly) positive, since `Reduce` appears to the left of an arrow in the `Arr` case. Allowing unrestricted such definitions destroys the soundness of our system. Here we note that this definition is stratified by the type: the recursive occurences of `Reduce` are at types `A` and `B` which are smaller than `arr A B`. This is how one justifies its existence. Beluga does not currently perform either a positivity check or a stratification check, so in the present we must convince ourselves that this predicate is well-defined. In Coq [Bertot and Castéran 2004] and Agda [Norell 2007] one typically justifies such definitions using large elimination, which Beluga does not currently support.

The next step we elided in the handwritten proof. It is a trivial lemma showing that halts is closed under expansion. Proofs in Beluga are written as recursive functions, analyzing the construction of the underlying LF objects, making recursive calls for appeals to the induction hypothesis. Here, we proceed by unboxing and pattern matching on our arguments to extract the witness of the existential and rebuilding them, appealing to transitivity of `mstep`.

```
rec haltsClosed : [. mstep M M’] → [. halts M’]
 → [. halts M] =
fn ms ⇒ fn h ⇒
 let [. h/val MS’ V] = h in
 let [. MS] = ms in
 [. h/val (trans MS MS’) V];
```

The trivial fact that reducible terms halt has a corresponding trivial proof, analyzing the construction of the the proof of `Reduce [.A] [.M]`:

```
rec cr1 : Reduce [.A] [.M] → [. halts M] =
fn r ⇒ case r of
| I h ⇒ h
| Arr h f ⇒ h;
```

Next we prove closure of `Reduce` under expansion. This follows the handwritten proof, proceeding by induction on the type `A` which is an implicit argument. In the base case we appeal to `haltsClosed`, while in the `Arr` case we must also appeal to the induction hypothesis at the range type, going inside the function position of applications.

```
rec closed : [. mstep M M’] → Reduce [.A] [.M’]
    → Reduce [.A] [.M] =
fn ms ⇒ fn r ⇒ case r of
| I h ⇒ I (haltsClosed ms h)
| Arr h f ⇒ let [.MS] = ms in
  Arr (haltsClosed ms h)
  (λ N ⇒ fn rn ⇒
   closed [. s/app MS] (f [. N] rn));
```

Note that we overload λ; here we use λ-abstractions when abstracting over an index argument explicitely, while we use **fn**-abstraction when abstracting over a computation-level variable. Now we arrive at the part of the proof requiring simultaneous substitutions. First we define a schema for contexts, describing a type of contexts which contains only (typed) term assumptions:

```
schema ctx = tm T;
```

Schema declarations are similar to world declarations in Twelf [Pfenning and Schürmann 1999]. In the example above, `ctx` classifies contexts which contain instances of `tm T`; for example, `x:tm i, y: tm (arr i i), w: tm i` is a well-formed context of schema `ctx`.

We now must state precisely what it means for a substitution to be reducible. We do this by employing another indexed recursive type: a predicate expressing that the substitution was built up as a list of reducible terms. The notation `#S` stands for a substitution variable. Its type is written `g[]`, meaning that it has domain `g` and empty range, i.e. it takes variables in `g` to closed terms of the same type. In the base case, the empty substitution is reducible. In the `Cons` case, we read this as saying: if `#S` is a reducible substitution (implicitly at type `g[]`) and `M` is a reducible term at type `A`, then `#S` with `M` appended is a reducible substitution (implicitly at type `(g,x:tm A)[]` – the domain has been extended with a variable of type `A`).

```
datatype RedSub : (g:ctx){#S:g[]} ctype =
| Nil : RedSub [. · ]
| Cons : RedSub [. #S] → Reduce [.A] [.M] →
    RedSub [. #S M ];
```

We implicitly quantify over the context `g` by using round parenthesis writing `(g:ctx)`; we explictly quantify over substitution variables using curly braces writing `{#S:g[]}`.

The main fact that we need about this definition is that if a substitution `#S` is reducible, then looking up a variable `#p` in it produces something reducible. The hash in front of `#p` distinguishes it as a *parameter* variable, i.e. standing for an object-level variable. To clarify, `#p #S` is notation for applying the substitution `#S` to the variable `#p` – informally, $\sigma(x)$. The proof proceeds by analyzing the variable. If it is the top variable of the context, the proof that `#p #S` is reducible is the one at the top. Otherwise, we continue inductively looking in the prefix of the context. The case of an empty context is impossible, as there are no variables in the empty context.

```
rec lookup : {g:ctx}{#p:[g.tm A]} RedSub [. #S] →
    Reduce [.A] [. #p #S] =
λg ⇒ λ#p ⇒ fn rs ⇒
case [g. #p ..] of
 | [g’,x:tm A. x] ⇒ let Cons rs’ rN = rs in rN
 | [g’,x:tm A. #q ..] ⇒ let Cons rs’ rN = rs in
     lookup [g’] [g’. #q ..] rs’;
```

In the above, we write `..` for the identity substitution associated with a context variable. In particular, when we write `[g’,x:tm A. #q ..]`, this means that `#q` is a parameter variable defined in `g’`, and hence is not `x`. If we wished to allow it to be `x`, we would instead write `#q .. x`, where now `.. x` is the expanded identity substitution for the context `g’,x:tm A`. One may choose to think of the `..` in this instance as a shift. Operationally, when we have a concrete variable for `#p`, this pattern performs a check to see if the variable is in the image of the `..` pattern substitution, i.e. if it is not the top variable `x`.

Finally, our main lemma is standard and takes the form we would expect from the handwritten proof: if M is a well-typed term, and we provide a reducible substitution #S with closed instantiations for each of the free variables of M, then M #S (that is, the application of #S to M) is reducible. We proceed by induction on the term. When it is a variable, we appeal to lookup. When it is an application, we straightforwardly apply the functional argument we obtain from the induction hypothesis for M1 to the induction hypothesis for M2. The lam case is the most interesting. Clearly the lambda abstraction halts, as it is a value. To show that applying it to any reducible term is reducible, we appeal directly to closure under expansion, and the induction hypothesis for M1 with the substitution extended with N. Recall that rN is a proof that N is reducible. We write _ for the type of the variable x and let reconstruction infer it.

```
rec main : {g:ctx}{M:[g.tm A]} RedSub [. #S]
          → Reduce [.A] [. M #S] =
λg ⇒ λM ⇒ fn rs ⇒ case [g. M ..] of
| [g. #p ..] ⇒ lookup [g] [g. #p ..] rs

| [g. lam (λx. M1 .. x)] ⇒
  Arr [. h/val s/refl val/lam]
   (λ N ⇒ fn rN ⇒ closed [. s/beta]
    (main [g,x:tm _] [g,x. M1 .. x] (Cons rs rN)))

| [g. app (M1 ..) (M2 ..)] ⇒
  let Arr h f = main [g] [g. M1 ..] rs in
  f [._] (main [g] [g. M2 ..] rs)

| [g. c] ⇒ I [. h/val s/refl val/c];
```

Notably, we did not have to concern ourselves with the property of substitutions that we wrote explicitly in the paper proof in the lam case. Using a low-level representation of substitution (e.g. de Bruijn indices) one must prove $[\sigma, N/x]M = [N/x][\sigma, x/x]M$ (where $x \notin \mathsf{FV}(\sigma)$) by hand, and this is actually a large bulk of the proof. We will see that in our setting, it is handled automatically by normalization during typechecking: the normal form of both sides of this equation is $[\sigma, N/x]M$.

Weak normalization is now a trivial corollary, taking M to be a closed term and #S to be the empty substitution:

```
rec weakNorm : {M:[. tm A]} [. halts M] =
λM ⇒ cr1 (main [] [. M] Nil);
```

Once again if one is being pedantic (as one must in a formalization), this requires the property that $[\cdot]M = M$, which is arranged to hold automatically in our system.

We wish to point out that the whole development is ≈60 lines, and it follows the handwritten proof very closely, with essentially no extra overhead. Compared to low-level techniques for variable binding, it is a huge win to not be burdened with proving properties of simultaneous substitution. Even in other systems employing higher-order abstract syntax, such as Abella [Gacek 2008], simultaneous substitution is not a first-class notion, and must be defined explicitly. This means that one must still prove properties of simultaneous substitution, although it is somewhat easier in Abella than with a low-level representation because simultaneous substitution can be built out of the provided individual substitution. However, one must still explicitly prove that the defined simultaneous substitution is a congruence in Abella, i.e. $([\sigma]M)\ ([\sigma]N) = [\sigma](M\ N)$ and similarly for the more complex λ-abstraction case. In our system, this burden is lifted completely.

For systems such as Twelf [Pfenning and Schürmann 1999] and Delphin [Poswolsky and Schürmann 2008] a direct formulation of normalization proofs is out of reach, since they lack first-class contexts and recursive types. Instead Schürmann and Sarnat [2008] proposed to represent and reason about an auxiliary logic to overcome the limited meta-logical strength of systems such as Twelf.

Preliminary results suggest that this proof can be elegantly extended to a $\beta\eta$-normalization proof which normalizes under binders. This proof is made substantially more difficult by the fact that one must invent an arbitrary number of fresh variables when η-expanding. A common technique is to employ a *Kripke* logical relation in which one quantifies over a variable-for-variable substitution representing an arbitrary weakening. This proof relies much more heavily on the equational theory of substitutions, so our approach is particularly promising here. More generally, our approach should prove useful for many (Kripke) logical relations proofs, such as parametricity, full abstraction, or various kinds of completeness proofs. This is especially so for larger languages (e.g. with case expressions) where one must use such properties of substitution repeatedly.

3. Contextual LF with Substitution Variables

Here we revisit contextual LF [Nanevski et al. 2008; Pientka 2008] which extends the logical framework LF [Harper et al. 1993] with contextual objects and types. A contextual type $A[\Psi]$ characterizes a LF object M of type A in a context Ψ; it is inhabited by the contextual object $[\hat{\Psi}.M]$ where $\hat{\Psi}$ lists the free variables occurring in M and can be obtained from the context Ψ by erasing the type annotations and keeping only the declared variable names. In Pientka [2008], contextual LF was extended to allow context variables which inhabit a context schema and parameter variables which inhabit the parameter type $\#A[\Psi]$. In this work, we revisit the addition of first-class substitutions and substitution variables.

3.1 Contextual LF

We summarize here contextual LF concentrating on the new parts which deal with substitution variables. Furthermore, we concentrate here on characterizing well-typed terms, but defining kinds and kinding rules for types is straightforward and omitted. We characterize only objects in $\beta\eta$ normal form, since these are the only meaningful objects in LF.

$$\begin{array}{lrcl}
\text{Atomic types} & P, Q & ::= & \mathbf{a} \cdot S \\
\text{Types} & A, B & ::= & P \mid \Pi x{:}A.B \\
\text{Heads} & H & ::= & x \mid \mathbf{c} \mid p[\sigma] \mid (\hat{\Psi}.x)[\rho] \\
\text{Spines} & S & ::= & \epsilon \mid N; S \\
\text{Neutral Terms} & R & ::= & H \cdot S \mid u[\sigma] \\
\text{Normal Terms} & M, N & ::= & R \mid \lambda x.M \\
\text{Substitutions} & \sigma, \tau & ::= & \cdot \mid \mathsf{id}_\psi \mid \sigma, M \mid \mathsf{id}_\psi[\rho] \\
\text{Sub. Closures} & \rho & ::= & s[\sigma] \\
\text{Contexts} & \Psi & ::= & \cdot \mid \psi \mid \Psi, x{:}A
\end{array}$$

We present the language in spine form [Cervesato and Pfenning 2003], as it makes the termination of hereditary substitution easier to establish. Instead of $x\ M_1 \ldots M_n$, we write $x \cdot M_1; \ldots; M_n; \epsilon$.

In addition to λ-abstraction, application, variables and constants, contextual LF also contains meta-variables u which represent a general open LF object and parameter variables p which represents an LF variable. They are associated with a postponed substitution σ representing a closure. The intention is to apply σ as soon as we know what u (or p resp.) stand for. Intuitively, we can justify the closure as follows: let u describe a hole in a term M; if we were to apply a substitution σ to M, then σ gets stuck when it encounters the hole u. However, as soon as we know what u stands for, we can further apply the substitution σ to it.

Meta-variables are considered neutral terms and our typing rules will enforce that meta-variables are of atomic type. This is not a restriction, since we can always lower a meta-variable; intuitively, we transform $u[\sigma]\ M$ (or in our syntax $u[\sigma] \cdot M; \epsilon$) where u has type

$(\Pi x{:}A.B)[\Psi]$ to $u'[\sigma, M]$ where u' has type $B[\Psi, x{:}A]$ by replacing u with $\hat{\Psi}.\lambda x.u'[\sigma, x]$ possibly η-expanding x, if necessary.

For this paper, we draw the reader's attention to substitution variables s. They appear in two forms in the grammar. If the domain of a substitution variable s is Ψ and x is a variable in Ψ then looking up the variable x in s (i.e. $s(x)$) is stuck until one has a concrete instantiation for s. Moreover, applying a substitution σ to $s(x)$ is also stuck until one has a concrete instantiation for s. This is the meaning of the term $(\hat{\Psi}.x)[s[\sigma]]$. One may also view the domain of a substitution as a form of product type, in which case the term $(\hat{\Psi}.x)[s[\sigma]]$ may be read as the x projection of $s[\sigma]$. The $\hat{\Psi}$ binds the x, allowing for α-renaming.

The second form, $\mathsf{id}_\psi[s[\sigma]]$ is morally the stuck composition $[\sigma]([s](\mathsf{id}_\psi))$, where id_ψ may be thought of as a weakening or shifting substitution. If the domain of s starts with a context variable (e.g. ψ,Ψ) then this corresponds to removing the Ψ components, leaving only the ψ components of the substitution.

The grammar and typing rules ensure that substitutions are η-long. By this we mean that if a substitution closure ρ ($= s[\sigma]$) has domain $\psi, x{:}A$, then one may write the substitution $(\mathsf{id}_\psi[\rho], (\psi,x.\, x)[\rho])$ which morally represents the same thing as ρ, but the term ρ by itself is not even grammatically a substitution. This has the practical impact that when writing Beluga programs, one never needs to do work to prove an equation such as $\rho = \mathsf{id}_\psi[\rho], (\psi,x.\, x)[\rho]$, since it is arranged to hold in the theory. Similarly, if ρ has domain $\cdot$, the η-long form of ρ is simply $\cdot$, the empty substitution. Note that our treatment of substitution variables differs here from the treatment in Pientka [2008] where we allow $s[\sigma]$ to stand by itself.

Next we define hereditary substitution, which performs a substitution and reduces any redexes which arise along the way. Hereditary substitution allows us to only consider normal forms in our grammar and typing rules, which makes the decidability of type-checking obvious. As usual, we annotate hereditary substitutions with an approximation of the type of the term we substitute for to guarantee termination.

$$\text{Type approximations} \quad \alpha, \beta \quad ::= \quad \mathbf{a} \mid \alpha \to \beta$$

We then define the dependency erasure operator $(-)^-$ as follows:

$$\begin{array}{lcl} (\mathbf{a} \cdot S)^- & = & \mathbf{a} \\ (\Pi x{:}A.B)^- & = & (A)^- \to (B)^- \end{array}$$

We will sometimes tacitly apply the dependency erasure operator $(-)^-$ in the following definitions. We can now define hereditary substitution, presented in Figure 1. The definition is mostly standard. As usual, when the substitution would create redexes, we proceed by hereditarily performing substitution. The addition of substitution variables does not pose any difficulties here. Termination is readily established:

Lemma 3.1. *The hereditary substitutions* $[M/x]^*_\alpha(N)$ *where* $* \in \{n, s, l\}$ *and* $\mathsf{reduce}(M : \alpha, S)$ *terminate, either by failing or successfully producing a result.*

Proof. By nested induction on α primarily and N secondarily. □

Normal terms

$$\begin{array}{lll}
[M/x]^n_\alpha(\lambda y.N) & = \lambda y.N' & \text{where } N' = [M/x]^n_\alpha(N) \\
 & & \text{choosing } y \notin \mathsf{FV}(M) \\
 & & \text{and } y \neq x \\
[M/x]^n_\alpha(u[\sigma]) & = u[\sigma'] & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
[M/x]^n_\alpha(\mathbf{c} \cdot S) & = \mathbf{c} \cdot S' & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(x \cdot S) & = N & \text{where } S' = [M/x]^l_\alpha(S) \\
 & & \text{and } N{=}\mathsf{reduce}(M{:}\alpha{,}S') \\
[M/x]^n_\alpha(y \cdot S) & = y \cdot S' & \text{where } y \neq x \\
 & & \text{and } S' = [M/x]^l_\alpha(S) \\
[M/x]^n_\alpha(p[\sigma] \cdot S) & = p[\sigma'] \cdot S' & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
 & & \text{and } S' = [M/x]^l_\alpha(S) \\
[M/x]^n_\alpha((\hat{\Psi}.y)[\rho]{\cdot}S) & = (\hat{\Psi}.y)[\rho']{\cdot}S' & \text{where } \rho' = [M/x]_\alpha(\rho) \\
 & & \text{and } S' = [M/x]^l_\alpha(S)
\end{array}$$

Spines

$$\begin{array}{lll}
[M/x]^l_\alpha(\epsilon) & = \epsilon & \\
[M/x]^l_\alpha(N; S) & = N'; S' & \text{where } N'{=}[M/x]^n_\alpha(N) \\
 & & \text{and } S' = [M/x]^l_\alpha(S)
\end{array}$$

Substitutions

$$\begin{array}{lll}
[M/x]^s_\alpha(\cdot) & = \cdot & \\
[M/x]^s_\alpha(\mathsf{id}_\psi) & = \mathsf{id}_\psi & \\
[M/x]^s_\alpha(\sigma, N) & = \sigma', N' & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
 & & \text{and } N' = [M/x]^n_\alpha(N) \\
[M/x]^s_\alpha(\mathsf{id}_\psi[\rho]) & = \mathsf{id}_\psi[\rho'] & \text{where } \rho' = [M/x]_\alpha(\rho)
\end{array}$$

Substitution Closures

$$\begin{array}{lll}
[M/x]_\alpha(s[\sigma]) & = s[\sigma'] & \text{where } \sigma' = [M/x]_\alpha(\sigma)
\end{array}$$

$$\begin{array}{ll}
\mathsf{reduce}(\lambda x.M : \alpha \to \beta, (N; S)) & = \mathsf{reduce}([N/x]^n_\alpha(M) : \beta, S) \\
\mathsf{reduce}(R : \mathbf{a}, \epsilon) & = R \\
\mathsf{reduce}(M, S) & \quad \text{fails otherwise}
\end{array}$$

Figure 1. Hereditary substitution

Now we can generalize hereditary substitution to simultaneous substitutions. It is this generalization (and later, meta-substitution) which needs a careful treatment in the presence of substitution variables. Note that previous formulations of contextual LF allow substitutions to be extended with variables in addition to arbitrary terms. This is motivated by the fact that as we push σ under a binder in $\lambda x.M$, we would like to extend it by the identity mapping x to itself. However, since we do not know the type of x, simply extending σ with x may not be meaningful, since x may not be in η-long form. Therefore, previous formulations allow a substitution σ to be extended with M where M is either an η-long normal form or a variable. Unfortunately, this complicated the grammar of substitutions and meant that there was not necessarily a unique representation of substitutions. This meant that substitutions still had to be compared up to η equality. Our formulation avoids these issues and requires that substitutions must be extended with η-long terms, thus guaranteeing unique normal forms for substitutions.

In Figure 2 we define an operation $[\sigma]^{\hat{\Phi}}_{\Psi}$. This operation is a true simultaneous substitution when $\hat{\Phi} = \cdot$. More generally it means substitute for the variables in Ψ, leaving the variables in $\hat{\Phi}$ untouched. It is a simultaneous substitution *under a context*. This means that when we go under the binder in $\lambda x.M$ we need only to extend $\hat{\Phi}$ with x. This is the key to avoiding the issues mentioned above. The typing lemma below should shed some light on the behaviour of this operation.

Since the domain of our substitutions is not explicit in the syntax, in the application $[\sigma]_\Psi(M)$ we consider the ith component of σ to correspond to the ith component of Ψ, thus order is relevant. We will write $[\sigma]_\Psi(-)$ to mean $[\sigma]^{\cdot}_\Psi(-)$, dropping the dot.

We highlight here a few key parts of this definition. Because substitutions are η-long, variable lookup is guaranteed to succeed if the variable is in the domain of the substitution. i.e. the case

Variable lookup

$$\begin{array}{lll}
(\sigma, M)_{\Psi,x:A}(x) & = M : (A)^{-} & \\
(\sigma, M)_{\Psi,y:A}(x) & = \sigma_{\Psi}(x) & \text{where } x \neq y \\
(\sigma)_{\Psi}(x) & \quad\text{fails otherwise} &
\end{array}$$

Truncation

$$\begin{array}{lll}
\mathsf{trunc}_{\psi}(\sigma) & = \sigma & \\
\mathsf{trunc}_{\Psi,x:A}(\sigma, N) & = \mathsf{trunc}_{\Psi}(\sigma) & \\
\mathsf{trunc}_{\Psi}(\sigma) & \quad\text{fails otherwise} &
\end{array}$$

Normal terms

$$\begin{array}{lll}
[\sigma]^{\hat{\Phi}}_{\Psi}(\lambda y.N) & = \lambda y.N' & \text{where } N' = [\sigma]^{\hat{\Phi},y}_{\Psi}(N) \\
 & & \text{choosing } y \notin \mathsf{FV}(\sigma) \\
 & & \text{and } y \notin \Psi \\
[\sigma]^{\hat{\Phi}}_{\Psi}(u[\tau]) & = u[\tau'] & \text{where } \tau' = [\sigma]^{\hat{\Phi}}_{\Psi}(\tau) \\
[\sigma]^{\hat{\Phi}}_{\Psi}(\mathbf{c} \cdot S) & = \mathbf{c} \cdot S' & \text{where } S' = [\sigma]^{\hat{\Phi}}_{\Psi}S \\
[\sigma]^{\hat{\Phi}}_{\Psi}(x \cdot S) & = \mathsf{reduce}(M{:}\alpha,S') & \text{where } x \notin \hat{\Phi} \\
 & & \text{and } \sigma_{\Psi}(x) = M : \alpha \\
 & & \text{and } S' = [\sigma]^{\hat{\Phi}}_{\Psi}(S) \\
[\sigma]^{\hat{\Phi}}_{\Psi}(y \cdot S) & = y \cdot S' & \text{where } y \in \hat{\Phi} \\
 & & \text{and } S' = [\sigma]^{\hat{\Phi}}_{\Psi}(S) \\
[\sigma]^{\hat{\Phi}}_{\Psi}(p[\tau] \cdot S) & = p[\tau'] \cdot S' & \text{where } \tau' = [\sigma]^{\hat{\Phi}}_{\Psi}(\tau) \\
 & & \text{and } S' = [\sigma]^{\hat{\Phi}}_{\Psi}(S) \\
[\sigma]^{\hat{\Phi}}_{\Psi}((\hat{\Psi}'.x)[\rho] \cdot S) & = (\hat{\Psi}'.x)[\rho'] \cdot S' & \text{where } \rho' = [\sigma]^{\hat{\Phi}}_{\Psi}(\rho) \\
 & & \text{and } S' = [\sigma]^{\hat{\Phi}}_{\Psi}(S)
\end{array}$$

Spines

$$\begin{array}{lll}
[\sigma]^{\hat{\Phi}}_{\Psi}(\epsilon) & = \epsilon & \\
[\sigma]^{\hat{\Phi}}_{\Psi}(N; S) & = N'; S' & \text{where } N' = [\sigma]^{\hat{\Phi}}_{\Psi}(N) \\
 & & \text{and } S' = [\sigma]^{\hat{\Phi}}_{\Psi}(S)
\end{array}$$

Substitutions

$$\begin{array}{lll}
[\sigma]^{\hat{\Phi}}_{\Psi}(\cdot) & = \cdot & \\
[\sigma]^{\hat{\Phi}}_{\Psi}(\mathsf{id}_{\psi}) & = \mathsf{trunc}_{\Psi}(\sigma) & \\
[\sigma]^{\hat{\Phi}}_{\Psi}(\tau, N) & = \tau', N' & \text{where } \tau' = [\sigma]^{\hat{\Phi}}_{\Psi}(\tau) \\
 & & \text{and } N' = [\sigma]^{\hat{\Phi}}_{\Psi}(N) \\
[\sigma]^{\hat{\Phi}}_{\Psi}(\mathsf{id}_{\psi}[\rho]) & = \mathsf{id}_{\psi}[\rho'] & \text{where } \rho' = [\sigma]^{\hat{\Phi}}_{\Psi}(\rho)
\end{array}$$

Substitution Closures

$$\begin{array}{lll}
[\sigma]^{\hat{\Phi}}_{\Psi}(s[\tau]) & = s[\tau'] & \text{where } \tau' = [\sigma]^{\hat{\Phi}}_{\Psi}(\tau)
\end{array}$$

Figure 2. Simultaneous hereditary substitution

$(s[\sigma])_{\Psi,x:A}(x)$ which is problematic in Pientka [2008] cannot arise here. The truncation operation $\mathsf{trunc}_{\Psi}(\sigma)$ has the effect of dropping components of σ, keeping only the portion supplying instantiations for a context variable. This arises when applying $[\sigma]^{\hat{\Phi}}_{\Psi}(\mathsf{id}_{\psi})$. From the typing, we will know that σ must begin with a substitution having domain ψ and that this operation must produce a substitution with domain ψ. So we simply chop off the suffix we do not need. That is, id_{ψ} acts as a weakening substitution. Since σ must be in fully expanded form, the problematic case of $\mathsf{trunc}_{\Psi,x:A}(s[\sigma])$ in Pientka [2008] is not possible here.

There is one last tool we need to define the typing rules. We will need certain expansion operations. In particular, we need to compute the expanded form of $s[\sigma]$ at domain Ψ, which in turn also requires η-expansion. We also define here the related operation of expanding identity substitutions. We will make more extensive use of these operations in the next section. Below we define η-expansion on the approximate type, using @ to mean appending a term on the end of a spine. In a spine calculus, η-expansion takes a head H and a spine S, and computes the η-long form of $H \cdot S$:

$$\begin{array}{rcl}
\eta\text{-}\mathsf{exp}_{\mathbf{a}}(H, S) & = & H \cdot S \\
\eta\text{-}\mathsf{exp}_{\alpha\to\beta}(H, S) & = & \lambda x.\eta\text{-}\mathsf{exp}_{\beta}(H, (S @ \eta\text{-}\mathsf{exp}_{\alpha}(x))) \\
 & & \text{where } x \notin \mathsf{FV}(H) \cup \mathsf{FV}(S)
\end{array}$$

We write $\eta\text{-}\mathsf{exp}_{\alpha}(H)$ as shorthand for $\eta\text{-}\mathsf{exp}_{\alpha}(H, \epsilon)$. In the arrow case, we introduce a fresh variable x and continue expanding H applied to the spine S extended with x. We can now expand the identity substitution for a context Ψ by appealing to η-expansion. We simply walk through the context, η-expanding each variable. When we encounter a context variable, we are stuck (that is, until we have a concrete instantiation for it), so we make use of the stuck syntactic form id_{ψ}:

$$\begin{array}{rcl}
\mathsf{id}(\cdot) & = & \cdot \\
\mathsf{id}(\psi) & = & \mathsf{id}_{\psi} \\
\mathsf{id}(\Psi, x{:}A) & = & \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{A}(x)
\end{array}$$

Next we define substitution expansion. The expansion $\mathsf{sexp}_{\Psi}(\rho)$ computes the expanded form of ρ when the domain of s is a general context Ψ. To do so, we must generalize this operation to perform expansion underneath a context $\hat{\Phi}$. We similarly traverse the context, for each variable x in the domain Ψ we add the x projection of ρ. When we encounter a context variable, we are similarly stuck, and so we use the stuck syntactic form $\mathsf{id}_{\psi}[\rho]$:

$$\begin{array}{rcl}
\mathsf{sexp}^{\hat{\Phi}}_{\cdot}(\rho) & = & \cdot \\
\mathsf{sexp}^{\hat{\Phi}}_{\psi}(\rho) & = & \mathsf{id}_{\psi}[\rho] \\
\mathsf{sexp}^{\hat{\Phi}}_{\Psi,x:A}(\rho) & = & \mathsf{sexp}^{x,\hat{\Phi}}_{\Psi}(\rho), \eta\text{-}\mathsf{exp}_{A}((\hat{\Psi},x,\hat{\Phi}.\ x)[\rho])
\end{array}$$

We now have the tools to define the typing of Contextual LF with substitution variables. We use the following judgements:

$\Delta; \Psi \vdash M \Leftarrow A$	Normal term M checks against type A
$\Delta; \Psi \vdash H \Rightarrow A$	Head H synthesizes type A
$\Delta; \Psi \vdash S > A \Rightarrow B$	Spine S synthesizes type B
$\Delta; \Psi \vdash \sigma \Leftarrow \Phi$	Substitution σ has domain Φ and range Ψ
$\Delta; \Psi \vdash \rho \Rightarrow \Phi$	Sub. Closure ρ has domain Φ and range Ψ

The bi-directional typing rules are mostly straightforward and are presented in Figure 3. The context Δ contains the various forms of meta-level variables and is described in more detail in the next section, while the context Ψ contains regular LF variables. 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-objects we tacitly apply α-renaming. We write Ψ, Φ for context concatenation. When we write this, we mean to imply that Φ does *not* start with a context variable, i.e. Φ is a pure list of the form $x_1{:}A_1, ..., x_n{:}A_n$.

We highlight here a few of the interesting rules. The rule for substitution closures ρ of the form $s[\sigma]$ types the stuck composition of a substitution variable with a substitution σ. If s takes Φ to Ψ' and σ takes Ψ' to Ψ, then the stuck composition $s[\sigma]$ takes Φ to Ψ. Substitution closures can be used in two ways: the first, $\mathsf{id}_{\psi}[\rho]$ projects out the ψ components of ρ. So if ρ provides instantiations for ψ and Φ then we can discard those for Φ, leaving only the instantiations for ψ. The second, $(\hat{\Phi}.x)[\rho]$ allows one to use the instantiations ρ provides for concrete variables. If ρ provides

Heads $\boxed{\Delta; \Psi \vdash H \Rightarrow A}$

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

$$\frac{\Sigma(\mathbf{c}) = A}{\Delta; \Psi \vdash \mathbf{c} \Rightarrow A} \qquad \frac{\Delta; \Psi \vdash \rho \Rightarrow \Phi \quad \Phi(x) = A}{\Delta; \Psi \vdash (\hat{\Phi}.x)[\rho] \Rightarrow [\mathsf{sexp}_\Phi(\rho)]_\Phi A}$$

Spines $\boxed{\Delta; \Psi \vdash S > A \Rightarrow B}$

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

$$\frac{}{\Delta; \Psi \vdash \epsilon > A \Rightarrow A}$$

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

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

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

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

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

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

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]_\Phi A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x{:}A} \qquad \frac{\Delta; \Psi \vdash \rho \Rightarrow \psi, \Phi}{\Delta; \Psi \vdash \mathsf{id}_\psi[\rho] \Leftarrow \psi}$$

Sub. Closures $\boxed{\Delta; \Psi \vdash \rho \Rightarrow \Phi}$

$$\frac{\Delta(s) = \Phi[\Psi'] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Psi'}{\Delta; \Psi \vdash s[\sigma] \Rightarrow \Phi}$$

Figure 3. Typing for contextual LF with substitution variables

instantiations for Φ, among which is a variable $x{:}A$, then the type of the stuck projection $(\hat{\Phi}.x)[\rho]$ is $[\mathsf{sexp}_\Phi(\rho)]_\Phi(A)$, in effect applying ρ to A. We must expand the substitution, since $[-]$ is only defined for fully-expanded substitutions. This has the unfortunate behaviour of potentially performing η expansions which then trigger spurious β reductions during hereditary substitution. This is sufficient for our purposes, although it could be avoided at the cost of directly defining an operation $[\rho](-)$.

The expected weakening properties hold in our type system. The following lemmas provide the typing for the expansion operations and hereditary substitution. Below, we use J to stand for any of the forms of judgements defined above.

Lemma 3.2 (Expansion lemmas)**.**

1. *If* $\Delta; \Psi \vdash H \Rightarrow A$ *and* $\Delta; \Psi \vdash S > A \Rightarrow B$ *and* $\eta\text{-}\mathsf{exp}_B(H, S) = M$ *then* $\Delta; \Psi \vdash M \Leftarrow B$
2. *If* $\mathsf{id}(\Psi) = \sigma$ *then* $\Delta; \Psi \vdash \sigma \Leftarrow \Psi$
3. *If* $\Delta; \Psi' \vdash \rho \Rightarrow \Psi, \Phi$ *and* $\mathsf{sexp}_\Psi^{\hat{\Phi}}(\rho) = \sigma$ *then* $\Delta; \Psi' \vdash \sigma \Leftarrow \Psi$

Lemma 3.3 (Substitution lemma)**.**

1. *If* $\Delta; \Psi \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x : A, \Phi \vdash J$ *then* $\Delta; \Psi, [M/x]_A(\Phi) \vdash [M/x]_A^*(J)$ *where* $* \in \{n, s, l\}$
2. *If* $\Delta; \Psi \vdash M \Leftarrow A$ *and* $\Delta; \Psi \vdash S > A \Rightarrow B$ *and* $\mathsf{reduce}(M : (A)^-, S) = M'$ *then* $\Delta; \Psi \vdash M' \Leftarrow B$

Lemma 3.4 (Simultaneous substitution lemma)**.**

1. *If* $\Delta; \Psi, \Phi \vdash J$ *and* $\Delta; \Psi' \vdash \sigma \Leftarrow \Psi$ *then* $\Delta; \Psi', [\sigma]^{\cdot}_\Psi(\Phi) \vdash [\sigma]_\Psi^{\hat{\Phi}}(J)$
2. *If* $\Delta; \Psi \vdash \sigma \Leftarrow \psi, \Phi$ *then* $\Delta; \Psi \vdash \mathsf{trunc}_{\psi,\Phi}(\sigma) \Leftarrow \psi$

Since hereditary substitution terminates, our typing rules are clearly syntax directed, and equality is simply syntactic (modulo α-equivalence), our typing rules immediately give rise to an algorithm for typechecking.

Theorem 3.5. *Typechecking is decidable.*

3.2 Meta-Objects and Meta-types

To give meaning to the substitution variables present in our syntax, we must describe what it means to substitute for one. We also need to carefully describe the interaction of our substitution variables with context variables and substitution for context variables. Indeed, since we intend to use Contextual LF with substitution variables as an index language in Beluga, we must lift it to a meta-level where we abstract over all the free variables of a term, and we must define the meta-substitution operation.

We base our presentation of the meta-level here on the presentation by Cave and Pientka [2012], highlighting the extensions necessary for substitution variables. We lift contextual LF objects to meta-types and meta-objects to treat abstraction over meta-objects uniformly. Meta-objects include contextual objects written as $\hat{\Psi}.R$, contexts Ψ, and now also contextual substitutions written $\hat{\Psi}.\sigma$. These are the index objects which can be used to index computation-level types in Beluga. There are four different flavours of meta-types: $P[\Psi]$ denotes the type of a meta-variable u and stands for a general contextual object $\hat{\Psi}.R$. $\#A[\Psi]$ denotes the type of a parameter variable p and it stands for a variable object, i.e. either $\hat{\Psi}.x$ or $\hat{\Psi}.p[\pi]$ where π is a variable substitution. A variable substitution π is a special case of general substitutions σ; however unlike $p[\sigma]$ which can produce a general LF object, $p[\pi]$ guarantees we are producing a variable. G describes the schema (i.e. type) of a context. Last, the type $\Psi[\Phi]$ characterizes substitutions with domain Ψ and range Φ – they take objects defined in the context Ψ to objects defined in the context Φ.

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; for context variables we use ψ and for substitution variables we use s.

Context schemas	G	$::=$	$\exists\overrightarrow{(x{:}A)}.B \mid G + \exists\overrightarrow{(x{:}A)}.B$
Meta Objects	C	$::=$	$\hat{\Psi}.R \mid \Psi \mid \hat{\Psi}.\sigma$
Meta Types	U	$::=$	$P[\Psi] \mid \#A[\Psi] \mid \Psi[\Phi] \mid G$
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. For example, a context $x{:}\mathsf{exp}\ \mathsf{nat}, y{:}\mathsf{exp}\ \mathsf{bool}$ will check against the schema $\exists T{:}\mathsf{tp}.\mathsf{exp}\ T$.

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

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

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

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

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

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

$$\frac{}{\Delta \vdash \cdot \Leftarrow \cdot} \quad \frac{\Delta \vdash \theta \Leftarrow \Delta' \quad \Delta \vdash C \Leftarrow \llbracket\theta\rrbracket_{\Delta'}(U)}{\Delta \vdash \theta, C/X \Leftarrow \Delta', X{:}U}$$

Figure 4. Typing for meta-terms

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

We define the typing rules for meta-objects in Figure 4. These are unchanged from Cave and Pientka [2012], with the exception of the added rule for substitution objects.

In Figure 5 we present the definition of meta-substitution, which substitutes a meta object for a metavariable, parameter variable, context variable, or substitution variable. We give a single definition of meta-substitution which can substitute for any of these four flavours of meta-types. We annotate the meta-substitution with the type of the object it is substituting, since we need this information to pass on to hereditary substitution.

The case for metavariables is standard for CTT. When substituting a meta-object $\hat{\Psi}.R$ for u in $u[\sigma]$, we first continue substituting for occurences of u in σ to obtain σ'. Now that we know more concretely what u is we trigger the suspended substitution $[\sigma']_\Psi(R)$, appealing now to hereditary substitution.

The cases of interest in this paper are those involving substitution variables. When we substitute a concrete τ for s in $(\hat{\Phi}.x)[s[\sigma]]\cdot S$, we compose it with the suspended σ, lookup x in the resulting substitution to obtain a normal term, and reduce the redexes which arise from its application to S.

Two interesting cases arise when substituting into $\mathsf{id}_\psi[s[\sigma]]$. When substituting a concrete τ for s, we must compose it with the suspended σ and drop the suffix of the substitution, keeping only the ψ component. When substituting a concrete Ψ for ψ, we must expand $s[\sigma]$ into a genuine substitution.

This operation obeys the following typing lemma:

Lemma 3.6 (Meta-substitution lemma)**.**
If $\Delta \vdash C \Leftarrow U$ *and* $\Delta, X{:}U, \Delta'; \Psi \vdash J$
then $\Delta, \llbracket C/X\rrbracket_U(\Delta'); \llbracket C/X\rrbracket_U(\Psi) \vdash \llbracket C/X\rrbracket_U(J)$

With meta-substitution properly defined, we can simply plug the meta level into the computation level framework described in Cave and Pientka [2012] to obtain the type safety results proven there. As a result, we obtain a language capable of formalizing the weak normalization example of Section 2.2.

4. Formalization and Implementation

We have carried out a partial mechanization of the results here in Agda [Norell 2007], a dependently typed programming language. Namely, we have implemented simply-typed variants of the definitions and results prior to Section 3.2, using strongly-typed term representations – for a reference on this technique, see Benton et al. [2012]. This follows work by Keller and Altenkirch [2010] on formalizing hereditary substitution for simple types. Our implementation in Agda serves to establish the totality (termination and coverage) of the hereditary substitution operations we define here, as well as simply-typed approximations of the substitution lemmas and expansion lemmas. Note that it does not attempt to prove any of the algebraic properties of substitution we expect to hold, as proving these is still burdensome using this technique. Nevertheless, this proved to be a valuable tool in assisting to develop the theory, in particular because the on-paper definitions are traditionally written first as partial functions before proving they are total on well-typed terms. The strongly-typed term representation, on the other hand, allowed us to directly define hereditary substitution as a total function, ruling out cases which are impossible due to typing.

We have also carried out a preliminary implementation of CTT^{s} in the Beluga system, which is capable of typechecking the example in Section 2.2, save for the fact that type reconstruction involving substitution variables is not presently as powerful as the example suggests, and we must supply more arguments explicitly. However, we see no fundamental obstacles to improving the reconstruction to the point where the example can be typechecked as-is. We plan to integrate substitution variables more smoothly into type reconstruction and make a public release in the near future.

5. Related Work

Abadi et al. [1990] briefly consider metavariables (including substitution variables) in the context of their explicit substitution calculus, concluding that the η rule (surjective pairing) for substitutions leads to non-confluence. The lack of confluence is not a concern in our setting, as we are only concerned with normal forms and not rewriting. Our work provides a precise analysis of the complex normal forms Abadi et al. allude to. We conjecture that our hereditary substitution gives rise to a sound and complete decision procedure for (a typed variant of) their calculus, although it remains to work out the details.

Subsequently, substitution variables, sometimes referred to as first-class environments, have been considered for simply-typed calculi [Hashimoto and Ohori 2001; Mason 1999; Nishizaki 2000; Sato et al. 2001, 2002]. Nishizaki [2000] for example extends a lambda-calculus with explicit substitutions following in the spirit Abadi et al. [1990] and representing variables via de Bruijn indices.

The present work also bears a similarity to the row polymorphism of Rémy [1991]. Row polymorphism provides a notion of record polymorphism which admits type inference. It allows one to polymorphically quantify over row variables representing a subset of the fields of a record. Here our context variables fill a role similar to row variables, and our substitutions fill a role similar to records. Our $(\hat{\Psi}.x)[\rho]$ construction acts as a field access. Through this lens, it would seem that our work provides a perspective on η-expansion and η-long forms for row-polymorphic records. One notable difference is that we consider our substitution types equivalent under α-renaming, while this is generally not the case for record fields. Our setting is also dependently typed instead of polymorphically typed.

6. Conclusion

We have presented a full theory of substitution variables within contextual type theory which solves several issues, previous pro-

Meta-substitution for terms

$$
\begin{array}{lll}
\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(x \cdot S) & = x \cdot S' & \text{where } \llbracket C/X \rrbracket_U(S) = S' \\
\llbracket C/X \rrbracket_U(\mathbf{c} \cdot S) & = \mathbf{c} \cdot S' & \text{where } \llbracket C/X \rrbracket_U(S) = S' \\
\llbracket C/X \rrbracket_U(p[\sigma] \cdot S) & = q[\sigma''] \cdot S' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \text{and } \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.q[\pi]/p \rrbracket_{\#A[\Psi]} \\
 & & \text{and } [\sigma']_\Psi(\pi) = \sigma'' \\
 & & \text{and } \llbracket C/X \rrbracket_U(S) = S' \\
\llbracket C/X \rrbracket_U(p[\sigma] \cdot S) & = M' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \text{and } \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.x/p \rrbracket_{\#A[\Psi]} \\
 & & \text{and } \sigma'_\Psi(x) = M : \alpha \\
 & & \text{and } \llbracket C/X \rrbracket_U(S) = S' \\
 & & \text{and } \mathsf{reduce}(M : \alpha, S') = M' \\
\llbracket C/X \rrbracket_U(p[\sigma] \cdot S) & = p[\sigma'] \cdot S' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.H/p \rrbracket_{\#A[\Psi]} \\
 & & \text{and } \llbracket C/X \rrbracket_U(S) = S' \\
\llbracket C/X \rrbracket_U((\hat{\Phi}.\, x)[s[\sigma]] \cdot S) & = M' & \text{where } \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.\tau/s \rrbracket_{\Phi[\Psi]} \\
 & & \text{and } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \text{and } [\sigma']_\Psi(\tau) = \tau' \\
 & & \text{and } \tau'_\Phi(x) = M : \alpha \\
 & & \text{and } \llbracket C/X \rrbracket_U(S) = S' \\
 & & \text{and } \mathsf{reduce}(M : \alpha, S') = M' \\
\llbracket C/X \rrbracket_U((\hat{\Phi}.\, x)[s[\sigma]] \cdot S) & = (\hat{\Phi}'.\, x)[s[\sigma']] \cdot S' & \text{where } \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.\tau/s \rrbracket_{\Phi[\Psi]} \\
 & & \text{and } \llbracket C/X \rrbracket_U(\hat{\Phi}) = \hat{\Phi}' \\
 & & \text{and } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \text{and } \llbracket C/X \rrbracket_U(S) = S'
\end{array}
$$

Meta-substitution for substitutions

$$
\begin{array}{lll}
\llbracket C/X \rrbracket_U(\cdot) & = \cdot & \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi) & = \sigma & \text{where } \llbracket \mathcal{C}/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G \text{ and} \\
 & & \mathsf{id}(\Psi) = \sigma \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi) & = \mathsf{id}_\psi & \text{where } \llbracket \mathcal{C}/X \rrbracket_U \neq \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\sigma, M) & = \sigma', M' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U(M) = M' \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi[s[\sigma]]) & = \mathsf{trunc}_\Phi(\tau') & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \text{and } \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.\tau/s \rrbracket_{\Phi[\Psi]} \\
 & & \text{and } [\sigma']_\Psi(\tau) = \tau' \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi[s[\sigma]]) & = \mathsf{sexp}_\Psi(s[\sigma']) & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \text{and } \llbracket C/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi[s[\sigma]]) & = \mathsf{id}_\psi[s[\sigma']] & \text{where } \llbracket C/X \rrbracket_U \neq \llbracket \Psi/\psi \rrbracket_G \\
 & & \text{and } \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.\tau/s \rrbracket_{\Phi[\Psi]} \\
 & & \text{and } \llbracket C/X \rrbracket_U(\sigma) = \sigma'
\end{array}
$$

Meta-substitution for context

$$
\begin{array}{lll}
\llbracket C/X \rrbracket_U(\cdot) & = \cdot & \\
\llbracket C/X \rrbracket_U(\psi) & = \Psi & \text{where } \llbracket C/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\psi) & = \psi & \text{where } \llbracket C/X \rrbracket_U \neq \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\Psi, x{:}A) & = \Psi', x{:}A' & \text{where } \llbracket C/X \rrbracket_U(\Psi) = \Psi' \text{ and} \\
 & & \llbracket C/X \rrbracket_U(A) = A'
\end{array}
$$

Figure 5. Meta-substitution

posals left open: it defines a canonical form for substitutions and we include the closure of a bound variable with a substitution variable as well as the closure of the identity substitution with a substitution variable. Our resulting type theory characterizes only normal forms and type checking is decidable.

We have also illustrated the elegance and usefulness of substitution variables, when implementing normalization proofs. A preliminary prototype for programming with first-class substitution variables is under way within the dependently typed programming and proof language, Beluga.

In the future, we plan to extend our preliminary prototype to fully support type reconstruction in the presence of substitution variables and demonstrate their power in encoding strong normalization proofs. We also plan to extend the coverage checker to guarantee that splitting over substitutions is defined on all cases. Finally, we plan to address the issue of termination to guarantee that our programs are total. This also requires us to check that a given recursive data-type definition is strictly positive or stratified.

Acknowledgments

We wish to thank the anonymous referees for their valuable feedback.

References

Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–46. ACM Press, 1990.

Nick Benton, Chung-Kil Hur, AndrewJ. Kennedy, and Conor McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

Andrew Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008.

David Gelernter, Suresh Jagannathan, and Thomas London. Environments as first class objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*, pages 98–110, New York, NY, USA, 1987. ACM.

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

Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001.

Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010.

Ian A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, 12(2):171–201, 1999.

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

Shin-Ya Nishizaki. A polymorphic environment calculus and its type-inference algorithm. *Higher Order Symbol. Comput.*, 13(3):239–278, 2000.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007. Technical Report 33D.

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

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 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 6173), pages 15–21. Springer-Verlag, 2010.

Adam B. Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *17th European Symposium on Programming (ESOP '08)*, volume 4960, pages 93–107. Springer, 2008.

Christian Queinnec and David De Roure. Sharing code through first-class environments. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 251–261. ACM, 1996.

Didier Rémy. Type inference for records in a natural extension of ML. Research Report 1431, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, May 1991.

Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. *Fundamenta Informaticae*, 45(1-2):79–115, 2001.

Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, 2002(4):1–41, March 2002.

Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In *23rd Annual Symposium on Logic in Computer Science (LICS), Pittsburgh, PA, USA*, pages 69–80. IEEE Computer Society, 2008.