Programming with Proofs and Explicit Contexts

[Extended Abstract]

Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

Joshua Dunfield
School of Computer Science
McGill University
Montreal, Canada
joshua@cs.mcgill.ca

Abstract

This paper explores a new point in the design space of functional programming: functional programming with dependently-typed higher-order data structures described in the logical framework LF. This allows us to program with proofs as higher-order data. We present a decidable bidirectional type system that distinguishes between dependently-typed data and computations. To support reasoning about open data, our foundation makes contexts explicit. This provides us with a concise characterization of open data, which is crucial to elegantly describe proofs. In addition, we present an operational semantics for this language based on higher-order pattern matching for dependently typed objects. Based on this development, we prove progress and preservation.

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

General Terms Theory, Languages

Keywords Type theory, Dependent types, Logical frameworks

1. Introduction

Various forms of dependent types have found their way into mainstream functional programming languages to allow programmers to express stronger properties about their programs [2, 16, 26, 29]. In this paper, we explore a new point in the design space of functional programming with dependent types where we can analyze and manipulate dependently-typed higher-order data described in the logical framework LF [9]. LF provides a rich meta-language for describing formal systems defined by axioms and inference rules (such as a type system, a logic, etc.) together with proofs within these systems (such as a typing derivation, a proof of a proposition, etc.). 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 meta-language. For example, the formula $\forall x.(x = 1) \supset \neg(x = 0)$ is represented as `forall λx. (eq x (Suc Zero)) imp (not (eq x Zero))`. This simple, but powerful idea avoids the need to implement common and tricky machinery for variables, such as capture-avoiding substitution and renaming.

However, implementing and verifying proofs about HOAS encodings has been notoriously difficult. There are two distinct challenges. First, encodings based on HOAS require us to recursively traverse binders and describe open data objects within a context. Second, dependent types add another degree of complexity: data objects themselves can refer to types and the structure of types can be observed.

Programming with HOAS has recently received widespread attention, although most work has focused on the simply-typed setting [5, 10, 21, 25]. Only a few approaches have been considered in the dependently-typed setting. Despeyroux and Leleu [3, 4] extended previous work by Despeyroux et al. [5] which provided a type-theoretic foundation for primitive recursive programming with HOAS. The Delphin language [23] extends these ideas to provide general recursion over HOAS encodings as well as dependent types.

In this paper, we present an alternative to these approaches, extending the first author's previous work on programming with HOAS encodings in the simply-typed setting [21] to dependent types. As in that work, we use contextual modal types [15] to separate HOAS data from computations about them. Open data M is characterized by the contextual modal type $A[\Psi]$ where M has type A in the *context* Ψ. The object M is closed with respect to the context Ψ, so M can refer to variables declared in Ψ. For example, the object `eq x (Suc Zero)` has type `o[x:nat]` (where `o` describes propositions). Since we want to allow recursion over open data and the local context Ψ associated with the type A may grow, our foundation supports *context variables*, which allow us to abstract over contexts. Just as types classify terms, and kinds classify types, we use *context schemas* to classify and characterize contexts.

In this paper, we revisit the key concepts from the simply-typed setting [21] and generalize them to dependent types. While extending the data language with dependent types follows from previous work [15], the generalization of context schemas to dependent types and, especially, the integration of dependently typed higher-order data into the computation language are novel. Our discussion will highlight the fact that the simply-typed foundation scales nicely to the setting of dependent types. This lays the foundation for programming with proofs and explicit contexts. Inductive proofs about formal systems can be implemented as recursive functions, and case analysis in the inductive proof corresponds to analyzing dependently-typed higher-order data via pattern matching. We call this language Beluga.

We make the following contributions. First, we present a syntax-directed decidable type system for dependently-typed open data.

PPDP'08, July 15–17, 2008, Valencia, Spain.

Our presentation only admits data objects in canonical form because only those represent meaningful data. This follows the ideas of Watkins et al.[28] and Nanevski et al.[15]. By cleanly separating the data language from the computation language, exotic terms that do not represent legal data objects are prevented. Our framework supports explicit context variables to abstract over the concrete context and parameterizes computations by contexts, classified by context schemas. While context schemas have been present in the simply-typed setting, characterizing contexts in the presence of dependent types is more complex.

Second, we present a type system for computation that allows recursion and pattern matching on dependently-typed open data. This type system is also syntax-directed and decidable. It allows dependent types, but only types indexed by data objects, not arbitrary, possibly recursive computations. Because data objects are in canonical form, equality between two data objects is easily decided by syntactic equality. This is crucial to achieve a decidable type system. The language we describe is intended as an internal language, into which a source level language will be compiled. It clearly distinguishes between *implicit* data objects that occur in a dependent type and may be kept implicit in the actual source program, and *explicit* data objects that are recursively analyzed with higher-order pattern matching [13, 17] and are explicit in the source program. Intuitively, implicit data arguments are the index objects that are reconstructed when translating a source language to this internal language. In addition, our type system is unique in that constraints due to dependent types are resolved eagerly during type checking using higher-order pattern unification, leading to an elegant decidable algorithm.

Finally, we present a small-step operational semantics based on higher-order pattern matching for dependently-typed objects [17, 22], and prove progress and preservation for our language with respect to this semantics.

This paper is a first important step in laying the type-theoretic foundation for programming with proofs and explicit contexts. Our foundation ensures that contexts are well-formed according to a user-specified schema, and we have formulated a coverage checking algorithm [6] for dependently-typed open data. At this point, the only missing piece is to verify that a given function terminates—we leave this for a separate paper. As such, our work may be thought of as an alternative to Twelf [19], an implementation of the logical framework LF [9] in which proofs about formal systems are encoded via relations, and to Delphin [23] (which, like our system, implements proofs via functions).

2. Motivation

To motivate the problem, we consider a program that counts the free occurrences of some variable x in a formula. For example, the formula $\forall y.(x = y) \supset (\mathsf{suc}\ y = \mathsf{suc}\ x)$ has two free occurrences of x. The data language here is first-order logic with quantification over natural numbers. We begin by defining a type `o` for propositions and we use higher-order abstract syntax to model the binder in the universal quantifier.

```
nat : type .          o      : type .
Zero: nat.            eq     : nat → nat → o.
Suc : nat → nat.      imp    : o → o → o.
                      forall: (nat → o) → o.
```

2.1 Counting free variable occurrences

We will approach this problem top-down and first consider the function `cntV` which will recursively analyze formulas using pattern matching. When it reaches the proposition `eq`, it will call a second function `cntVN`. The modal type `o[x:nat, y:nat]` describes a formula that can refer to the variables `x` and `y`. One element of this type is the formula `((eq x y) imp (eq (Suc x) (Suc y)))`.

When analyzing a formula with a universal quantifier, the set of free variables grows. Hence, we need to abstract over the contexts in which the formula makes sense. Context variables ψ provide this functionality. The function `cntV` takes a context ψ of natural numbers, a formula `f`, and returns an integer. Just as types classify data objects and kinds classify types, we introduce *schemas* to classify contexts. In the type declaration for the function `cntV` we say that the context variable ψ has the schema $(\mathtt{nat})^*$, meaning that ψ stands for a data-level context $\mathtt{x}_1\mathtt{:nat},\ldots,\mathtt{x}_n\mathtt{:nat}$.

```
ctx schema natCtx = (nat)*

rec cntV : Π ψ:natCtx.o[ψ,x:nat] → nat[.] =
Λ ψ ⇒ fn f ⇒ case f of
  box(ψ,x. imp U[id_ψ,x] W[id_ψ,x]) ⇒
   add (cntV ⌈ψ⌉ box(ψ,x. U[id_ψ,x]))
       (cntV ⌈ψ⌉ box(ψ,x. W[id_ψ,x]))
| box(ψ,x. forall(λy.U[id_ψ,x,y])) ⇒
     cntV⌈ψ,y:nat⌉ box(ψ,y,x. U[id_ψ,x,y])
| box(ψ,x. eq U[id_ψ,x] V[id_ψ, x]) ⇒
    add (cntVN ⌈ψ⌉ box(ψ,x.U[id_ψ,x]))
        (cntVN ⌈ψ⌉ box(ψ,x.V[id_ψ,x]))
```

The function `cntV` is built by a context abstraction $\Lambda\ \psi$ that introduces the context variable ψ and binds every occurrence of ψ in the body. Next, we introduce the computation-level variable `f` which has type `o[`ψ`,x:nat]`. In the body of the function `cntV` we case-analyze objects of type `o[`ψ`,x:nat]`. The `box` construct separates data from computations.

When we encounter an object built from a constructor `eq`, `imp`, or `forall`, we need to access the subexpression(s) underneath. Pattern variables are characterized by a closure `U[`σ`]` consisting of a contextual variable `U` and a *postponed substitution* σ. As soon as we know what the contextual variable stands for, we apply the substitution σ. In the example, the postponed substitution associated with `U` is the identity substitution which essentially corresponds to α-renaming. We write id_ψ for the identity substitution with domain ψ. Intuitively, one may think of the substitution associated with contextual variables occurring in patterns as a list of variables which may occur in the hole. In the data object `U[`id_ψ`]`, for example, the contextual variable `U` can be instantiated with any formula that is either closed (i.e. it does not refer to any bound variable listed in the context ψ) or contains a bound variable from the context ψ. Since we want to allow subformulas to refer to all variables in `(`ψ`, x:nat)`, we write `U[`id_ψ`, x]`. We use capital letters for meta-variables.

In the first case for `imp` we recursively analyze the subformulas described by `U[`id_ψ`,x]` and `W[`id_ψ`,x]`. The context ψ is the same, so we pass it (explicitly) in the recursive calls.

In the case for `box(`ψ`,x. forall (`λ`y.W[`id_ψ`,x,y]))`, we analyze the formula `W[`id_ψ`,x,y]` under the assumption that `y` is a natural number. To do this, we pass an extended context `(`ψ`,y:nat)` to `cntV`.

Finally, for `eq`, we call `cntVN` to count the occurrences of `x` in the natural numbers `U[`id_ψ`,x]` and `V[`id_ψ`, x]`, explicitly passing the context ψ. The function `cntVN` counts the occurrences of a variable `x` in an object of type `nat[`ψ`,x:nat]`.

```
rec cntVN : Π ψ:natCtx.nat[ψ,x:nat] → nat[.] =
 Λ ψ ⇒ fn n ⇒ case n of
   box(ψ,x. x) ⇒ box( . Suc Z)
 | box(ψ,x. p[id_ψ]) ⇒ box( . Z)
 | box(ψ,x. Zero) ⇒ box( . Z)
 | box(ψ,x. Suc U[id_ψ,x]) ⇒
     cntVN ⌈ψ⌉ box(ψ,x. U[id_ψ,x])
```

The first and second cases are the interesting ones. The first patttern, box(ψ,x. x), matches an occurrence of x. The second pattern, box(ψ,x. p[id_ψ]), matches a variable that is not x and occurs in ψ. For this case, we use a *parameter variable* p (the small letter distinguishes it from a meta-variable). This represents a bound data-level variable. The substitution id_ψ associated with p characterizes the possible instantiations of p. This allows us to write a generic base case where we encounter a parameter from the context that ψ stands for. This combination of explicit context variables and parameter variables to describe generic base cases is unique to our work. Comparison and explicit pattern matching for variables are features typically associated with nominal systems [7, 27]. However, unlike nominal systems, variable names here are not global, but local and subject to α-renaming.

In Twelf [19], one can write a relation that counts variable occurrences, but there is no generic base case for counting variables in a natural number. Instead, one dynamically adds the appropriate base case for every variable introduced when traversing a universal quantifier.

2.2 Example: Programming with proofs

We illustrate the idea of programming with proofs by considering bracket abstraction, a key lemma that arises in the translation of natural deduction proofs to Hilbert-style proofs. For this discussion we concentrate on the fragment consisting of implications. This example has been extensively discussed in the literature [18, 23] and so highlights the differences between approaches. This example again uses explicit context variables and parameter variables. The Hilbert-style axiomatization can be formalized as follows:

```
hil : o → type .

k : hil (A imp (B imp A)).
s : hil ((A imp (B imp C)) imp
         ((A imp B) imp (A imp C))).
mp : hil (A imp B) → hil A → hil B.
```

We omit leading Πs from the types when they can be reconstructed, as is common practice in Twelf [19].

The bracket abstraction lemma states that a derivation $\mathcal{D}$ of hil B that depends on the hypothesis x:hil A can be translated into a derivation $\mathcal{E}$ of hil (A imp B) that does not refer to the hypothesis x:

LEMMA (BRACKET ABSTRACTION)
If $\mathcal{D} :: \Gamma, x{:}\mathtt{hil}\ A \vdash \mathtt{hil}\ B$ *then* $\mathcal{E} : \Gamma \vdash \mathtt{hil}\,(A\ \mathtt{imp}\ B)$

The context Γ has the form $x_n{:}\mathtt{hil}\ A_n, \ldots, x_1{:}\mathtt{hil}\ A_1$, and the proof follows by induction on the derivation $\mathcal{D}$. There are four base cases. Two arise from the constants k and s, but we concentrate on the other two cases, which arise from using an assumption in $\Gamma, x{:}\mathtt{hil}\ A$: we could have used the assumption $x{:}\mathtt{hil}\ A$, or an assumption $x_i{:}A_i$ from Γ, to prove hil B.

Explicit contexts allow us to easily characterize the lemma as a dependent type, and implement the inductive proof as a recursive program using pattern matching. Moreover, every case in the informal proof directly corresponds to a case in our recursive program. As a first step, we characterize the context Γ with a context schema:

```
ctx schema hilCtx = (all A:o. hil A)*.
```

This schema describes contexts $x_n{:}\mathtt{hil}\ A_n, \ldots, x_1{:}\mathtt{hil}\ A_1$. Next, we represent the lemma as a dependent type. The recursive function that realizes this type describes the proof transformation on Hilbert derivations. Implicit data objects that characterize index objects occurring in a dependent type are introduced with $\Pi^\square$, but we omit these implicit type arguments from computations and from data for readability, since we expect them to be reconstructed in practice.

```
rec ded: Π γ:hilCtx.Π□A:o[γ] Π□B:o[γ].
       (hil B[id_γ])[γ, x:hil A[id_γ]]
    → (hil (A[id_γ] imp B[id_γ]))[γ] =
 Λ γ ⇒ fn D ⇒
 case D of
   box(γ, x. k) ⇒ box(γ. mp k k)
 | box(γ, x. s) ⇒ box(γ. mp k s)
 | box(γ, x. x) ⇒ box(γ. mp (mp s k) k)
 | box(γ, x. p[id_γ]) ⇒ box(γ. mp k p[id_γ])
 | box(γ, x. mp D1[id_γ, x] D2[id_γ, x]) ⇒
   let
     box(γ. E1[id_γ]) = ded ⌈γ⌉ box(γ,x. D1[id_γ,x])
     box(γ. E2[id_γ]) = ded ⌈γ⌉ box(γ,x. D2[id_γ,x])
   in
     box(γ. mp (mp s E1[id_γ]) E2[id_γ])
   end
```

We analyze Hilbert-style derivations by pattern matching on D, which describes an object of type hil B[id_γ] in the context (γ, x:hil A[id_γ]). Intuitively, we can construct all derivations of the appropriate type by either using one of the constructors or an element from the context. This gives rise to the first two cases: either we have used the specific assumption x, or we have used one of the assumptions in γ. To describe this last generic variable case, we again use a parameter variable.

The other base cases are straightforward, so we concentrate on the last case where we use modus ponens. To pattern match on derivations constructed using the constant mp, we describe the subderivations using meta-variables $\mathtt{D}_1[\mathsf{id}_\gamma]$ and $\mathtt{D}_2[\mathsf{id}_\gamma]$ for which higher-order pattern matching will find appropriate instantiations during runtime. The appeal to the induction hypothesis corresponds to the recursive call to the function ded. We first pass to this function the context γ by context application, and then the derivation described by box(γ,x. $\mathtt{D}_1[\mathsf{id}_\gamma$, x]).

2.3 Summary of key ideas

We summarize here the five key ideas underlying our work: First, we separate data from computations via the modality box. Second, every data object is closed with respect to a local context. For example, box(x. eq x z) denotes an object of type o[x:nat], i.e. a proposition that may contain the variable x. The box-construct introduces the bound variables x. Third, we allow context variables ψ and abstract over them on the computation level. This is necessary since the concrete bound variables occurring in a data object are only exposed when we recursively traverse a binder, and the context describing these variables may grow. More importantly, this allows us to program with explicit contexts and capture more invariants. Fourth, we support pattern matching on higher-order data using meta-variables and, importantly, parameter variables. While meta-variables allow us to deconstruct arbitrary objects with binders, parameter variables allow us to manipulate names of bound variables directly in computation. Finally, we support dependent types using the type $\Pi^\square u{::}A[\Psi].\tau$, and we clearly distinguish implicit data objects occurring as index objects in data-level types from explicit data objects that can be analyzed by pattern matching.

3. Data-level terms and contexts

We begin by describing the data layer of our dependently-typed intermediate language. We support the full logical framework LF together with Σ-types. Our data layer closely follows contextual modal type theory [15], extended with parameter variables and context variables [21], and finally with dependent pairs and projections. The syntax is given in Figure 1. We only characterize normal terms since only these are meaningful in the logical framework, following Watkins et al. [28] and Nanevski et al. [15]. This is achieved by distinguishing between normal terms M and neutral terms R. While

Kinds	K	::=	$\mathsf{type} \mid \Pi x{:}A.K$
Atomic types	P	::=	$a\ M_1 \ldots M_n$
Types	A, B	::=	$P \mid \Pi x{:}A.B \mid \Sigma x{:}A.B$
Normal terms	M, N	::=	$\lambda x.\, M \mid (M, N) \mid R$
Neutral terms	R	::=	$c \mid x \mid u[\sigma] \mid p[\sigma] \mid R\ N \mid \mathsf{proj}_k R$
Substitutions	σ, ρ	::=	$\cdot \mid \sigma\,;\, M \mid \sigma\,,\, R \mid \mathsf{id}_\psi$

Figure 1. Data-level syntax

the syntax only guarantees that terms N contain no β-redexes, the typing rules will also guarantee that all well-typed terms are fully η-expanded.

We distinguish between four kinds of variables in our theory: *Ordinary bound variables* are used to represent data-level binders and are bound by λ-abstraction. *Contextual variables* stand for open objects, and include *meta-variables* u that represent general open objects and *parameter variables* p that can only be instantiated with an ordinary bound variable.[1] Contextual variables are introduced in computation-level case expressions, and can be instantiated via pattern matching. They are associated with a postponed substitution σ thereby representing a closure. This substitution precisely characterizes the dependencies we allow when instantiating the meta-variable with an open term. Our intention is to apply σ as soon as we know which term the contextual variable should stand for. The domain of σ thus describes the free variables of the object the contextual variable stands for, and the type system statically guarantees this.

Substitutions σ are built of either normal terms (in $\sigma\,;\, M$) or atomic terms (in $\sigma\,,\, R$). The two forms are necessary since not every neutral term is also a normal term. Only neutral terms of atomic type are in fact normal. We do not make the domain of the substitutions explicit, which simplifies the theoretical development and avoids having to rename the domain of a given substitution σ. We also require a first-class notion of identity substitution id_ψ. Data-level substitutions, as defined operations on data-level terms, are written $[\sigma]N$.

While contextual variables are declared in a meta-level context Δ, ordinary bound variables are declared in a data-level context Ψ. Our foundation supports *context variables* ψ which allow us to reason abstractly with contexts. Context variables are declared in Ω. Unlike previous uses of context variables [12], a context may contain at most one context variable. In the same way that types classify objects, and kinds classify types, we introduce the notion of a schema W that classifies contexts Ψ.

Context variables	ψ, ϕ		
Contexts	Ψ, Φ	::=	$\cdot \mid \psi \mid \Psi, x{:}A$
Meta-contexts	Δ	::=	$\cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, p{::}A[\Psi]$
Schema contexts	Ω	::=	$\cdot \mid \Omega, \psi{::}W$
Element types	$\widetilde{A}$	::=	$\Pi x{:}A.\widetilde{A} \mid a\ N_1 \ldots N_n$
Schema elements	F	::=	$\mathsf{all}\, x_1{:}\widetilde{B_1}, \ldots, x_n{:}\widetilde{B_n}.$ $\Sigma y_1{:}\widetilde{A_1}, \ldots, y_k{:}\widetilde{A_k}.\widetilde{A}$
Schemas	W	::=	$(F_1 + \cdots + F_n)^*$

As the earlier example illustrated, contexts play an important part in programming with open data objects, and contexts, which are explicitly constructed at the computation level, belong to a specific context schema. For example, the schema $(\mathtt{nat})^*$ represented contexts of the form $\mathtt{x_1{:}nat,\ \ldots,\ x_n{:}nat}$, and the schema $\mathsf{all}\,\mathtt{A{:}o.hil\ A}$ characterized the context Γ consisting of assumptions $x{:}\mathtt{hil\ A}$. In general, we allow an even richer form of contexts to be captured. Consider how the schema of the context Γ will change when we extend the Hilbert-style calculus with universal quantifiers. In this case, we must also consider assumptions about possible parameters introduced when we traverse a universal quantifier. On paper, we may write the following to characterize this context:

$$\text{Hilbert Contexts}\ \ \Gamma \quad ::= \quad \cdot \mid \Gamma, a{:}\mathtt{nat} \mid \Gamma, x{:}\mathtt{hil}\ A$$

We provide $+$ to denote a choice of possible elements in a context. Hence we would modify our schema for Hilbert contexts as follows:

```
ctx schema hilCtx = nat + all A:o. hil A.
```

Context schemas are built of elements $F_1, \ldots F_n$, each of which must have the form $\mathsf{all}\,\widetilde{\Phi}.\Sigma\widetilde{\Psi}.\,\widetilde{A}$ where $\widetilde{\Phi}$ and $\widetilde{\Psi}$ are Σ-free contexts, i.e. $x_1{:}\widetilde{B_1}, \ldots x_k{:}\widetilde{B_k}$. In other words, the element is of $\Sigma\Pi$-type, where we first introduce some Σ-types, followed by pure Π-types. We disallow arbitrary mixing of Σ and Π. This restriction makes it easier to describe the possible terms of this type, which is a crucial step towards ensuring coverage [6].

Schemas resemble Schürmann's worlds [24], but while similar in spirit, we use dependent pairs to express the relationship between multiple objects in a context. While worlds impose a similar $\Sigma\Pi$-structure, schemas differ from worlds in the sense that schemas are pure. They only keep track of assumptions about binders occurring in a data object of type A. This is unlike worlds as realized in Twelf [19], which also track dynamic computation-level extensions.

3.1 Data-level typing

In this section, we present a bidirectional type system for data-level terms. We assume that type constants and object constants are declared in a signature Σ, which we suppress since it is the same throughout a typing derivation. However, we will keep in mind that all typing judgments have access to a well-formed signature. Typing is defined via the following judgments:

$\Omega; \Delta; \Psi \vdash M \Leftarrow A$	Check normal object M against A
$\Omega; \Delta; \Psi \vdash R \Rightarrow A$	Synthesize A for neutral object R
$\Omega; \Delta; \Phi \vdash \sigma \Leftarrow \Psi$	Check σ against context Ψ

For readability, we omit Ω in the subsequent development since it is constant; we also assume that Δ and Ψ are well-formed. First, we show in Figure 2 the typing rules for objects.

We assume that data level type constants a together with constants c have been declared in a signature. 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, not on modal variables u. We also require the usual conditions on bound variables. For example, in the rule $\underline{\Pi\Pi}$ the bound variable x must be new and cannot already occur in the context Ψ. This can always be achieved via α-renaming. The typing rules for data-level neutral terms rely on *hereditary substitutions* that preserve canonical forms [15, 28].

The idea is to define a primitive recursive functional that always returns a canonical object. In places where the ordinary substitution would construct a redex $(\lambda y.\, M)\, N$ we must continue, substituting N for y in M. Since this could again create a redex, we must continue and hereditarily substitute and eliminate potential redexes. Hereditary substitution can be defined recursively, considering both the structure of the term to which the substitution is applied and the type of the object being substituted. Hence we annotate the substitution with the type of the argument being substituted. We also indicate with the superscript a that the substitution is applied to a type. We give the definition of hereditary substitutions in the appendix.

[1] Prior work also considered substitution variables. Although our theory extends to substitution variables, we omit them here to make the presentation compact and focus on aspects related to dependent types.

Data-level normal terms

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

$$\frac{\Delta; \Psi \vdash M_1 \Leftarrow A_1 \qquad \Delta; \Psi \vdash M_2 \Leftarrow [M_1/x]^a_{A_1} A_2}{\Delta; \Psi \vdash (M_1, M_2) \Leftarrow \Sigma x{:}A_1.A_2}\ \Sigma\mathrm{I}$$

$$\frac{\Delta; \Psi \vdash R \Rightarrow P' \qquad P' = P}{\Delta; \Psi \vdash R \Leftarrow P}\ \mathsf{turn}$$

Data-level neutral terms

$$\frac{x{:}A \in \Psi}{\Delta; \Psi \vdash x \Rightarrow A}\ \mathsf{var} \qquad \frac{c{:}A \in \Sigma}{\Delta; \Psi \vdash c \Rightarrow A}\ \mathsf{con}$$

$$\frac{u{::}A[\Phi] \in \Delta \qquad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]^a_\Phi A}\ \mathsf{mvar}$$

$$\frac{p{::}A[\Phi] \in \Delta \qquad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash p[\sigma] \Rightarrow [\sigma]^a_\Phi A}\ \mathsf{param}$$

$$\frac{\Delta; \Psi \vdash R \Rightarrow \Pi x{:}A.B \qquad \Delta; \Psi \vdash N \Leftarrow A}{\Delta; \Psi \vdash R\ N \Rightarrow [N/x]^a_A B}\ \Pi\mathrm{E}$$

$$\frac{\Delta; \Psi \vdash R \Rightarrow \Sigma x{:}A_1.A_2}{\Delta; \Psi \vdash \mathsf{proj}_1 R \Rightarrow A_1}\ \Sigma\mathrm{E}_1$$

$$\frac{\Delta; \Psi \vdash R \Rightarrow \Sigma x{:}A_1.A_2}{\Delta; \Psi \vdash \mathsf{proj}_2 R \Rightarrow [\mathsf{proj}_1 R/x]^a_{A_1} A_2}\ \Sigma\mathrm{E}_2$$

Data-level substitutions

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

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \qquad \Delta; \Psi \vdash R \Rightarrow A' \qquad [\sigma]^a_\Phi A = A'}{\Delta; \Psi \vdash (\sigma\ ,\ R) \Leftarrow (\Phi, x{:}A)}$$

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

Figure 2. Data-level typing for normal terms, neutral terms, and substitutions

Finally, we define context checking.

Context Ψ checks against a schema $W : \Omega \vdash \Psi \Leftarrow W$

for some k

$$\frac{\Omega; \Delta; \Psi \vdash B \in F_k \qquad \Omega; \Delta \vdash \Psi \Leftarrow (F_1 + \cdots + F_n)^*}{\Omega; \Delta \vdash \Psi, x{:}B \Leftarrow (F_1 + \cdots + F_n)^*}$$

$$\frac{\psi{::}W \in \Omega}{\Omega; \Delta \vdash \psi \Leftarrow W} \qquad \frac{}{\Omega; \Delta \vdash \cdot \Leftarrow W}$$

To verify that a context Ψ belongs to a schema $W = (F_1 + \cdots + F_n)^*$, we check that for every declaration B_i in the context Ψ, there exists a k s.t. B_i is an instance of a schema element $F_k = \mathsf{all}\ \widetilde{\Phi}.\Sigma y_1{:}\widetilde{A}_1, \ldots, y_j{:}\widetilde{A}_j.\ \widetilde{A}$. This means we must find an instantiation σ for all the variables in $\widetilde{\Phi}$ s.t. $[\sigma]^a_{\widetilde{\Phi}}(\Sigma y_1{:}\widetilde{A}_1, \ldots, y_j{:}\widetilde{A}_j.\widetilde{A})$ is equal to B_i. This is done in practice by higher-order pattern matching.

Theorem 1 (Decidability of Typechecking)**.**
All judgments in the contextual modal type theory are decidable.

Proof. The typing judgments are syntax-directed and therefore clearly decidable assuming hereditary substitution is decidable. □

3.2 Substitution operations

The different variables (ordinary variables x, context variables ψ, and contextual variables $u[\sigma]$ and $p[\sigma]$) give rise to different substitution operations. We already remarked on the hereditary substitution operation for ordinary variables x and we give its definition in the appendix. The remaining substitution operations do not require any significant changes from earlier work [15, 21] to handle dependent types and we revisit them in this section.

3.2.1 Substitution for context variables

If we encounter a context variable ψ, we simply replace it with the context Ψ. This is possible because context variables occur only in leftmost position. When we substitute some context Ψ for ψ in the context (ψ, Φ), the context Ψ cannot depend on Φ. This would not hold if we allowed contexts of the form (Φ, ψ).

Data-level context

$$\begin{array}{lll} [\![\Psi/\psi]\!](\cdot) & = \cdot & \\ [\![\Psi/\psi]\!](\Phi, x{:}A) & = (\Phi', x{:}A') & \text{if } x \notin \mathsf{V}(\Phi') \text{ and } [\![\Psi/\psi]\!]A = A' \\ & & \text{and } [\![\Psi/\psi]\!]\Phi = \Phi' \\ [\![\Psi/\psi]\!](\psi) & = \Psi & \\ [\![\Psi/\psi]\!](\phi) & = \phi & \text{if } \phi \neq \psi \end{array}$$

When we apply the substitution $[\![\Psi/\psi]\!]$ to the context $\Phi, x{:}A$, we apply the substitution to the type A, yielding some new type A', and to the context Φ, yielding some new context Φ'. Applying the substitution to the type A is necessary in the dependently-typed setting, since A may contain terms and in particular identity substitutions id_ψ. When we replace ψ with Ψ in the substitution id_ψ, we unfold the identity substitution. Expansion of the identity substitution is defined by the operation $\mathsf{id}(\Psi)$ for valid contexts Ψ:

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

Lemma 1 (Unfolding identity substitution)**.**
If $\mathsf{id}(\Psi) = \sigma$ *then* $\Delta; \Psi, \Psi' \vdash \sigma \Leftarrow \Psi$.

Proof. By induction on the structure of Ψ. □

When we combine Φ' and the declaration $x{:}A'$ to yield a new context, we must ensure that x is not already declared in Φ'. This can always be achieved by appropriately renaming bound variable occurrences. We write $\mathsf{V}(\Phi')$ for the set of variables declared in Φ'. The rest of the definition is mostly straightforward.

Theorem 2 (Substitution for context variables)**.**
If $\Omega, \psi{::}W, \Omega'; \Delta; \Phi \vdash J$ *and* $\Omega \vdash \Psi \Leftarrow W$
then $\Omega, \Omega'; [\![\Psi/\psi]\!]\Delta; [\![\Psi/\psi]\!](\Phi) \vdash [\![\Psi/\psi]\!]J$.

Proof. By induction on the first derivation using Lemma 1. □

3.2.2 Contextual substitution for contextual variables

Substitution for contextual variables is a little more difficult, but is essentially similar to Pientka [21]. We can think of $u[\sigma]$ as a closure where, as soon as we know which term u should stand for, we can apply σ to it. The typing will ensure that the type of M and the type of u agree, i.e. we can replace u of type $A[\Psi]$ with a normal term M if M has type A in the context Ψ. Because of α-conversion, the variables substituted at different occurrences of u may differ, and we write the contextual substitution as $[\![\hat{\Psi}.M/u]\!](N)$, $[\![\hat{\Psi}.M/u]\!](R)$, and $[\![\hat{\Psi}.M/u]\!](\sigma)$, where $\hat{\Psi}$ binds

all free variables in M. Applying $[\![\hat{\Psi}.M/u]\!]$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\hat{\Psi}.M/u]\!]\sigma$, but instead of returning $M[\sigma']$, it eagerly applies σ' to M. Similar ideas apply to parameter substitutions, which are written as $[\![\hat{\Psi}.x/p]\!](M)$, $[\![\hat{\Psi}.x/p]\!](R)$, and $[\![\hat{\Psi}.x/p]\!](\sigma)$. Parameter substitution could not be achieved with the previous definition of contextual substitution for meta-variables, since it only allows us to substitute a normal term for a meta-variable, and x is only normal if it is of atomic type. We give its definition in the appendix.

Finally, we will rely in the subsequent development on simultaneous contextual substitutions, built of either meta-variables, $(\theta, \hat{\Psi}.M/u)$, or parameter variables, $(\theta, \hat{\Psi}.x/p)$. The judgment $\Delta \vdash \theta \Leftarrow \Delta'$ checks that the contextual substitution θ maps contextual variables from Δ' to the contextual variables in Δ.

Simultaneous contextual substitution

$$\frac{}{\Delta \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta \vdash \theta \Leftarrow \Delta' \quad \Delta; [\![\theta]\!]\Psi \vdash M \Leftarrow [\![\theta]\!]A}{\Delta \vdash (\theta, \hat{\Psi}.M/u) \Leftarrow \Delta', u{::}A[\Psi]}$$

$$\frac{\Delta \vdash \theta \Leftarrow \Delta' \quad \Delta; [\![\theta]\!]\Psi \vdash x \Rightarrow A' \quad A' = [\![\theta]\!]A}{\Delta \vdash (\theta, \hat{\Psi}.x/p) \Leftarrow \Delta', p{::}A[\Psi]}$$

4. Computation-level expressions

We cleanly separate the data level from the computation level (Figure 3), which describes programs operating on data. Computations analyze data of type $A[\Psi]$, which denotes an object of type A that may contain the variables specified in Ψ. To allow quantification over context variables ψ, we introduce the type $\Pi\psi{:}W.\tau$ and context abstraction $\Lambda\psi.e$. We write $\rightarrow$ for computation-level functions.

As mentioned earlier, the language we describe in this section may be thought of as an internal language into which source level programs will be compiled. The extension to dependent types is novel in that we clearly distinguish between (1) *implicit data objects* which occur in a type as index and are kept implicit in the source program and (2) *explicit data objects* which are analyzed recursively by pattern matching. The intuition is that implicit arguments can be reconstructed when translating source programs to our intermediate representation. We introduce abstraction over implicit data objects occurring in a dependent type A on the computation level using the dependent type $\Pi^{\square} u{::}A[\Psi].\tau$.

Moreover, we require that patterns occurring in the branches of case expressions are annotated with types, since the type of each pattern need not be identical to the type of the expression being case analyzed in the dependently-typed setting. We expect that these annotations can be reconstructed during the compilation from source programs to this intermediate language.

We will enforce that all context variables are bound by Λ-abstractions. To support α-renaming of ordinary bound variables, we write $\mathsf{box}(\hat{\Psi}.M)$ where $\hat{\Psi}$ is a list of variables that can possibly occur in M. Index objects of dependent types are introduced by $\lambda^{\square} u.\, e$ of type $\Pi^{\square} u{::}A[\Psi].\tau$. In other words, index objects are characterized by contextual variables. $i\ \lceil\hat{\Psi}.M\rceil$ describes the application of an index argument to an expression. The contextual variables in branches b, declared in Δ, are instantiated using higher-order pattern matching. We only consider patterns à la Miller [14] where meta-variables that are subject to instantiation must be applied to a distinct set of bound variables. In our setting, this means all contextual variables must be associated with a substitution such as $x_{\Phi(1)}/x_1, \ldots, x_{\Phi(n)}/x_n$. This fragment is decidable and has efficient algorithms for pattern matching [20, 22].

Types	$\tau ::= A[\Psi] \mid \tau_1 \rightarrow \tau_2 \mid \Pi\psi{::}W.\tau \mid \Pi^{\square} u{::}A[\Psi].\tau$
Expressions (checked)	$e ::= i \mid \mathsf{rec}\ f.e \mid \mathsf{fn}\ y.e \mid \Lambda\psi.e \mid \lambda^{\square} u.\, e$ $\mid \mathsf{box}(\hat{\Psi}.\, M) \mid \mathsf{case}\ i\ \mathsf{of}\ bs$
Expressions (synth.)	$i ::= y \mid i\ e \mid i\ \lceil\Psi\rceil \mid i\ \lceil\hat{\Psi}.M\rceil \mid (e : \tau)$
Branch	$b ::= \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \mapsto e$
Branches	$bs ::= \cdot \mid (b \mid bs)$
Contexts	$\Gamma ::= \cdot \mid \Gamma, y{:}\tau$

Figure 3. Computation-level syntax

Patterns in case expressions are annotated with their types, since in the dependently-typed setting, the type of each pattern need not be identical to the type of the expression being analyzed.

4.1 Computation-level typing

We describe computation-level typing using the following judgments:

$\Omega; \Delta; \Gamma \vdash e \Leftarrow \tau$	e checks against τ
$\Omega; \Delta; \Gamma \vdash i \Rightarrow \tau$	i synthesizes τ
$\Omega; \Delta; \Gamma \vdash b \Leftarrow_{\tau'} \tau$	branch b checks against τ, when case-analyzing a τ'

The rules are given in Figure 4. The names of computation-level rules are written with a line above to distinguish them from the names of data-level rules, which are underlined. There are two interesting rules. The first is $\overline{\mathsf{turn}}$. In this rule we check that the computation-level types τ and τ' are equal. At the data level, we only characterize canonical forms, so equality between two dependent types A and A' is simply syntactic equality. Hence, equality between two computation-level types is also just syntactic equality. This is in stark contrast to dependently-typed languages such as Cayenne [1] and Epigram [11] that allow computations within the index objects of dependent types. In these systems, we cannot simply compare two types syntactically, but must evaluate the index arguments first, before comparing the final values. Even weaker dependently-typed systems such as DML [29], where types are indexed by a decidable constraint domain such as integers with linear inequalities, need efficient constraint solvers to decide type equality.

The second interesting rule in the bidirectional type checking algorithm is the one for branches, since the type $A_k[\Psi_k]$ of each of the patterns must be considered equal to the type $A[\Psi]$, the type of the expression we analyze by cases. In some approaches to dependently typed programming [29], branches are checked under certain equality constraints. Instead we propose here to solve these constraints eagerly using higher-order pattern unification [14]. Hence, we restrict the contextual variables $u[\sigma]$ that occur in patterns to be higher-order patterns, i.e. the substitution σ is simply a renaming substitution. Higher-order pattern unification is decidable, so unification of the contexts and types is decidable. For a formal description of a higher-order pattern unification algorithm for contextual variables, see Pientka [20].

We use the judgment $\Omega; \Delta \vdash A \doteq A_k \ / \ (\theta, \Delta')$ to describe higher-order pattern unification for types. Δ describes all the meta-variables occurring in types A and A_k. The result of unifying A and A_k is a contextual substitution θ that maps the meta-variables in Δ to the meta-variables in Δ' s.t. $[\![\theta]\!]A$ is equal to $[\![\theta]\!]A_k$. This operation can be extended to unify contexts Ψ and Ψ_k.

Unlike the simply-typed setting, where it is natural to require linearity, i.e. that every contextual variable in a pattern occurs only once, we cannot enforce a similar condition in the dependently-

Expression e checks against type τ

$$\frac{\Omega; \Delta; \Gamma, f{:}\tau \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \mathsf{rec}\ f.e \Leftarrow \tau}\ \overline{\mathsf{rec}} \qquad \frac{\Omega, \psi{:}W; \Delta; \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \Lambda\psi.e \Leftarrow \Pi\psi{:}W.\tau}\ \overline{\Pi\mathrm{I}} \qquad \frac{\Omega; \Delta; \Gamma, y{:}\tau_1 \vdash e \Leftarrow \tau_2}{\Omega; \Delta; \Gamma \vdash \mathsf{fn}\ y.e \Leftarrow \tau_1 \to \tau_2}\ \overline{\to\mathrm{I}}$$

$$\frac{\Omega; \Delta, u{::}A[\Psi]; \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \lambda^{\square} u.\, e \Leftarrow \Pi^{\square} u{::}A[\Psi].\tau}\ \overline{\Pi^{\square}\mathrm{I}} \qquad \frac{\Omega; \Delta; \Psi \vdash M \Leftarrow A}{\Omega; \Delta; \Gamma \vdash \mathsf{box}(\hat{\Psi}.\, M) \Leftarrow A[\Psi]}\ \overline{\mathsf{box}}$$

$$\frac{\Omega; \Delta; \Gamma \vdash i \Rightarrow A[\Psi] \qquad \text{for all } k\ \Omega; \Delta; \Gamma \vdash b_k \Leftarrow_{A[\Psi]} \tau}{\Omega; \Delta; \Gamma \vdash \mathsf{case}\ i\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \Leftarrow \tau}\ \overline{\mathsf{case}} \qquad \frac{\Delta; \Gamma \vdash i \Rightarrow \tau' \qquad \Omega; \Delta \vdash \tau' = \tau}{\Omega; \Delta; \Gamma \vdash i \Leftarrow \tau}\ \overline{\mathsf{turn}}$$

Expression i synthesizes type τ

$$\frac{\Omega; \Delta; \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash (e : \tau) \Rightarrow \tau}\ \overline{\mathsf{anno}} \qquad \frac{y{:}\tau \in \Gamma}{\Omega; \Delta; \Gamma \vdash y \Rightarrow \tau}\ \overline{\mathsf{var}} \qquad \frac{\Omega; \Delta; \Gamma \vdash i \Rightarrow \tau_2 \to \tau \qquad \Omega; \Delta; \Gamma \vdash e \Leftarrow \tau_2}{\Omega; \Delta; \Gamma \vdash i\ e \Rightarrow \tau}\ \overline{\to\mathrm{E}}$$

$$\frac{\Omega; \Delta; \Gamma \vdash i \Rightarrow \Pi\psi{:}W.\tau \qquad \Omega; \Delta \vdash \Psi \Leftarrow W}{\Omega; \Delta; \Gamma \vdash i\ \lceil\Psi\rceil \Rightarrow [\![\Psi/\psi]\!]\tau}\ \overline{\Pi\mathrm{E}} \qquad \frac{\Omega; \Delta; \Gamma \vdash i \Rightarrow \Pi^{\square} u{::}A[\Psi].\tau \qquad \Omega; \Delta; \Psi \vdash M \Leftarrow A}{\Omega; \Delta; \Gamma \vdash i\ \lceil\hat{\Psi}.M\rceil \Rightarrow [\![\hat{\Psi}.M/u]\!]\tau}\ \overline{\Pi^{\square}\mathrm{E}}$$

Body e_k checks against type τ, assuming the value cased upon has type $A[\Psi]$

$$\frac{\begin{array}{lll} & \Omega; \Delta, \Delta_k & \vdash \Psi \doteq \Psi_k/(\theta_1, \Delta') \\ \Omega; \Delta_k; \Psi_k \vdash M_k \Leftarrow A_k & \Omega; \Delta' & \vdash [\![\theta_1]\!]A \doteq [\![\theta_1]\!]A_k \,/\, (\theta_2, \Delta'') \\ & \Omega; \Delta''; [\![\theta_2]\!][\![\theta_1]\!]\Gamma \vdash [\![\theta_2]\!][\![\theta_1]\!]e_k \Leftarrow [\![\theta_2]\!][\![\theta_1]\!]\tau \end{array}}{\Omega; \Delta; \Gamma \vdash \Pi\Delta_k.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi_k] \mapsto e_k \Leftarrow_{A[\Psi]} \tau}$$

Figure 4. Computation-level typing rules

typed setting, because the object may become ill-typed. To illustrate, consider pattern matching against the Hilbert derivation $\mathsf{box}(\psi.\ \mathtt{mp\ k\ D[id}_\psi\mathtt{]})$. After reconstructing the omitted implicit arguments, we get

$$\mathsf{box}(\psi.\ \mathtt{mp\ A[id}_\psi\mathtt{]\ (A[id}_\psi\mathtt{]\ imp\ B[id}_\psi\mathtt{])\ k\ D[id}_\psi\mathtt{]})$$

It is clear that enforcing linearity in dependently-typed patterns is impossible.

Let us now return to the rule for branches. Branches b have the form $\Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) \mapsto e$ where Δ contains all the contextual variables introduced and occurring in the guard $\mathsf{box}(\hat{\Psi}.\, M)$. We concentrate here on the last rule for checking the pattern in a case expression. After typing the pattern in the case expression, we unify the type of the subject of the case expression with the type of the pattern. First, however, we must unify the context Ψ with the context Ψ_i of the pattern. Finally, we apply the result of higher-order unification θ to the body e and check its type.

This approach will simplify the preservation and progress proof. It is also closer to a realistic implementation, which would support early failure and indicate the offending branch.

Theorem 3 (Decidability of Typechecking)**.**
Computation-level typechecking is decidable.

Proof. The typing rules are syntax-directed. Computation-level types are in canonical form, so the equality in rule $\overline{\mathsf{turn}}$ is syntactic equality. Moreover, higher-order pattern matching is decidable. Thus, typechecking is decidable. □

Finally, we briefly remark on computation-level substitutions. There are four varieties: $[e/x]e'$ describes the substitution of the computation-level expression e for x in the computation-level expression e'. This operation is straightforward and capture-avoiding in the case of functions $\mathsf{fn}\ y.e$. It does not affect data-level terms, since the computation-level variable x cannot occur in them.

The operation $[\![\Psi/\psi]\!](e)$ extends the previous definition of context substitution to the computation level. It is capture-avoiding in the case for context abstraction, and is propagated to the data level.

The last two varieties, $[\![\theta]\!](e)$ and $[\![\hat{\Psi}.M/u]\!](e)$, extend the previous contextual substitution operation to the computation level in a straightforward manner. We ensure it is capture-avoiding in $\lambda^{\square}$-abstraction and in branches of case expressions.

4.2 Operational semantics

Next, we define a small-step evaluation judgment:

$$e \text{ evaluates in one step to } e' \qquad e \longrightarrow e'$$

$$\text{Branch } b \text{ matches } \mathsf{box}(\hat{\Psi}.\, M) \text{ and steps to } e' \qquad (\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \doteq b) \longrightarrow e'$$

In the presence of full LF we cannot erase type information completely during evaluation, since not all implicit type arguments can be uniquely determined. In the simply-typed setting, the patterns in the branches of a case expression must have the same type as the case expression's subject. However, with dependent typing, the patterns may have different types. Hence, we first match against the pattern's type before matching against the pattern itself, to ensure that the types of the subject and pattern agree. Thus, we must translate the computational language into one where type annotations of the form $(e : \tau)$ are erased but all expressions of type $A[\Psi]$, in particular patterns in branches, carry their corresponding type. We denote this translation by $|e|$ and $|i|$ for checked and synthesizing expressions, respectively[2].

Otherwise, the semantics is straightforward. In function application, values for program variables are propagated by computation-level substitution. Instantiations for context variables are propa-

[2] Technically, this erasure is type-directed, and adds an annotation for the object $\hat{\Psi}.M$ in $i\ \lceil\hat{\Psi}.M\rceil$ with its corresponding type. This is necessary since contextual substitution is defined recursively on the structure of this type (see the appendix). For conciseness we omit this type here.

Evaluation of computation:

$$\frac{}{\mathsf{rec}\ f.e \longrightarrow [\mathsf{rec}\ f.e/f]e} \quad \frac{}{(\mathsf{fn}\ y.e)\ v \longrightarrow [v/y]e} \quad \frac{}{(\lambda^{\square} u.\, e)\ulcorner \hat{\Psi}.M \urcorner \longrightarrow [\![\hat{\Psi}.M/u]\!]e}$$

$$\frac{i_1 \longrightarrow i_1'}{i_1\ e_2 \longrightarrow i_1'\ e_2} \quad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} \quad \frac{i \longrightarrow i'}{i\ulcorner \Psi \urcorner \longrightarrow i'\ulcorner \Psi \urcorner} \quad \frac{}{(\Lambda\psi.e)\ulcorner \Psi \urcorner \longrightarrow [\![\Psi/\psi]\!]e} \quad \frac{i \longrightarrow i'}{\mathsf{case}\ i\ \mathsf{of}\ bs \longrightarrow \mathsf{case}\ i'\ \mathsf{of}\ bs}$$

$$\frac{(\mathsf{box}(\hat{\Psi}.\, M){:}A[\Psi]) \not\doteq (\Pi\Delta_1.\mathsf{box}(\hat{\Psi}.\, M_1){:}A_1[\Psi_1]) \qquad b = \Pi\Delta_1.\mathsf{box}(\hat{\Psi}.\, M_1){:}A_1[\Psi_1] \mapsto e_1}{(\mathsf{case}\ (\mathsf{box}(\hat{\Psi}.\, M){:}A[\Psi])\ \mathsf{of}\ b \mid bs) \longrightarrow \mathsf{case}\ (\mathsf{box}(\hat{\Psi}.\, M){:}A[\Psi])\ \mathsf{of}\ bs}$$

$$\frac{(\mathsf{box}(\hat{\Psi}.\, M){:}A[\Psi]) \doteq (\Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_1){:}A_1[\Psi_1])\ /\ \theta \qquad b = \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_1){:}A_1[\Psi_1] \mapsto e_1}{(\mathsf{case}\ (\mathsf{box}(\hat{\Psi}.\, M){:}A[\Psi])\ \mathsf{of}\ b \mid bs) \longrightarrow [\![\theta]\!]e_1}$$

Evaluation of branch:

$$\frac{\Delta \vdash \Psi_k \doteq \Psi\ /\ (\theta_1, \Delta_1) \qquad \Delta_1; \Psi \vdash A \doteq [\![\theta_1]\!]A_k\ /\ (\theta_2, \Delta_2) \qquad \theta = [\![\theta_2]\!]\theta_1 \qquad \Delta_2; \Psi \vdash M \doteq [\![\theta]\!]M_k\ /\ (\theta_3, \cdot)}{(\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \doteq \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi_k])\ /\ [\![\theta_3]\!]\theta}$$

Figure 5. Operational semantics

gated by applying a concrete context Ψ to a context abstraction $\Lambda\psi.e$. Index arguments are propagated in $(\lambda^{\square} u.\, e)\ulcorner \hat{\Psi}.M \urcorner$ by replacing u with the concrete data object $\hat{\Psi}.M$.

Evaluation in branches relies on higher-order pattern matching against data-level terms to instantiate the contextual variables occurring in a branch. Data-level instantiations are propagated via simultaneous contextual substitution. We write $\mathsf{box}(\hat{\Psi}.\, M){:}A[\Psi] \not\doteq \Pi\Delta_1.\mathsf{box}(\hat{\Psi}.\, M_1){:}A_1[\Psi_1]$ to mean that higher-order pattern matching between $\mathsf{box}(\hat{\Psi}.\, M_1)$ and $\mathsf{box}(\hat{\Psi}.\, M)$ failed, i.e. there exists no instantiation for the contextual variables in Δ_1 that makes these terms equal.

We assume that $\mathsf{box}(\Psi.\, M)$ does not contain any meta-variables, i.e. it is closed, and that its type $A[\Psi]$ is known. Because of dependent types in Ψ_k we must first match Ψ against Ψ_k, and then proceed to match M against M_k.

Before we prove type safety for our dependently-typed functional language with higher-order abstract syntax, we briefly discuss the issue of coverage. To prove progress, we must check that the set of patterns in a case expressions covers all possible values. In the first-order setting, this is straightforward. Given a datatype `nat` with constructors `Zero` and `Suc`, the set of patterns $Z = \{\mathtt{Zero}, \mathtt{Suc}\ u\}$ covers the type `nat`. However, in the higher-order setting we have open terms that can depend on assumptions. Intuitively, given a type $A[\Psi]$ we can generate a set of patterns by generating patterns for type A and in addition elements from Ψ. To generate all possible elements covering Ψ, we must inspect its shape. If $\Psi = \psi, x_1{:}A_1, \ldots, x_n{:}A_n$, then we generate cases for $x_1, \ldots, x_n$ along with a general parameter case, $p[\mathsf{id}_\psi]$. For example, given a type `nat[ψ,x:nat]` the set of patterns $Z = \{\mathtt{p}[\mathsf{id}_\psi], \mathtt{x}, \mathtt{Zero}, \mathtt{Suc\ u}[\mathsf{id}_\psi]\}$ covers all elements of this type. For a detailed discussion of coverage for higher-order data of type $A[\Psi]$, see Dunfield and Pientka [6]. Here, we simply assume that patterns cover all cases. First we state and prove a canonical forms lemma.

Lemma 2 (Canonical Forms)**.**

(1) If i is a value and $\cdot;\cdot;\cdot \vdash i \Rightarrow \tau \to \tau'$ then $|i| = \mathsf{fn}\ y.|e'|$ and $\cdot;\cdot;y{:}\tau \vdash e' \Leftarrow \tau'$.
(2) If i is a value and $\cdot;\cdot;\cdot \vdash i \Rightarrow A[\Psi]$ then $|i| = (\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi])$ and $\cdot;\cdot;\cdot \vdash \mathsf{box}(\hat{\Psi}.\, M) \Leftarrow A[\Psi]$.
(3) If i is a value and $\cdot;\cdot;\cdot \vdash i \Rightarrow \Pi^{\square} u{::}A[\Psi].\tau$ then $|i| = \lambda^{\square} u.\, |e'|$ and $\cdot; u{::}A[\Psi]; \cdot \vdash e' \Leftarrow \tau$.

Proof. By induction on the typing derivation. □

Theorem 4 (Preservation and Progress)**.**

(1) If $\cdot;\cdot;\cdot \vdash e \Leftarrow \tau$ and e coverage checks then either e is a value or there exists e' such that $|e| \longrightarrow |e'|$ and $\cdot;\cdot;\cdot \vdash e' \Leftarrow \tau$.
(2) If $\cdot;\cdot;\cdot \vdash i \Rightarrow \tau$ and i coverage checks then either i is a value or there exists i' such that $|i| \longrightarrow |i'|$ and $\cdot;\cdot;\cdot \vdash i' \Rightarrow \tau$.

Proof. By induction on the given typing derivation, using Lemma 2, the obvious substitution properties, and coverage soundness [6] as needed. □

5. Related Work

Implementing proofs about HOAS encodings is an ambitious project. One approach is realized in Twelf [19], an implementation of the logical framework LF where inductive proofs are implemented as relations between derivations and the fact that relations constitute total functions is verified separately. While the logical framework LF itself is well-understood and has a small type-theoretic core, the external checkers guaranteeing totality of the implemented relations still remain mysterious to many users. Moreover, how to automate induction proofs about LF signatures and develop such proofs interactively has remained a major problem despite the seminal groundwork laid in Schürmann's dissertation [24]. An alternative approach to proving properties about HOAS encodings is based on generic judgments [8] and realized in the system Abella. This approach enhances intuitionistic logic with generic judgments, which allow for recursive definitions and induction over natural numbers. Contexts are explicit, and the user needs to explicitly manage and prove properties about contexts. This is a powerful approach grounded in proof theory. In contrast, we aim for a type-theoretic approach for programming with proofs and explicit contexts. In particular, we characterize contexts type-theoretically using context schemas. A potential advantage of our approach is that we use the same functional paradigm of writing programs and writing proofs, and hence they can live within the same language.

Enriching functional programming with dependently-typed higher-order data structures is a longstanding open problem and is presently receiving widespread attention. Most closely related to our work is Delphin [23], a dependently-typed functional programming language that supports HOAS encodings. However, our theoretical approach differs substantially. Most of these differences arise because we build our type-theoretical foundation on contextual modal types where $A[\Psi]$ denotes an object M of type A in

a context Ψ. This means that the object M can refer to the variables declared in Ψ. Every data object is therefore locally closed. Moreover, the context Ψ is explicit. An important consequence is that, when pattern matching against data objects of type $A[\Psi]$, it is easy to understand intuitively when all cases are covered. A set of patterns is exhaustive if it contains all constructors of type A together with all the variables from Ψ. Furthermore, the power of context variables allows us to capture more general invariants about our programs. In Delphin, contexts are implicit and there are no context variables. This has several consequences. For example, the whole function and all its arguments are executed in a global context. This is less precise and can express fewer properties on individual data objects. Thus, one cannot express that the first argument of a function is closed, while its second argument may not be (see the `cntV` example).

Despeyroux et al. [5] presented a type-theoretic foundation for programming with HOAS that supports primitive recursion. To separate data from computation, they introduced modal types $\Box A$ that can be injected into computation. However, all data is closed and can only be analyzed by a primitive recursive iterator. Despeyroux and Leleu [3, 4] extended this work to dependent types.

In recent years, various forms of dependent types have found their way into mainstream functional programming to allow programmers to express stronger properties about their programs. Generalized algebraic datatypes [2, 16, 26] can index types by other types and have entered mainstream languages such as Haskell. The Dependent ML approach [29] uses indexed types with a fixed set of constraint domains, such as integers with linear inequalities, for which efficient decision procedures exist. However, all these systems lack the power to manipulate and analyze higher-order data. Moreover, they do not support user-defined index domains.

Finally, there have been several proposals in the functional programming community to allow full dependent types in languages such as Cayenne [1] and Epigram [11]. Neither of these supports HOAS encodings. Moreover, their approach to allowing a user-defined index domain for dependent types is quite different: types can be indexed with arbitrary computations. This is problematic since equality between two types is not simple syntactic equality. Instead, one must first evaluate index arguments. To ensure termination and hence decidability of type checking, Cayenne imposes a heuristic, while Epigram only allows index objects defined by well-founded recursion. This approach violates the idea that type checking should be independent of the operational semantics, and may be costly.

6. Conclusion

We have presented an intermediate language for programming with dependently-typed higher-order data. This paper extends the first author's simply-typed work [21] to the dependently-typed setting and lays the foundation for programming with proofs. Our framework distinguishes between implicit data objects that occur in a dependent type and explicit data objects that are recursively analyzed with higher-order pattern matching. To focus on issues related to dependent types, we omitted first-class substitutions—a concept present in the simply-typed framework [21]—from our presentation, but there is no inherent difficulty in adding them.

When contexts are implicit, as in Twelf and Delphin, implicit world subsumption allows one to elegantly write structured proofs involving lemmas. In our work, we have explicit contexts. We plan to explore an explicit notion of *schema subsumption*, along with existential quantification over contexts, which should be even more flexible and expressive.

We are in the process of completing an implementation of a type checker and interpreter for the internal language described in this paper. However, to make Beluga practical, we plan to address two important questions in the near future. First, we need to reconstruct implicit data objects occurring in computations and data. This follows similar ideas as employed in Twelf [19]. Second, we aim to reconstruct typing annotations at the branches of case expressions. In the long term, we plan to consider the issue of termination checking to ensure that the implemented functions actually do correspond to inductive proofs.

References

[1] L. Augustsson. Cayenne—a language with dependent types. In *3rd International Conference on Functional Programming (ICFP '98)*, pages 239–250. ACM, 1998.

[2] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[3] J. Despeyroux and P. Leleu. Recursion over objects of functional type. *Mathematical Structures in Computer Science*, 11 (4):555–572, 2001.

[4] J. Despeyroux and P. Leleu. Primitive recursion for higher order abstract syntax with dependent types. In *International Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, 1999.

[5] J. Despeyroux, F. Pfenning, and C. 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.

[6] J. Dunfield and B. Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, June 2008.

[7] M. Gabbay and A. 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.

[8] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2008.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[10] D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2008.

[11] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[12] A. McCreight and C. Schürmann. A meta-linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, 2004.

[13] D. Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.

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

[15] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9 (3), 2008.

[16] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *11th ACM SIGPLAN Int'l Conference on Functional Programming (ICFP '06)*, pages 50–61, Sept. 2006.

[17] F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[18] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1063–1147. Elsevier, 2001.

[19] F. Pfenning and C. 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)*, pages 202–206. Springer LNAI 1632, 1999.

[20] B. Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2003. CMU-CS-03-185.

[21] B. 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, 2008.

[22] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, 2003.

[23] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, Mar. 2008.

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

[25] C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In P. 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.

[26] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *Int'l Workshop on Logical Frameworks and Meta-languages (LFM '04)*, pages 106–124, 2004.

[27] M. R. Shinwell, A. M. Pitts, and M. 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.

[28] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.

[29] H. Xi and F. Pfenning. Dependent types in practical programming. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.

A. Appendix

A.1 Ordinary substitution

In the definition for ordinary data-level substitutions, we need to be careful because the only meaningful data-level terms are those in canonical form. To ensure that substitution preserves canonical form, we use a technique pioneered by Watkins et al. [28] and described in detail in Nanevski et al. [15]. The idea is to define *hereditary substitution* as a primitive recursive functional that always returns a canonical object.

In the formal development, it is simpler if we can stick to non-dependent types. We therefore first define type approximations α and an erasure operation $()^-$ that removes dependencies. Before applying any hereditary substitution $[M/x]^a_A(B)$ we first erase dependencies to obtain $\alpha = A^-$ and then carry out the hereditary substitution formally as $[M/x]^a_\alpha(B)$. A similar convention applies to the other forms of hereditary substitutions. Types relate to type approximations via an erasure operation $()^-$ which we overload to work on types.

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

$$\begin{array}{lcl} (a\ N_1 \ldots N_n)^- & = & a \\ (\Pi x{:}A \ldots B)^- & = & A^- \to B^- \\ (\Sigma x{:}A \ldots B)^- & = & A^- \times B^- \end{array}$$

We can define $[M/x]^n_\alpha(N)$, $[M/x]^r_\alpha(R)$, and $[M/x]^s_\alpha(\sigma)$ by nested induction, first on the structure of the type approximation α and second on the structure of the objects N, R and σ. In other words, we either go to a smaller type approximation (in which case the objects can become larger), or the type approximation remains the same and the objects become smaller. The following hereditary substitution operations are defined in Figure 6.

$[M/x]^n_\alpha(N)$	=	N'	Normal terms N
$[M/x]^r_\alpha(R)$	=	R' or $M' : \alpha'$	Neutral terms R
$[M/x]^s_\alpha(\sigma)$	=	σ'	Substitutions σ

We write $\alpha \leq \beta$ and $\alpha < \beta$ if α occurs in β (as a proper subexpression in the latter case). 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. This substitution operation can be extended to types for which we write $[M/x]^a_\alpha(A)$.

Theorem 5 (Termination)**.**
The operation $[M/x]^*_\alpha(\_)$ where $* = \{n, r, s, a\}$ terminates, either by returning a result or failing after a finite number of steps.

Theorem 6 (Hereditary Substitution Principles)**.**
If $\Delta; \Psi \vdash M \Leftarrow A$ and $\Delta; \Psi, x{:}A, \Psi' \vdash J$ then
$\Delta; \Psi, [M/x]^*_\alpha \Psi' \vdash [M/x]^*_\alpha(J)$ where $* = \{n, r, s, a\}$.

Building on Nanevski et al. [15], we can also define simultaneous substitution $[\sigma]^n_{\bar\psi}(M)$ (respectively $[\sigma]^r_{\bar\psi}(R)$ and $[\sigma]^s_{\bar\psi}(\sigma)$). We write $\bar\psi$ for the context approximation of Ψ which is defined using the erasure operation $()^-$.

$$\begin{array}{lcl} (\cdot)^- & = & \cdot \\ (\psi)^- & = & \psi \\ (\Psi, x{:}A)^- & = & (\Psi)^-, x{:}(A)^- \end{array}$$

A.2 Substitution for contextual variables

Substitutions for contextual variables are a little more difficult. We have two kinds of contextual variables: meta-variables u and parameter variables p [21].

Data-level normal terms

$$
\begin{array}{lll}
[M/x]^n_\alpha(\lambda y.\, N) & = \lambda y.\, N' & \text{where } N' = [M/x]^n_\alpha(N) \\
 & & \text{choosing } y \notin \mathsf{FV}(M), \text{and } y \neq x \\
[M/x]^n_\alpha(M_1, M_2) & = (N_1, N_2) & \text{if } [M/x]^n_\alpha(M_1) = N_1 \\
 & & \text{and } [M/x]^n_\alpha(M_2) = N_2 \\
[M/x]^n_\alpha(R) & = M' & \text{if } [M/x]^r_\alpha(R) = M' : \alpha' \\
[M/x]^n_\alpha(R) & = R' & \text{if } [M/x]^r_\alpha(R) = R' \\
[M/x]^n_\alpha(N) & \text{fails} & \text{otherwise}
\end{array}
$$

Data-level neutral terms

$$
\begin{array}{lll}
[M/x]^r_\alpha(x) & = M : \alpha & \\
[M/x]^r_\alpha(y) & = y & \text{if } y \neq x \\
[M/x]^r_\alpha(u[\sigma]) & = u[\sigma'] & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
[M/x]^r_\alpha(p[\sigma]) & = p[\sigma'] & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
[M/x]^r_\alpha(R\ N) & = R'\ N' & \text{where } R' = [M/x]^r_\alpha(R) \\
 & & \text{and } N' = [M/x]^n_\alpha(N) \\
[M/x]^r_\alpha(R\ N) & = M'' : \beta & \\
 & \text{if } [M/x]^r_\alpha(R) = \lambda y.\, M' : \alpha_1 \to \beta \text{ where} & \\
 & \alpha_1 \to \beta \leq \alpha \text{ and } N' = [M/x]^n_\alpha(N) & \\
 & \text{and } M'' = [N'/y]^n_{\alpha_1}(M') & \\
[M/x]^r_\alpha(\mathsf{proj}_i\ R) & = N_i : \alpha_i & \text{if } [M/x]^r_\alpha(R) = (N_1, N_2) : \alpha_1 \times \alpha_2 \\
[M/x]^r_\alpha(\mathsf{proj}_i\ R) & = \mathsf{proj}_i\ R' & \text{if } [M/x]^r_\alpha(R) = R' \\
[M/x]^r_\alpha(R) & \text{fails} & \text{otherwise}
\end{array}
$$

Data-level substitutions

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

Figure 6. Ordinary substitution

Contextual substitution for meta-variables We write the contextual substitution operations for normal objects, neutral objects, and substitutions as follows.

$$
\begin{array}{lcll}
[\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N) & = & N' & \text{Normal terms } N \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R) & = & R' \text{ or } M' : \alpha' & \text{Neutral terms } R \\
[\![\hat\Psi.M/u]\!]^s_{\alpha[\bar\psi]}(\sigma) & = & \sigma' & \text{Substitutions } \sigma
\end{array}
$$

As mentioned earlier, $u[\sigma]$ represents a closure where, as soon as we know which term u should stand for, we can apply σ to it. Because of α-conversion, the variables substituted at different occurrences of u may differ, and we write $\hat\Psi.M$ to allow for necessary α-renaming. The contextual substitution is indexed with the type of u. This typing annotation is necessary since we apply the substitution σ hereditarily once we know which term u represents, and hereditary substitution requires the type to ensure termination.

We define the operations in Figure 7. Note that applying $[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\hat\Psi.M/u]\!]^s_{\alpha[\bar\psi]}\sigma$, but instead of returning $M[\sigma']$, it eagerly applies σ' to M. However, before that we recover its domain with $[\sigma'/\bar\psi]$. To ensure that we return a normal object as a result of contextual substitution, we use ordinary hereditary substitution. For a thorough discussion, see Nanevski et al. [15].

Data-level normal terms

$$
\begin{array}{lll}
[\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(\lambda y.\, N) & = \lambda y.\, N' & \text{where } [\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}N = N' \\
[\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N_1, N_2) & = (N_1', N_2') & \\
\multicolumn{3}{l}{\text{where } [\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N_1) = N_1' \text{ and } [\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N_2) = N_2'} \\
[\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(R) & = R' & \text{where } [\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R) = R' \\
[\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(R) & = M' & \text{where } [\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R) = M' : \beta \\
[\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N) & \text{fails} & \text{otherwise}
\end{array}
$$

Data-level neutral terms

$$
\begin{array}{lcll}
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(x) & = & x & \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(u[\sigma]) & = & N : \alpha & \text{where } [\![\hat\Psi.M/u]\!]^s_{\alpha[\bar\psi]}\sigma = \sigma' \\
 & & & \text{and } [\sigma'/\bar\psi]^n_{\bar\psi}M = N \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(u'[\sigma]) & = & u'[\sigma'] & \text{where } [\![\hat\Psi.M/u]\!]^s_{\alpha[\bar\psi]}\sigma = \sigma' \\
 & & & \text{choosing } u' \neq u \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(p[\sigma]) & = & p[\sigma'] & \text{where } [\![\hat\Psi.M/u]\!]^s_{\alpha[\bar\psi]}\sigma = \sigma' \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R\ N) & = & (R'\ N') & \text{where } [\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}R = R' \\
 & & & \text{and } [\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N) = N' \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R\ N) & = & M' : \alpha_2 & \\
\multicolumn{4}{l}{\text{if } [\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}R = \lambda x.\, M_0 : \alpha_1 \to \alpha_2 \text{ for } \alpha_1 \to \alpha_2 \leq \alpha[\bar\psi]} \\
\multicolumn{4}{l}{\text{and } [\![\hat\Psi.M/u]\!]^n_{\alpha[\bar\psi]}(N) = N' \text{ and } [N'/x]^n_{\alpha_1}(M_0) = M'} \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(\mathsf{proj}_i R) & = & \mathsf{proj}_i R' & \text{if } [\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R) = R' \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(\mathsf{proj}_i R) & = & M_i : \alpha_i & \\
 & & \multicolumn{2}{l}{\text{if } [\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R) = (M_1, M_2) : \alpha_1 \times \alpha_2} \\
[\![\hat\Psi.M/u]\!]^r_{\alpha[\bar\psi]}(R) & & \text{fails} & \text{otherwise}
\end{array}
$$

Figure 7. Substitution for meta-variables

Contextual substitution for parameter variables Contextual substitution for parameter variables follows similar principles, but substitutes an ordinary variable for a parameter variable. We write parameter substitutions as $[\![\hat\Psi.x/p]\!]^*_{\alpha[\bar\psi]}$, where $* \in \{n, r, s, a\}$. When we encounter a parameter variable $p[\sigma]$, we replace p with the ordinary variable x and apply the substitution $[\![\hat\Psi.x/p]\!]^s_{\alpha[\bar\psi]}$ to σ obtaining a substitution σ'. Instead of returning a closure $x[\sigma']$ as the final result we apply σ' to the ordinary variable x. This may again yield a normal term, so we must ensure that contextual substitution for parameter variables preserves normal forms.

$$
\begin{array}{lcll}
[\![\hat\Psi.x/p]\!]^r_{\alpha[\bar\psi]}(p[\sigma]) & = & M : \alpha & \text{if } [\![\hat\Psi.x/p]\!]^s_{\alpha[\bar\psi]}\sigma = \sigma' \\
 & & & \text{and } [\sigma'/\bar\psi]^r_{\bar\psi}x = M : \alpha \\
[\![\hat\Psi.x/p]\!]^r_{\alpha[\bar\psi]}(p[\sigma]) & = & R & \text{if } [\![\hat\Psi.x/p]\!]^s_{\alpha[\bar\psi]}\sigma = \sigma' \\
 & & & \text{and } [\sigma'/\bar\psi]^r_{\bar\psi}x = R \\
[\![\hat\Psi.x/p]\!]^r_{\alpha[\bar\psi]}(p'[\sigma]) & = & p'[\sigma'] & \text{where } [\![\hat\Psi.x/p]\!]^s_{\alpha[\bar\psi]}\sigma = \sigma'
\end{array}
$$

Theorem 7 (Termination)**.**
$[\![\hat\Psi.M/u]\!]^*_{\alpha[\bar\phi]}(\_)$ and $[\![\hat\Psi.x/p]\!]^*_{\alpha[\bar\phi]}(\_)$ where $* \in \{n, r, s, a\}$ terminate, either by returning a result or failing after a finite number of steps.

Theorem 8 (Contextual Substitution Principles)**.**

1. *If $\Delta_1; \Phi \vdash M \Leftarrow A$ and $\Delta_1, u{::}A[\Phi], \Delta_2; \Psi \vdash J$ then $\Delta_1, [\![\hat\Psi.M/u]\!]^*_{\alpha[\bar\phi]}\Delta_2; [\![\hat\Psi.M/u]\!]^*_{\alpha[\bar\phi]}\Psi \vdash [\![\hat\Psi.M/u]\!]^*_{\alpha[\bar\phi]}J$*
2. *If $\Delta_1; \Phi \vdash x \Rightarrow A$ and $\Delta_1, p{::}A[\Phi], \Delta_2; \Psi \vdash J$ then $\Delta_1, [\![\hat\Psi.x/p]\!]^*_{\alpha[\bar\phi]}\Delta_2; [\![\hat\Psi.x/p]\!]^*_{\alpha[\bar\phi]}\Psi \vdash [\![\hat\Psi.x/p]\!]^*_{\alpha[\bar\phi]}J$*

where $* = \{n, r, s, a\}$.