A Type-theoretic Foundation for Programming with Higher-order Abstract Syntax and First-class Substitutions

Brigitte Pientka

McGill University, Canada

bpientka@cs.mcgill.ca

Abstract

Higher-order abstract syntax (HOAS) is a simple, powerful technique for implementing object languages, since it directly supports common and tricky routines dealing with variables, such as capture-avoiding substitution and renaming. This is achieved by representing binders in the object-language via binders in the meta-language. However, enriching functional programming languages with direct support for HOAS has been a major challenge, because recursion over HOAS encodings requires one to traverse λ-abstractions and necessitates programming with open objects.

We present a novel type-theoretic foundation based on contextual modal types which allows us to recursively analyze open terms via higher-order pattern matching. By design, variables occurring in open terms can never escape their scope. Using several examples, we demonstrate that our framework provides a name-safe foundation to operations typically found in nominal systems. In contrast to nominal systems however, we also support capture-avoiding substitution operations and even provide first-class substitutions to the programmer. The main contribution of this paper is a syntax-directed bi-directional type system where we distinguish between the data language and the computation language together with the progress and preservation proof for our language.

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

General Terms Theory, Languages

Keywords Type theory, logical frameworks

1. Introduction

Typed functional programming languages are particularly suited for analyzing and manipulating syntactic structures and are pervasively used for implementing object languages. Although many object languages include binding constructs, it is striking that typed functional languages still lack direct support for binders and common tricky operations such as renaming, capture-avoiding substitution, and fresh name generation. The most common approach in practice is to implement binders via de Bruijn indices, which at least provides for α-renaming. While this leads to an efficient implementation, analyzing and especially manipulating data can be cumbersome and error-prone. Capture-avoiding substitution must be implemented separately. Nominal approaches (Gabbay and Pitts 1999) as found in FreshML (Shinwell et al. 2003) provide first-class names and α-renaming. This approach is appealing because it gives us direct access to names of bound variables. The generation of a new name and binding names are separate operations and fresh name generation is an observable side effect. Unfortunately, this means that it is possible to generate data which contains accidentally unbound names. To address this problem, Pottier (2007) recently proposed pure FreshML where one can reason about the set of names occurring in an expression via a Hoare-style proof system. This static analysis approach is quite expressive since the language of constraints includes subset relations, equality, and intersection. Nevertheless, the programmer needs to implement capture-avoiding substitution manually.

In higher-order abstract syntax (HOAS) we represent binders in the object-language via binders in our meta-language (i.e. functional programming language). One of the key benefits is that we not only get support for renaming and fresh name generation, but also for capture-avoiding substitution. Consequently, it is typically easier to ensure correctness and reason about HOAS representations, since tedious lemmas about substitutions and fresh names do not need to be proven. The strengths of HOAS have been impressively demonstrated within the logical framework LF (Harper et al. 1993) and its implementation in the Twelf system (Pfenning and Schürmann 1999) over the past decade. However, HOAS has rarely been considered in real programming languages (example include Washburn and Weirich (2006); Guillemette and Monnier (2006)). To illustrate the difficulty, we define a small language with let-expressions and arithmetic operations. The let-expression is defined via higher-order abstract syntax, i.e. the binder in the let-expression is represented via a binder in our meta-language. We distinguish between natural numbers and expressions, and variables in expressions represent only values.

```
exp: type .                                nat: type .
Nat: nat → exp.                            z: nat.
Add: exp → exp → exp.                      suc: nat → nat.
Let: exp → (nat → exp) → exp.
```

The expression let val x = 1 in Add(0, x) end is represented as `Let (Nat(suc z)) (λx.Add (Nat z) (Nat x))` in our meta-language. When we recursively analyze the body of the let-expression, we must analyze `Add (Nat z) (Nat x)` which contains a free variable `x` and hence is an open term. Moreover, we often also want to manipulate the variable `x` and compare it to other variables. However, HOAS encodings do typically not allow us to directly access, manipulate and compare bound variables.

The message of this paper is that we can recursively analyze and manipulate open data, which is defined via HOAS and may contain variables, and safely add this functionality to typed functional programming. In particular, we can support operations such as analyzing and comparing bound variables. By design, variables occurring in open data can never escape their scope thereby avoiding a problem prevalent in previous attempts. Our framework therefore may be seen as a name-safe alternative foundation for the operations typically supported in nominal systems. However, in addition to supporting binders and α-renaming, we provide capture-avoiding substitution operation together with first-class substitution.

Building on ideas by Nanevski et al. (2006), we design a type-theoretic foundation for programming with HOAS and substitutions based on contextual modal types. The contextual modal type $A[\Psi]$ classifies open data M where M has type A in the *context* Ψ. Consequently, the object M may refer to the variables declared in the context Ψ, but M is closed with respect to the context Ψ. The data-object `Add (Nat (suc z)) (Nat x)` from the previous example has type `exp` in the context `x:nat`.

Generalizing ideas from Despeyroux et al. (1997), data of type $A[\Psi]$ may be injected into the computation level and analyzed via pattern matching via the **box**-construct thereby separating data from computations. Since we want to allow recursion over open data objects and the local context Ψ which is associated with the type A may grow, our foundation supports *context variables* and abstraction over contexts. Consequently, different arguments to a computation may have different local contexts and we can distinguish between data of type $A[\cdot]$, which is closed, and open data of type $A[\Psi]$. This gives us fine-grained control and allows us to specify strong invariants. Our type-theoretic foundation based on contextual modal types is substantially different from previous proposals to marry HOAS with functional programming as proposed by Schürmann et al. (2005). In Schürmann et al. (2005), the necessity modality $\Box\tau$ describes *computation* of type τ which can be executed in every world where we have at least one context extension. The context containing binding occurrences is hence left implicit and associated with a computation. As a consequence, every argument of the computation must share one global context. While Schürmann's work does provide for capture-avoiding substitution via β-reduction, it lacks the support to construct substitutions as first-class objects. The type of a substitution in our framework will be $\Psi[\Phi]$ where Ψ describes the domain and Φ the range. In other words, the substitution of type $\Psi[\Phi]$ maps bound variables declared in the context Ψ to objects which may refer to the bound variables declared in Φ. We believe this foundation provides general insights into how open data-objects can be understood type-theoretically and incorporated as first-class into programming languages. The main contribution of this paper are:

- We present a syntax-directed decidable bi-directional *type system for open data and substitutions based on contextual types*. By design, variables occurring in open data objects cannot escape their scope, and no separate reasoning about scope is required (Section 3). Following a recent presentation technique for logical frameworks due to Watkins et al. (2002) our syntax only allows for objects which are in canonical form since only these are meaningful for representing object-languages. Consequently, exotic terms that do not represent legal objects from our object-language are prevented.
- Extending our previous work (Pientka and Pfennning 2003), we present a *linear higher-order pattern matching algorithm for open data and substitutions* together with its correctness proof (Section 3.4). We also discuss the trade-offs and costs of considering the full pattern fragment as defined by Miller (1991) versus linear higher-order patterns (Section 2 and Section 3.4).
- Using several examples (Section 2), we show that our framework may be seen as a name-safe foundation to operations typically found in nominal systems while at the same time also providing for capture-avoiding substitutions together with first-class substitutions.
- We present a syntax-directed decidable bi-directional *type system for computation which allows recursion and pattern matching on open data and substitutions* (Section 4) together with a *small-step operational semantics* (Section 5). Open data is injected into computation via the **box**-construct. The driving force behind the operational semantics is the higher-order matching algorithm described in Section 3.4. The key to a clean and elegant meta-theory is our distinction between *bound variables in data*, *contextual variables* which may be instantiated via pattern matching with data, *context variables*, and *bound variables in computations*. Each of these variables gives rise to their own substitution definition and they play a central role in defining our operational semantics. Building on different substitution properties, we prove type preservation and progress.

We believe our calculus is an important step towards understanding syntactic structures with binders, and how one can provide direct support for binders in the setting of typed functional programming. More generally, it provides a type-theoretic foundation for open data-objects which play an important role in many areas of computer science beyond programming with higher-order abstract syntax, such as explaining linking of code, or staged computation (see for example (Kim et al. 2006)).

2. Motivation

In this section we briefly discuss four small examples to illustrate the main ideas behind our approach. The examples are purposely simple to emphasize the key features of our work and we keep the notation as close as possible to the theoretical foundation. In all our examples, we consider a small language with let-expressions and arithmetic operations which we introduced earlier.

Counting occurrences of a variable In the first example, we show how to count the occurrences of a particular variable in a given data-object. We exploit the power of higher-order patterns to accomplish this. The function `cntV` takes in a context ψ of natural numbers, and a data-object `e` of type `exp[`ψ`,x:nat]`, i.e. an expression which may refer to the bound variables listed in the context (`$\psi$,x:nat`), and returns as a result an integer. Just as types classify data-objects and kinds classify types, we introduce *context schemas* to classify contexts. In the type declaration for the function `cntV` we state that the context variable ψ has the context schema `(nat)`*. In other words ψ stands for a data-level context of the form $\mathtt{x}_1$`:nat, ..., `$\mathtt{x}_n$`:nat`. We represent contextual variables which are instantiated via higher-order pattern matching with capital letters.

```
rec  cntV : Π ψ:(nat)*.exp[ψ,x:nat] → int =
Λ ψ ⇒ fn  e ⇒ case  e of
  box (ψ,x. Nat U[idψ]) ⇒ 0
| box (ψ,x. Nat U[x]) ⇒ 1
| box (ψ,x. Let U[idψ,x] (λy.W[idψ,x,y])) ⇒
   cntV ⌈ψ⌉ box(ψ,x. U[idψ,x]) +
   cntV ⌈ψ,y:nat ⌉ box(ψ,y,x. W[idψ,x,y])
| box (ψ,x. Add U[idψ,x] W[idψ,x]) ⇒
   cntV ⌈ψ⌉ box(ψ,x. U[idψ,x])
 + cntV ⌈ψ⌉ box(ψ,x. W[idψ,x])
```

The function `cntV` is built by a context abstraction $\Lambda\ \psi$ which introduces the context variable ψ and binds every occurrence of ψ in the body of the function. Next, we introduce the computation-level variable `e` which has type `exp[`ψ`,x:nat]` by function-abstraction. In the body of the function `cntV` we analyze objects of type `exp[`ψ`,x:nat]` by case-analysis. As mentioned earlier, we separate data from computations via the **box**-construct.

"Holes" in the pattern which are instantiated via higher-order pattern matching 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 $\mathbf{id}_\psi$ for the identity substitution with domain ψ. Intuitively, one may think of the substitution associated with contextual variables which occur in patterns as a list of variables which may occur in the hole. In `U[`$\mathbf{id}_\psi$`]`, for example, the contextual variable `U` can be instantiated with any natural number which either is closed, i.e. it does not refer to any bound variable listed in the context ψ or it contains a bound variable from the context ψ.

To recursively analyze expressions we have to consider different cases. The first case **box** `(`ψ`,x. Nat U[`$\mathbf{id}_\psi$`])` captures the idea that if we encounter a natural number which does not contain occurrences of the bound variable `x` then we return `0`. In the second case **box** `(`ψ`,x. Nat (U[x]))` we pattern match against a "hole" which may refer to the bound variable `x`. The only time this case now succeeds, is for a natural number which actually refers to `x` since this is the only case not already covered by the first case. In the third case **box** `(`ψ`,x.Let U[`$\mathbf{id}_\psi$`,x] (`λ`y.W[`$\mathbf{id}_\psi$`,x,y]))` we analyze the let-expression. To count the occurrences of the variable `x`, we first count the occurrences in `U[`$\mathbf{id}_\psi$`,x]`, and then count the occurrences in the body of the let-expression. To accomplish this, we must extend the context with the declaration `y:nat` and pass the extended context `(`ψ`,y:nat)` to the recursive call of `cntV`. Context application is described by `cntV` $\lceil$$\psi$`,y:nat`$\rceil$.

The **box**-construct binds occurrences of data-level variables only. For example in **box** `(`ψ`,y,x.U[`$\mathbf{id}_\psi$`,x,y])`, the variables `x` and `y` are bound and subject to α-renaming. However we emphasize that the context variable ψ is not bound by the **box**-construct in the branch of a case-expression, but bound by the context abstraction $\Lambda\ \psi \Rightarrow \ldots$. In particular, ψ is not instantiated via pattern matching. One may think that listing the bound variables explicitly in the **box**-construct is not necessary because they are determined by the type `exp[`ψ`,x:nat]`. However to support α-renaming of data-level variables, we explicitly list the names of the bound variables, and enforce that this list can be obtained by erasing all types from the context ψ`,x:nat`. In an implementation of our language based on de Bruijn indices this complication can be eliminated.

Extracting variables Next, we show how to compute the variables occurring in a data-object. We will write a function which accepts a natural number of type `nat[`ψ`]` which may refer to bound variables listed in the context ψ, and returns a data-object of type `nat[`ψ`] option`. If a bound variable occurs in the natural number then we return it, otherwise we return `NONE`.

```
rec  FVnat:Π ψ:(nat)*.nat[ψ] → (nat[ψ]) option =
Λ ψ ⇒ fn  e ⇒ case  e of
  box (ψ.z) ⇒ NONE
| box (ψ.p[id_ψ]) ⇒ SOME box(ψ.p[id_ψ])
| box (ψ.suc U[id_ψ]) ⇒ FVnat ⌈ψ⌉ box(ψ. U[id_ψ])
```

The key question is how do we detect and pattern match against a data-level variable? – To accomplish this, we use the *parameter variable* `p`. A parameter variable represents a bound variable and can only be instantiated with a variable from the object level. Similar to meta-variables they are treated as closures. In contrast to meta-variables which can be instantiated with an arbitrary object and are represented by capital letters, we will use small letters for parameter variables. Parameter variables allow us to write explicitly a case for matching against variables and allow us to collect them, and even compare them. Given the function `FVnat`, it should be obvious how to write a function which collects all the free variables occurring in a let-expression.

Closed value So far our examples only utilized one context and didn't exploit the power that we are able to distinguish between different contexts. For example, we may want to write a simple function which tests whether a given natural number is closed, and hence constitutes a value. It may be in fact important in the later part of the program that we know that we have a closed value. This can be achieved by the function `isVal` which not only tests whether a given natural number is closed but also strengthens the result.

```
val  isVal:Π ψ:(nat)*. nat[ψ] → (nat[.]) option =
Λ ψ ⇒ fn  e ⇒ case  e of
  box (ψ.U[.]) ⇒ SOME box(U[.])
| box (ψ.U[id_ψ]) ⇒ NONE
```

In the first case we test whether the input is closed, and if it is we return it in a strengthened context. While in other approaches we can recursively analyze objects and thereby check whether they are a closed value, the property of being closed is usually harder to capture in current type systems.

Environment-based interpreter Finally, we give an example which uses first-class substitutions and substitution variables which our foundation provides. The task is to write a simple environment-based interpreter for the language we have defined earlier, where we take in a context ψ of natural numbers, an expression `e` of type `exp[`ψ`]` and an environment `r` which maps variables declared in the context ψ to closed values. The environment is represented as a substitution with domain ψ and range empty and has the type ψ`[.]`. The result of the interpreter is a closed value. Similar to the **box**-construct which injects open data-objects into the computation, we use the **sbox**-construct to inject data-level substitutions into the computation. We will use capital letters `S` to describe substitution variables which may occur in patterns, and similar to other contextual variables we think of substitution variables as closures, providing a built-in operation for composing substitutions.

```
rec  eval:Π ψ:(nat)*. exp[ψ] → ψ[.] → nat[.] =
Λ ψ ⇒ fn  e ⇒ fn  r ⇒ let  sbox (S[.]) = r in
case  e of
  box (ψ.Nat U[id_ψ]) ⇒ box(U[S[.]])
| box (ψ.Add U1[id_ψ] U2[id_ψ]) ⇒
  let  val  a = eval ⌈ψ⌉ box(ψ.U1[id_ψ]) r
       val  b = eval ⌈ψ⌉ box(ψ.U2[id_ψ]) r
  in  add(a, b)  end
| box (ψ.Let W[id_ψ] (λx. U[id_ψ, x])) ⇒
  let
    box  V[.] = eval ⌈ψ⌉ box(ψ.W[id_ψ]) r
  in
    eval ⌈ψ,x:nat⌉
         box (ψ,x. U[id_ψ,x]) sbox(S[.], V[.])
  end
end
```

When we encounter a natural number as in the first case, we can simply apply the substitution `S[.]` to the object `U`. Since the substitution `S[.]` has domain ψ and range empty, applying it to the meta-variable `U` yields a closed object. Because we apply `S[.]` as soon as we know what `U` stands for, the variable occurring in the instantiation for U will now be replaced by its correct corresponding value. Closures thereby provide us with built-in support for substitutions. The type system guarantees that the environment `r` provides closed instantiations for every variable in the local context ψ, and applying the substitution `S[.]` to the contextual variable `U` must yield a closed natural number.

When evaluating **box** (ψ.Let W[**id**$_\psi$] (λx.U[**id**$_\psi$,x])), we evaluate the expression **box** (ψ.W[**id**$_\psi$]) in the environment r to some closed value V, and then evaluate **box** (ψ,x.W[**id**$_\psi$,x]) in the extended environment where we associate the binder x with the value V. Since we think of substitutions by position, we do not make their domain explicit and simply write **sbox** (S[.], V[.])

In traditional approaches where names are first-class, as in FreshML, or where variables are represented by strings or de Bruijn indices, environments are usually implemented as a list of pairs consisting of variable name and their corresponding value. However, the type system cannot easily guarantee that the environment indeed provides closed values for all free variables occurring in the expression. Moreover, one needs to write a lookup-function for retrieving a value from the environment and a substitution function for replacing the occurrence of the free variable with its corresponding value. In contrast, our foundation enforces a strong invariant about the relationship between the expression we are analyzing and the environment.

Remark: Higher-order pattern matching and its trade-offs

We would like to emphasize that our interest is in designing a type-theoretic foundation for programming with HOAS and the code snippets presented are intended to model our theory closely to provide an intuition. It is not necessarily intended as the source language which a programmer would use.

As mentioned earlier, we treat contextual variables uniformly as closures and allow full higher-order pattern matching to instantiate contextual variables. In *higher-order patterns* (Miller 1991), the substitution associated with contextual variables must consist of *some distinct bound variables*, and pattern matching must enforce checks on bound variables. The cost of checking for variable dependencies is hidden from the user (see for example the function cntV or isVal). *Linear higher-order patterns* (Pientka and Pfennning 2003) restrict higher-order patterns such that every contextual variable must be applied to *all the bound variables* in whose scope it occurs. In this case no bound variable dependency checks are necessary (see Section 3.4) yielding an efficient matching algorithm closely resembling first-order matching. Our foundation leaves implementors a choice of whether to enforce linear higher-order patterns dynamically or statically. To enforce them dynamically, one translates every pattern into a linear one with potentially additional constraints (see (Pientka and Pfennning 2003)). Consequently, only if bound variable checks are necessary, they will be done. Alternatively, if we enforce it statically, then all contextual variables must be applied to all the bound variables. In this case, contextual variables occurring in patterns can be simply described by U or p and not as closure U[**id**$_\psi$] and p[**id**$_\psi$]. To check whether an object depends on a bound variable we can write a separate function. To illustrate, we present a function for counting occurrences of a variable x in a natural number.

```
rec cntVN : Π ψ:(nat)*.nat[ψ] → nat[ψ] → int =
Λ ψ ⇒ fn e ⇒ fn e' ⇒ let box(ψ. p'[idψ]) = e' in
case e of
  box(ψ. z) ⇒ 0
| box(ψ. p[idψ]) ⇒
  if box(ψ.p'[idψ]) = box(ψ. p[idψ]) then 1 else 0
| box(ψ. suc U[idψ]) ⇒ cntVN ⌈ψ⌉ box(ψ. U[idψ]) e'
end
```

If we elide the identity substitutions associated with the contextual variables, we obtain a program which closely resembles the one we write in a nominal style[1]. Closures however provide us with direct built-in substitution operation. As shown in the environment-based interpreter, the closure of meta-variable and substitution allows us to retrieve a value. Similarly, when implementing a substitution-based interpreter, we can use closures to propagate instantiation. Providing not only direct support for binders but also substitution, is the essence in higher-order abstract syntax. However, this discussion seems to suggest that basic operations provided by nominal systems can be explained by imposing special restrictions on our foundation.

Summary of key ideas

We summarize here the four key ideas underlying our work: First, we separate the data from the computation via the modality **box**. Second, every data-object is closed with respect to a local context. For example, **box** (x1,x2.Add (Nat(suc x1)) (Nat x2)) denotes a data-object of type exp[x1:nat,x2:nat]. The box-construct introduces the bound variables x1 and x2. 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 once we recursively traverse a binder, and the context describing these variables may grow. Context abstraction via Λ binds every occurrence of ψ in the expression e. Fourth, we provide closures consisting of a contextual variable and a postponed substitution. When replacing the contextual variable with a concrete object, we apply the substitution thereby providing built-in support for substitutions. We support three kinds of contextual variables, meta-variables which can be instantiated with an arbitrary object, parameter variables which can be instantiated with bound variables only, and substitution variables which represent first-class substitutions. While meta-variables allow us to deconstruct arbitrary objects with binders, parameter variables allows us to manipulate names of bound variables directly in computation.

3. Data-level terms, substitutions, contexts

In this section, we concentrate on the formal definition and type system for data-objects. The definition of computation-level expressions which allow recursion and pattern matching on data-objects is discussed later in Section 4.

Our theoretical development is closely based on contextual modal type theory by Nanevski et al. (2006) which we extend with pairs and projections and more importantly with parameter variables and substitution variables as well as context variables. For simplicity, we also restrict our data-objects to the simply-typed fragment, however the ideas can be extended to the dependently typed setting (see Nanevski et al. (2006) for more details).

Types	A, B	$::=$	$P \mid A \to B \mid A \times B$
Normal Terms	M, N	$::=$	$\lambda x.\, M \mid (M, N) \mid R$
Neutral Terms	R	$::=$	$c \mid x \mid u[\sigma] \mid p[\sigma] \mid R\ N \mid \mathsf{proj}_i R$
Substitutions	σ, ρ	$::=$	$\cdot \mid \sigma\,;\, M \mid \sigma\,,\, R \mid s[\sigma] \mid \mathsf{id}_\psi$
Context Schema	W	$::=$	$A \mid (W)^* \mid W_1 + W_2$
Contexts	Ψ, Φ	$::=$	$\cdot \mid \psi \mid \Psi, x{:}A$
Meta-contexts	Δ	$::=$	$\cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, p{::}A[\Psi] \mid \Delta, s{::}\Psi[\Phi]$
Context Schema context	Ω	$::=$	$\cdot \mid \Omega, \psi{::}W$

Following a recent presentation technique for logical frameworks due to Watkins et al. (2002) our syntax only allows for objects which are in canonical form since only these are meaningful

[1] There are two remaining differences: First, our foundation makes the context (= set of names) explicit. This is necessary if we want to reason about the closedness of an object, and is also done for example in pure FreshML by Pottier (2007). Since context schemas classify contexts, we naturally can distinguish between different sets of names. Second, unlike nominal type systems which have a special type atom for names, our type system does not distinguish between a type for names and objects.

for representing object-languages. This is achieved by distinguishing between normal terms M and neutral terms R. While the syntax only guarantees that terms N contain no β-redices, the typing rules will also guarantee that all well-typed terms are fully η-expanded.

We distinguish between four different 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 they include *meta-variables* u which represent general open objects and *parameter variables* p which can only be instantiated with an ordinary bound variable, and *substitution variables* s which represent a mapping from one context to another. Contextual variables are introduced in case-expressions on the computation level, and can be instantiated via pattern matching. They are associated with a postponed substitution σ thereby representing a closure. Our intention is to apply σ as soon as we know which term the contextual variable should stand for. The domain of σ therefore describes the free variables which can possibly occur in the object which represents the contextual variable, and our type system will ensure statically that this is indeed the case.

Substitutions σ are built of either normal terms (in $\sigma\,;\,M$) or atomic terms (in $\sigma\,,\,R$). We do not make the domain of the substitutions explicit. This will simplify the theoretical development and avoid having to rename the domain of a given substitution σ. Similar to meta-variables, substitution variables are closures with a postponed substitution. We also require a first-class notion of identity substitution id_ψ. Our convention is that substitutions, as defined operations on data-level terms, are written in prefix notation $[\sigma]N$ for a data-level substitution. Contextual variables such as the meta-variables u, parameter variables p, and substitution variables s are declared in the meta-context Δ, while ordinary bound variables are declared in the context Ψ.

Finally, our foundation supports *context variables* ψ which allow us to reason abstractly with contexts. Abstracting over contexts is an interesting and essential next step to allow recursion over higher-order abstract syntax. Context variables are declared in the context Ω. Unlike previous uses of context variables for example in (McCreight and Schürmann 2004), a context may at most contain one context variable. In the same way as types classify objects, and kinds classify types, we will introduce the notion of a context schema W which classifies contexts Ψ. We will say a context Ψ belongs to context schema W or a context Ψ is an element of the context schema W, if it consists of declarations $x{:}A$ where A occurs in W. Context schemas are described by a subset of regular expressions. For example, a context schema $A + B$ describes any context which contains declarations of the form $x{:}A$ or $y{:}B$. Concept schemas resemble the notion of worlds described in (Schürmann 2000), however while similar in spirit, we simplify matters by drawing on the power of pairs and cross-products to express the relationship between multiple objects in a context.

We assume that type constants and object constants are declared in a signature Σ which we typically suppress since it never changes during a typing derivation. However, we will keep in mind that all typing judgments have access to a well-formed signature.

3.1 Data-level typing

Next, we present a bi-directional type system for data-level terms. 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
Ω	$\vdash \Psi \Leftarrow W$	Context Ψ checks against schema W

For better readability, we omit Ω in the subsequent development since it is constant and assume that context Δ and Ψ are well-formed. First, the typing rules for objects. 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. We are also explicit about α-renaming in the rule for substitution variables where $\overset{\alpha}{=}$ describes equality between two contexts up to renaming.

Data-level normal terms

$$\frac{\Delta;\Psi,x{:}A \vdash M \Leftarrow B}{\Delta;\Psi \vdash \lambda x.\,M \Leftarrow A \to B} \qquad \frac{\Delta;\Psi \vdash R \Rightarrow P' \quad P' = P}{\Delta;\Psi \vdash R \Leftarrow P}$$

$$\frac{\Delta;\Psi \vdash M_1 \Leftarrow A_1 \quad \Delta;\Psi \vdash M_2 \Leftarrow A_2}{\Delta;\Psi \vdash (M_1, M_2) \Leftarrow A_1 \times A_2}$$

Data-level neutral terms

$$\frac{x{:}A \in \Psi}{\Delta;\Psi \vdash x \Rightarrow A} \qquad \frac{c{:}A \in \Sigma}{\Delta;\Psi \vdash c \Rightarrow A} \qquad \frac{\Delta;\Psi \vdash R \Rightarrow A_1 \times A_2}{\Delta;\Psi \vdash \mathsf{proj}_i R \Rightarrow A_i}$$

$$\frac{u{::}A[\Phi] \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi}{\Delta;\Psi \vdash u[\sigma] \Rightarrow A} \qquad \frac{p{::}A[\Phi] \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi}{\Delta;\Psi \vdash p[\sigma] \Rightarrow A}$$

$$\frac{\Delta;\Psi \vdash R \Rightarrow A \to B \quad \Delta;\Psi \vdash N \Leftarrow A}{\Delta;\Psi \vdash R\;N \Rightarrow B}$$

Data-level substitutions

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

$$\frac{s{::}\Phi_1[\Phi_2] \in \Delta \quad \Delta;\Psi \vdash \rho \Leftarrow \Phi_2 \quad \Phi \overset{\alpha}{=} \Phi_1}{\Delta;\Psi \vdash (s[\rho]) \Leftarrow \Phi}$$

$$\frac{\Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \Delta;\Psi \vdash R \Rightarrow A' \quad A = A'}{\Delta;\Psi \vdash (\sigma\,,\,R) \Leftarrow (\Phi, x{:}A)}$$

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

In the simultaneous substitutions σ, we do not make its domain explicit. Rather we think of a substitution as a list of normal and neutral terms, and if σ has domain Ψ the i-th element in σ corresponds to the i-th declaration in Ψ. We can turn any substitution σ which does not make its domain Ψ explicit into one which does by σ/Ψ. We distinguish between substituting a normal term M and a neutral term R. This is justified by the nature of hypothetical judgments, since an assumption $x{:}A$ represents $x \Rightarrow A$ so we can substitute R for x if $R \Rightarrow A$. This distinction between normal and neutral terms is necessary since we can extend a given simultaneous substitution σ with x and obtain $(\sigma\,,\,x)$ when traversing a binding operator in a type-free way. We could not extend the simultaneous substitutions with $(\sigma\,;\,x)$, since x is not a canonical term unless it is of atomic type. Identity substitutions can now have the form $(x_1\,,\ldots,\,x_n)$. The typing rules for substitutions also make clear that the identity substitution id_ψ is necessary once we introduce context variables. Without the identity substitution id_ψ, we wouldn't be able to construct substitutions where the domain is described abstractly by the context variable ψ. Finally, we present context schema checking.

Context Ψ checks against a context schema W

$$\frac{A \in W \quad \Omega \vdash \Psi \Leftarrow W}{\Omega \vdash \Psi, x : A \Leftarrow W} \qquad \frac{\psi{::}W \in \Omega}{\Omega \vdash \psi \Leftarrow W} \qquad \frac{}{\Omega \vdash \cdot \Leftarrow W}$$

Essentially a context is well-formed, if every declaration $x_i{:}A_i$ is declared in the context schema $W = A_1 + A_2 + \ldots + A_n$. $A \in W$ succeeds if $W = (A_1 + \ldots + A_n)^*$ and there exists an A_i s.t. $A = A_i$.

Theorem 3.1. [Decidability of Type Checking]
All judgments in the contextual modal type theory are decidable.

3.2 Substitution

Since we have different kinds of variables, context variables ψ, ordinary variables x, and contextual variables, this gives rise to different substitution operations. These different substitution operations are key to the elegant and simple preservation and progress proof.

Substitution for context variables

We begin by considering the substitution operation for context variables. The most interesting cases are where actual substitution happens. If we encounter a context variables ψ, then we simply replace it with the context Ψ. When we apply the substitution $[\![\Psi/\psi]\!]$ to the context $\Phi, x{:}A$, we apply the substitution to the context Φ to yield some new context Φ'. However, we must check whether x occurs in the variables declared in Φ', i.e. $x \notin \mathsf{V}(\Phi')$, to avoid name clashes. This side condition can always be met by appropriately renaming bound variable occurrences.

Data-level context

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

The remaining definition is mostly straightforward. Since, context variables occur in the identity substitution id_ψ, we must apply the context substitution to objects and in particular to substitutions. When we replace ψ with Ψ in id_ψ, we unfold the identity substitution. Expansion of the identity substitution is defined by the operation $\mathsf{id}(\Psi)$ for valid contexts Ψ as follows:

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

Lemma 3.1. [Unfolding identity substitution]
If $\mathsf{id}(\Psi) = \sigma$ then $\Delta; \Psi, \Psi' \vdash \sigma \Leftarrow \Psi$.

Theorem 3.2. [Substitution for context variables]
If $\Omega, \psi{::}W, \Omega'; \Delta; \Phi \vdash J$ and $\Omega \vdash \Psi \Leftarrow W$
then $\Omega, \Omega'; [\![\Psi/\psi]\!]\Delta; [\![\Psi/\psi]\!](\Phi) \vdash [\![\Psi/\psi]\!]J$.

Ordinary substitution

In the definition for ordinary data-level substitutions, we need to be a little bit careful because the only meaningful data-level terms are those which are in canonical forms. To ensure that substitution preserves canonical forms, we will employ a technique pioneered by Watkins *et. al.* (Watkins et al. 2002) and described in detail in (Nanevski et al. 2006). The idea is to define *hereditary substitution* as a primitive recursive functional which will always return 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 redices. We define the hereditary substitution operations for normal object, neutral objects and substitutions next.

$$\begin{array}{ll}
[M/x]^n_A(N) = N' & \text{Hereditary substitution into } N \\
[M/x]^r_A(R) = R' \text{ or } M' : A' & \text{Hereditary substitution into } R \\
[M/x]^s_A(\sigma) = \sigma' & \text{Hereditary substitution into } \sigma
\end{array}$$

Each of these hereditary substitution operations will be defined by nested induction, first on the structure of the type A and second on the structure of the objects N, R, and σ. In other words, we either go to a smaller type, in which case the objects themselves can become larger, or the type remains the same and the objects become smaller. We write $A \leq B$ and $A < B$ if A occurs in B (as a proper subexpression in the latter case). For an in depth discussion, we refer the reader to (Nanevski et al. 2006).

Data-level normal terms

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

Data-level neutral terms

$$\begin{array}{lcll}
[M/x]^r_A(x) & = & M : A & \\
[M/x]^r_A(y) & = & y & \text{if } y \neq x \\
[M/x]^r_A(u[\sigma]) & = & u[\sigma'] & \text{where } \sigma' = [M/x]^s_A(\sigma) \\
[M/x]^r_A(p[\sigma]) & = & p[\sigma'] & \text{where } \sigma' = [M/x]^s_A(\sigma) \\
[M/x]^r_A(R\; N) & = & R'\; N' & \text{where } R' = [M/x]^r_A(R) \text{ and} \\
 & & & N' = [M/x]^n_A(N) \\
[M/x]^r_A(R\; N) & = & M'' : B & \\
 & & \multicolumn{2}{l}{\text{if } [M/x]^r_A(R) = \lambda y.\, M' {:} A_1 \to B \text{ where}} \\
 & & \multicolumn{2}{l}{A_1 \to B \leq A \text{ and } N' = [M/x]^n_A(N)} \\
 & & \multicolumn{2}{l}{\text{and } M'' = [N'/y]^n_{A_1}(M')} \\
[M/x]^r_A(\mathsf{proj}_i\; R) & = & N_i : A_i & \\
 & & \multicolumn{2}{l}{\text{if } [M/x]^r_A(R) = (N_1, N_2) {:} A_1 \times A_2} \\
[M/x]^r_A(\mathsf{proj}_i\; R) & = & \mathsf{proj}_i\; R' & \text{if } [M/x]^r_A(R) = R' \\
[M/x]^r_A(R) & & \text{fails} & \text{otherwise}
\end{array}$$

Data-level substitution

$$\begin{array}{lcll}
[M/x]^s_A(\cdot) & = & \cdot & \\
[M/x]^s_A(\sigma\,;\, N) & = & (\sigma'\,;\, N') & \text{where } \sigma' = [M/x]^s_A(\sigma) \\
 & & & \text{and } N' = [M/x]^n_A(N) \\
[M/x]^s_A(\sigma\,,\, R) & = & (\sigma'\,,\, R') & \text{if } [M/x]^r_A(R) = R' \\
 & & & \text{and } \sigma' = [M/x]^s_A(\sigma) \\
[M/x]^s_A(\sigma\,,\, R) & = & (\sigma'\,;\, M') & \text{if } [M/x]^r_A(R) = M' : A' \\
 & & & \text{and } \sigma' = [M/x]^s_A(\sigma) \\
[M/x]^s_A(s[\sigma]) & = & s[\sigma'] & \text{where } \sigma' = [M/x]^s_A(\sigma) \\
[M/x]^s_A(\mathsf{id}_\psi) & = & \mathsf{id}_\psi & \\
[M/x]^s_A(\sigma) & & \text{fails} & \text{otherwise}
\end{array}$$

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.

Theorem 3.3. [Termination]

1. If $[M/x]^r_A(R) = M' : A'$ then $A' \leq A$
2. $[M/x]^*_A(\_)$ terminates, either by returning a result or failing after a finite number of steps.

Theorem 3.4. [Hereditary Substitution Principles]
If $\Delta; \Psi \vdash M \Leftarrow A$ and $\Delta; \Psi, x{:}A, \Psi' \vdash J$ then
$\Delta; \Psi, \Psi' \vdash [M/x]^*_A(J)$ where $* = \{n, r, s\}$.

Building on the discussed ideas and extending (Nanevski et al. 2006), we can define also simultaneous substitution $[\sigma]^n_\Psi(M)$ ($[\sigma]^r_\Psi(R)$, $[\sigma]^s_\Psi(\sigma)$ resp.).

Contextual substitution for contextual variables

Substitutions for contextual variables are a little more difficult. We discuss next the three kinds of contextual variables, meta-variables u, parameter-variables p, and substitution variables s.

Contextual substitution for meta-variables 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 which has type $A[\Psi]$ with a normal term M if M has type A in the context Ψ. Because of α-conversion, the variables that are substituted at different occurrences of u may be different, and we write the contextual substitution as $[\![\hat{\Psi}.M/u]\!]^n_{A[\Psi]}(N)$, $[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}(R)$, and $[\![\hat{\Psi}.M/u]\!]^s_{A[\Psi]}(\sigma)$, where $\hat{\Psi}$ binds all free variables in M. This complication can be eliminated in an implementation of our calculus based on de Bruijn indexes. The typing annotation $A[\Psi]$ is necessary since we apply the substitution σ hereditarily once we know which term u represents, and hereditary substitution requires this information to ensure termination. In defining the substitution we must pay attention that normal forms are preserved. We show contextual substitution into data-level terms next. For better readability, we will write a as an abbreviation for the type $A[\Psi]$.

Data-level normal terms

$$\begin{array}{lcll}
[\![\hat{\Psi}.M/u]\!]^n_a(\lambda y.\, N) & = & \lambda y.\, N' & \text{where } [\![\hat{\Psi}.M/u]\!]^n_a N = N' \\
[\![\hat{\Psi}.M/u]\!]^n_a(N_1, N_2) & = & (N_1', N_2') & \text{where } [\![\hat{\Psi}.M/u]\!]^n_a(N_1) = N_1' \\
& & & \text{and } [\![\hat{\Psi}.M/u]\!]^n_a(N_2) = N_2' \\
[\![\hat{\Psi}.M/u]\!]^n_a(R) & = & R' & \text{where } [\![\hat{\Psi}.M/u]\!]^r_a(R) = R' \\
[\![\hat{\Psi}.M/u]\!]^n_a(R) & = & M' & \text{where } [\![\hat{\Psi}.M/u]\!]^r_a(R) = M' : A \\
[\![\hat{\Psi}.M/u]\!]^n_a(N) & & \text{fails} & \text{otherwise}
\end{array}$$

Data-level neutral terms

$$\begin{array}{lcll}
[\![\hat{\Psi}.M/u]\!]^r_a(x) & = & x & \\
[\![\hat{\Psi}.M/u]\!]^r_a(c) & = & c & \\
[\![\hat{\Psi}.M/u]\!]^r_a(u[\sigma]) & = & N : A & \text{where } [\![\hat{\Psi}.M/u]\!]^s_a \sigma = \sigma' \\
& & & \text{and } [\sigma'/\Psi]^n_{\Psi} M = N \\
[\![\hat{\Psi}.M/u]\!]^r_a(u'[\sigma]) & = & u'[\sigma'] & \text{where } [\![\hat{\Psi}.M/u]\!]^s_a \sigma = \sigma' \\
& & & \text{choosing } u' \neq u \\
[\![\hat{\Psi}.M/u]\!]^r_a(p[\sigma]) & = & p[\sigma'] & \text{where } [\![\hat{\Psi}.M/u]\!]^s_a \sigma = \sigma' \\
[\![\hat{\Psi}.M/u]\!]^r_a(R\ N) & = & (R'\ N') & \text{where } [\![\hat{\Psi}.M/u]\!]^r_a R = R' \\
& & & \text{and } [\![\hat{\Psi}.M/u]\!]^n_a(N) = N' \\
[\![\hat{\Psi}.M/u]\!]^r_a(R\ N) & = & M' : A_2 & \\
\multicolumn{4}{r}{\text{if } [\![\hat{\Psi}.M/u]\!]^r_a R = \lambda x.\, M_0 : A_1 \to A_2 \text{ for } A_1 \to A_2 \leq A[\Psi]} \\
\multicolumn{4}{r}{\text{and } [\![\hat{\Psi}.M/u]\!]^n_a(N) = N' \text{ and } [N'/x]^n_{A_1}(M_0) = M'} \\
[\![\hat{\Psi}.M/u]\!]^r_a(\mathsf{proj}_i R) & = & \mathsf{proj}_i R' & \text{if } [\![\hat{\Psi}.M/u]\!]^r_a(R) = R' \\
[\![\hat{\Psi}.M/u]\!]^r_a(\mathsf{proj}_i R) & = & M_i : A_i & \\
\multicolumn{4}{r}{\text{if } [\![\hat{\Psi}.M/u]\!]^r_a(R) = (M_1, M_2) : A_1 \times A_2} \\
[\![\hat{\Psi}.M/u]\!]^r_a(R) & & \text{fails} & \text{otherwise}
\end{array}$$

Applying $[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\hat{\Psi}.M/u]\!]^s_{A[\Psi]}\sigma$, but instead of returning $M[\sigma']$, it proceeds to eagerly apply σ' to M. However before we apply σ' to M we recover its domain by $[\sigma'/\Psi]$. To enforce that we always return a normal object as a result of contextual substitution, we carry the type of the meta-variable u which will be replaced. In the case where we would possibly obtain a non-normal object, we resort to ordinary hereditary substitution in order to guarantee that the final result of contextual substitution is in normal form.

Contextual substitution for parameter variables Contextual substitution for parameter variables follows similar principles, but it substitutes an ordinary variable for a parameter variable. This 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 a normal term if it is of atomic type. In the case where we encounter a parameter variable $p[\sigma]$, we replace p with the ordinary variable x and apply the substitution $[\![\hat{\Psi}.x/p]\!]$ to σ. This may yield a normal term, and hence we must again ensure that our contextual substitution for parameter variables preserves normal forms. We only show here the case where substitution actually happens and again write a as an abbreviation for $A[\Psi]$.

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

The distinction between parameter variables and meta-variables is mainly interesting from an operational point of view.

Substitution for substitution variables in data-level terms Finally, we give a brief definition for substituting for substitution variables. To ensure it works correctly with the previously defined substitution operations, we also annotate it with the type of the substitution variable. We will abbreviate $\Phi[\Psi]$ as c, and concentrate on the case for substitution.

$$\begin{array}{lcll}
[\![\hat{\Psi}.\sigma/s]\!]^s_c(\cdot) & = & \cdot & \\
[\![\hat{\Psi}.\sigma/s]\!]^s_c(\rho\,;\, N) & = & (\rho'\,;\, N') & \text{if } [\![\hat{\Psi}.\sigma/s]\!]^s_c \rho = \rho' \\
& & & \text{and } [\![\hat{\Psi}.\sigma/s]\!]^n_c N = N' \\
[\![\hat{\Psi}.\sigma/s]\!]^s_c(\rho\,,\, R) & = & (\rho'\,,\, R') & \text{if } [\![\hat{\Psi}.\sigma/s]\!]^s_c \rho = \rho' \\
& & & \text{and } [\![\hat{\Psi}.\sigma/s]\!]^r_c R = R' \\
[\![\hat{\Psi}.\sigma/s]\!]^s_c(s[\rho]) & = & \sigma' & \text{if } ([\![\hat{\Psi}.\sigma/s]\!]^s_c \rho) = \rho' \\
& & & \text{and } [\rho'/\Psi]^s_{\Psi} \sigma = \sigma' \text{ and } c = \Phi[\Psi] \\
[\![\hat{\Psi}.\sigma/s]\!]^s_c(s'[\rho]) & = & s'[\rho'] & \text{if } [\![\hat{\Psi}.\sigma/s]\!]^s_c \rho = \rho' \\
[\![\hat{\Psi}.\sigma/s]\!]^s_c(\mathsf{id}_\phi) & = & \mathsf{id}_\phi & \\
[\![\hat{\Psi}.\sigma/s]\!]^s_c(\rho) & & \text{fails} & \text{otherwise}
\end{array}$$

Applying $[\![\hat{\Psi}.\sigma/s]\!]^s_{\Phi[\Psi]}$ to the closure $s[\rho]$ first obtains the simultaneous substitution $\rho' = [\![\hat{\Psi}.\sigma/s]\!]^s_{\Phi[\Psi]}\rho$, but instead of returning $\sigma[\rho']$, it proceeds to eagerly apply ρ' to σ.

Theorem 3.5. [Termination] $[\![\hat{\Psi}.M/u]\!]^*_{A[\Phi]}(\_)$, $[\![\hat{\Psi}.x/p]\!]^*_{A[\Phi]}(\_)$ and $[\![\hat{\Psi}.\sigma]\!]^*_{\Phi[\Psi]}(\_)$ terminate, either by returning a result or failing after a finite number of steps.

Theorem 3.6. [Contextual Substitution Principles]

1. If $\Delta_1; \Phi \vdash M \Leftarrow A$ and $\Delta_1, u{::}A[\Phi], \Delta_2; \Psi \vdash J$ then $\Delta_1, \Delta_2; \Psi \vdash [\![\hat{\Phi}.M/u]\!]^*_{A[\Phi]} J$ where $* = \{n, r, s\}$.
2. If $\Delta_1; \Phi \vdash x \Rightarrow A$ and $\Delta_1, p{::}A[\Phi], \Delta_2; \Psi \vdash J$ then $\Delta_1, \Delta_2; \Psi \vdash [\![\hat{\Phi}.x/p]\!]^*_{A[\Phi]} J$ where $* = \{n, r, s\}$.
3. If $\Delta; \Phi \vdash \sigma \Leftarrow \Psi_1$ and $\Delta_1, s{::}\Psi_1[\Phi], \Delta_2; \Psi \vdash J$ then $\Delta_1, \Delta_2; \Psi \vdash [\![\hat{\Phi}.\sigma/s]\!]^*_{\Psi_1[\Phi]} J$ where $* = \{n, r, s\}$.

3.3 Simultaneous contextual substitution

Often it is convenient to allow simultaneous contextual substitution θ. Typing rules for simultaneous contextual substitutions can be defined via the judgment $\Delta' \vdash \theta \Leftarrow \Delta$, where Δ denotes the domain of the contextual substitution θ and Δ' describes its range. Just as we annotate the contextual substitution $[\![\hat{\Psi}.M/u]\!]^*_{A[\Psi]}$ with the type of the meta-variable u, we annotate the simultaneous contextual substitution θ with its domain Δ. This is necessary because when applying the substitution $\theta = (\theta_1, \hat{\Psi}.M/u, \theta_2)$ to a closure $u[\sigma]$, we instantiate u with a term M and then apply to it the substitution $[\![\theta]\!]\sigma$. Since the ordinary simultaneous substitution operation is annotated with its domain, we annotate also the simultaneous contextual substitution with its domain.

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

$$\frac{\Delta' \vdash \theta \Leftarrow \Delta \quad x{:}A \in \Psi}{\Delta' \vdash (\theta, \hat{\Psi}.x/p) \Leftarrow \Delta, p{::}A[\Psi]} \quad \frac{\Delta' \vdash \theta \Leftarrow \Delta \quad \Delta'; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta' \vdash (\theta, \hat{\Psi}.\sigma/s) \Leftarrow \Delta, s{::}\Phi[\Psi]}$$

Definition of the simultaneous contextual substitution is a straightforward extension of the previous substitution operation.

Theorem 3.7. [Simultaneous contextual substitution]
If $\Delta' \vdash \theta \Leftarrow \Delta$ and $\Delta; \Psi \vdash J$ then $\Delta'; \Psi \vdash [\![\theta]\!]^*_\Delta J$ where $* = \{n, r, s\}$.

3.4 Linear higher-order pattern matching

Data-level terms represent our data which we analyze via pattern matching in computations. In this section, we describe a pattern matching algorithm for linear higher-order patterns. Linear higher-order patterns (Pientka and Pfennning 2003) are data-level terms where we impose the following two restrictions on contextual variables: First, contextual variables must occur uniquely. Second, they are applied to *all bound variables* in whose scope they occur. In our setting, this means the postponed substitution σ associated with a contextual variable must be a substitution which maps all variables to distinct variables. We will write π for the substitution which has domain and range Ψ. The identity substitution is the simplest form of π, but in general we can allow permutations of variables. As mentioned earlier, linear higher-order patterns refine the notion of higher-order patterns as identified by Miller (Miller 1991), and eliminate expensive checks for bound variable dependencies. For the theoretical development, we enforce that every meta-variable must be of base type. This can always be achieved by lowering. Pattern matching for data-level terms is then defined as follows:

$\Delta; \hat{\Psi} \vdash M_1 \doteq M_2/\theta$	Ground term M_2 matches M_1
$\Delta; \hat{\Psi} \vdash R_1 \div R_2/\theta$	Ground term R_2 matches R_1
$\Delta; \hat{\Psi} \vdash \sigma_1 \doteq \sigma_2/\theta$	Ground substitution σ_1 matches σ_2

In the judgments describing matching we will keep a context $\hat{\Psi}$ which describes the ordinary variables occurring in M_1 and M_2 and R_1 and R_2 respectively. Let Δ describe the meta-variables, parameter variables and substitution variables. Only M_1, R_1, and σ_1 may contain contextual variables which we will instantiate via matching. The result of matching M_2 against M_1 will be a contextual simultaneous substitution θ for all the contextual variables in M_1, s.t. $[\![\theta]\!]^n_\Delta M_1 = M_2$ and $[\![\theta]\!]^r_\Delta R_1 = R_2$.

Matching normal objects

$$\frac{\Delta; \hat{\Psi}, x \vdash M \doteq N/\theta}{\Delta; \hat{\Psi} \vdash \lambda x.\, M \doteq \lambda x.\, N/\theta} \quad \frac{\Delta; \hat{\Psi} \vdash R \div R'/\theta}{\Delta; \hat{\Psi} \vdash R \doteq R'/\theta}$$

$$\frac{\Delta_1; \hat{\Psi} \vdash M_1 \doteq M_2/\theta_1 \quad \Delta_2; \hat{\Psi} \vdash N_1 \doteq N_2/\theta_2}{\Delta_1, \Delta_2; \hat{\Psi} \vdash (M_1\,,\; N_1) \doteq (M_2\,,\; N_2)/(\theta_1, \theta_2))}$$

Matching neutral objects

$$\frac{}{\cdot; \hat{\Psi} \vdash x \div x/\cdot} \quad \frac{}{\cdot; \hat{\Psi} \vdash c \div c/\cdot}$$

$$\frac{}{u{::}P[\Psi']; \hat{\Psi} \vdash u[\pi] \div R/(\hat{\Psi}'.[\pi]^{-1}R/u)}$$

$$\frac{}{p{::}A[\Psi']; \hat{\Psi} \vdash p[\pi] \div x/(\hat{\Psi}'.[\pi]^{-1}x/p)}$$

$$\frac{\Delta; \hat{\Psi} \vdash R \div R'/\theta}{\Delta; \hat{\Psi} \vdash \mathsf{proj}_i R \div \mathsf{proj}_i R'/\theta}$$

$$\frac{\Delta_1; \hat{\Psi} \vdash R_1 \div R_2/\theta_1 \quad \Delta_2; \hat{\Psi} \vdash N_1 \doteq N_2/\theta_2}{\Delta_1, \Delta_2; \hat{\Psi} \vdash R_1\ N_1 \div R_2\ N_2/(\theta_1, \theta_2)}$$

This matching algorithm extends ideas on higher-order pattern unification (Pientka and Pfennning 2003; Pientka 2003) to handle parameter variables and substitution variables. Note that we will only match a term against another if both have the same type. The interesting cases are when we match a neutral term R against a contextual modal variable $u[\pi]$, a substitution σ against a substitution variable $s[\pi]$, and a parameter x against a parameter variable $p[\pi]$. We first consider matching a neutral term R against the meta-variable $u[\pi]$. Since we require that all meta-variables u are applied to *all bound variables* whose scope they occur in, we can simply apply the inverse substitution $[\pi]^{-1}$ to the object R. If π is a substitution with domain Ψ' and range Ψ then $[\pi]^{-1}$ is the substitution with domain Ψ and range Ψ'. Since we require that Ψ' is a permutation of Ψ, applying the inverse substitution $[\pi]^{-1}$ to R will only rename variables and must always succeed since all variables possibly occurring in R are in the range of π. If π is the identity substitution then we have $[\pi]^{-1}R = R$. We note that the algorithm above is strikingly similar to first-order matching algorithms. If one would like to allow the full pattern fragment where meta-variables are only required to be applied to *some bound variables*, then we must check whether applying the inverse substitution to the object R does in fact exist. This requires a traversal of R. For a more detailed discussion we refer the reader to (Pientka 2003).

The rules for matching against substitutions are straightforward. We note that we do not consider the case of matching against the identity substitution id_ψ since our operational semantics will enforce that all context variables have been instantiated and therefore all identity substitutions have been unrolled.

Matching substitutions

$$\frac{}{\cdot; \hat{\Psi} \vdash \cdot \doteq \cdot/\cdot} \quad \frac{}{s{::}\Phi[\Psi]; \hat{\Psi} \vdash s[\pi] \doteq \rho/\hat{\Psi}.[\pi]^{-1}\rho/s}$$

$$\frac{\Delta_1; \hat{\Psi} \vdash \sigma \doteq \rho/\theta_1 \quad \Delta_2; \hat{\Psi} \vdash M \doteq N/\theta_2}{\Delta_1, \Delta_2; \hat{\Psi} \vdash (\sigma\,;\, M) \doteq (\rho\,;\, N)/(\theta_1, \theta_2)}$$

$$\frac{\Delta_1; \hat{\Psi} \vdash \sigma \doteq \rho/\theta_1 \quad \Delta_2; \hat{\Psi} \vdash R \doteq R'/\theta_2}{\Delta_1, \Delta_2; \hat{\Psi} \vdash (\sigma\,,\, R) \doteq (\rho\,,\, R')/(\theta_1, \theta_2)}$$

Soundness of linear higher-order pattern matching ensures that if an object N pattern matches against an object M then $[\![\theta]\!]^n_\Delta(M) = N$. We require that M is indeed well-typed and is a linear higher-order pattern, i.e. all contextual variables occur uniquely and are applied to all bound variables whose scope they occur in, which is described by the judgment $\Delta; \Psi \overset{l}{\vdash} J$.

Theorem 3.8. [Soundness of linear higher-order pattern matching]

1. If $\Delta; \Psi \overset{l}{\vdash} M \Leftarrow A$ and $\cdot; \Psi \vdash N \Leftarrow A$ and $\Delta; \hat{\Psi} \vdash M \doteq N/\theta$ then $\cdot \vdash \theta \Leftarrow \Delta$ and $[\![\theta]\!]^n_\Delta M = N$.

2. If $\Delta; \Psi \vdash^{l} R \Rightarrow A$ and $\cdot; \Psi \vdash R' \Rightarrow A'$ and $A = A'$, $\Delta; \hat{\Psi} \vdash R \doteqdot R'/\theta$ then $\cdot \vdash \theta \Leftarrow \Delta$ and $[\![\theta]\!]^{r}_{\Delta} R = R'$.
3. If $\Delta; \Psi \vdash^{l} \sigma \Leftarrow \Phi$ and $\cdot; \Psi \vdash \rho \Leftarrow \Phi$ and $\Delta; \hat{\Psi} \vdash \sigma \doteq \rho/\theta$ then $\cdot \vdash \theta \Leftarrow \Delta$ and $[\![\theta]\!]^{s}_{\Delta} \sigma = \rho$.

Proof. By structural induction on the matching judgment. □

Theorem 3.9. [Completeness of higher-order pattern matching]

1. If $\Delta; \Psi \vdash^{l} M \Leftarrow A$ and $\cdot; \Psi \vdash N \Leftarrow A$ and $[\![\theta]\!]^{n}_{\delta} M = N$ and $\cdot \vdash \theta \Leftarrow \Delta$ then $\Delta; \hat{\Psi} \vdash M \doteq N/\theta$.
2. If $\Delta; \Psi \vdash^{l} R \Rightarrow A$ and $\cdot; \Psi \vdash R' \Rightarrow A'$, and $A = A'$, and $\cdot \vdash \theta \Leftarrow \Delta$ and $[\![\theta]\!]^{r}_{\delta} R = R'$, then $\Delta; \hat{\Psi} \vdash R \doteqdot R'/\theta$.
3. If $\Delta; \Psi \vdash \sigma \Leftarrow \Phi$ and $\cdot; \Psi \vdash \rho \Leftarrow \Phi$ and $\cdot \vdash \theta \Leftarrow \Delta$ and $[\![\theta]\!]^{s}_{\delta} \sigma = \rho$ then $\Delta; \hat{\Psi} \vdash \sigma \doteq \rho/\theta$.

Proof. Induction on M, R and σ respectively. □

4. Computation-level expressions

Our goal is to cleanly separate the object level and the computation level. While the object level describes data, the computation level describes the programs which operate on data. Computation-level types may refer to data-level types via the contextual type $A[\Psi]$ which denotes an object of type A which may contain the variables specified in Ψ. To allow quantification over context variables ψ, we introduce a dependent type $\Pi\psi{:}W.\tau$ where W denotes a context schema and context abstraction via $\Lambda\psi.e$. We overload the $\rightarrow$ which is used to denote function types at the object level as well as the computation level. However, it should be obvious from the usage which one we mean.

Types	$\tau ::=$	$A[\Psi] \mid \Phi[\Psi] \mid \tau_1 \rightarrow \tau_2 \mid \Pi\psi{::}W.\tau$
Expressions	$e ::=$	$y \mid \mathsf{rec}\ f.e \mid \mathsf{fn}\ y.e \mid \Lambda\psi.e \mid e_1\ e_2 \mid$
		$\mathsf{box}(\hat{\Psi}.\, M) \mid \mathsf{sbox}(\hat{\Psi}.\, \sigma) \mid e\ \lceil\Psi\rceil \mid$
		$(e : \tau) \mid (\mathsf{case}\ e\ \mathsf{of}\ b_1 \mid \ldots \mid b_n\ )$
Branch	$b ::=$	$\mathsf{box}(\hat{\Psi}.\, M) \mapsto e \mid \mathsf{sbox}(\hat{\Psi}.\, \sigma) \mapsto e \mid$
		$\Pi p{::}A[\Psi].b \mid \Pi u{::}P[\Psi].b \mid \Pi s{::}\Phi[\Psi].b$
Contexts	$\Gamma ::=$	$\cdot \mid \Gamma, y{:}\tau$

Data can be injected into programs via the box-construct $\mathsf{box}(\hat{\Psi}.\, M)$. Sine we do not need types inside objects, we write $\hat{\Psi}$ for a list of variables $x_1, \ldots, x_n$ which we think of as a context Ψ without types. Here M denotes an data-level term which has type A in the context Ψ. Annotating the box-construct with $\hat{\Psi}$, i.e. the list of variables occurring in M, is necessary due to α-conversion, and some renaming of bound variables may be necessary to bring the variables in $A[\Psi]$ in accordance with the variables in $\hat{\Psi}.M$. We keep Ψ as a proper context since in the dependent type case A may depend on it. It is worth pointing out that in an implementation with de Bruijn indices this complication can be eliminated.

Similarly, we can inject substitutions $\mathsf{sbox}(\hat{\Phi}.\, \sigma)$ which are of type $\Psi[\Phi]$ where Ψ is the domain of the substitution σ and Φ is its range. Due to α-conversion issues we list the variables occurring in the range of the substitution. Since substitutions can be viewed as pairs between variables and data-level terms, this facility essentially allows us to model explicit environments. Finally, we allow pattern matching on data-level terms via case-expression. In our case-expression, we explicitly abstract over contextual variables which occur in the pattern using the Π-quantifier, however this prefix can always be reconstructed. This will simplify our meta-theoretic development.

4.1 Typing rules for computation level

Next, we present bi-directional typing rules for programs which will minimize the amount of typing annotations. We distinguish here between typing of expressions and branches. In the typing judgment, we will distinguish between the context Ω for context variables, the context for contextual variables Δ, and the context Γ which includes declarations of computation-level variables. Context variables will be introduced via context abstraction. The contextual variables in Δ are introduced in the branch of a case-expression, and computation-level variables in Γ are introduced by recursion or functions.

$\Omega; \Delta; \Gamma \vdash e \Leftarrow \tau$	check an expression e against τ
$\Omega; \Delta; \Gamma \vdash e \Rightarrow \tau$	synthesize τ for expression e
$\Omega; \Delta; \Gamma \vdash b : \tau' \Leftarrow \tau$	branch b checks against $\tau' \Leftarrow \tau$

Branches b are of the form $\Pi\Delta'.\mathsf{box}(\hat{\Psi}.\, M) \mapsto e$, where Δ' contains all the contextual variables occurring in $\mathsf{box}(\hat{\Psi}.\, M)$. The judgment for checking branches then stipulates that the guard $\mathsf{box}(\hat{\Psi}.\, M)$ checks against τ' and the expression e checks against the type τ. The typing rules for expressions are next.

Expressions

$$\frac{\Omega, \psi{:}W; \Delta; \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \Lambda\psi.e \Leftarrow \Pi\psi{:}W.\tau} \qquad \frac{\Omega; \Delta; \Gamma, f{:}\tau \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \mathsf{rec}\ f.e \Leftarrow \tau}$$

$$\frac{\Omega; \Delta; \Gamma, y{:}\tau_1 \vdash e \Leftarrow \tau_2}{\Omega; \Delta; \Gamma \vdash \mathsf{fn}\ y.e \Leftarrow \tau_1 \rightarrow \tau_2}$$

$$\frac{\Omega; \Delta; \Psi \vdash M \Leftarrow A}{\Omega; \Delta; \Gamma \vdash \mathsf{box}(\hat{\Psi}.\, M) \Leftarrow A[\Psi]} \qquad \frac{\Omega; \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Omega; \Delta; \Gamma \vdash \mathsf{sbox}(\hat{\Psi}.\, \sigma) \Leftarrow \Phi[\Psi]}$$

$$\frac{\Omega; \Delta; \Gamma \vdash e \Rightarrow A[\Psi] \quad \text{for all } i\ \Omega; \Delta; \Gamma \vdash b_i : A[\Psi] \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \ \Leftarrow \tau}$$

$$\frac{\Omega; \Delta; \Gamma \vdash e \Rightarrow \Phi[\Psi] \quad \text{for all } i\ \Omega; \Delta; \Gamma \vdash b_i : \Phi[\Psi] \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \ \Leftarrow \tau}$$

$$\frac{\Omega; \Delta; \Gamma \vdash e \Rightarrow \tau' \quad \tau' = \tau}{\Omega; \Delta; \Gamma \vdash e \Leftarrow \tau} \qquad \frac{\Omega; \Delta; \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash (e : \tau) \Rightarrow \tau}$$

$$\frac{y{:}\tau \in \Gamma}{\Omega; \Delta; \Gamma \vdash y \Rightarrow \tau} \qquad \frac{\Omega; \Delta; \Gamma \vdash e \Rightarrow \Pi\psi{:}W.\tau \quad \Omega \vdash \Psi \Leftarrow W}{\Omega; \Delta; \Gamma \vdash e\ \lceil\Psi\rceil \Rightarrow [\![\Psi/\psi]\!]\tau}$$

$$\frac{\Omega; \Delta; \Gamma \vdash e_1 \Rightarrow \tau_2 \rightarrow \tau \quad \Omega; \Delta; \Gamma \vdash e_2 \Leftarrow \tau_2}{\Omega; \Delta; \Gamma \vdash e_1\, e_2 \Rightarrow \tau}$$

Branches

$$\frac{\Omega; \Delta'; \Psi \vdash^{l} M \Leftarrow A \quad \Omega; (\Delta, \Delta'); \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \Pi\Delta'.\mathsf{box}(\hat{\Psi}.\, M) \mapsto e : A[\Psi] \Leftarrow \tau}$$

$$\frac{\Omega; \Delta'; \Psi \vdash^{l} \sigma \Leftarrow \Phi \quad \Omega; (\Delta, \Delta'); \Gamma \vdash e \Leftarrow \tau}{\Omega; \Delta; \Gamma \vdash \Pi\Delta'.\mathsf{sbox}(\hat{\Psi}.\, \sigma) \mapsto e : \Phi[\Psi] \Leftarrow \tau}$$

We observe the usual bound variable renaming conditions in the rule for function abstraction, recursion, and context abstraction. Context variables are explicitly quantified and bound by $\Lambda\psi.e$. There are a few interesting issues which deserve attention: First the typing rule for $\mathsf{box}(\hat{\Psi}.\, M)$. M denotes a data-level term whose free variables are defined in the context Ψ, i.e. it is closed with respect to a context Ψ. To type $\mathsf{box}(\hat{\Psi}.\, M)$ we switch to data-level typing, and forget about the previous context Γ which only describes assumptions on the computation level. Our typing rules will ensure that all variables occurring in M must have been declared in the context Ψ. Similar reasoning holds for the typing rule for $\mathsf{sbox}(\hat{\Psi}.\, \sigma)$. In both cases, some renaming may be necessary to apply the typing rule to bring the variables in $A[\Psi]$ in accordance with the variables in $\hat{\Psi}.M$. To access data, we provide a case-expression with pattern matching. The intention is to match against the contextual modal variables occurring in the pattern. When type-checking a branch, we appeal to a linear typing judgment $\Omega; \Delta'; \Psi \overset{l}{\vdash} M \Leftarrow A$, which ensures that all contextual variables occur linearly and are higher-order patterns.

Theorem 4.1. [Decidability of Type Checking]
Type-checking computation-level expressions is decidable.

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

Due to space constraints, we omit here the substitution definitions $[e/x]e'$ and extensions of previous substitution operations such as $[\![\Psi/\psi]\!](e)$ and $[\![\theta]\!]_\Delta(e)$. The definitions are mostly straightforward.

5. Operational semantics

In this section, we describe a small-step operational semantics for the presented language. During execution type annotations are unnecessary, and we define evaluation only on expressions where all type annotations have been erased. First, we define the values in this language.

$$\text{Value} \quad v \quad ::= \quad \mathsf{fn}\ y.e \mid \Lambda\psi.e \mid \mathsf{box}(\hat{\Psi}.\, M) \mid \mathsf{sbox}(\hat{\Psi}.\, \sigma)$$

Next, we define a small-step evaluation judgment:

$e \longrightarrow e'$	e evaluates in one step to e'.
$(\mathsf{box}(\hat{\Psi}.\, M) \doteq b) \longrightarrow e'$	Branch b matches $\mathsf{box}(\hat{\Psi}.\, M)$ and steps to e'
$(\mathsf{sbox}(\hat{\Psi}.\, \sigma) \doteq b) \longrightarrow e'$	Branch b matches $\mathsf{sbox}(\hat{\Psi}.\, \sigma)$ and steps to e'

Evaluation relies on pattern matching and evaluating branches. The case for function application is straightforward. Values for program variables are propagated by computation-level substitution. Instantiations for context variables are propagated by applying a concrete context Ψ to a context abstraction $\Lambda\psi.e$.

$$\frac{}{\mathsf{rec}\ f.e \longrightarrow [\mathsf{rec}\ f.e/f]e} \qquad \frac{}{(\mathsf{fn}\ y.e)\ v \longrightarrow [v/y]e}$$

$$\frac{e_1 \longrightarrow e_1'}{(e_1\ e_2) \longrightarrow (e_1'\ e_2)} \qquad \frac{e_2 \longrightarrow e_2'}{(v\ e_2) \longrightarrow (v\ e_2')}$$

$$\frac{e \longrightarrow e'}{e\ \lceil\Psi\rceil \longrightarrow e'\ \lceil\Psi\rceil} \qquad \frac{}{(\Lambda\psi.e)\ [\Psi] \longrightarrow [\![\Psi/\psi]\!]e}$$

$$\frac{e \longrightarrow e'}{(\mathsf{case}\ e\ \mathsf{of}\ b_1 \mid \ldots \mid b_n\ ) \longrightarrow (\mathsf{case}\ e'\ \mathsf{of}\ b_1 \mid \ldots \mid b_n\ )}$$

$$\frac{\cdot \vdash \mathsf{box}(\hat{\Psi}.\, M) \doteq b_i \longrightarrow e'}{(\mathsf{case}\ (\mathsf{box}(\hat{\Psi}.\, M))\ \mathsf{of}\ b_1 \mid \ldots \mid b_n\ ) \longrightarrow e'}$$

$$\frac{\cdot \vdash \mathsf{sbox}(\hat{\Psi}.\, \sigma) \doteq b_i \longrightarrow e'}{(\mathsf{case}\ (\mathsf{sbox}(\hat{\Psi}.\, \sigma))\ \mathsf{of}\ b_1 \mid \ldots \mid b_n\ ) \longrightarrow e'}$$

Evaluation in branches relies on higher-order pattern matching against data-level terms to instantiate the contextual variables occurring in a branch and data-level instantiations are propagated via contextual simultaneous substitution. We assume that $\mathsf{box}(\hat{\Psi}.\, M)$ does not contain any meta-variables, i.e. it is closed.

$$\frac{\Delta; \hat{\Psi} \vdash M' \doteq M/\theta}{(\mathsf{box}(\hat{\Psi}.\, M) \doteq \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M') \mapsto e) \longrightarrow [\![\theta]\!]_\Delta e}$$

$$\frac{\Delta; \hat{\Psi} \vdash \sigma' \doteq \sigma/\theta}{(\mathsf{sbox}(\hat{\Psi}.\, \sigma) \doteq \Pi\Delta.\mathsf{sbox}(\hat{\Psi}.\, \sigma') \mapsto e) \longrightarrow [\![\theta]\!]_\Delta e}$$

Given the current setup, we can prove type safety for our proposed functional language with higher-order abstract syntax and explicit substitutions. We assume that patterns cover all cases here, but coverage checking can be incorporated by following ideas described in (Schürmann and Pfenning 2003). First we state and prove the necessary canonical forms lemma where $|e|$ denotes the expression e where typing annotations in $e : \tau$ are erased.

Lemma 5.1. [Canonical forms]

1. If $|e|$ is a value and $\cdot; \cdot; \cdot \vdash e \Rightarrow A[\Psi]$
then $|e| = \mathsf{box}(\hat{\Psi}.\, M)$ and $\cdot; \cdot; \cdot \vdash \mathsf{box}(\hat{\Psi}.\, M) \Leftarrow A[\Psi]$
2. If $|e|$ is a value and $\cdot; \cdot; \cdot \vdash e \Rightarrow \Phi[\Psi]$
then $|e| = \mathsf{sbox}(\hat{\Psi}.\, \sigma)$ and $\cdot; \cdot; \cdot \vdash \mathsf{sbox}(\hat{\Psi}.\, \sigma) \Leftarrow \Phi[\Psi]$
3. If $|e|$ is a value and $\cdot; \cdot; \cdot \vdash e \Rightarrow \tau_1 \rightarrow \tau_2$
then $|e| = \mathsf{fn}\ y.e'$ and $\cdot; \cdot; \cdot \vdash \mathsf{fn}\ y.e' \Leftarrow \tau_1 \rightarrow \tau_2$
4. If $|e|$ is a value and $\cdot; \cdot; \cdot \vdash e \Rightarrow \Pi\psi{::}W.\tau$
then $|e| = \Lambda\psi.e'$ and $\cdot; \cdot; \cdot \vdash \Lambda\psi.e' \Leftarrow \Pi\psi{::}W.\tau$

Proof. By induction on the typing judgment and case analysis. □

Theorem 5.1. [Preservation and Progress]

1. If $\cdot; \cdot; \cdot \vdash e \Rightarrow \tau$ and e coverage checks then either $|e|$ is a value or there exists an expression e' s.t. $|e| \longrightarrow |e'|$ and $\cdot; \cdot; \cdot \vdash e' \Rightarrow \tau$.
2. If $\cdot; \cdot; \cdot \vdash e \Leftarrow \tau$ and e coverage checks then either $|e|$ is a value or there exists an expression e' s.t. $|e| \longrightarrow |e'|$ and $\cdot; \cdot; \cdot \vdash e' \Leftarrow \tau$.

Proof. By structural induction on the first derivation using canonical forms lemma, correctness of higher-order pattern matching, and various substitution properties we proved earlier. □

6. Related Work

One of the first proposals for functional programming with support for binders and higher-order abstract syntax was presented by Miller (1990). Later, Despeyroux et al. (1997) developed a type-theoretic foundation for programming which supports primitive recursion. To separate data from computation, they introduce modal types $\Box A$ which can be injected into computation. However, data in their work must always be closed and can only be analyzed via a primitive recursive iterator. Our work essentially continues the path set out in (Despeyroux et al. 1997), and generalizes their work to allow for open data-objects and first-class substitutions.

Closely related to our approach is the work by Schürmann et al. (2005) where the authors present the ∇-calculus which provides a foundation for programming with higher-order abstract syntax as found in the simply typed *Elphin* language. In Schürmann et al. (2005) the necessity modality $\Box\tau$ describes *computation* of type τ which can be executed in every world where we have at least one context extension. The context containing binding occurrences is hence left implicit and associated with a computation. When a new context extension is introduced, the computation moves from the present world to a world where the context is extended. To return the computation to the present world again where the context extension is not present, the programmer has to explicitly use a pop-operation which removes the introduced binding. The fact that binders can never escape their scope during execution is a meta-theoretic property which is proven. Since $\Box\tau$ describes computation which may be executed in every world where we have at least one context extension, the whole function and all its arguments must be well-typed in this extended context.

This is in stark contrast to our work based on *contextual modal type*, where $A[\Psi]$ denotes some object of type A which is well-typed in every world where we have a context Ψ and hence every data-object carries its own local context. Therefore we can for example return an object which is closed and distinguish it from an object which is not closed. The fact that local binders can never escape their scope is an inherent property of contextual types $A[\Psi]$. Ψ denotes exactly the bound variables which are allowed to occur in an object of type $A[\Psi]$, and the type system will detect if this is violated. *Elphin* seems a special case of our foundation where there is only one context variable which all arguments depend on. Because we can distinguish between different contexts, we believe our foundation is more expressive and is likely to scale better when we compose different computations, since it provides more fine-grained control. Finally, we propose to extend the framework with first-class substitutions, which seem interesting on their own and are absent from *Elphin*. Although their full impact still needs to be explored, explicit substitutions have been used to model records, closures, modules, classes and abstract data types with one single versatile and powerful construct. The lack of first-class substitutions generally forces the inclusion of several different name-space mechanisms.

Most recently, Poswolsky and Schürmann (2007) proposed a dependently typed language for programming with higher-order encodings. While substantially different from *Elphin*, it also does not provide fine-grained control to distinguish between multiple different contexts and does not support first-class substitutions.

A different more pragmatic approach to allow manipulation of binding structures is pursued in nominal type systems which serve as a foundation of FreshML (Shinwell et al. 2003). In this approach names and α-renaming are supported but implementing substitution is left to the user. The type system distinguishes at the type-level between expressions and names, and provides a special type `atom` which is inhabited by all names. The distinction between different categories of names is usually more difficult. Generation of a new name and binding names are separate operations which means it is possible to generate data which contains accidentally unbound names since fresh name generation is an observable side effect. To address this problem, Pottier (2007) describes pure FreshML where we can reason about the set of names occurring in an expression via a Hoare-style proof system. The system relies on assertions written by the programmer to reason about the scope of names. This static-analysis approach is quite expressive since the language of constraints includes subset relations, equality, intersection etc. In contrast, our work aims to provide a type-theoretic understanding of open data, binders, and substitutions. This has various benefits: For example, it should be possible to provide precise error messages on where names escape their scope. In contrast to nominal systems, our foundation also leaves flexibility as to how data-level bound variables are implemented and in fact lends itself to an implementation based on de Bruijn indices. While the exact relationship between nominal types, static analysis of names and contextual types still needs to be investigated, we believe the presented work is a start towards comparing both approaches.

From a more theoretical perspective, various λ-calculi supporting contexts as a primitive programming construct have been considered (Sato et al. 2001, 2002; Nishizaki 2000; Mason 1999; Hashimoto and Ohori 2001). Nishizaki (2000) for example extends a lambda-calculus with explicit substitutions in the spirit of the explicit substitution calculus proposed by Abadi et al. (1990). However, unlike Abadi's work, the author proposes a polymorphic calculus where we can quantify over explicit substitutions. This work crucially relies on de Bruijn indices. Although the use of de Bruijn indices is useful in an implementation, nameless representation of variables via de Bruijn indices are usually hard to read and critical principles are obfuscated by the technical notation.

Sato et al. (2002) introduce a simply typed λ-calculus which has both contexts and environments (=substitutions) as first-class values, called $\lambda_{\kappa,\epsilon}$-calculus. There are many distinctions, however, between their work and the contextual modal type theory we propose as a foundation. Most of these differences arise because the former is not based on modal logic. For example, they do not allow α-conversion on open objects and they do not require open objects to be well-formed with respect to a local context. Moreover, they do not cleanly distinguish between meta-variables and ordinary variables. All these restrictions together make their system quite heavy, and requires fancy substitution operations and levels attached to ordinary variables to maintain decidability and confluence. None of these approaches allows for pattern matching and recursion on open data-objects.

7. Conclusion

We have presented a type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions based on contextual modal types together with its type preservation and progress proof. We support recursion over data defined in HOAS-style, and allow pattern matching against open data and variables. By design bound variables in data cannot escape their scope. This is to our knowledge the first type-theoretic proposal to achieve this. This work also provides interesting insights into the relationship between nominal systems and higher-order abstract syntax. In the future we plan to address the following questions:

Existential quantification over context variables In some examples, we would like to be able to write a function which returns an open object of type exp[ψ'] for some context ψ'. This is for example necessary if we want to write a small-step environment based interpreter, or even if we want to translate a name-based representation of terms into a higher-order representation. Adding existential quantification for context variables is in fact possible and straightforward.

$$\begin{array}{lll}
\text{Types} & \tau ::= & \ldots \mid \Sigma\psi{::}W.\tau \\
\text{Expressions} & e ::= & \mathsf{pack}(\Psi, e) \mid \mathsf{let\ pack}(\psi, x) = e \ \mathsf{in}\ e' \ \mathsf{end}
\end{array}$$

The typing and evaluation rules are then straightforward.

Typing rules

$$\frac{\Omega;\Delta;\Gamma \vdash e \Leftarrow [\![\Psi/\psi]\!]\tau \quad \Omega \vdash \Psi \Leftarrow W}{\Omega;\Delta;\Gamma \vdash \mathsf{pack}(\Psi, e) \Leftarrow \Sigma\psi{::}W.\tau}$$

$$\frac{\Omega;\Delta;\Gamma \vdash e \Rightarrow \Sigma\psi{::}W.\tau \quad \Omega,\psi{::}W;\Delta;\Gamma, x:\tau \vdash e' \Rightarrow \tau'}{\Omega;\Delta;\Gamma \vdash \mathsf{let\ pack}(\psi, x) = e\ \mathsf{in}\ e'\ \mathsf{end} \Rightarrow \tau'}$$

Evaluation rules

$$\frac{}{\mathsf{let\ pack}(\psi, x) = \mathsf{pack}(\Psi, e)\ \mathsf{in}\ e'\ \mathsf{end} \longrightarrow [e/x][\![\Psi/\psi]\!]e'}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{let\ pack}(\psi, x) = e_1\ \mathsf{in}\ e'\ \mathsf{end} \longrightarrow \mathsf{let\ pack}(\psi, x) = e_2\ \mathsf{in}\ e'\ \mathsf{end}}$$

The full impact of existential types in practice however still needs to be addressed.

Mixing data with computations At the moment our foundation for programming with HOAS and substitutions is pure, i.e. we never mix data and computation. However, there may be good reasons to allow some form of computation inside of data-definitions. For example, consider the definition of arithmetic expressions. We defined natural numbers inductively, and then included them in the expressions via the coercion `Nat`. However, clearly we may want to use the integer-type given by our functional language when we define the language of arithmetic expressions, and thereby be able to rely on the built-in arithmetic operations instead of redefining them. How to mix data and computation and retain all our good properties is an important question we plan to address in the future. Also the interaction with other features realistic programming languages provide such as mutable state, exceptions, etc. needs to be investigated.

Reconstruction of context variables In the presented foundation we explicitly abstract over context variables and pass them explicitly. An interesting question in practice is whether we can reconstruct some of these context variables and keep contexts implicit. This also leads to the question whether we should support pattern matching against contextual variables. At the moment, context variables will be instantiated to some concrete context via context application before we use pattern matching to decide which branch to pick in a case-expression. This in fact is important to achieve elegant meta-theoretic properties. In the future, we plan to consider in more detail how to reconstruct context variables and how to add matching for context variables.

Acknowledgments

I would like to thank Frank Pfenning, Aleks Nanevski, and Adam Poswolsky for many insightful discussion regarding this topic. Moreover, this work benefited from comments and suggestions by Derek Dreyer, Stefan Monnier, and François Pottier. Special thanks go to Joshua Dunfield for helping to typeset the code.

References

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, Lecture Notes in Computer Science (LNCS) 1210, Springer-Verlag, 1997.

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

Louis-Julien Guillemette and Stefan Monnier. Statically verified type-preserving code transformations in Haskell. In *Programming Languages meets Program Verification (PLPV), Seattle, USA*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2006.

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

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

Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL '06)*, pages 257–268, New York, NY, USA, ACM Press, 2006.

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

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

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

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

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. *ACM Transactions on Computational Logic (accepted, to appear in 2008)*, page 56 pages, 2006.

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

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

Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Sciences, Carnegie Mellon University, 2003. CMU-CS-03-185.

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

Adam Poswolsky and Carsten Schürmann. Programming with higher-order encodings and dependent types. Technical Report YALEU/DCS/TR1375, Department of Computer Science, Yale University, July 2007.

François Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 356–365, Wroclaw, Poland, IEEE Computer Society, 2007.

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), March 2002.

Carsten Schürmann. *Automating the meta theory of deductive systems.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, Available as Technical Report CMU-CS-00-146, 2000.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, Lecture Notes in Computer Science (LNCS) 2758, Springer-Verlag, 2003.

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

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

Geoff Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 2007 (to appear).

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