Functional programming with dependently-typed higher-order data

Brigitte Pientka Joshua Dunfield
McGill University
School of Computer Science
Montreal, Canada
{bpientka,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. The contributions of this paper are twofold: First, we present a syntax-directed bidirectional type system that distinguishes between dependently typed data and computations and locally resolves constraints arising from dependent types using unification. Our foundation is unique in the sense that data objects may be open and encoded in higher-order abstract syntax. In addition, our language supports first-class substitutions, a feature which is absent in the competing proposals. Second, we describe an operational semantics for this language based on higher-order pattern matching for dependently typed objects and prove progress and preservation for our language.

1. Introduction

Over the last decade, various forms of dependent types have found their way into mainstream functional programming languages to allow programmers to express stronger properties about their programs. Generalized algebraic datatypes (GADTs) [22, 2, 25, 13] can index types by other types and have entered mainstream languages such as Haskell. Other approaches, such as DML [26], use indexed types with a fixed constraint domain, such as integers with linear inequalities, for which efficient decision procedures exist.

This paper explores a new point in the design space of functional programming with dependent types. Specifically, we are interested in functional programming with dependently typed higher-order data structures described in the logical framework LF [7]. LF provides a rich meta-language that represents binders in the object language as binders in the meta-language. This technique is called higher-order abstract syntax (HOAS) encoding and supports renaming, fresh name generation, and capture-avoiding substitution. Proofs can be directly encoded as higher-order dependently typed objects, and there is built-in support for applying substitution lemmas. Over the past decade, the strengths of HOAS have been compellingly proven in LF and its implementation in the Twelf system [14].

But combining dependent types and HOAS encodings with functional programming has been plagued with difficulties. HOAS encodings are not inductive in the usual sense, since we must traverse binding structures and recursively analyze and manipulate open data containing binders. How, then, do we know that a variable occurring in data does not extrude its scope? How do we access and manipulate variables in data objects?

It has been said that "[t]he whole HOAS approach by its very nature disallows a feature that we regard of key practical importance: *the ability to manipulate names of bound variables explicitly* in computation and proof" [6]. Thus, nominal approaches [6], which provide first-class names and α-renaming, but not capture-avoiding substitution, have been advocated as a good compromise. More recently, this approach has been extended to support dependent types [19]. However, nominal systems themselves are not free from difficulties. In FreshML [23], for example, name generation and binding are separate operations, and fresh name generation is an observable side effect. Unfortunately, this means that data can accidentally contain unbound names. To address this problem, Pottier [18] proposed *pure* FreshML, in which one reasons about sets of free variables via a Hoare-style proof system.

In this paper, we tackle the problem of combining dependent types and HOAS with functional programming, and show that these concepts can be elegantly incorporated into functional programming using contextual modal type theory [12]. The contextual modality $A[\Psi]$ characterizes data objects of type A in the context Ψ, that is, data objects referring to variables declared in Ψ. Every data object is therefore locally closed. Since we want to allow recursion over open data, and the local context Ψ associated with the type A can grow, our foundation supports *context variables* that abstract over contexts. Consequently, different arguments to a computation can have different local contexts and we can distinguish between closed data of type $A[\cdot]$ and open data of type $A[\Psi]$. By design, variables occurring in open data cannot escape their scope, avoiding a problem common in previous attempts and yielding a name-safe foundation for the operations typical in nominal systems. In addition to supporting binders and α-renaming, we provide capture-avoiding substitution together with first-class substitutions. Our foundation provides direct access to variables and allows us to manipulate and compare them in computations and proofs, providing an essential feature which typically has been difficult to marry with HOAS.

This paper extends the first author's prior work on programming with higher-order abstract syntax in the simply-typed setting [16] to dependent types. The programmer can directly construct and manipulate proofs as dependently typed higher-order data structures. This allows the programmer to directly express why properties about data objects hold. More importantly, our functional language is powerful enough to be a general proof language for implementing, as functions, inductive properties about formal deductive systems. As such, it is an alternative to the approach taken in Twelf, in which metatheory about deductive systems is encoded via relations. Our contribution is twofold: First, we present a syntax-directed bidirectional type system that distinguishes between dependently typed data and computations and locally resolves constraints arising from dependent types using unification. Our foundation is unique in the sense that dependently typed data objects may be open and supports encodings based on higher-order abstract syntax and provides first-class substitutions. This facilitates the encodings of proofs as data. We achieve this by cleanly separating dependently-typed higher-order data objects from computations using contextual modal types.

Second, we describe the operational semantics for this language based on higher-order pattern matching for dependently typed objects and prove progress and preservation for our language, assuming that programs are covering all cases. The formal development of a coverage and termination checker which guarantees the totality of the functions is beyond the scope of this paper.

We believe our calculus is an important step towards understanding syntactic structures with binders and dependent types, and providing direct support for binders in the setting of typed functional programming.

2. Related work

Enriching functional programming with dependently typed higher-order data structures, a longstanding open problem, is presently receiving widespread attention. There are two central challenges: First, we must support higher-order encodings in which objects refer to variables. Second, we must support dependent types, including matching on the index objects of dependent types.

Most closely related to our work is the Delphin language [17], a dependently typed functional programming language which 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. In Delphin, however, 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. Delphin's ∇-quantifier introduces a new parameter in a computation and extends the current context. However, the actual context of dynamic assumptions is kept implicit, while we have explicit context variables. Finally, we provide first-class substitutions.

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, data is always closed and can only be analyzed by a primitive recursive iterator. Despeyroux and Leleu [4, 3] extended this work to dependent types. Our work may be seen as a continuation and extension of theirs, in which we relativize the modal necessity.

Our work shares many of the same goals as the work by Licata and Harper [8] who proposed an extensible theory of indexed domains. However, their current work does not consider full LF, but a framework of abstract binding trees that is expressive enough to describe higher-order abstract syntax, but not judgments. In this sense, their framework lacks the power to manipulate and analyze proofs as data. Technically, their work is quite different from ours and provides no inherent type-theoretic foundation for open objects.

Finally, there have been several proposals in the functional programming community to allow full dependent types but not HOAS encodings. Languages such as Cayenne [1] and Epigram [9] support full dependent types but do not distinguish between dependently typed data and computations.

3. Motivation

To illustrate our approach and motivate the problem, we give two examples: a type-preserving environment-based interpreter and a self-certifying type inference engine.

3.1. Type-preserving evaluator

First, we show an environment-based interpreter for a simple language with natural numbers and **let** expressions. Expressions are indexed with their types to ensure that only well-typed expressions are evaluated. We first define the basic types, in the LF style.

```
tp: type .          nat: tp.
```

Next, we define the object terms, which are indexed by the types `tp`. We write dependent types explicitly, though explicit abstraction over index arguments in the dependent type is not necessary in practice.

```
exp: tp → type .
z   : exp nat.
Suc : exp nat → exp nat.
Add : exp nat → exp nat → exp nat.
Let : Π t1:tp Π t2:tp.
      exp t1 → (exp t1 → exp t2) → exp t2.
```

The expression **let val** $x = 1$ **in** $\mathsf{Add}(0, x)$ **end**, for example, is represented as `Let nat nat (Suc z) (λx.Add z x)`. When we recursively analyze objects, we will analyze `Add z x`, in which `x` is free. The expression `Add z x` has type `exp nat` in the context `x:exp nat`. We express this by the contextual modal type `(exp nat)[x:exp nat]`.

This example illustrates the need to characterize open data objects. Since the context of free variables grows during recursion, we have *context variables* that abstract over concrete contexts. Just as types classify objects, *schemas* classify contexts. Schemas resemble Schürmann's worlds [20], but our type-theoretic foundation makes context abstraction and schemas explicit. In this example, the context $\psi = \mathtt{x}_1\mathtt{:exp\ t}_1, \ldots, \mathtt{x}_n\mathtt{:exp\ t}_n$ is classified by the schema $(\mathsf{all}\,\mathtt{t:tp.\ exp\ t})^*$. All declarations in ψ are instances of the schema where `t` is instantiated with a concrete type $\mathtt{t}_i$.

The interpreter takes as input an expression `e` of type `t` that may have free variables $\mathtt{x}_1\mathtt{:exp\ t}_1, \ldots, \mathtt{x}_n\mathtt{:exp\ t}_n$ together with an environment `r` that maps all free variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ to closed well-typed values such that $\mathtt{r} = v_1/\mathtt{x}_1, \ldots, v_n/\mathtt{x}_n$ and every v_i has type $\mathtt{t}_i$. The result is a closed value of type `t`. The environment that maps all free variables in the context ψ to closed values of the appropriate type is represented as a first-class substitution with domain ψ and empty range; its type is written ψ`[.]`.

```
ctx schema W = all  t:tp . exp t.

rec  eval : Π ψ:(W)*. Π□T:tp[.].
          (exp T[.])[ψ] → ψ[.] → (exp T[.])[.] =
Λ ψ ⇒ λ□T ⇒ fn e ⇒ fn r ⇒ let  sbox S[.] = r in
 case  e of
   box(ψ. z) : (exp nat)[ψ] ⇒ box( . z)

 | Π□U::(exp nat)[ψ] .
    box(ψ. Suc U[id_ψ]) : (exp nat)[ψ]
  ⇒ let  box( . V[.]) =
        eval ⌈ψ⌉ < . nat> box(ψ. U[id_ψ]) r
     in
       box( . Suc V[.])
     end

 | Π□T::tp[.]. Π□p::(exp T[.])[ψ] .
   box(ψ. p[id_ψ]) : (exp T[.])[ψ]
   ⇒ box( . p[S[.]])

 | Π□T1::tp[.].Π□ T2::tp[.].
   Π□W::(exp T1[.])[ψ].
   Π□U::(exp T2[.])[ψ, x:exp T1[.]].
    box(ψ.Let T1[.] T2[.] W[id_ψ]
         (λx. U[id_ψ, x])) : (exp T2[.])[ψ]
 ⇒ let  box( . V[.]) =
      eval ⌈ψ⌉ <. T1[.]> box(ψ.W[id_ψ]) r
    in
      eval ⌈ψ,x:exp T1[.] ⌉ <. T2[.]>
           box(ψ,x. U[id_ψ,x]) (sbox S[.], V[.])
    end
```

In the function, we first introduce the context variable ψ using Λ-abstraction and which binds every occurrence of ψ in the body. Next, we introduce the index variable `T`, followed by the computation-level variable `e` which has type `(exp T[.])[`ψ`]` and which will be analyzed by pattern matching.

We use contextual variables written in capital letters to represent open data. There are two situations which necessitate the use of contextual variables: describing data indexing types, as in `(exp T[.])[`ψ`]`, and supporting pattern matching in branches. In the above program, we explicitly declared all the contextual variables as well as the type of each pattern, reflecting the type-theoretic foundation presented in this paper. We will begin by first discussing contextual variables in patterns.

In patterns, the closure consisting of a contextual variable `U` and a *postponed substitution* characterizes "holes" instantiated using higher-order pattern matching. For example, in **box**`(`ψ`. Suc U[`$\mathbf{id}_\psi$`])`, the closure `U[`$\mathbf{id}_\psi$`]` describes a "hole" that is to be filled. As soon as we know what `U` stands for, we apply the 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 as a list of variables that may occur in the hole. In `U[`$\mathbf{id}_\psi$`]` the contextual variable `U` can be instantiated with an expression containing variables declared in ψ.

Similarly, index objects are characterized by contextual variables. $\Pi^\square$`T:tp[.].`τ introduces a contextual variable `T` denoting a closed index object of type `tp`. In addition to meta-variables `U`, which may be instantiated with an arbitrary data object, we also support *parameter variables* `p`. A parameter variable represents a bound variable and can only be instantiated with a variable from the data level. Similar to meta-variables, they are treated as closures. We use small letters for parameter variables, in contrast to capital letters for meta-variables which can be instantiated with an arbitrary object. Parameter variables allow us to write, in a case expression, a pattern that only matches variables, and allow us to collect and even compare variables.

When we encounter a parameter from the context as in the third case, we can simply apply the substitution `S[.]` to the object `p`. Since the substitution `S[.]` has domain ψ and range empty, applying it to the parameter variable `p` yields a closed object. Because we apply `S[.]` as soon as we know what `p` stands for, the variable occurring in the instantiation for `p` 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` has closed instantiations for all variables in the local context ψ, and applying the substitution `S[.]` to the parameter variable `p` must yield a closed value.

To recursively analyze expressions we have to consider different cases. The cases for zero and successor are straightforward. In the recursive call in the case for successor, we write `eval` $\lceil\psi\rceil$ for context application and `< . nat>` for applying an index argument. To evaluate the **let** expression, we evaluate **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 that associates 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[.]).

3.2. Self-certifying type inference

In this example, we consider the implementation of a simple type inference algorithm which will not only infer the type of a given term but also return its typing derivation. We consider the same expressions as above, but instead of indexing the expressions with their types, we define typing judgments separately. We begin by defining expressions, types and typing derivations.

```
exp: type .
z: exp.        Suc: exp → exp.
Let exp → (exp → exp) → exp.
```

Typing rules are represented as a dependent type which describes the relation between object-level expressions and their types. A typing judgment $\Gamma \vdash t : T$ is translated into a dependent type declaration in LF and a typing derivation concluding $\Gamma \vdash t : T$ is described by an object of type type `oft t T`. We use higher-order abstract syntax in the specification of the typing rule to model hypothetical typing derivations, eliminating need to explicitly manipulate Γ. Every typing rule for $\Gamma \vdash t : T$ is translated into a constant whose type encodes the actual typing rule.

```
oft: exp → tp → type .
o_z: oft z nat.
o_s: Π e:exp. oft e nat → oft (Suc e) nat.
o_l: Π e1:exp. Π t1:tp. Π t2:tp. Π e2:exp→exp.
       oft e1 t1 →
      (Π x:exp. oft x t1 → oft (e2 x) t2)
         → oft (Let e1 λx.e2 x) t2.
```

The function `infer` accepts an `exp` and returns a tuple consisting of its type and the typing derivation. Since we will traverse binding constructs, the function will have to handle open objects, which depend on assumptions about types of variables. The context keeps track of the variable `x:exp` together with its proof, which has the (object-level) type `oft x T`. The following schema classifies exactly those contexts.

```
ctx schema W = all  t:tp. Σ x:exp. oft x t
```

Next, we show the implementation of the function `infer`.

```
rec  infer: Πγ:(W)*. Π□U::exp[γ] .
            Σ□T:tp[.]. (oft U[idγ] T[.])[γ] =
Λ γ ⇒ λ□U ⇒
 case  box(γ. U[idγ]) of
   box(γ. z) : exp[γ] ⇒
     pack< . nat, box(γ. o_z) >

 | Π□E1:exp[γ].Π□ E2:exp[γ, x:exp].
  box(γ. Let E1[idγ] λx.E2[idγ,x]) : exp[γ] ⇒
   let  pack< . T1[.], box(γ.D1[idγ]) >
          = infer [γ] <γ . E1[idγ]>

        pack< . T2[.], box(γ. D2[idγ]) >
          = infer [γ, d:Σ x:exp. oft x S1[.]]
               <γ, d . E2[idγ, proj1 d] >
   in
     pack< . S2[.],
           box(γ. o_l D1[idγ]
                   λx.λu.D2[idγ, <x,u>]) >
   end

 | Π□T::tp[.] Π□p::(Σ x::exp. oft x T[.])[γ]
  box(γ. proj1 p[idγ]) : exp[γ] ⇒
       pack< . T[.], box(γ. proj2 p[idγ])
```

Typically a type inference algorithm is implemented by recursively analyzing expressions. Because the expression is an index object in our function and we cannot directly match against index objects, we apply **box** to lift it to a data object that we can pattern-match. To return a dependent tuple, we use **pack**, which pairs a data object with a computation-level expression. In general, a dependent pair has the form **pack**<Ψ.M,e> where `M` is a data object with free variables Ψ and `e` is a computation-level expression.

In the recursive call for `Let`, we extend the context with the declaration Σ`x:exp.oft x S1[.]`. Our context therefore keeps track not only of parameters occurring in the expression, but also assumptions about their type. Thus, our typing context is implicit, and does not have to be modeled with a list or some other explicit data structure.

In the case for a variable, we use a parameter variable `p` to match against declarations in γ. When matching succeeds, it instantiates not only `p` but also the index argument `T`. We can thus return `T` and the witness **proj**$_2$ p[**id**$_\gamma$].

4. Data-level terms, substitutions, contexts

We begin describing the type-theoretic foundation of our languages by presenting the data layer. We essentially support the full logical framework LF together with Σ-types. Our data layer closely follows contextual modal type theory [12], extended with parameter variables, substitution variables, and context variables [16], and finally with dependent pairs and projections. Perhaps most importantly, we formalize schemas, which classify contexts.

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Atomic types	P	$::=$	$a\ M_1 \dots M_n$
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	$::=$	$\cdot \mid \sigma\,;\,M \mid \sigma\,,\,R$ $\mid s[\sigma] \mid \mathsf{id}_\psi$
Element types	A^*	$::=$	$\Pi x{:}A.A^* \mid a\ N_1 \dots N_n$
Schema elements	F	$::=$	$\mathsf{all}\ \Phi.\Sigma y_1{:}A_1^*, \dots, y_j{:}A_j^*.\ A^*$
Schema sums	S	$::=$	$F_1 + \dots + F_n$
Schemas	W	$::=$	$(S)^*$
Context variables	ψ, ϕ		
Contexts	Ψ, Φ	$::=$	$\cdot \mid \psi \mid \Psi, x{:}A$
Meta-contexts	Δ	$::=$	$\cdot \mid \Delta, u{::}A[\Psi]$ $\mid \Delta, p{::}A[\Psi] \mid \Delta, s{::}\Psi[\Phi]$
Schema contexts	Ω	$::=$	$\cdot \mid \Omega, \psi{::}W$

We only characterize normal terms since only these are meaningful in the logical framework, following Watkins et al. [24] and Nanevski et al. [12]. This is achieved by distinguishing between normal terms M and neutral terms R. While 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, which represent general open objects, *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 computation-level case expressions, 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 thus describes the free variables that can possibly occur in the object which represents the contextual variable, and the type system statically guarantees this.

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, which simplifies the theoretical development and avoids 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 data-level substitutions, as defined operations on data-level terms, are written $[\sigma]N$.

Contextual variables such as meta-variables u, parameter variables p, and substitution variables s are declared in a meta-level context Δ, while ordinary bound variables are declared in a 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 Ω. Unlike previous uses of context variables [10], 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 which classifies contexts Ψ. We say that context Ψ checks against a schema $W = (F_1 + \dots + F_n)^*$ if there exists an element $F_k = \mathsf{all}\ \Phi.\Sigma y_1{:}A_1^*, \dots, y_j{:}A_j^*.\ A^*$ such that Ψ is an instance of $\Sigma y_1{:}A_1^*, \dots, y_j{:}A_j^*.A^*$ where all variables in Φ have been appropriately instantiated.

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.

4.1. Data-level typing

We present a bidirectional 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

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

Data-level normal terms

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

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

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

Data-level neutral terms

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

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

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

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

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

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

Data-level substitutions

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

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

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

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

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 for lambda-abstraction 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* which ensure that canonical forms are preserved [24, 12].

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. For this operation to terminate, it suffices to keep track of the approximate (non-dependent) type of the object being substituted. We omit the definition of ordinary hereditary substitutions and refer the reader to [12] for details. For the subsequent development we use the operation $[M/x]^*_A(N)$ where $* \in \{n, r, s, a\}$.

Schema checking relies on higher-order pattern matching. If Ψ checks against W, then every declaration $x_i{:}A_i$ in Ψ will be independent from any other declaration $x_j{:}A_j$ in Ψ because dependencies are only allowed *within* a schema element F.

Theorem 4.1. [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 operation is decidable. □

4.2. Substitution

The different variables (context variables ψ, ordinary variables x, and contextual variables) give rise to different substitution operations. We briefly summarize the substitution operations for context variables ψ and contextual variables in this section, and refer the reader to previous work [16, 12] for a more detailed discussion.

We begin by considering substitution for context variables. If we encounter a context variable ψ, we simply replace it with the context Ψ.

Data-level context

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

When we apply the substitution $[\![\Psi/\psi]\!]$ to the context $\Phi, x{:}A$, we apply the substitution to the context Φ to yield some new context Φ'. However, we must make sure 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. 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:

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

Substitutions for contextual variables are a little more difficult. We have three kinds of contextual variables: meta-variables u, parameter variables p, and substitution variables (see [16]).

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 $[\![\hat{\Psi}.M/u]\!]^n_{A[\Psi]}(N)$, $[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}(R)$, and $[\![\hat{\Psi}.M/u]\!]^s_{A[\Psi]}(\sigma)$, where $\hat{\Psi}$ binds all free variables in M. The annotation $A[\Psi]$ 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.

In defining substitution, we must make sure that normal forms are preserved. For example, 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 eagerly applies σ' to M. To enforce that we always return a normal object as a result of contextual substitution, we resort to ordinary hereditary substitution. Since ordinary hereditary substitution requires the type of the argument being substituted, we carry the type of the meta-variable u which will be replaced. For a thorough explanation, see [12]. For clarity, we omit the typing annotation when we subsequently write contextual substitution.

Contextual substitution for parameter variables follows similar principles, but 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 normal if it is of atomic type. We write parameter substitutions as $[\![\hat{\Psi}.x/p]\!]^*_{A[\Psi]}$. When we encounter a parameter variable $p[\sigma]$, we replace p with the ordinary variable x and apply the substitution $[\![\hat{\Psi}.x/p]\!]^s_{A[\Psi]}$ 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.

Substituting for substitution variables follows the same ideas. To ensure it works correctly with the previously defined substitution operations, we also annotate it with the type of the substitution variable. 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.

5. Computation-level expressions

Our goal is to cleanly separate the data level from the computation level, which describes programs operating on data. Computation-level types may refer to data-level types via the contextual 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 a dependent type $\Pi\psi{:}W.\tau$ and context abstraction $\Lambda\psi.e$. We write $\rightarrow$ for computation-level functions. We introduce abstraction over the arguments which could occur in a dependent type A on the computation level using the dependent type $\Pi^\square A[\Psi].\tau$.

$$
\begin{array}{llcl}
\text{Types} & \tau & ::= & A[\Psi] \mid \Phi[\Psi] \mid \tau_1 \rightarrow \tau_2 \mid \Pi\psi{::}W.\tau \\
& & & \mid \Pi^\square u{::}A[\Psi].\tau \mid \Sigma^\square u :: A[\Psi].\tau \\
\text{Expressions} & e & ::= & i \mid \mathsf{rec}\ f.e \mid \mathsf{fn}\ y.e \mid \Lambda\psi.e \mid \lambda^\square u.\, e \\
\text{(checked)} & & & \mid \mathsf{box}(\hat{\Psi}.\, M) \mid \mathsf{sbox}(\hat{\Psi}.\, \sigma) \\
& & & \mid \mathsf{pack}(\hat{\Psi}.M\ ,\ e) \mid \mathsf{case}\ i\ \mathsf{of}\ bs \\
\text{Expressions} & i & ::= & y \mid i\ e \mid i\ \lceil\Psi\rceil \mid i\ \lceil\hat{\Psi}.M\rceil \mid (e:\tau) \\
\text{(synth.)} & & & \mid \mathsf{let}\ \mathsf{pack}(u\ ,\ x) = e\ \mathsf{in}\ e'\ \mathsf{end} \\
\text{Patterns} & \zeta & ::= & \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \\
& & & \mid \Pi\Delta.\mathsf{sbox}(\hat{\Psi}.\, \sigma) : \Phi[\Psi] \\
\text{Branch} & b & ::= & \zeta \mapsto e \\
\text{Branches} & bs & ::= & \cdot \mid (b \mid bs) \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, y{:}\tau
\end{array}
$$

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 which can possibly occur in M. Index objects to dependent types are introduced using the constructor $\lambda^\square u.\, e$ which has 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 which are declared in Δ are instantiated using higher-order pattern matching. We only consider patterns à la Miller [11] 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.

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 case analyzed.

5.1. Computation-level typing

We describe computation-level typing using the following judgments:

$$
\begin{array}{ll}
\Omega;\Delta;\Gamma \vdash e \Leftarrow \tau & e \text{ checks against } \tau \\
\Omega;\Delta;\Gamma \vdash i \Rightarrow \tau & i \text{ synthesizes } \tau \\
\Omega;\Delta;\Gamma \vdash b \Leftarrow_{\tau'} \tau & \text{branch } b \text{ checks against } \tau, \\
& \quad \text{when case-analyzing a } \tau'
\end{array}
$$

The most interesting rule in the bidirectional typechecking algorithm is the one for branches, since the type $A_k[\Psi_k]$

Expression e checks against type τ

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

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

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

$$\frac{\Omega;\Delta;\Psi\vdash M\Leftarrow A\qquad\Omega;\Delta;\Gamma\vdash e\Leftarrow[\![\hat{\Psi}.M/u]\!]\tau}{\Omega;\Delta;\Gamma\vdash\mathsf{pack}(\hat{\Psi}.M\ ,\ e)\Leftarrow\Sigma^{\square}u{::}A[\Psi].\tau}\ \overline{\mathsf{pack}}$$

$$\frac{\Omega;\Delta;\Gamma\vdash e'\Rightarrow\Sigma^{\square}u{::}A[\Psi].\tau'\qquad\Omega;\Delta,u{::}A[\Psi];\Gamma,x{:}\tau'\vdash e\Leftarrow\tau}{\Omega;\Delta;\Gamma\vdash\mathsf{let}\,\mathsf{pack}(u\ ,\ x)=e'\ \mathsf{in}\ e\ \mathsf{end}\Leftarrow\tau}\ \overline{\mathsf{let\text{-}pack}}\qquad\frac{\Delta;\Gamma\vdash i\Rightarrow\tau'\qquad\Omega;\Delta\vdash\tau'=\tau}{\Omega;\Delta;\Gamma\vdash i\Leftarrow\tau}\ \overline{\mathsf{turn}}$$

Expression i synthesizes type τ

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

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

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

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

Figure 1. Computation-level typing rules

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 [26], branches are checked under certain equality constraints. Instead we propose here to solve these constraints eagerly using higher-order pattern unification. This approach simplifies the later preservation and progress proof. It is also closer to a realistic implementation, which would support early failure and point out the offending branch.

Branches b are of the form $\Pi\Delta.\mathsf{box}(\hat{\Psi}.\,M)\mapsto e$ (resp. sbox), 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 of the branch and check that it is well-typed.

Since we restrict all occurrences of contextual variables in the patterns of case expressions such that they are higher-order patterns [11], and higher-order pattern unification is decidable, unification of the contexts and types is decidable. For a formal description of a higher-order pattern unification algorithm for contextual variables, see [15].

5.2. Operational semantics

Next, we define a small-step evaluation judgment:

e evaluates in one step to e' $\qquad e\ \longrightarrow\ e'$

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

Branch b matches $\mathsf{sbox}(\hat{\Psi}.\,\sigma)$ and steps to e'
$$(\mathsf{sbox}(\hat{\Psi}.\,\sigma):\Phi[\Psi]\doteq b)\ \longrightarrow\ e'$$

In the presence of full LF we cannot erase type information completely during evaluation, since not all implicit type arguments can be uniquely determined. Because evaluation relies on pattern matching, and higher-order pattern matching requires that we only match two terms if they have the same type, we match against the pattern's type before

Evaluation of computations:

$$\frac{}{\mathsf{rec}\ f.e \longrightarrow [\mathsf{rec}\ f.e/f]e} \quad \frac{}{(\mathsf{fn}\ y.e : \tau_1 \to \tau_2)\ v \longrightarrow [v/y]e} \quad \frac{}{(\lambda^{\square} u.\, e)\ \lceil \hat{\Psi}.M \rceil \longrightarrow [\![\hat{\Psi}.M/u]\!]e}$$

$$\frac{i_1 \longrightarrow i_1'}{i_1\ e_2 \longrightarrow i_1'\ e_2} \quad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} \quad \frac{i \longrightarrow i'}{i\ \lceil \Psi \rceil \longrightarrow i'\ \lceil \Psi \rceil} \quad \frac{}{(\Lambda\psi.e)\ \lceil \Psi \rceil \longrightarrow [\![\Psi/\psi]\!]e}$$

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

$$\frac{i \longrightarrow i'}{\mathsf{case}\ i\ \mathsf{of}\ bs \longrightarrow \mathsf{case}\ i'\ \mathsf{of}\ bs} \quad \frac{i \Rightarrow\Leftarrow \zeta_1}{(\mathsf{case}\ i\ \mathsf{of}\ \zeta_1 \mapsto e_1 \mid bs) \longrightarrow \mathsf{case}\ i\ \mathsf{of}\ bs}$$

$$\frac{i \doteq \zeta_1 \,/\, \theta \qquad \zeta_1 = \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi]}{(\mathsf{case}\ i\ \mathsf{of}\ \zeta_1 \mapsto e_1 \mid bs) \longrightarrow [\![\theta]\!]e_1} \quad \frac{i \doteq \zeta_1 \,/\, \theta \qquad \zeta_1 = \Pi\Delta.\mathsf{sbox}(\hat{\Psi}.\, \sigma) : \Phi[\Psi]}{(\mathsf{case}\ i\ \mathsf{of}\ \zeta_1 \mapsto e_1 \mid bs) \longrightarrow [\![\theta]\!]e_1}$$

Evaluation of branches:

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

$$\frac{\Delta \vdash \Psi \doteq \Psi_k \,/\, (\theta_1, \Delta_1) \qquad \Delta_1 \vdash \Phi \doteq [\![\theta_1]\!]\Phi_k \,/\, (\theta_2, \Delta_2) \qquad \theta = [\![\theta_2]\!]\theta_1 \qquad \Delta_2; \Psi \vdash \sigma \doteq [\![\theta]\!]\sigma_k \,/\, (\theta_3, \cdot)}{(\mathsf{sbox}(\hat{\Psi}.\, \sigma) : \Phi[\Psi] \doteq \Pi\Delta.\mathsf{sbox}(\hat{\Psi}.\, \sigma_k) : \Phi_k[\Psi_k]) \,/\, [\![\theta_3]\!]\theta}$$

Figure 2. Operational semantics

matching against the pattern itself. This also means we implicitly translate the computational language into one where general type annotations are erased, but all box expressions, including patterns in branches, carry their corresponding type. We denote this translation by $|e|$ and $|i|$ for checked and synthesizing expressions, respectively.

Otherwise, the operational semantics is straightforward. In function application, 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 $\Lambda\psi.e$. Index arguments are propagated by applying a concrete data object $\hat{\Psi}.M$ to the expression $\lambda^{\square} u.\, e$.

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 $\mathsf{box}(\Psi.\, M)$ does not contain any meta-variables, i.e. it is closed and 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. During execution, the only type annotations needed are on box expressions. We define evaluation on expressions with all other type annotations erased. Given the current setup, we can now 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 it should be possible to incorporate coverage checking following the ideas of [21]. First we state and prove a canonical forms lemma.

Lemma 5.1. [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 $\cdot;u{::}A[\Psi];\cdot \vdash e' \Leftarrow \tau$.

(4) If i is a value and $\cdot;\cdot;\cdot \vdash i \Rightarrow \Sigma^{\square} u :: A[\Psi].\tau$
then $|i| = \mathsf{pack}(\hat{\Psi}.M\ ,\ |e'|)$ and $\cdot;\cdot;\Psi \vdash M \Leftarrow A$
and $\cdot;\cdot;\cdot \vdash e' \Leftarrow [\![\hat{\Psi}.M/u]\!]\tau$.

Proof. By induction on the typing derivation. □

Theorem 5.1. [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 5.1, the obvious substitution properties, and coverage checking as needed. □

6. Conclusion

We have presented a type-theoretic foundation for programming with higher-order dependently typed data that supports higher-order abstract syntax and first-class substitution. Our framework supports recursion over data defined with HOAS, and allows pattern matching against open data and variables based on higher-order patterns. By design, bound variables in data cannot escape their scope. Although we have not considered coverage and termination checking in this paper, we believe this is an important step towards programming with proofs in a functional setting thereby providing an alternative functional framework for mechanizing the meta-theory of formal systems. More generally, our work explores a novel point in combining a rich dependently typed data language with functional programming while preserving decidability of typing.

References

[1] L. Augustsson. Cayenne—a language with dependent types. In *3rd International Conference Functional Programming (ICFP '98)*, pages 239–250, ACM Press, 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. Primitive recursion for higher order abstract syntax with dependent types. In *International Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, 1999.

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

[5] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *3rd International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, 1997. Springer.

[6] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, 1999. IEEE Computer Society.

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

[8] D. R. Licata and R. Harper. An extensible theory of indexed types. Unpublished manuscript, July 2007.

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

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

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

[12] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*. Accepted, to appear in 2008.

[13] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *11th International Conference on Functional Programming (ICFP '06)*, ACM Press, 2006.

[14] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer, LNAI 1632, 1999.

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

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

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

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

[19] U. Schöpp and I. Stark. A dependent type theory with names and binding. In *13th Annual Conference on Computer Science Logic (CSL)*, pages 235–249. Springer, LNCS 3210, 2004.

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

[21] C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, pages 120–135. Springer, LNCS 2758, 2003.

[22] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *4th International Workshop on Logical Frameworks and Meta-languages (LFM '04)*, pages 106–124, 2004.

[23] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. In *8th International Conference Functional Programming (ICFP'03)*, pages 263–274. ACM Press, 2003.

[24] 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.

[25] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *30th ACM Symposium Principles of Programming Languages (POPL '03)*, pages 224–235. ACM Press, 2003.

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