Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 174 (2007) 41–60

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Functional Programming With Higher-order Abstract Syntax and Explicit Substitutions

Brigitte Pientka[1]

School of Computer Science
McGill University
Montreal, Canada

Abstract

This paper sketches a foundation for programming with higher-order abstract syntax and explicit substitutions based on contextual modal type theory [9]. Contextual modal types not only allows us to cleanly separate the representation of data objects from computation, but allow us to recurse over data objects with free variables. In this paper, we extend these ideas even further by adding first-class contexts and substitutions so that a program can pass and access code with free variables and an explicit environment, and link them in a type-safe manner. We sketch the static and operational semantics of this language, and give several examples which illustrate these features.

Keywords: Logical frameworks, type systems

1 Introduction

Higher-order abstract syntax is a simple well-recognized technique for implementing languages with variables and binders. This issue typically is key when implementing evaluators, compilers or automated reasoning systems. The central idea behind higher-order abstract syntax is to implement object-level variables and binders by variables and binders in the meta-language (i.e. functional programming language). One of the key benefits behind higher-order abstract syntax representations is that one can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation.

Higher-order abstract syntax and its usefulness have long been demonstrated within the logical framework LF [3] and its implementation in the Twelf system [12]. However it has been difficult to extend mainstream functional programming languages with direct support for higher-order syntax encodings. The difficulty is due to the fact that higher-order abstract syntax encodings are not inductive in

[1] Email: bpientka@cs.mcgill.ca

1571-0661
doi:10.1016/j.entcs.2006.10.037

the usual sense. The problem is that recursion over higher-order abstract syntax requires one to traverse a λ-abstraction and hence we need to be able to reason about "open" terms. Building on ideas in [9], we present a novel approach based on contextual modal types which allows recursion over open terms and also supports first-class environments and contexts.

Most closely related to our work are previous proposals by Despeyroux, Pfenning and Schürmann [2] where the authors present a modal λ-calculus which supports primitive recursion over higher-order encodings via an iterator. However, function definitions via iteration do not support pattern matching and do not easily support reasoning with dynamic assumption. This problem is addressed in [15] and the ∇-calculus is proposed as a remedy. It also serves as a foundation for the *Elphin* language. Their work requires scope stacks and explicit operations on them such as popping an element of the scope stack. A function is then executed within a certain scope. The necessity of keeping track of the current scope also permeates the operational semantics, which explicitly keeps track of scope stacks.

In contrast to these approaches we believe that the contextual modal type theory [9] can provide a clean and elegant framework for extending functional programming with support for higher-order abstract syntax and pattern matching. Since we allow abstraction over context variables, we can associate a scope (context) with each argument passed to a function rather with the function itself. This allows us to write programs which accept open data objects as inputs and enforces stronger invariants about data objects and programs. We also believe this will facilitate the reasoning about the programs we write. Previous approaches by [15] only allow open objects during recursion but require the objects to be closed at the beginning of the computation. Moreover, our proposal extends to support first-class substitutions which are independently interesting and useful. First-class substitutions are potentially interesting in programming with explicit environments, which has a wide range of applications such as an environment based interpreter, or constant elimination algorithm. Our underlying contextual modal type system guarantees that programs can pass and access open terms and that it can be safely linked with an environment.

In this paper we take a first step towards designing a foundation for programming with higher-order abstract syntax and explicit substitution based on contextual modal type theory. We first introduce an example to highlight some of the issues when programming with higher-order abstract syntax and then present a theoretical foundation by adapting and extending contextual modal type theory. Finally, we give several examples illustrating the basic ideas of programming with explicit substitutions. Since explicit substitutions can be viewed as value-variable pairs, this facility essentially allows us to model explicit environments. Moreover, our underlying type system will ensure the correct usage of this environment and statically enforce crucial invariants such as every "free" variable occurring in a term M is bound in some environment.

2 Motivating example

In this section we briefly discuss some of our main ideas and concerns using a simple example which analyzes and compares the structure of lambda-terms. To add higher-order encodings to a functional programming language we follow the approach taken in [2,15] to separate the representation and computation level via a modal operator. Data objects M can make use of higher-order abstract syntax and are injected into programs via the box-construct $\mathsf{box}\ M$ where M denotes object-level data. The computation level describes our functional programs which operate on object-level terms. On this level, we allow recursion and pattern matching against object-level terms to extract sub-terms. Unlike previous proposals however, every object M carries its own local context Ψ such that $\mathsf{box}(\Psi.\, M)$ thereby allowing programming with open objects.

To illustrate consider first the following two definitions of lambda-terms based on higher-order abstract syntax.

Object-level expressions	Object-level terms
$\mathsf{lam} : (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}.$	$\mathsf{lam}' : (\mathsf{term} \to \mathsf{term}) \to \mathsf{term}.$
$\mathsf{app} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$	$\mathsf{app}' : \mathsf{term} \to \mathsf{term} \to \mathsf{term}.$

We are interested in defining a function related which checks whether an expression is related to a term. This function just compares the basic shape of a term and an expression but it does not check for equality of variable names. For example, we consider the term $\mathsf{lam}\ \lambda x.\mathsf{lam}\ \lambda y.\mathsf{app}\ x\ y$ to be related to the expression $\mathsf{lam}'\ \lambda x.\mathsf{lam}'\ \lambda y.\mathsf{app}'\ y\ x$. Hence our computation would start of with

$$\mathsf{related}\ (\mathsf{box}(\cdot.\, \mathsf{lam}\ \lambda x.\mathsf{lam}\ \lambda y.\mathsf{app}\ x\ y))\ (\mathsf{box}(\cdot.\, \mathsf{lam}'\ \lambda x.\mathsf{lam}'\ \lambda y.\mathsf{app}'\ y\ x))$$

Recursively, we traverse the lambda-binder and in the next iteration we compute

$$\mathsf{related}\ (\mathsf{box}(x{:}\mathsf{exp}.\, \mathsf{lam}\ \lambda y.\mathsf{app}\ x\ y)\ (\mathsf{box}(x{:}\mathsf{term}.\, \mathsf{lam}'\lambda y.\ \mathsf{app}'\ y\ x))$$

As we see every argument carries its own local context, which is extended when we traverse the binder. In the next iteration, the local contexts are again extended.

$$\mathsf{related}\ (\mathsf{box}(x{:}\mathsf{exp}, y{:}\mathsf{exp}.\, \mathsf{app}\ x\ y))\ (\mathsf{box}(x{:}\mathsf{term}, y{:}\mathsf{term}.\, \mathsf{app}'\ y\ x))$$

Finally, we need to check

$$\mathsf{related}\ (\mathsf{box}(x{:}\mathsf{exp}, y{:}\mathsf{exp}.\, x))\ (\mathsf{box}(x{:}\mathsf{term}, y{:}\mathsf{term}.\, y))$$

How could one write a recursive function related and what type should it have? The function related takes as input an open object of type term and open object of type exp. The local context associated with each open object will be reflected in its type. While in previous proposals object-level data of type term was given the type $\Box\mathsf{term}$, we write $\mathsf{term}[x{:}\mathsf{term}]$ for an open object M which has type term

in the context x:term. Similarly, we write exp[x:exp, y:exp] for an open object M which has type exp in the context x:exp, y:exp. To actually write recursive programs and assign a type to them, we need to be able to abstract over the concrete context, since it changes during execution and we must characterize valid instances for context variables. Valid instances of context variables can be described by context schemas or worlds (see also [14]). Let exprW be the constant describing the context schema x_1:exp, ..., x_n:exp, and termW be the constant describing the context schema y_1:term, ..., y_k:term. Then we can declare a dependent type for the function related as:

$$\mathsf{related} : \Pi\gamma{:}\mathsf{exprW}.\Pi\psi{:}\mathsf{termW}.\mathsf{exp}[\gamma] \to \mathsf{term}[\psi] \to \mathsf{bool}.$$

Before we present the code for the function related, we discuss a few considerations here. First, a recursive function will be defined via pattern matching and needs to extract sub-expression. To describe "holes" which can be matched against open sub-expressions, we draw upon ideas from [11,9] and use contextual modal variables $u[\sigma]$. $u[\sigma]$ denotes a closure with the postponed substitution σ. Second, we need to be able to pattern match against object-level variables. This will be achieved by imposing constraints on occurrences of context variables. The context variable $\psi(x$:term) denotes a context of type termW which must contain at least one element x:term. We can now give the code for the function related next.

```
fun related [g] [g'] (box g'(y:exp).y) (box g(x:term).x) =
     true
  | related [g] [g'] (box g. lam [x]. u[id_g, x/x])
                     (box g'. lam' [y]. v[id_g',y/y']) =
     related [g, x:exp] [g', y:term]
             (box g, x:exp. u[id_g, x/x])
             (box g', y:term. v[id_g', y/y'])
  | related [g] [g'] (box g. app u1[id_g] u2[id_g])
                     (box g'. app' v1[id_g'] v2[id_g']) =
      related [g] [g'] (box g. u1[id_g]) (box g'.v1[id_g'])
       andalso
      related [g] [g'] (box g. u2[id_g]) (box g'.v2[id_g'])
```

One interesting question is how to modify this program such that we check for exact shapes. One possible solution is to modify the type of the function related such that both objects share the same context. Such a generalized context has the following shape: (x_1:term, y_1: exp), ..., (x_n:term, y_n: exp). We use brackets to emphasize the fact that this context is built up of tuples (x_i:term, y_i:exp). Let termExpW be the constant describing this world. Then we can declare a dependent type for the function rel as $\Pi\gamma$:termExpW.exp[γ] $\to$ term[γ] $\to$ bool.

```
fun rel [g] (box g(x:term, y:exp).x)  (box g(x:term,y:exp).y) =
     true
  | rel [g] (box g. lam [x]. u[id_g, x/x'])
            (box g. lam' [y]. v[id_g,y/y']) =
     rel [g, x:term, y:exp] (box g, x:term, y:exp. u[id_g, x/x'])
                            (box g, x:term, y:exp. v[id_g, y/y'])
  | rel [g] (box g. app u1[id_g] u2[id_g])
            (box g. app' v1[id_g] v2[id_g]) =
      rel [g] (box g. u1[id_g]) (box g.v1[id_g])
       andalso
      rel [g] (box g. u2[id_g]) (box g.v2[id_g])
```

An important issue in our setting is α-equivalence and the scope of context variables. Do we consider $\mathsf{box}(\psi(x{:}\mathsf{term}).\, x)$ α-equivalent to $\mathsf{box}(\psi(y{:}\mathsf{term}).\, y)$? Is $\mathsf{box}(x{:}\mathsf{term}, y{:}\mathsf{term}.\, y)$ α-equivalent to $\mathsf{box}(z{:}\mathsf{term}, w{:}\mathsf{term}.\, w)$? What operations do we allow on context variables? – In the following theoretical development, we will pay careful attention to these issues.

3 Formalities

Object-level terms and typing

In this section, we introduce the formal definition and type system based on contextual modal type theory. We start with th object language which is defined following ideas in [9] and define only objects which are in canonical form since only these are meaningful for representing object-languages. For simplicity, we restrict it to the simply-typed fragment.

Types	A, B, C	$::= \alpha \mid A \to B$
Normal Terms	M, N	$::= \lambda x.\, M \mid R$
Neutral Terms	R	$::= x \mid c \mid R\, N \mid u[\sigma]$
Substitutions	σ, ρ	$::= \cdot \mid \sigma, M/x \mid s[\sigma] \mid \mathsf{id}_{\psi(\omega)}$
Contexts	Ψ, Φ	$::= \cdot \mid \Psi, x{:}A \mid \psi(\omega)$
Meta-contexts	Δ	$::= \cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, s{::}\Psi[\Phi] \mid \psi{::}W$
Constraints	ω	$::= \cdot \mid \omega, x{:}A$

There are several interesting aspects about the simply-typed modal lambda-calculus. First, we distinguish between ordinary bound variables x and contextual modal variables $u[\sigma]$. Contextual modal variable $u[\sigma]$ denotes a closure with the postponed substitution σ. As we briefly alluded to in the previous section, contextual modal variables will be used to define pattern matching. Sometimes we also call contextual modal variables meta-variables. Our intention is to apply σ as soon as we know which term u should stand for. The domain of σ describes the free variables which can possibly occur in the term which represents u. For more details

on contextual modal variables we refer the interested reader to [9]. Here we propose to extend the calculus with context variables ψ and substitution variables $s[\sigma]$. Context variables ψ may be annotated with constraints ω which impose condition on the context ψ. For example $\psi(x{:}A)$ denotes a context which is built out of blocks $x_i{:}A$ and contains at least one such block. In other words the variable x of type A must occur in the context. We also note that $\psi(\omega), \Psi$ is only well-formed if the variables mentioned in ω are not also declared in Ψ. At the moment, we restrict the use of constraints to context variables occurring in object. Context variables occurring in types are not allowed to be associated with any constraints [2]. We also require a first-class notion of identity substitution $\mathsf{id}_{\psi(\omega)}$. Abstracting over contexts and substitution seems an interesting and essential next step, if we aim at using contextual modal types as a foundation for programming with higher-order abstract syntax. Contextual modal variables u, substitution variables s and context variables ψ are declared in the meta-context Δ. Context variables are declared to belong to a context schema W, which we assume to be declared in the a signature Σ similar to type and world declarations in Twelf [14]. We omit here the exact definition for worlds and context schemas and refer the interested reader to [14,13]. Typically we also suppress the signature Σ since it never changes during a typing derivation, but keep in mind that all typing judgments have access to a well-formed signature. Checking a type A is well-formed in a signature Σ is straightforward since there are no dependencies. Next, we follow essentially ideas in [9] to describe object-level canonical forms only. We assume only simple types (no dependencies) and object level type constants a together with constants denoting context schemas have been declared in a signature. Next, we describe the main typing judgments and typing rules.

$$\Delta; \Psi \vdash M \Leftarrow A \quad \text{Check normal object } M \text{ against type } A$$

$$\Delta; \Psi \vdash R \Rightarrow A \quad \text{Synthesize type } A \text{ for atomic object } R$$

$$\Delta; \Phi \vdash \sigma \Leftarrow \Psi \quad \text{Check substitution } \sigma \text{ against context } \Psi$$

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 modal variables u. We also streamline the calculus slightly by always substituting simultaneously for all ordinary variables. This is not essential, but saves some tedium in relating simultaneous and iterated substitution. We will omit here the definitions for well-formed contexts and well-formed constraints, but focus on typing of terms and substitutions. During typing we refer to $\omega : W$ which guarantees that the constraints ω correspond to the context schema W.

[2] Allowing constraints in types would yield to complications when defining substitution

Object-level terms

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

$$\frac{x{:}A \in \Psi}{\Delta; \Psi \vdash x \Rightarrow A} \qquad \frac{x{:}A \in \omega \quad x{:}A \notin \Psi \quad \psi{:}W \in \Delta \quad \omega : W}{\Delta; \psi(\omega), \Psi \vdash x \Rightarrow A} \qquad \frac{c{:}A \in \Sigma}{\Delta; \Psi \vdash c \Rightarrow A}$$

$$\frac{u{::}A[\Phi] \in \Delta \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow A} \qquad \frac{\Delta; \Psi \vdash M \Rightarrow A \to B \quad \Delta; \Psi \vdash N \Leftarrow A}{\Delta; \Psi \vdash M\, N \Rightarrow B}$$

Object-level substitutions

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

$$\frac{s{::}\Phi_1[\Phi_2] \in \Delta \quad \Phi = \Phi_1 \quad \Delta; \Psi \vdash \rho \Leftarrow \Phi_2}{\Delta; \Psi \vdash (s[\rho]) \Leftarrow \Phi} \qquad \frac{\psi : W \in \Delta \quad \omega : W}{\Delta; \psi(\omega), \Psi \vdash \mathsf{id}_{\psi(\omega)} \Leftarrow \psi(\omega)}$$

We note that we 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 be always achieved via alpha-renaming.

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 object-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 $\Lambda\psi.e$.

$$\begin{array}{lll}
\text{Types} & \tau & ::= A[\Psi] \mid \Phi[\Psi] \mid \tau_1 \to \tau_2 \mid \Pi\psi{:}W.\tau \\
\text{Expressions} & e & ::= y \mid \mathsf{rec}\ f.e \mid e_1\ e_2 \mid \mathsf{fn}\ y.e \mid \Lambda\psi.e \mid e\ \lceil\Psi\rceil \mid (e : \tau) \\
 & & \quad \mid \mathsf{box}(\Psi.\, M) \mid \mathsf{sbox}(\Psi.\, \sigma) \mid \mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n \\
\text{Branch} & p, q & ::= \epsilon u{::}A[\Psi].p \mid \epsilon s{::}\Phi[\Psi].p \mid \mathsf{box}(\Psi.\, M) \mapsto e \mid \mathsf{sbox}(\Psi.\, \sigma) \mapsto e \\
\text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, y{:}\tau
\end{array}$$

Data can be injected into programs via the box-construct $\mathsf{box}(\Psi.\, M)$. Here M denotes an object-level term M which may contain the variables specified in the context Ψ. Similarly, we can inject substitutions $\mathsf{sbox}(\Psi.\, \sigma)$ where Ψ is the range of

the substitution σ. Since substitutions can be viewed as pairs between variables and object-level terms, this facility essentially allows us to model explicit environments. Finally, we allow pattern matching on object-level terms via case-statement. To simplify the theoretical development, we require that all contextual modal variables occurring in a pattern are explicitly specified. However we do not yet consider matching against context variables. We overload the $\rightarrow$ which is used to denote function types on the object level as well as the computation level. Also we may use x and y for object-level variables and program variables. However, it should be clear from the usage which one we mean.

Next, we consider typing rules for programs. We distinguish here between typing of expressions and branches. Note that in order to type expressions and branches we will refer to the typing of object-level terms. Moreover, we adopt a bi-directional view.

$\Delta; \Gamma \vdash e \Leftarrow \tau$	expression e has type τ
$\Delta; \Gamma \vdash e \Rightarrow \tau$	synthesize type τ for expression e
$\Delta; \Delta'; \Gamma \vdash p : \tau' \Leftarrow \tau$	branch p checks against $\tau' \Leftarrow \tau$

The typing rules for expressions are next. We only point out a few interesting issues. First the typing rule for $\mathsf{box}(\Psi.\, M)$. M denotes a object-level term whose free variables are defined in the context Ψ, i.e. it is closed with respect to a context Ψ. To type $\mathsf{box}(\Psi.\, M)$ we switch to object-level typing, and forget about the previous context Γ which only describes assumptions on the computation-level. Similar reasoning holds for the typing rule for $\mathsf{sbox}(\Psi.\, \sigma)$. To access data, we provide a case-statement with pattern matching. The intention is to match against the contextual modal variables occurring in the pattern.

Expressions

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

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

$$\frac{\Delta; \Gamma \vdash e \Rightarrow A[\Psi] \quad \text{for all } i\ \Delta; \cdot; \Gamma \vdash p_i : A[\Psi] \Leftarrow \tau}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ \Leftarrow \tau}$$

$$\frac{\Delta; \Gamma \vdash e \Rightarrow \Phi[\Psi] \quad \text{for all } i\ \Delta; \cdot; \Gamma \vdash p_i : \Phi[\Psi] \Leftarrow \tau}{\Delta; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ \Leftarrow \tau}$$

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

$$\frac{\Delta; \Gamma \vdash e \Rightarrow \Pi\psi{:}W.\tau \quad \vdash \Psi : W}{\Delta; \Gamma \vdash e\ \lceil \Psi \rceil \Rightarrow [\![\Psi/\psi]\!]\tau} \qquad \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_2 \rightarrow \tau \quad \Delta; \Gamma \vdash e_2 \Leftarrow \tau_2}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow \tau}$$

Branches

$$\frac{\Delta; (\Delta', u{::}A[\Phi]); \Gamma \vdash p : \tau_1 \Leftarrow \tau}{\Delta; \Delta'; \Gamma \vdash \epsilon u{::}A[\Phi].p : \tau_1 \Leftarrow \tau} \qquad \frac{\Delta, (\Delta', s{::}\Psi[\Phi]); \Gamma \vdash p : \tau_1 \Leftarrow \tau}{\Delta; \Delta'; \Gamma \vdash \epsilon s{::}\Psi[\Phi].p : \tau_1 \Leftarrow \tau}$$

$$\frac{\Delta'; \Gamma \vdash \mathsf{box}(\Psi.\, M) \Leftarrow \tau_1 \quad (\Delta, \Delta'); \Gamma \vdash e \Leftarrow \tau}{\Delta; \Delta'; \Gamma \vdash \mathsf{box}(\Psi.\, M) \mapsto e : \tau_1 \Leftarrow \tau}$$

$$\frac{\Delta'; \Gamma \vdash \mathsf{sbox}(\Psi.\, \sigma) \Leftarrow \tau_1 \quad (\Delta, \Delta'); \Gamma \vdash e \Leftarrow \tau}{\Delta; \Delta'; \Gamma \vdash \mathsf{sbox}(\Psi.\, \sigma) \mapsto e : \tau_1 \Leftarrow \tau}$$

Contexts

$$\frac{}{\psi(\omega) \leq \psi} \qquad \frac{\Psi \leq \Psi'}{\Psi, x{:}A \leq \Psi', x{:}A} \qquad \frac{}{\cdot \leq \cdot}$$

Here we also observe the usual bound variable renaming conditions. In the function rule we assume that the variable x is new and does not occur already in Γ. Context variables are explicitly quantified and bound by $\Lambda\psi.e$. In particular, the context variable ψ in $\mathsf{box}(\psi.\, M)$ is not bound by box. Recall also, that we do not allow constraints at binding occurrences of context variables in types. As a consequence, comparing two contexts Ψ and Ψ' as in the rule for box for example, just checks whether we can obtain Ψ' from Ψ by erasing any constraints ω which may be present in a context variable ψ.

In the rules for explicit substitutions, we need to possibly rename the domain of σ. This can always be achieved. Renaming of the domain of a substitution can be done explicitly by σ'/Ψ. Similarly, in the rule for $\mathsf{box}(\Psi.\, M)$ we may need to rename the variables in Ψ' to match the variables in Ψ.

4 Ordinary and contextual substitutions

In this section we define the operations of substitution. There are multiple substitution operations because we have several different kinds of variables. We will consider each of these operations in turn.

Ordinary substitution for program and object-level variables

The operations are capture-avoiding and defined in a standard manner. Our convention is that substitutions as defined operations on object-level terms and expressions are written in prefix notation $[M/x]N$ for an object-level substitution and $[e/x]e'$ for computation-level substitution. Note that in $[M/x]N$ the bound variable x denotes an object-level term, while in the $[e/x]e'$ the bound variable x denotes a computation-level expression. We only show the substitutions on the computation level to illustrate some basic principles. The details for $[M/x]N$ can

be found in [9]. Next, we define substitution for computation-level expressions and branches.

$$
\begin{array}{lll}
[e/x](x) & = e & \\
[e/x](y) & = y \quad \text{if } y \neq x & \\
[e/x](\Lambda\psi.e') & = \Lambda\psi.[e/x]e' & \text{provided } \psi \notin \mathsf{FV}(e) \\
[e/x](e'\ \lceil\Psi\rceil) & = [e/x]e'\ \lceil\Psi\rceil & \\
[e/x](\mathsf{fn}\ y.e') & = \mathsf{fn}\ y.[e/x]e' & \text{provided } y \notin \mathsf{FV}(e) \text{ and } y \neq x \\
[e/x](\mathsf{rec}\ f.e') & = \mathsf{rec}\ f.[e/x]e' & \text{provided } f \notin \mathsf{FV}(e) \text{ and } f \neq x \\
[e/x](e_1\ e_2) & = ([e/x]e_1)\ ([e/x]e_2) & \\
[e/x](\mathsf{box}(\Psi.\, M)) & = \mathsf{box}(\Psi.\, M) & \\
[e/x](\mathsf{sbox}(\Psi.\, \sigma)) & = \mathsf{sbox}(\Psi.\, \sigma) & \\
[e/x](\mathsf{case}\ e'\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) & = \mathsf{case}\ [e/x]e'\ \mathsf{of}\ [e/x]p_1 \mid \ldots \mid [e/x]p_n & \\
[e/x](\epsilon u{::}A[\Phi].p) & = \epsilon u{::}A[\Phi].[e/x]p & \\
[e/x](\epsilon s{::}\Psi[\Phi].p) & = \epsilon s{::}\Psi[\Phi].[e/x]p & \\
[e/x](\mathsf{box}(\Psi.\, M) \mapsto e') & = \mathsf{box}(\Psi.\, M) \mapsto [e/x]e' & \\
[e/x](\mathsf{sbox}(\Psi.\, \sigma) \mapsto e') & = \mathsf{sbox}(\Psi.\, \sigma) \mapsto [e/x]e' &
\end{array}
$$

Note that $\mathsf{box}(\Psi.\, M)$ does not contain any free occurrences of program variables x, and therefore substitution has no effect. Similarly, the case for $\mathsf{sbox}(\Psi.\, \sigma)$ where no change is visible when $[e/x]$ is applied to it.

Theorem 4.1 (Substitution on computation-level variables)
If $\Delta; \Gamma \vdash e \Leftarrow \tau$ *and* $\Delta; \Gamma, x{:}\tau, \Gamma' \vdash J$ *then* $\Delta; \Gamma, \Gamma' \vdash [e/x]J$.

Proof. By induction on the structure of the second given derivation. □

A similar substitution principle holds for object-level variables.

Theorem 4.2 (Substitution on object-level variables)
If $\Delta; \Psi \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x{:}A, \Psi' \vdash J$ *then* $\Delta; \Psi, \Psi' \vdash [M/x]J$.

Proof. By induction on the structure of the second given derivation. □

Contextual substitution for meta-variables

Substitutions for contextual variables u are a little more difficult. 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. Because of α-conversion, the variables that are substituted at different occurrences of u may be different, contextual substitution for a meta-variable must carry a context, written as $[\![\Psi.M/u]\!]N$, $[\Psi.M/u]\sigma$, and $[\![\Psi.M/u]e$ where Ψ binds

all free variables in M. This complication can be eliminated in an implementation of our calculus based on de Bruijn indexes. We show contextual substitution into objects-level terms next.

$$
\begin{array}{lll}
[\![\Psi.M/u]\!](x) & = & x \\
[\![\Psi.M/u]\!](\lambda y.N) & = & \lambda y.[\![\Psi.M/u]\!]N \\
[\![\Psi.M/u]\!](R\ N) & = & ([\![\Psi.M/u]\!]R)\ ([\![\Psi.M/u]\!]N) \\
[\![\Psi.M/u]\!](u[\sigma]) & = & [[\![\Psi.M/u]\!]\sigma/\Psi]M \\
[\![\Psi.M/u]\!](v[\sigma]) & = & v[[\![\Psi.M/u]\!]\sigma] \quad \text{provided } v \neq u \\
[\![\Psi.M/u]\!](\cdot) & = & \cdot \\
[\![\Psi.M/u]\!](\sigma, N/y) & = & [\![\Psi.M/u]\!]\sigma, ([\![\Psi.M/u]\!]N)/y \\
[\![\Psi.M/u]\!](s[\rho]) & = & [\![s[([\![\Psi.M/u]\!]\rho)]\!] \\
[\![\Psi.M/u]\!](\mathsf{id}_\psi) & = & \mathsf{id}_\psi
\end{array}
$$

Applying $[\![\Psi.M/u]\!]$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\Psi.M/u]\!]\sigma$, but instead of returning $M[\sigma']$, it proceeds to eagerly apply σ' to M. Before σ' can be carried out, however, it's domain must be renamed to match the variables in Ψ, denoted by σ'/Ψ. Contextual substitution into computation-level expressions is next.

$$
\begin{array}{lll}
[\![\Psi.M/u]\!](x) & = & x \\
[\![\Psi.M/u]\!](\Lambda\psi.e) & = & \Lambda\psi.[\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](e\ \lceil\Psi\rceil) & = & ([\![\Psi.M/u]\!]e)\ \lceil\Psi\rceil \\
[\![\Psi.M/u]\!](\mathsf{rec}\ f.e) & = & \mathsf{rec}\ f.[\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](\mathsf{fn}\ y.e) & = & \mathsf{fn}\ y.[\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](e_1\ e_2) & = & ([\![\Psi.M/u]\!]e_1)\ ([\![\Psi.M/u]\!]e_2) \\
[\![\Psi.M/u]\!](\mathsf{box}(\Phi.\ N)) & = & \mathsf{box}(\Phi.\ [\![\Psi.M/u]\!]N) \\
[\![\Psi.M/u]\!](\mathsf{sbox}(\Phi.\ \sigma)) & = & \mathsf{sbox}(\Phi.\ [\![\Psi.M/u]\!]\sigma) \\
[\![\Psi.M/u]\!](\mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) & = & \mathsf{case}\ [\![\Psi.M/u]\!]e_1\ \mathsf{of}\ [\![\Psi.M/u]\!]p_1 \mid \ldots \mid [\![\Psi.M/u]\!]p_n
\end{array}
$$

The cases for function and recursion do not have to consider capture-avoiding side conditions. Since M must be closed with respect to Ψ and meta-variables u are distinct from computation-level variables x no clashes can happen. Finally, contextual substitution into computation-level branches.

$$
\begin{aligned}
[\![\Psi.M/u]\!](\epsilon v{::}B[\Phi].p) &= \epsilon v{::}B[\Phi].[\![\Psi.M/u]\!]p \quad \text{if } v \notin \mathsf{FMV}(M) \text{ and } v \neq u \\
[\![\Psi.M/u]\!](\epsilon s{::}\Psi[\Phi].p) &= \epsilon s{::}\Psi[\Phi].[\![\Psi.M/u]\!]p \quad \text{if } s \notin \mathsf{FMV}(M) \\
[\![\Psi.M/u]\!](\mathsf{box}(\Phi.\, N) \mapsto e)) &= \mathsf{box}(\Phi.\, N) \mapsto [\![\Psi.M/u]\!]e \\
[\![\Psi.M/u]\!](\mathsf{sbox}(\Phi.\, \sigma) \mapsto e)) &= \mathsf{sbox}(\Phi.\, \sigma) \mapsto [\![\Psi.M/u]\!]e
\end{aligned}
$$

Finally, the cases for the branches are interesting. We require that all the contextual variables occurring in a pattern $N \mapsto e$ are explicitly quantified by ϵ and hence we do not apply the contextual substitution $[\![\Psi.M/u]\!]$ to the object N describing the pattern, but only to e. Contextual substitution satisfies the following substitution property.

Theorem 4.3 (Contextual modal substitution)

(i) *If* $\Delta; \Psi \vdash M \Leftarrow A$ *and* $\Delta, u{::}A[\Psi]; \Gamma \vdash J$ *then* $\Delta; \Gamma \vdash [\![\Psi.M/u]\!]J$.

(ii) *If* $\Delta; \Psi \vdash M \Leftarrow A$ *and* $\Delta, u{::}A[\Psi]; \Phi \vdash J$ *then* $\Delta; \Phi \vdash [\![\Psi.M/u]\!]J$.

Proof. By structural induction on the second derivation. □

Contextual substitution for context variables

Next, we consider substitutions for context variables. Unlike the previous substitution operations which were total, substitution of a context Ψ into a context variable $\psi(\omega)$ may fail if Ψ does not satisfy ω. We start with considering context substitution into computation-level expressions.

$$
\begin{aligned}
[\![\Psi/\psi]\!](x) &= x \\
[\![\Psi/\psi]\!](\Lambda\gamma.e) &= \Lambda\gamma.[\![\Psi/\psi]\!]e \quad \text{provided that } \gamma \notin \mathsf{FV}(\Psi) \\
[\![\Psi/\psi]\!](e\ \lceil\Phi\rceil) &= ([\![\Psi/\psi]\!]e)\ \lceil[\![\Psi/\psi]\!]\Phi\rceil \\
[\![\Psi/\psi]\!](\mathsf{fn}\ y.e) &= \mathsf{fn}\ y.[\![\Psi/\psi]\!]e \\
[\![\Psi/\psi]\!](\mathsf{rec}\ f.e) &= \mathsf{rec}\ f.[\![\Psi/\psi]\!]e \\
[\![\Psi/\psi]\!](e_1\ e_2) &= ([\![\Psi/\psi]\!]e_1)\ ([\![\Psi/\psi]\!]e_2) \\
[\![\Psi/\psi]\!](\mathsf{box}(\Phi.\, N)) &= \mathsf{box}([\![\Psi/\psi]\!]\Phi.\, [\![\Psi/\psi]\!]N) \\
[\![\Psi/\psi]\!](\mathsf{sbox}(\Phi.\, \sigma)) &= \mathsf{sbox}(\Phi.\, [\![\Psi/\psi]\!]\sigma) \\
[\![\Psi/\psi]\!](\mathsf{case}\ e\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) &= \mathsf{case}\ [\![\Psi/\psi]\!]e_1\ \mathsf{of}\ q_1 \mid \ldots \mid q_k \\
&\qquad \text{where } [\![\Psi/\psi]\!]p_i = q_i \text{ for some } i
\end{aligned}
$$

In the case for $\mathsf{box}(\Phi.\, N)$ we apply the substitution $[\![\Psi/\psi]\!]$ to both the context Φ and the object N. While the object N does not contain context variables ψ,

it may contain the identity substitution $\mathsf{id}_{(\psi)}$ which needs to be unfolded. Similar considerations hold for the case $\mathsf{sbox}(\Phi.\,\sigma)$. Note that applying substitution $[\![\Psi/\psi]\!]$ to some of the branches p_i may actually fail, and the substitution operation eliminates these branches, since they are unreachable. Coverage checking [14] will guarantee that there is at least one branch where applying substitution $[\![\Psi/\psi]\!]$ to it will succeed and we can guarantee progress. Hence applying the substitution $[\![\Psi/\psi]\!]$ to an expression is only total if we covered all possible cases which guarantees that there must be at least one case where applying $[\![\Psi/\psi]\!]$ to the branch succeeds. The most interesting case is where actual substitution must happen.

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

$$
\begin{array}{lll}
[\![\Psi/\psi]\!](\cdot) & = \cdot & \\
[\![\Psi/\psi]\!](\sigma, M/x) & = [\![\Psi/\psi]\!]\sigma, [\![\Psi/\psi]\!]M/x & \\
[\![\Psi/\psi]\!](s[\rho]) & = s[[\![\Psi/\psi]\!]\rho] & \\
[\![\Psi/\psi]\!](\mathsf{id}_{\psi(\omega)}) & = \mathsf{id}(\Psi) & \text{if } \Psi \text{ satisfies } \omega. \\
[\![\Psi/\psi]\!](\mathsf{id}_{\phi(\omega)}) & = \mathsf{id}_{\phi(\omega)} &
\end{array}
$$

We recall that the only construct binding a context variable ψ is context abstraction $\Lambda\psi.e$ and $\mathsf{box}(\psi.\,M)$ or $\mathsf{sbox}(\psi.\,\sigma)$ does not bind ψ. Expansion of the identity substitution is defined as follows:

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

Lemma 4.4 *If* $\mathsf{id}(\Psi) = \sigma$ *then* $\Delta; \Psi, \Psi' \vdash \sigma \Leftarrow \Psi$.

Proof. Induction on the structure of Ψ. □

Lemma 4.5

(i) *If* $\Delta, \psi{:}W; \psi(\omega), \Phi \vdash M \Leftarrow A$ *and* $\Psi : W$ *and* Ψ *satisfies* ω *then* $\Delta; \Psi, \Phi \vdash M \Leftarrow A$.

(ii) *If* $\Delta, \psi{:}W; \psi(\omega), \Phi \vdash e \Leftarrow \tau$, *e coverage checks,* $\Psi : W$ *and* Ψ *satisfies* ω *then* $\Delta; \Psi, \Phi \vdash e \Leftarrow \tau$.

Proof. Structural induction on the first derivation. □

Next, we give a brief definition for substituting for substitution variables.

$$
\begin{array}{lll}
[\![\Psi.\sigma/s]\!](x) & = & x \\
[\![\Psi.\sigma/s]\!](\lambda y.N) & = & \lambda y.[\![\Psi.\sigma/s]\!]N \\
[\![\Psi.\sigma/s]\!](N_1\ N_2) & = & ([\![\Psi.\sigma/s]\!]N_1)\ ([\![\Psi.\sigma/s]\!]N_2) \\
[\![\Psi.\sigma/s]\!](u[\rho]) & = & u[[\![\Psi.\sigma/s]\!]\rho] \\
[\![\Psi.\sigma/s]\!](\cdot) & = & \cdot \\
[\![\Psi.\sigma/s]\!](\sigma, N/y) & = & [\![\Psi.\sigma/s]\!]\sigma, ([\![\Psi.\sigma/s]\!]N)/y \\
[\![\Psi.\sigma/s]\!](s[\rho]) & = & [\![[([\![\Psi.\sigma/s]\!]\rho)/\Psi]\sigma \\
[\![\Psi.\sigma/s]\!](s'[\rho]) & = & [\![s'[([\![\Psi.\sigma/s]\!]\rho)] \\
[\![\Psi.\sigma/s]\!](\mathsf{id}_\phi) & = & \mathsf{id}_\phi
\end{array}
$$

Applying $[\![\Psi.\sigma/s]\!]$ to the closure $s[\rho]$ first obtains the simultaneous substitution $\rho' = [\![\Psi.\sigma/s]\!]\rho$, but instead of returning $\sigma[\rho']$, it proceeds to eagerly apply ρ' to σ. Before ρ' can be carried out, however, it's domain must be renamed to match the variables in Ψ, denoted by ρ'/Ψ.

Lemma 4.6

(i) *If* $\Delta; \Psi \vdash \sigma \Leftarrow \Psi'$ *and* $\Delta, s{::}\Psi'[\Psi]; \Phi \vdash J$ *then* $\Delta; \Phi \vdash [\![\Psi.\sigma/s]\!]J$.

(ii) *If* $\Delta; \Psi \vdash \sigma \Leftarrow \Psi'$ *and* $\Delta, s{::}\Psi'[\Psi]; \Gamma \vdash J$ *then* $\Delta; \Gamma \vdash [\![\Psi.\sigma/s]\!]J$.

Proof. By structural induction on the second derivation. □

5 Operational semantics

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

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

Next, we define a small-step evaluation judgment:

$$
\begin{array}{rl}
e \longrightarrow e' & \text{Expression } e \text{ evaluates in one step to } e'. \\
\Delta \vdash \mathsf{box}(\Psi.\, M) \doteq p \longrightarrow e' & \text{Branch } p \text{ matches } \mathsf{box}(\Psi.\, M) \text{ and steps to } e' \\
\Delta \vdash \mathsf{sbox}(\Psi.\, \sigma) \doteq p \longrightarrow e' & \text{Branch } p \text{ matches to } \mathsf{sbox}(\Psi.\, \sigma) \text{ and steps to } e'
\end{array}
$$

In the judgment for branches, we note that $\mathsf{box}(\Psi.\, M)$ does not contain any meta-variables, i.e. it is closed, and Δ characterizes the meta-variables occurring

in the branch p. We only concentrate on three interesting cases, where actual computation happens. 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 $\Lambda\psi.e$. Finally, the case for pattern matching against $\mathsf{box}(\Psi.\, M)$ and $\mathsf{sbox}(\Psi.\, \sigma)$. Here we need to propagate object-level terms via contextual substitution.

$$\frac{}{(\mathsf{fn}\ y.e)\ v \longrightarrow [v/y]e} \qquad \frac{}{(\Lambda\psi.e)\ [\Psi] \longrightarrow [\![\Psi/\psi]\!]e}$$

$$\frac{\cdot \vdash \mathsf{box}(\Psi.\, M) \doteq p_i \longrightarrow e'}{(\mathsf{case}\ (\mathsf{box}(\Psi.\, M))\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) \longrightarrow e'}$$

$$\frac{\cdot \vdash \mathsf{sbox}(\Psi.\, \sigma) \doteq p_i \longrightarrow e'}{(\mathsf{case}\ (\mathsf{sbox}(\Psi.\, \sigma))\ \mathsf{of}\ p_1 \mid \ldots \mid p_n\ ) \longrightarrow e'}$$

Since evaluation relies on pattern matching object-level terms, we describe briefly this process. In particular, we rely on higher-order pattern matching to match $\mathsf{box}(\Psi.\, M)$ against $\mathsf{box}(\Psi.\, M') \mapsto e$. Higher-order patterns in the sense of Miller [7] restrict syntactically the occurrences of contextual modal variables $u[\sigma]$. The pattern restriction enforces that the substitution σ which is associated with the contextual modal variable u only maps variables to variables and has the following form: $y_1/x_1, \ldots, y_n/x_n$. This ensures that higher-order pattern matching remains decidable in the presence of λ-abstraction. The judgment for higher-order pattern matching can be described as follows:

$$\Delta; \Psi \vdash M \doteq M'/\theta \ \ M \text{ matches } M' \text{ s.t. } [\![\theta]\!]M' = M$$

A description of higher-order pattern matching for contextual modal variables can be found in [10]. It seems feasible to extended this description to incorporate also substitution while preserving correctness and crucial invariants of higher-order pattern matching such as

(i) θ has domain Δ and instantiates all modal variables in Δ s.t. $\cdot \vdash \theta : \Delta$.

(ii) $M = [\![\theta]\!]N$, i.e. object M is syntactically equal to $[\![\theta]\!]N$.

We are now in a position to describe computation-level pattern matching of $\mathsf{box}(\Psi.\, M)$ against a pattern p. Pattern matching of $\mathsf{sbox}(\Psi.\, \sigma)$ follows similar ideas.

$$\frac{\Delta, v{::}A[\Phi] \vdash \mathsf{box}(\Psi.\, M) \doteq p \longrightarrow e'}{\Delta \vdash \mathsf{box}(\Psi.\, M) \doteq (\epsilon v{::}A[\Phi].p) \longrightarrow e'}$$

$$\frac{\Delta, s{::}\Phi'[\Phi] \vdash \mathsf{box}(\Psi.\, M) \doteq p \longrightarrow e'}{\Delta \vdash \mathsf{box}(\Psi.\, M) \doteq (\epsilon s{::}\Phi'[\Phi].p) \longrightarrow e'}$$

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

Given the current setup, we can prove type safety for our proposed functional language with higher-order abstract syntax and explicit substitutions. Let $|e|$ be the erasure of all type assignments of e.

Theorem 5.1

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

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

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

6 Examples

In this section, we will show several examples to illustrate the potential applications of the ideas presented. All the examples require context variables to denote an open world we recurse over. This world is known during run-time, but changes. We believe that the foundation is strong enough to handle most of the examples using higher-order abstract syntax from the Twelf repository [12]. The examples do not yet make essential use of substitution variables and first-class substitutions.

Variable counting

First, we show a very simple function which counts the bound variables occurring in an expression defined using higher-order abstract syntax. We assume, we have declared a data-type exp for expressions using higher-order abstract syntax which contains the objects $\mathsf{lam} : (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$ and $\mathsf{app} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$.. We assume we have available basic computation-level types such as nat, string or bool.

$$
\begin{array}{lll}
\mathsf{rec}\ cnt. & & \\
\Lambda\gamma.\mathsf{fn}\ e.\mathsf{case}\ e\ \mathsf{of} & \mathsf{box}(\gamma(x{:}exp).\, x) & \mapsto 1 \\
\quad \mid \epsilon u{::}\mathsf{exp}[\gamma, x'{:}\mathsf{exp}]. & \mathsf{box}(\gamma.\, \mathsf{lam}\ \lambda x{:}\mathsf{exp}.u[\mathsf{id}_\gamma, x/x']) & \mapsto \\
 & \quad cnt\ \lceil \gamma, x{:}\mathsf{exp} \rceil\ (\mathsf{box}(\gamma, x{:}\mathsf{exp}.\, u[\mathsf{id}_\gamma, x/x'])) & \\
\quad \mid \epsilon u{::}\mathsf{exp}[\gamma]\ \epsilon v{::}\mathsf{exp}[\gamma]. & \mathsf{box}(\gamma.\, \mathsf{app}\ u[\mathsf{id}_\gamma]\ v[\mathsf{id}_\gamma]) & \mapsto \\
 & \quad cnt\ \lceil \gamma \rceil\ (\mathsf{box}(\gamma.\, u[\mathsf{id}_\gamma])) + cnt\ \lceil \gamma \rceil\ (\mathsf{box}(\gamma.\, v[\mathsf{id}_\gamma])) &
\end{array}
$$

Note we need to use a context variable γ to denote our context of variables which may occur in the term e and which will be built up during recursion. Only during runtime, do we know the actual context. Omitting some type information (which we think of being implicit) and following Twelf-like syntax where λ-abstraction is denoted by `[x] ...` this can be beautified to:

```
rec cnt: Pi g:expW. exp[g] -> int =
Lam g => fn e =>
case e of (box g(x). x)  => 1
        |(box g. (app u[id_g] v[id_g])) =>
           cnt [g] (box g. u[id_g]) +  cnt(box g. v[id_g])
        |(box g. (lam [x] u[id_g, x/x]) =>
           cnt [g,x] (box g,x. u[id_g, x/x])
```

Double negation translation

In this example, we give a functional implementation of translating first-order formulas using double negation. Formulas can be defined using higher-order abstract syntax in a straightforward way, where we define a type `i` for individuals and a type `prop` for propositions. We only concentrate on the fragment for implication and universal quantification here, and include an equality predicate here. The constant `eq` will have type `i -> i -> prop`, the constant `all` has type `(i -> prop) -> prop` and the constant `imp` has type `prop -> prop -> prop`. Next, we present a program which translates propositions via double-negation. Let `iW` denote a context where we have individuals, i.e. `x1:i, ...  xn:i`. To ease readability, we use `let`-expressions `let box g'.u = e in e' end` as an abbreviation for `case e of box g'.u => e'`.

```
rec dneg:  Pi g: iW. prop[g] -> prop[g] =
Lam g => fn e  =>
case e of (box g. eq t1[id_g] t2[id_g]) =>
            (box g. neg (neg (eq t1[id_g] t2[id_g])))
        | (box g. imp f1[id_g] f2[id_g]) =>
          let box g. u1 = dneg [g] (box g. f1[id_g])
              box g. u2 = dneg [g] (box g. f2[id_g])
          in (box g. neg(neg (imp u1[id_g] u2[id_g]))) end
        | (box g. all [x] f[id_g, x/x]) =>
          let box g,x. u = dneg [g,x] (box g,x. f[id_g, x/x'])
          in box (g. neg(neg(all [x] u[id_g, x/x']))) end
```

Substitution-based evaluator

Finally, we the functional implementation of a substitution-based evaluator. We first define the following data-type declaration for numbers and expressions and use higher-order abstract syntax to denote the binder in the let-expression.

```
z nat.                          Add: exp -> exp -> exp.
suc: nat -> nat.                Let: exp -> (nat -> exp) -> exp.
num: nat -> exp
```

We assume we defined a function for addition `add:nat[]*nat[] -> nat[]`. Then we can define a simple evaluator in a straightforward way. If we encounter a let-expression `let x = e in e' end` then we first evaluate the expression `e` to some value `v`, and then we replace all occurrences of the binder `x` in `e'` with the value `v`.

This is handled by building the closure `u[id_g,v[]/x]`. It should be obvious how to extend this evaluator to other forms of arithmetic expressions.

```
rec eval: exp[] ->  nat[] =
fn e => case e of(box . Nat n[])  => (box . n[])
                | (box . Add(e1[], e2[])) =>
                  let val a = eval (box . e1[])
                      val b = eval (box . e2[])
                  in add (a, b) end
                | (box . Let(e[], [x]. u[x/x]))  =>
                  let box v = eval (box . e[])
                  in eval(box . u[v[]/x]) end
```

While the substitution-model has many advantages from a theoretical point of view, in an implementation it is usually considered too expensive. Alternatively, we can use an environment model where we associates variables with values in an environment. When we evaluate a let-expression `let x = e in e' end` then we evaluate the expression `e` to some value `v`, and then evaluate `e'` in an environment where we associate the binder `x` with the value `v`. When we encounter a variable `x`, we lookup its value in the environment. Explicit substitutions seem ideally suited to model the run-time environment. In the future, we plan to explore this direction further.

7 Related Work

Techniques for supporting higher-order abstract encodings in functional programming languages have received wide spread attention. One of the first proposals for functional programming with support for binders and higher-order abstract syntax was presented by Miller [6]. Later, Despeyroux, Pfenning, and Schürmann, have developed proof-theoretic foundations for programming and reasoning with higher-order abstract syntax [2,15] based on modal types. However, there are no contextual types and their theoretical development lacks first-class meta-variables and a context of meta-variables. This has deep consequences for the theoretical development and leads to difficulties in proving progress of the proposed language. Since in our framework the type of a meta-variable determines its local scope, local scope is naturally enforced. No scope stacks and operations on them is required, which simplifies the theoretical development. For example, the operational semantics does not have to take into account a stack of contexts and there are no explicit operations for popping a context of the stack in the proposed language. Instead we naturally enforce that we can only evaluate closed expressions, i.e. expressions which do not contain any meta-variables. This seems to be critical to achieve a proof for progress and preservation. In addition, context abstraction in our setting allows us to enforce stronger invariants about programs since we can distinguish between different context and different worlds. The nature of the nabla-quantifier allows only reasoning within one world or context. This seems to be contained in the fragment we present where we have only one context variable γ and all argu-

ments depend on this context γ. However, our framework is richer in the sense that it allows us to consider different context variables. Moreover, we propose to extend the framework with explicit substitutions, which seem interesting in its own, although their full impact still needs to be explored.

In functional programming, various formulations of context as a primitive programming construct have been considered [16,17,8,5,4]. Nishizaki [8] for example extends a lambda-calculus with explicit substitutions in the spirit of the explicit substitution calculus proposed by Abadi *et. al.*[1]. 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 M. Sato et al. [17] 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 $\lambda\kappa\epsilon$ 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 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.

8 Conclusion future plans

We have sketched a foundation for functional programming with higher-order abstract syntax and explicit substitution based on contextual modal types. Our proposal builds on earlier ideas by [2,15] where modal types have been used to distinguish between object-level terms and computation-level programs. In contrast to earlier proposals, we distinguish between contextual meta-variables and ordinary variables which we believe leads to a cleaner and more expressive framework for programming with higher-order abstract syntax. The distinction between contextual meta-variables and ordinary bound variables has already provided interesting insights into higher-order proof search, higher-order unification and logical frameworks in general (see for example [11,10]). It has allowed us to clarify many theoretical issues and invariants related to the interplay of meta-variables (= contextual variables) and ordinary variables. Contextual modal types also have been applied to staged functional programming to generate code which may possibly be open [9]. In this paper, we apply contextual modal types to programming with open terms based on higher-order abstract syntax.

There are many aspects left to consider. One important aspect of this work is that our proposal ensures the adequate encoding of on-paper formulations. We believe that it is possible to prove adequacy about our examples since our object-level theory draws on ideas from logical frameworks, and adequacy proofs for encodings

within logical frameworks are standard. In the future, we also hope to gain a better understanding of the expressiveness of the presented language. Some features such as substitution variables have not been used to their full power in practice yet. The status of context variables and the operations allowed on them may also need to be expanded for some practical examples. Finally, an interesting aspect is how we can combine datatypes defined via higher-order abstract syntax with ordinary types.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Symposium on Principles of Programming Languages, POPL'90*, pages 31–46, San Francisco, California, 1990. ACM.

[2] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *International Conference on Typed Lambda Calculus and Applications, TLCA'97*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[3] 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.

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

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

[6] 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.

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

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

[9] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *submitted*, 2005.

[10] Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Computer Science Department, Carnegie Mellon University, December 2003.

[11] Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *International Conference on Automated Deduction, CADE'03*, Lecture Notes in Computer Science (LNAI 2741), pages 473–487, Miami, Florida, 2003. Springer.

[12] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction, CADE'99*, volume 1632 of *Lecture Notes in Artificial Inteligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.

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

[14] 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, September 2003. Springer-Verlag LNCS 2758.

[15] 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), Nara, Japan, April 21-23, 2005*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

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

[17] 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.