Programming with Binders and Indexed Data-Types

Andrew Cave   Brigitte Pientka
McGill University
{acave1,bpientka}@cs.mcgill.ca

Abstract

We show how to combine a general purpose type system for an existing language with support for programming with binders and contexts by refining the type system of ML with a restricted form of dependent types where index objects are drawn from contextual LF. This allows the user to specify formal systems within the logical framework LF and index ML types with contextual LF objects. Our language design keeps the index language generic only requiring decidability of equality of the index language providing a modular design. To illustrate the elegance and effectiveness of our language, we give programs for closure conversion and normalization by evaluation.

Our three key technical contribution are: 1) a bi-directional type system for our core language which is centered around refinement substitutions instead of constraint solving. As a consequence, type checking is decidable and easy to trust, although constraint solving may be undecidable. 2) a big-step environment based operational semantics with environments which lends itself to efficient implementation. 3) We prove our language to be type safe and have mechanized our theoretical development in the proof assistant Coq using the fresh approach to binding.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Design, Languages

Keywords Logical frameworks, higher-order abstract syntax, dependent types, recursive types

1. Introduction

To reason about the runtime behavior of software, we routinely design and use formal systems, given by axioms and inference rules, such as logics to reason about access control [Abadi et al. 1993, 1999; Garg and Pfenning 2006] and information flow [Miyamoto and Igarashi 2004], logics to reason about memory access [Nanevski et al. 2008a] or simply the scope of names [Pottier 2007]. Over the last decade we have come closer to narrowing the gap between programming software systems and reasoning about them [Chen and Xi 2005; Sheard 2004; Westbrook et al. 2005]. The general mantra is to design rich type systems which allow programmers to specify and enforce statically powerful invariants about their programs

Yet, existing approaches lack rich abstractions that allow users to describe formal systems and proofs on a high-level, factor out common and recurring issues, make it easy to use, and at the same time have a small trusted kernel.

In this paper, we extend a general purpose language to support programming with formal systems and ultimately proofs. This is achieved by indexing types with contextual LF objects [Pientka 2008]. Contextual LF extends the logical framework LF [Harper et al. 1993] with the power of contextual objects $\hat{\Psi}.M$ of type $A[\Psi]$. M denotes an object which may refer to the bound variables listed in $\hat{\Psi}$ and has type A in the context Ψ (see also [Nanevski et al. 2008b]). It also supports first-class contexts and allows us to abstract over contexts. This allows the user to specify formal systems within the logical framework LF and obtaining support for representing and managing binders, renaming, fresh name generation, and capture-avoiding substitutions. Contextual LF allows programmers to pack "open" LF objects together with the context in which they are meaningful thereby obtaining closed objects which can be passed and manipulated. In particular, contextual LF objects can be used to index types and track rich formal properties. We demonstrate the advantages of combining contextual LF with data-types by discussing implementations of closure conversion and normalization by evaluation. These examples have been a good benchmark in comparing systems and demonstrate the benefits and elegance of our approach.

Fundamentally, our approach follows the tradition of indexed types (see Xi and Pfenning [1999]; Zenger [1997]) choosing as an index domain contextual LF which allows us to express properties about open objects, the scope of variables and contexts which previous systems such as ATS/LF [Xi 2004] lack. Instead of generating and propagating constraints which is common in indexed type systems, we will associate patterns with refinements substitution and work only with constraints in solved form. This leads to a small trusted kernel.

Compared to languages such as Beluga [Pientka and Dunfield 2010] or Delphin [Poswolsky and Schürmann 2009, 2008] which support programming with binders already, the language we propose in this paper supports recursive types which the former languages lack; this extension is key to tackle important problems such as normalization by evaluation which are typically out of reach for these languages. Having the ability to define data-types is convenient; it is also more efficient than supporting data-types via a Church-encoding which is in principle already possible.

In contrast to Licata and Harper [2009] which supports mixing binding and computation, we keep the separation of data and computations. This has several advantages: Because our index language remains pure, it is straightforward to establish adequacy of the formal system and its encoding using standard techniques (see for example Harper and Licata [2007]). This allows us to maintain all the good properties of contextual objects, namely strong normalization and decidable equality. Moreover, our computation language remains close to traditional ML-like languages and is designed to be modular in the index domain. As a consequence, one can easily replace contextual LF with for example a higher-order logic with inductive types without affecting the computations. Compared to other systems with a uniform language for computations and types such as the Calculus of Construction or Martin Löf type theory, our computation language can easily be combined with imperative features, allows non-termination computation, and requires fewer annotations to make type checking decidable.

The main technical contributions of this paper are:

- We present a core language with dependent types where we separate the types from computations. One can think of this core language as the target of elaborating a surface language where we may omit implicit indices via type reconstruction. Our language is a conservative extension of a general purpose language where types are indexed by contextual LF objects and contexts and at the same time supports pattern matching on its index objects. Because it may be viewed as an extension of Beluga with recursive types, we call our language Beluga$^\mu$. However, we emphasize that our design of the computation language is generic and we can replace the index language which in our case is contextual LF with any other language where equality between two index objects is decidable.
- We present a bi-directional type system for our core language. Theoretically, we model dependently typed recursive types as fixpoints with explicit equality constraints on contextual objects. This is similar to Xi et al. [2003] and Sulzmann et al. [2007] where recursive types are endowed with equalities between types to model GADTs. However, instead of accumulating and solving constraints during type checking, our approach relies on refinement substitutions in branches. Although constraint solving may be undecidable in our setting, type checking based on refinements remains decidable. Type checking is hence easier to trust.
- We give a big-step environment-based operational semantics. Because we allow matching on computation-level expressions and index objects, we distinguish between the environment for computation-level values and index objects and show that types are preserved. By extending our operational semantics to also track diverging computations following Cousot and Cousot [1992]; Leroy and Grall [2009], we prove progress.
- We have mechanized the typing rules, the operational semantics and the type safety proof (preservation and progress) in the proof assistant Coq. We use the fresh approach to binding by Pouillard and Pottier [2010] to model variables in our language.

The proposed programming language with support for contextual objects and data-types is a prime candidate for programming code transformations and certified programming. More generally, it provides a foundation for programming with domain-specific logics and demonstrates how to endow a general purpose programming language with direct support for programming with logics and proofs.

The remainder of the paper is organized as follows. To illustrate the main idea of defining types indexed with contextual objects, we discuss in detail three examples (Section 2): closure-based evaluator, closure conversion and normalization by evaluation. We then introduce Beluga$^\mu$, a language which supports contextual objects and computation-level data-types in Section 3. We begin by introducing our index domain in Section 3.1 which in our case is contextual LF [Pientka 2008]. This will review and summarize previous work in this area. We then present the computation language in Section 4 which includes indexed types, recursive types, variants and general pattern matching. We will leave out polymorphism and imperative features which are orthogonal issues and which are straightforward to add. The typing rules for Beluga$^\mu$ and an environment-based big-step semantics together with the type safety proof are presented in Section 4.4. We explain our mechanization of the type safety proof in Section 5. The rest of the paper is concerned with some related work, current status and future research directions.

2. Motivating Examples

In this section, we discuss three examples which illustrate the utility of a language combining contextual types with indexed recursive types. We use an informal surface syntax inspired by both Beluga and Agda [Norell 2007]. This surface syntax is intended to elaborate to the core language presented in Section 4.

2.1 Closure-based evaluator for the lambda-calculus

We begin with a demonstration of a closure-based evaluator for the untyped lambda-calculus which is directly comparable to that of Licata and Harper [2009]. This allows us to explain contextual objects in a simple setting. We first represent the untyped lambda calculus in LF with higher order abstract syntax (HOAS).

```
lam: (tm → tm) → tm.
app: tm → tm → tm.
```

To implement an evaluator, we must analyze and pattern match on lambda-terms and consequently we must be able to handle open objects. Hence, the evaluator will be parameterized with a context ψ which keeps track of variables of type `tm` and manipulates contextual objects of type `tm[ψ]`. To type contexts we define a *context schema* `ctx` as follows: **schema** `ctx = tm;` To express that ψ contains only variables of type `tm` we write `ψ:ctx`.

The highlight of this example, when compared to Beluga or Delphin, is that we can write the closure of a term under an environment of bindings as a computation-level data-type with a single constructor `cl` as follows:

```
datatype  clos :  ctype =
 cl : {ψ:ctx} → tm[ψ,x:tm] → (#tm[ψ] → clos) → clos
type  envr ψ = #tm[ψ] → clos
```

We write `tm[ψ]` for the type of terms whose free variables come only from the context ψ. We write `#tm[ψ]` for the type of variables of type `tm` in context ψ – i.e. elements of ψ. We overload $\rightarrow$ using it for LF types as well as computation-level types; since **datatype** is part of the *computation* language, the arrow in `#tm[ψ ] → clos` does not indicate the LF function space, but rather the usual computational function space. Hence this datatype represents the bodies of lambda abstractions in some context together with a binding for each of the free variables minus one. Note that we wrap arguments in $\{\}$ to indicate that they are passed *implicitly*.

We can now proceed to evaluate a term in a context ψ and in an environment which provides bindings for the variables in ψ.

```
rec  eval : {ψ:ctx} tm[ψ] → envr ψ  → clos =
fn  e ⇒ fn  env ⇒ case  e of
| ψ. #p .. ⇒ env (ψ. #p ..)
| ψ. lam (λx. E .. x) ⇒ cl (ψ,x:exp . E .. x) env
| ψ. app (E1 ..) (E2 ..) ⇒
 let  cl (φ,x:tm . E .. x) env' = eval (ψ. E1 ..) env
 let  v = eval (ψ. E2 ..) env in
 in  eval  (φ,x:tm . E .. x)  (fn  var ⇒ case  var of
                 | φ,x:tm . x ⇒ v
                 | φ,x:tm . #p .. ⇒ env' (φ. #p .. ))
```

When we pattern match on a contextual object `e : tm[ψ]`, we might obtain a variable in ψ, which we write as `ψ. #p ...`. The item `#p` is a *parameter variable*, standing for a position in ψ. Its association with the identity substitution for ψ (written `..`) turns it from a position into a genuine `tm[ψ]`. In this case, we look it up in the environment. A lambda abstraction simply evaluates to a closure. We think of the ψ in `ψ. lam (λx. E .. x)` as binding all the free variables of `lam (λx. E .. x)`. We explicitly apply ψ's identity substitution `..` to `E` in the pattern to indicate that the variables in ψ *are* permitted to occur in `E`. Conversely, writing the pattern as `ψ. lam (λx. E x)` attempts to strengthen the variables of ψ out of `E`, which is not the intention.

An application evaluates the function position to obtain a body `E` in the *possibly different* context ϕ with an associated `env'` providing bindings for the variables in ϕ. We now evaluate the body `E` in the environment `env'` extended with the appropriate binding for `x`.

This implementation is quite close to that of Licata and Harper [2009]. One noticeable difference is that contexts appear explicitly in our definition of `clos`, which is arguably more readable.

2.2 Relating Contexts: Closure Conversion

When implementing a transformation between languages, as is common in compilers, we often need the resulting terms to be in a different but related context. In systems such as Twelf [Pfenning and Schürmann 1999] and Beluga, the solution is somewhat unsatisfactory. The programmer states results in a combined context, and relies on sophisticated subordination and subsumption mechanisms. However, by encoding this relation on contexts as an inductive predicate, we can express this directly and do away with world subsumption.

We demonstrate this idea with an implementation of closure conversion. Guillemette and Monnier [2007] provide a good overview of closure conversion. Our implementation resembles theirs, although we use HOAS for our term representations. The particulars of closure conversion are not tremendously important here. We wish primarily to demonstrate how data-types complement contextual types.

The source language is the language of Section 2.1. The target language is augmented with constructs for explicit closures, shown in part below:

```
clam : (envr → ctm) → ctm.   proj : envr → nat → ctm.
close : ctm → envr → ctm.    nil : envr.
create : envr → ctm.         snoc : envr → ctm → envr
```

An `envr` is an arbitrary length tuple of terms. Closure conversion will turn contexts into explicit `ctm` objects such that the bodies of lambda abstractions will only refer to their `envr` and no other variables. The reason is that we want to be able to hoist them to the top level. `close` constructs an explicit closure which binds all but the last argument. `proj E N` projects out the `N`th component of the environment `E`.

We must now do as we promised and turn the *open* bodies of lambdas into *closed* terms which instead project from their environment. To characterize the free variables in the open bodies we define the context `cctx`. We depart from Guillemette and Monnier [2007] by performing substitutions instead of let-bindings, which is easy thanks to HOAS.

```
schema cctx = ctm;
```

```
rec addProjs : (φ:cctx) (N:nat[]) (M:cexp[φ,e:envr])
  → cexp[e:envr] =
λ φ ⇒ λ N ⇒ λ M ⇒ case  φ of
| [] ⇒ [e:envr]. M e
| φ,x:ctm ⇒ addProjs φ (s N) (φ,e. M .. (proj e N) e)
```

We write `[]` for the empty context and `[e:envr]` for a singleton context containing a single variable of type `envr`.

Of course if we insist that contexts be passed explicitly as environments, we had better be able to turn contexts into environments:

```
rec ctxToEnv : (φ:cctx) envr[φ] =
λ φ ⇒ case  φ of
| [] ⇒ []. nil
| φ,x:ctm ⇒ let  φ. env .. = ctxToEnv φ in
  φ,x:ctm. snoc (env ..) x
```

So far this is more or less standard Beluga code. The *raison d'être* for this example is the recursive relation on contexts:

```
datatype  ctx_rel : ctx → cctx → ctype
| rnil : ctx_rel [] []
| rsnoc : {ψ φ} → ctx_rel ψ φ
  → ctx_rel (ψ,x:tm) (φ,x:ctm)
```

We freely omit types where they might reasonably be inferred.

`ctx_rel ψ φ` states only that ψ and ϕ are the same length. We illustrate more sophisticated relations later.

The closure conversion function takes vanilla terms in a context ψ into target language terms in a related context ϕ. We only explain the cases for variables and lambda-abstraction. The full implementation can be found in the appendix.

```
rec  conv : {ψ:ctx} (φ:cctx) → ctx_rel ψ φ → tm[ψ]
  → ctm[φ] = λ φ ⇒ fn  cr  ⇒ fn  m ⇒ case  m of
```

Variables are taken to corresponding variables. When we learn that ψ is non-empty, we learn by inspecting `cr` that so too is ϕ. The $\{\phi = ...\}$ syntax passes the implicit argument ϕ explicitly.

```
| ψ',x:tm. x ⇒ let  rsnoc {φ=φ',x:ctm} _ = cr  in  φ',x. x
| ψ',x:tm. #p .. ⇒ let  rsnoc {φ=φ',x:ctm} cr' = cr  in
 let  φ'. M .. = conv _ cr' (ψ'. #p ..) in φ',x. M ..
```

To closure convert a lambda, we closure convert the body in the extended context `ψ,x` using an extended context relation `rsnoc cr`. We *close* it and return an explicit closure. For simplicity, we cheat and do not compute the set of variables occuring in the body: we instead close over the entire context.

```
| ψ. lam (λx. M .. x) ⇒
 let  φ, x:ctm. M' .. x =
   conv _ (rsnoc cr) (ψ,x. M .. x) in
 let  [ev:envr]. M'' ev =
   addProjs _ z (φ, x:ctm, ev:envr. M' .. x) in
 let  φ. Env .. = ctxToEnv φ in
 φ.  close (clam (λev. M'' ev)) (Env ..)
```

We omit here the case for applications which is straightforward.

A more sophisticated example of a context relation appears if we wish to express that a transformation is type preserving. In type-preserving closure conversion (see Minamide et al. [1996] and Guillemette and Monnier [2007]), there is also a non-trivial relation between source language types and target language types. Assuming now that we use *intrinsically-typed* terms, we might wish to use a relation such as the following:

```
datatype  ctx_rel : ctx → cctx → ctype =
| rnil : ctx_rel [] []
| rsnoc : {ψ φ} {T:tp[]} {S:ctp[]} → ctx_rel ψ  φ
  → tp_rel T S → ctx_rel (ψ, x:tm T) (φ, x:ctm S)
```

2.3 Logical Relations: Normalization by Evaluation

Implementing normalization proofs or normalization by evaluation (NbE) in a system such Beluga has been difficult to do directly. Here we demonstrate an implementation of typed normalization by evaluation [Berger and Schwichtenberg 1991] in our language which supports both contextual types and indexed data-types.

The essence of normalization by evaluation is to normalize object level terms by reusing the evaluation of the computation level. There are therefore two levels of terms: object language terms using the LF function space, and computation level (semantic) terms using the computation level function space. Normalization proceeds by interpreting object level terms as semantic terms and reifying the result.

The source language is `tm`: a standard (intrinsically) simply-typed family of LF-level terms with `lam` and `app` as constructors.

We have an open type `atomic_tp` of atomic (base) types which our implementation is essentially parametric over. The target is simply-typed terms in β-normal η-long form:

```
nlam : (neut T → norm S) → norm (arr T S).
rapp : neut (arr T S) → norm T → neut S.
embed : neut (atomic P) → norm (atomic P).
```

To characterize the free variable context of the source language, we define **schema** `ctx = ` **some** `[T:tp] tm T`. The context of the target language is described by **schema** `tctx = ` **some** `[T:tp] neut T`.

Semantic terms must be defined as a datatype, because we must use the computation-level function space:

```
type  sub ψ φ = {T:tp[]} → #(neut T)[ψ] → #(neut T)[φ]

datatype   sem : ctx → tp[] → ctype =
| syn : {ψ} {P:atomic_tp[]}
   → (neut (atomic P))[ψ] → sem ψ (atomic P)
| slam : {ψ A B} ({φ} → sub ψ φ → sem φ A → sem φ B)
   → sem ψ (arr A B)
```

The type `{T:tp[]} → #(neut T)[ψ] → #(neut T)[φ]` is read as the type of (type preserving) substitutions of variables in ψ for variables in ϕ. The need for this will become clear in our implementations of substitution and reification. Observe that we restrict the embedding of `neut` into `sem` to atomic types. This enforces η-longness.

We must manually implement substitution of variables for variables in `sem`, since we do not provide substitution for free for datatypes. In fact, it is substantially different from LF substitution. Licata [2011] explains this difference in depth.

The interesting case is `slam`. It makes essential use of the quantification over substitutions in `slam`. Similar mechanisms appear in both Licata and Harper [2009] and Pouillard and Pottier [2010].

```
rec  subst:{ψ φ S} → sub ψ φ → sem ψ S → sem φ S =
fn  σ ⇒ fn  e ⇒ case  e of
| syn (ψ. R ..) ⇒ nsubst σ (ψ. R ..)
| slam f ⇒ slam (fn σ' ⇒ fn s ⇒ f (σ' ∘ σ) s)

rec  nsubst : {ψ φ S} → sub ψ φ → (neut S)[ψ]
 → (neut S)[φ] = ...
```

Where `nsubst` performs substitution on syntactic neutral terms. We show the full code for this example in the appendix.

Embedding `neut` into `sem` is only possible at atomic types, so we must η-expand in the general case:

```
rec  reflect : (ψ A) (R:(neut A)[ψ]) → sem ψ A =
λ ψ ⇒ λ A ⇒ λ R ⇒ case  []. A of
| []. atomic P ⇒ syn (ψ. R ..)
| []. arr T S ⇒ slam (λ {φ} ⇒ fn σ ⇒ fn  s ⇒
 let  φ. R' .. = nsubst σ (ψ. R ..) in
 let  φ. N .. = reify _ T s in
 reflect _ S (φ. rapp (R' ..) (N ..)))
```

We can then reify semantic terms as object level terms by calling the computation level functions on fresh variables:

```
rec  reify : (ψ A) → sem ψ A → (norm A)[ψ] =
λ ψ ⇒ λ A ⇒ fn  s ⇒ case  []. A of
| []. atomic P ⇒ let  syn (ψ. R ..) = s
 in  ψ. embed (R ..)
| []. arr T S ⇒ let  slam f = s in
 let  ψ,x:tm T. E .. x =
   reify (f weaken (reflect _ T (ψ,x:neut T. x)))
 in  ψ. nlam (λx. E .. x)
```

Where `weaken` is the weakening substitution of type:

```
{ψ:ctx} {S:tp[]} → sub ψ (ψ,x:tm S)
```

We can now implement evaluation with the help of an environment of bindings. The `lam` case evaluates in the extended environment. In the application case, the evaluation of `E1` must produce an `slam` since `syn` is only applicable to atomic types.

```
rec  eval : {ψ φ S} →
  → ({T} #(tm T)[ψ] → sem φ T)
  → (tm S)[ψ] → sem φ S =
fn  r ⇒ fn  σ ⇒ fn  e ⇒ case  e of
| ψ. #p ..  ⇒ σ (ψ . #p .. )
| ψ. lam (λx. E .. x) ⇒ slam (fn σ' ⇒ fn  s ⇒
    eval (extend ((subst σ') ∘ σ) s) (ψ,x. E .. x)
| ψ. app (E1 ..) (E2 ..) ⇒
   let  slam f = eval σ (ψ. E1 ..) in
   f id (eval σ (ψ. E2 ..))
```

We have used `extend` to extend the domain of the environment in the `lam` case. Its type is shown below.

```
rec  extend : {ψ:tctx} {φ:ctx} {S}
  → ({T} → #(tm T)[ψ] → sem φ T) → sem φ S
  → ({T} → #(tm T)[ψ,x:tm S] → sem φ T)
```

Normalization is then simply evaluation followed by reification:

```
rec  nbe : {A} → (tm A)[ ] → (norm A)[ ] =
fn  e ⇒ reify [] A (eval (fn  y ⇒ impossible y) e)
```

Notably, this implementation cleanly enforces η-longness and type preservation by employing dependent types. Expressing these invariants together with a clean approach to variable binding in NbE is rarely found in other work. Licata and Harper [2009] and Shinwell et al. [2003] lack dependent types. Pouillard and Pottier [2010] have this ability, although their implementation is untyped NbE, and hence does not employ it. Further, this illustrates that arbitrary mixing of computation and LF function spaces is not crucial to NbE, as suggested by Licata and Harper [2009]. Their framework, however, obtains weakening for free for `sem` while we have to do a modicum of work to implement it.

From a logic perspective, this can be seen as a (partial) completeness and consistency proof for a natural deduction system by the method of *logical relations*. We say *partial* because we postpone the issues of totality checking to future work. We anticipate that this can be scaled to normalization proofs for the simply-typed lambda calculus without any difficulty. The addition of indexed data-types hence brings substantial benefit to programming and proof systems such as Beluga, since it makes such systems capable of proofs by logical relations.

3. A Review of Contextual LF

Here we describe the index domain, which in our case is contextual LF [Pientka 2011, 2008] which builds on contextual types which were first introduced in Nanevski et al. [2008b].

3.1 Contextual LF

Contextual LF extends the logical framework LF [Harper et al. 1993] with the power of contextual objects $\hat{\Psi}.M$ of type $A[\Psi]$. M denotes an object which may refer to the bound variables listed in $\hat{\Psi}$ and has type A in the context Ψ (see also [Nanevski et al. 2008b]). $\hat{\Psi}$ can be obtained from the context Ψ by simply dropping the type annotations and keeping only the declared variable names. We characterize only objects in $\beta\eta$ normal form, since these are the only meaningful objects in LF. Furthermore, we concentrate here on characterizing well-typed terms, but defining kinds and kinding rules for types is straightforward and omitted.

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

Normal objects may contain *ordinary bound variables* which are used to represent object-level binders and are bound by λ-abstraction or in a context Ψ. They may also contain meta-variables $u[\sigma]$ and parameter variables $p[\sigma]$ which we call *contextual variables*. Contextual variables are associated with a post-poned substitution σ. The meta-variable u stands for a contextual object $\hat{\Psi}.R$ where $\hat{\Psi}$ describes the ordinary bound variables which may occur in R. This allows us to rename the free variables occurring in R when necessary. The parameter variable p stands for a contextual object $\hat{\Psi}.R$ where R must be either an ordinary bound variable from $\hat{\Psi}$ or another parameter variable.

In the simultaneous substitutions σ, we do not make its domain explicit. Rather we think of a substitution together with its domain Ψ and the i-th element in σ corresponds to the i-th declaration in Ψ. We have two different ways of building a substitution: either by using a normal term M or a variable x. Note that a variable x is only a normal term M if it is of base type. However, as we push a substitution σ through a λ-abstraction $\lambda x.M$, we need to extend σ with x. The resulting substitution σ, x may not be well-typed, since x may not be of base type and in fact we do not know its type. Hence, we allow substitutions not only to be extended with normal terms M but also with variables x. Without loss of generality we require that meta-variables have base type.

A bound variable context Ψ contains bound variable declarations in addition to context variables. A context may only contain at most one context variable and it must occur at the left. This will make it easier to ensure bound variable dependencies are satisfied in the dependently typed setting.

Following Pientka [2008] we use a bi-directional type system where we check normal terms against a type and synthesize a type for neutral terms. LF objects may depend on variables declared in the context Ψ and the meta-context Δ which contains contextual variables such as meta-variables u, parameter variables p and context variables ψ. We introduce Δ more formally in the next section. All typing judgments have access to both contexts and a well-typed signature Σ where we store constants together with their types and kinds.

$\Delta; \Psi \vdash M \Leftarrow A$	Normal term M checks against type A
$\Delta; \Psi \vdash R \Rightarrow A$	Neutral term R synthesizes type A
$\Delta; \Psi \vdash \sigma \Leftarrow \Psi'$	Substitution σ has domain Ψ' and range Ψ.

The bi-directional typing rules are mostly straightforward and are presented in Figure 1. We will tacitly rename bound variables, and maintain that contexts and substitutions declare no variable more than once. Note that substitutions σ are defined only on ordinary variables x and not contextual variables. Moreover, we require the usual conditions on bound variables. For example in the rule for λ-abstraction the bound variable x must be new and cannot already occur in the context Ψ. This can be always achieved via α-renaming. Similarly, in meta-terms we tacitly apply α-renaming.

As is common, we rely on hereditary substitutions, written as $[N/x]_A(B)$ (or $[\sigma]_\Psi(B)$) to guarantee that when we substitute a term N which has type A for the variable x in the type B, we obtain a type B' which is in normal form. Hereditary substitutions continue to substitute, if a redex is created; for example, when replacing naively x by $\lambda y.c\ y$ in the object $x\ z$, we would obtain $(\lambda y.c\ y)\ z$ which is not in normal form and hence not a valid term in our grammar. Hereditary substitutions continue to substitute z for y in $c\ y$ to obtain $c\ z$ as a final result. For a more detailed description of hereditary substitution, we refer the reader to for example Nanevski et al. [2008b].

Finally, we remark on equality checking. When checking $A = B$ we must take into account η-contraction, because we have two ways to build substitutions. If x has type $\Pi y{:}A.B$ then we may have written $\sigma; x$ or $\sigma, \lambda y.x\ y$.

Neutral Terms $\boxed{\Delta; \Psi \vdash R \Rightarrow A}$

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

$$\frac{\Sigma(\mathbf{c}) = A}{\Delta; \Psi \vdash \mathbf{c} \Rightarrow A} \qquad \frac{\Delta(u) = P[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]_\Phi P}$$

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

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

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

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

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash H \Rightarrow B \quad B = [\sigma]_\Phi A}{\Delta; \Psi \vdash \sigma; H \Leftarrow \Phi, x{:}A}$$

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

Figure 1. Typing for contextual LF

3.2 Meta-Objects and Meta-types

We lift contextual LF objects to meta-types and meta-objects to treat abstraction over meta-objects uniformly. Meta-objects are either contextual objects written as $\hat{\Psi}.R$ or contexts Ψ. These are the index objects which can be used to index computation-level types. There are three different meta-types: $P[\Psi]$ denotes the type of a meta-variable u and stands for a general contextual object $\hat{\Psi}.R$. $\#A[\Psi]$ denotes the type of a parameter variable p and it stands for a variable object, i.e. either $\hat{\Psi}.x$ or $\hat{\Psi}.p[\pi]$ where π is a variable substitution. A variable substitution π is a special case for general substitutions σ; however unlike $p[\sigma]$ which can produce a general LF object, $p[\pi]$ guarantees we are producing a variable. G describes the schema (i.e. type) of a context. The tag $\#$ on the type of parameter variables is a simple syntactic device to distinguish between the type of meta-variables and parameter variables. It does not introduce a subtyping relationship between the type $\#A[\Psi]$ and the type $A[\Psi]$. The meta-context in which an LF object appears uniquely determines if X denotes a meta-variable, parameter variable or context variable. We use the following convention: if X denotes a meta-variable we usually write u or v; if it stands for a parameter-variable, we write p and for context variables we use ψ.

Context schemas	G	$::=$	$\exists\overrightarrow{(x{:}A)}.B \mid G + \exists\overrightarrow{(x{:}A)}.B$
Meta Objects	C	$::=$	$\hat{\Psi}.R \mid \Psi$
Meta Types	U	$::=$	$P[\Psi] \mid \#A[\Psi] \mid G$
Meta substitutions	θ	$::=$	$\cdot \mid \theta, C/X$
Meta-context	Δ	$::=$	$\cdot \mid \Delta, X{:}U$

Context schemas consist of different schema elements $\exists\overrightarrow{(x{:}A)}.B$ which are built using $+$. Intuitively, this means a concrete declaration in a context must be an instance of one of the elements specified in the schema. For example, a context $x{:}\mathsf{exp\ nat}, y{:}\mathsf{exp\ bool}$ will check against the schema $\exists T{:}\mathsf{tp}.\mathsf{exp}\ T$.

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

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

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

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

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

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

$$\frac{}{\Delta \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta \vdash \theta \Leftarrow \Delta' \quad \Delta \vdash C \Leftarrow [\![\theta]\!]_{\Delta'}(U)}{\Delta \vdash \theta, C/X \Leftarrow \Delta', X{:}U}$$

Figure 2. Typing for meta-terms

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

We omit here the rules stating when meta-types and meta-contexts are well-formed and show only the typing rules for meta-term and meta-substitutions in Figure 2.

A consequence of the uniform treatment of meta-terms is that the design of the computation language is modular and parameterized over meta-terms and meta-types. This has two main advantages: First, we can in principle easily extend meta-terms and meta-types without affecting the computation language; in particular, it is straightforward to add substitution variables which were present in Pientka [2008] or allow for richer context schemas. Second, it will be key to a modular, clean design of computations.

The single meta-substitution, written as $[\![C/X]\!]_U(*)$ where $*$ stands for A, M, R, σ, Ψ, is defined inductively on the structure of the given object. (see for example Pientka [2011] or the appendix). We only discuss briefly here some of the fundamental ideas. Let us first consider the case where X stands for a meta-variable u and C is a meta-object $\hat{\Psi}.R$. We note that there is no capture issues when we push $[\![\hat{\Psi}.R/u]\!]$ through a lambda-expression and the only interesting issue arises when we encounter an object $u[\sigma]$. In this case, we apply $[\![\hat{\Psi}.R/u]\!]$ to σ to obtain σ'. Subsequently, we apply σ' to R to obtain the final result.

Next, we consider the case where X stands for a parameter variable p and C is a meta-object $\hat{\Psi}.x$ or $\hat{\Psi}.q[\pi]$. The only interesting case is when we encounter $p[\sigma]$. Similar to the case for meta-variables, we apply the meta-substitution to σ to obtain σ' and subsequently apply σ' to x or $q[\pi]$. There is however a small caveat: since σ' is an arbitrary substitution, applying it to x, may yield a normal object M. Hence, simply M may produce a non-normal term which is not meaningful in our grammar. The solution to this problem is to define meta-substitutions hereditarily; hence we index the meta-substitution with its domain.

Finally, the case where X stands for a context variable ψ and C is a meta-object Ψ. There are two interesting cases: 1) when we encounter the identity substitution id_ψ, we unroll Ψ and create at the same time a concrete identity substitution which maps all variables from Ψ to themselves. 2) when we encounter a context variable ψ in a context, then we simply replace it with the concrete context Ψ. The full definition of meta-substitutions is given in the appendix and has been previously been described in [Nanevski et al. 2008b; Pientka 2011, 2008].

The simultaneous meta-substitution, written as $[\![\theta]\!]$, is a straightforward extension of the single substitution.

Theorem 3.1 (Meta-substitution property)**.**
If $\Delta' \vdash \theta \Leftarrow \Delta$ *and* $\Delta;\Psi \vdash J$ *then* $\Delta'; [\![\theta]\!]_\Delta \Psi \vdash [\![\theta]\!]_\Delta J$.

4. Beluga$^\mu$: a language with binding support and recursive types

We present in this section a dependently typed programming language Beluga$^\mu$ along the lines of Mini-ML, including recursive types, variants and general pattern matching which is critical in practice and whose theory in this setting is non-trivial. Polymorphism, on the other hand, is largely orthogonal and therefore postponed. The type index objects are drawn from the domain of meta-objects presented in the previous section, but we emphasize that this language is parametric over the index domain, requiring only decidable equality.

4.1 Types and Kinds

Our type language supports function types (written as $T_1 \to T_2$), products (written as $T_1 \times T_2$), labelled sums (written as $\langle\overrightarrow{l{:}T}\rangle$), dependent types (written as $\Pi X{:}U.T$) and dependent products (written as $\Sigma X{:}U.T$). We only allow dependencies on meta-terms not on arbitrary computation-level expressions. The novel part in our type language is our definition of recursive types together with the equality constraint which may be associated with a given type. The recursive type is written as $\mu Z.\lambda\vec{X}.T$; while Z denotes a type variable, $\lambda\vec{X}.T$ describes a type-level function which expects meta-terms.

$$\begin{array}{lrl}
\text{Kinds} & K ::= & \mathsf{ctype} \mid \Pi X{:}U.K \\
\text{Types} & T ::= & \mathsf{Unit} \mid Z \mid T_1 \to T_2 \mid T_1 \times T_2 \mid \langle\overrightarrow{l : T}\rangle \\
 & & \mid \Pi X{:}U.T \mid \Sigma X{:}U.T \mid C_1 = C_2 \wedge T \\
 & & \mid \mu Z.\,\lambda\vec{X}.T \mid T\;\vec{C} \mid U \\
\text{Context} & \Gamma ::= & \cdot \mid \Gamma, Z : K \mid \Gamma, x : T
\end{array}$$

We note that we can directly refer to meta-types and embed them in our computation-level types. Hence meta-objects can be directly analyzed and manipulated by our computation language. This is convenient in our setting, however, it also prevents a naive erasure of all the meta-objects.

We also note that equalities cannot occur just by themselves in our grammar. The reason is that equalities are treated silently during type checking and equality proofs which establish $C_1 = C_2$ do not pollute our computation-level expressions. In fact, equalities typically occur inside a recursive type and they are trivially true once we have chosen the correct instantiation for the existentially quantified variable. We illustrate this idea shortly.

To illustrate, we give here three examples.

Example 1 The type of a vector of booleans which keeps track of their length can be defined as follows. We assume an LF signature which declares `nat:type` together with two constants, `0:nat` and `s: nat → nat`. Because all index objects to the recursive type V are closed, we omit writing the empty context and simply write nat for $\mathsf{nat}[]$ and Y for $\cdot.Y$.

$$\begin{array}{lll}
\mathrm{Vec} = \mu\mathrm{Vec}.\lambda X.\;\langle\; \mathsf{nil} & : & X = 0 \wedge \mathsf{Unit}\;, \\
 \qquad\qquad\qquad\qquad\; \mathsf{cons} & : & \Sigma Y{:}\mathsf{nat}.X = \mathsf{s}\;Y \wedge \mathsf{bool} \times \mathrm{Vec}\;Y\;\rangle
\end{array}$$

Example 2 The relation between contexts from Section 2.2 can be written as follows:

$$\begin{array}{l} \mu\mathsf{Ctx\_rel}.\lambda\psi\lambda\phi.\ \langle\ \mathsf{rnil}\ \ : \psi = \cdot \wedge \phi = \cdot \wedge \mathsf{Unit}\ , \\ \qquad\qquad\qquad \mathsf{rsnoc}\!: \Sigma\psi'\!:\!\mathsf{ctx}.\Sigma\phi'\!:\!\mathsf{cctx}. \\ \qquad\qquad\qquad\quad \psi = \psi', x\!:\!\mathsf{tm}\ \wedge\ \phi = \phi', y\!:\!\mathsf{ctm} \\ \qquad\qquad\qquad\quad \wedge\ \mathsf{Ctx\_rel}\ \psi'\ \phi'\ \rangle \end{array}$$

Our treatment of recursive types with equalities is similar to Xi et al. [2003] and Sulzmann et al. [2007] where recursive types are endowed with equalities between types to model GADTs. Recently fixed points with equalities between terms have appeared in Licata [2011] and in Baelde et al. [2010] for example. In our setting, we treat equality between contextual objects. Next, we give the kinding rules for computation-level types.

Well-kinded computation-level types $\boxed{\Delta;\Gamma \vdash T : K}$

$$\frac{\Delta, \overrightarrow{X{:}U}\ ;\ \Gamma, Z : \Pi\overrightarrow{X{:}U}.K \vdash T : K}{\Delta;\Gamma \vdash \mu Z.\,\lambda\vec{X}.\,T : \Pi\overrightarrow{X{:}U}.K}$$

$$\frac{\Delta;\Gamma \vdash T_1 :\ \mathsf{ctype} \quad \Delta;\Gamma \vdash T_2 :\ \mathsf{ctype} \quad * \in \{\rightarrow, \times\}}{\Delta;\Gamma \vdash T_1 * T_2 :\ \mathsf{ctype}}$$

$$\frac{\Delta\ ;\ \Gamma \vdash U \Leftarrow\ \mathsf{mtype} \quad \Delta, X{:}U\ ;\ \Gamma \vdash T :\ \mathsf{ctype}}{\Delta;\Gamma \vdash \Pi X{:}U.T :\ \mathsf{ctype}}$$

$$\frac{\Delta\ ;\ \Gamma \vdash U \Leftarrow\ \mathsf{mtype} \quad \Delta, X{:}U\ ;\ \Gamma \vdash T :\ \mathsf{ctype}}{\Delta;\Gamma \vdash \Sigma X{:}U.T :\ \mathsf{ctype}}$$

$$\frac{\Delta \vdash C_1 \Leftarrow U \quad \Delta \vdash C_2 \Leftarrow U \quad \Delta;\Gamma \vdash T :\ \mathsf{ctype}}{\Delta;\Gamma \vdash C_1 = C_2 \wedge T :\ \mathsf{ctype}}$$

$$\frac{\Delta \vdash U \Leftarrow\ \mathsf{mtype}}{\Delta;\Gamma \vdash U :\ \mathsf{ctype}} \qquad \frac{}{\Delta;\Gamma \vdash \mathsf{Unit} :\ \mathsf{ctype}}$$

$$\frac{\Gamma(Z) = K}{\Delta;\Gamma \vdash Z : K} \qquad \frac{\Delta;\Gamma \vdash T : \Pi X{:}U.K \quad \Delta \vdash C : U}{\Delta;\Gamma \vdash T\ C : [\![C/X]\!]K}$$

4.2 Computations

Our language of computations includes recursive functions (written as $\mathsf{rec}\ f.E$), nameless functions (written as $\mathsf{fn}\ x.E$) and dependent functions (written as $\lambda X.\,E$). We also include pairs (written as (E_1, E_2)) and dependent pairs (written as $\mathsf{pack}\ (C,\ E)$). Finally, we include labeled variants (written as $\langle l = E\rangle$) and a fold constructor for recursive types.

$$\begin{array}{lll} \text{Expressions (synth.)} & I & ::= y \mid I\ E \mid I\ \ C \mid (E : T) \\ \text{Expressions (checked)} & E & ::= I \mid C \mid \mathsf{fn}\ y.E \mid \lambda X.\,E \mid \mathsf{rec}\ f.E \mid \mathsf{unit} \\ & & \quad \mid \mathsf{fold}\ E \mid \langle l = E\rangle \mid \mathsf{pack}\ (C, E) \\ & & \quad \mid (E_1, E_2) \mid \mathsf{case}\ I\ \mathsf{of}\ \vec{B} \\ \text{Pattern} & pat & ::= x \mid C \mid \mathsf{unit} \mid \mathsf{fold}\ pat \mid \langle l = pat\rangle \\ & & \quad \mid \mathsf{pack}\ (C, pat) \mid (pat_1, pat_2) \\ \text{Branch} & B & ::= \Delta;\Gamma\ .\ pat : \theta \mapsto E \\ \text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, y{:}T \end{array}$$

Our language is split into expressions for which we synthesize types and expressions which are checked against a type. This minimizes the necessary type annotations and provides a syntax directed recipe for a type checker. Intuitively, the expressions which introduce a type are expressions which are checked and expressions which eliminate a type are in the synthesis category. We have two different kinds of function applications, one for applying computation-level functions to an expression and the other to apply a dependent function to a meta-object C. Pairs and dependent pairs are analyzed by pattern matching.

$\boxed{\Delta;\Gamma \vdash I \Rightarrow T}$ Expression I synthesizes type T

$$\frac{y{:}T \in \Gamma}{\Delta;\Gamma \vdash y \Rightarrow T} \qquad \frac{\Delta;\Gamma \vdash I \Rightarrow C = C \wedge T}{\Delta;\Gamma \vdash I \Rightarrow T}$$

$$\frac{\Delta;\Gamma \vdash I \Rightarrow T_2 \rightarrow T \qquad \Delta;\Gamma \vdash E \Leftarrow T_2}{\Delta;\Gamma \vdash I\ E \Rightarrow T}$$

$$\frac{\Delta;\Gamma \vdash I \Rightarrow \Pi X{:}U.T \qquad \Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash I\ \ C \Rightarrow [\![C/X]\!]T} \qquad \frac{\Delta;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash (E : T) \Rightarrow T}$$

$\boxed{\Delta;\Gamma \vdash E \Leftarrow T}$ Expression E checks against type T

$$\frac{\Delta;\Gamma \vdash E_i \Leftarrow T_i \quad \text{where } l_i : T_i \in \overrightarrow{l : T}}{\Delta;\Gamma \vdash \langle l_i = E_i\rangle \Leftarrow \langle\overrightarrow{l : T}\rangle} \qquad \frac{\Delta;\Gamma,\ f : T\ \vdash\ E\ \Leftarrow\ T}{\Delta;\Gamma\ \vdash\ \mathsf{rec}\ f.E\ \Leftarrow\ T}$$

$$\frac{\Delta;\Gamma, y{:}T_1 \vdash E \Leftarrow T_2}{\Delta;\Gamma \vdash \mathsf{fn}\ y.E \Leftarrow T_1 \rightarrow T_2} \qquad \frac{\Delta, X{:}U;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash \lambda X.\,E \Leftarrow \Pi X{:}U.T}$$

$$\frac{\Delta;\Gamma \vdash E_1 \Leftarrow T_1 \quad \Delta;\Gamma \vdash E_2 \Leftarrow T_2}{\Delta;\Gamma \vdash (E_1, E_2) \Leftarrow T_1 \times T_2} \qquad \frac{\Delta;\Gamma \vdash I \Rightarrow T \quad T = T'}{\Delta;\Gamma \vdash I \Leftarrow T'}$$

$$\frac{\Delta \vdash C \Leftarrow U \quad \Delta;\Gamma \vdash E \Leftarrow [\![C/X]\!]T}{\Delta;\Gamma \vdash \mathsf{pack}\ (C, E) \Leftarrow \Sigma X{:}U.T} \qquad \frac{\Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash C \Leftarrow U}$$

$$\frac{\Delta;\Gamma \vdash E \Leftarrow [\mu Z.\lambda\vec{X}.\,S/Z][\![\vec{C}/\vec{X}]\!]S}{\Delta;\Gamma \vdash \mathsf{fold}\ E \Leftarrow (\mu Z.\lambda\vec{X}.\,S)\ \vec{C}} \qquad \frac{\Delta;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash E \Leftarrow C = C \wedge T}$$

$$\frac{\Delta;\Gamma \vdash I \Rightarrow S \quad \text{for all } i\ \Delta;\Gamma \vdash B_i \Leftarrow S \rightarrow T}{\Delta;\Gamma \vdash \mathsf{case}\ I\ \mathsf{of}\ \vec{B} \Leftarrow T}$$

$\boxed{\Delta;\Gamma \vdash B \Leftarrow S \rightarrow T}$ Branch B with pattern of S checks against T

$$\frac{\Delta_i \vdash \theta_i \Leftarrow \Delta \quad \Delta_i;\Gamma_i \vdash pat \Leftarrow [\![\theta_i]\!]S \quad \Delta_i;[\![\theta_i]\!]\Gamma, \Gamma_i \vdash E \Leftarrow [\![\theta_i]\!]T}{\Delta;\Gamma \vdash \Delta_i;\Gamma_i\,.\,pat : \theta_i \mapsto E \Leftarrow S \rightarrow T}$$

Figure 3. Typing for computations

Branches are modelled by $\Delta;\Gamma\ .\ pat\ :\ \theta\ \mapsto\ E$ where Δ describes the meta-variables occurring in the pattern which are often left implicit in the surface language, while Γ corresponds to the explicit arguments. The refinement substitution θ describes how the type of the scrutinee is instantiated so the given branch is applicable.

Example 3 We show next the elaboration of a simple program to compute the tail of a vector and its elaboration:

```
rec  tail : {N:nat[]} → vec (s N) → vec N =
fn  l ⇒ case  l of  cons h t ⇒ t
```

Which can be elaborated into:

$$\begin{array}{l} \mathsf{rec\ tail}.\lambda N.\,\mathsf{fn}\ l.\ \mathsf{case}\ \ l\ \ \mathsf{of} \\ \mid M : \mathsf{tp}\ ;\ h : \mathsf{int},\ t{:}\mathsf{Vec}\ M\ . \\ \quad \mathsf{fold}\ \langle\ \mathsf{cons} = \mathsf{pack}\ (M,\ (h, t)\ )\ \rangle \quad : \quad M/N \quad \mapsto \quad t \end{array}$$

We insert the length argument to `cons` which was left implicit in the source-level program and the refinement `M/N` which guarantees that the type of the pattern is compatible with the type of the scrutinee. We also list explicitly the type of the index variable `M` as well as the type of the arguments `h` and `t`.

4.3 Typing rules

Next, we give the typing rules for computations in Figure 3. We present bi-directional typing rules for computations which will minimize the amount of typing annotations. We distinguish between typing of expressions and branches. In the typing judgment, we will distinguish between the context Δ for contextual variables from our index domain and the context Γ which includes declarations of computation-level variables. Contextual variables will be introduced via λ-abstraction. The contextual variables in Δ are also introduced in the branch of a case-expression. Computation-level variables in Γ are introduced by recursion or functions and in addition in branches. We use the following judgments:

$\Delta;\Gamma \vdash E \Leftarrow T$	Expression E checks against type T
$\Delta;\Gamma \vdash I \Rightarrow T$	Expression I synthesizes type T
$\Delta;\Gamma \vdash B \Leftarrow S \to T$	Branch B with pattern of type S checks against T

The typing rules are given in Figure 3. We will tacitly rename bound variables, and maintain that contexts declare no variable more than once. Moreover, we require the usual conditions on bound variables. For example in the rule for λ-abstraction the contextual variable X must be new and cannot already occur in the context Δ. This can be always achieved via α-renaming. Similarly, in the rule for recursion and function abstraction, the variable x must be new and cannot already occur in Γ.

The rules which synthesize and checking are mostly standard and we only point out a few rules. We have two rules for applications: to synthesize the type S of a non-dependent application $(I\ \ E)$, we synthesize the type for I to be $T \to S$ and check E against T. For the dependent application $I\ \ C$, we synthesize the type $\Pi X{:}U.T$ for I and check that C is a well-typed meta-object of type U. Note that we drop the computation context Γ when we transition to type check a meta-object, since meta-objects cannot refer to computations. The final type for $I\ \ C$ is $[\![C/X]\!]T$. If we have synthesized a type T together with a trivial equality constraint, we simply drop the constraint and return T.

To check a λ-abstraction against $\Pi X{:}U.T$, we add $X{:}U$ to the meta-context Δ and continue to check that the body of the abstraction has type T. To check that a function $\mathsf{fn}\ y.E$ has type $T_1 \to T_2$, we add the assumption $y{:}T_1$ to the computation context Γ and continue to check the body E against T_2. For checking a non-dependent pair $(E_1,\ E_2)$ against the type $T_1 \times T_2$, we check each part of the pair against their respective type. For checking a dependent pair $\mathsf{pack}\ (C,\ E)$ against $\Sigma X{:}U.T$, we check that C is a well-typed meta-object of type U switching to the typing rules for the meta-level and dropping the context Γ. In addition, we check that E has type $[\![C/X]\!]T$.

When we check a contextual meta-object C against a type U, we simply convert to the type checking rules for meta-objects and forget about the computation-level context Γ. Our typing rules will ensure that meta-objects are pure objects and do not contain any computation-level expressions. When checking an expression against $C = C \wedge T$, we can simply drop the constraint $C = C$, since it is trivially true. We check $(\mathsf{fold}\ \ E)$ against the recursive type $(\mu Z.\lambda\vec{X}.\,S)\ \ \vec{C}$ by unrolling the fixed point definition and checking E against $[\mu Z.\lambda\vec{X}.\,S/Z][\![\vec{C}/\vec{X}]\!]S$. The difference to simply-typed recursive types is that dependently typed recursive types are applied to index objects $\vec{C}$.

To illustrate that our data carries enough information to ensure that the equality constraints are trivially true, if the term is well-typed, we show the typing derivation for `cons (true,nil)` in Figure 4.

In the rule for case expressions, we first infer the type S of the scrutinee and then proceed to check that each branch B_i has a pattern compatible with S and its body has a type compatible with T. To check a branch $\Delta_i;\Gamma_i\ \vdash pat_i\ :\ \theta_i\ \mapsto\ E_i$, we check that the refinement substitution θ provides instantiations from the outer meta-context Δ to the current meta-context Δ_i. Moreover, the pattern has type $[\![\theta_i]\!]S$, i.e. it is compatible with the type of the scrutinee, and only refers to the local variables Δ_i and Γ_i. We then proceed to check the body E_i against $[\![\theta_i]\!]T$. Because the pattern may refine the types, we must make sure to apply θ_i to the appropriate parts and extend the computation context $[\![\theta_i]\!]\Gamma$ with the bindings Γ_i introduced in the branch.

The typing rules for patterns are given in Fig. 5. They duplicate some of the type checking rules for tuples, dependent pairs, meta-objects, recursive types, and variants. We ensure that the computation variables occurring in patterns occur uniquely and we split the computation context in the rule for tuples. The meta-context on the other hand remains. As a consequence, contextual variables may occur more than once, which is also often necessary to obtain well-typed expressions, but we enforce linearity for computation variables occurring in patterns.

Theorem 4.1 (Decidability of Type Checking)**.**
Type-checking computation-level expressions is decidable.

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

4.4 Big-step operational semantics

Next, we define the operational semantics for computations in Fig. 7. We adopt an environment-based approach where we do not eagerly propagate values. Recall that we distinguish between meta-variables in Δ and program variables in Γ. To work elegantly with refinement substitutions in branches, we hence define two environments: θ denotes the instantiation for meta-variables in Δ; ρ provides instantiations for program variables in Γ.

Values	V	$::=$	$F\ [\theta\ ;\ \rho] \mid \mathsf{unit} \mid \mathsf{fold}\ V \mid \langle l = V\rangle$
			$\mid \mathsf{pack}\ (C, V) \mid (V_1, V_2)$
Function Values	F	$::=$	$\mathsf{fn}\ y.E \mid \lambda X.\,E$
Extended Values	W	$::=$	$V \mid (\mathsf{rec}\ f.E)[\theta;\rho]$
Closures	L	$::=$	$E\ [\theta\ ;\ \rho]$
Environments	ρ	$::=$	$\cdot \mid \rho, W/y$

Values are either meta-terms C, unit, pairs (V_1, V_2), dependent pairs $\mathsf{pack}\ (C,\ V)$, variants $\langle l = V\rangle$, $\mathsf{fold}\ V$, or functions as closures. Since we have non-dependent and dependent functions, we have two corresponding closures. Closures are snapshots of computation inside an environment. The environment is represented by the two suspended substitutions θ and ρ for each of the two contexts Δ and Γ respectively. We write $E[\theta;\rho]$ for a closure consisting of the expression E and the suspended meta-substitution θ and the program environment ρ. The intended meaning is that first meta-substitution θ is applied to E and then ordinary substitution ρ to the result. For clarification, we show the typing for environments and values in Figure 6.

We give a big-step semantics for computations in Figure 7. To evaluate a variable y in the environment θ and ρ, we simply look up its binding in the computation environment ρ. Since ρ contains extended values, in particular y may be bound to a recursive function which in itself is not a valid result, we continue to evaluate the extended value we retrieve from ρ to a proper value. When we encounter a meta-object C in the environment θ and ρ, we apply θ to C to compute a closed meta-object. unit simply evaluates to itself regardless of the environment. Evaluating a function $\mathsf{fn}\ y.E$ in the environment θ and ρ simply returns the closure $\mathsf{fn}\ y.E\ [\theta\ ;\ \rho]$. When evaluating a recursive function $\mathsf{rec}\ f.E$ in an environment θ and ρ, we evaluate the body E and extend the computation environment ρ binding f to itself.

Evaluating a tuple $(E_1,\ E_2)$ in the environment θ and ρ is straightforward: we evaluate E_1 in the environment θ and ρ and we proceed similarly to evaluate E_2. The evaluation rules for fold and variants are straightforward.

$$\dfrac{\vdash 0 \Leftarrow \mathsf{nat} \qquad \dfrac{\dfrac{\vdash \mathsf{true} \Leftarrow \mathsf{bool} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{() \Leftarrow \mathsf{Unit}}}{() \Leftarrow 0 = 0 \wedge \mathsf{Unit}}}{\vdash \langle\ \mathsf{nil} = ()\rangle \Leftarrow \langle \mathsf{nil} : 0 = 0 \wedge \mathsf{Unit}\ ,\ \mathsf{cons} : \Sigma Y{:}\mathsf{nat}.0 = \mathsf{s}\ Y \wedge \mathsf{bool} \times \mathsf{Vec}\ Y\ \rangle}}{\vdash \mathsf{fold}\ \langle\ \mathsf{nil} = ()\rangle \Leftarrow \mathsf{Vec}\ 0}}{\vdash (\mathsf{true}, \mathsf{fold}\ \langle\ \mathsf{nil} = ()\ \rangle\ )\mathsf{bool} \times \mathsf{Vec}\ 0}}{\vdash (\mathsf{true}, \mathsf{fold}\ \langle\ \mathsf{nil} = ()\ \rangle\ ) \Leftarrow \mathsf{s}\ 0 = \mathsf{s}\ 0 \wedge \mathsf{bool} \times \mathsf{Vec}\ 0}}{\dfrac{\dfrac{\vdash \mathsf{pack}\ (0,\ (\mathsf{true},\ \mathsf{fold}\ \langle\ \mathsf{nil} = ()\ \rangle\ ))\ \Leftarrow \Sigma Y{:}\mathsf{nat}.\mathsf{s}\ 0 = \mathsf{s}\ Y \wedge \mathsf{bool} \times \mathsf{Vec}\ Y}{\vdash \langle\ \mathsf{cons} = \mathsf{pack}\ (0,\ (\mathsf{true}, \mathsf{fold}\ \langle\ \mathsf{nil} = ()\ \rangle\ ))\ \rangle \Leftarrow \langle \mathsf{nil} : \mathsf{s}\ 0 = 0 \wedge \mathsf{Unit}\ ,\ \mathsf{cons} : \Sigma Y{:}\mathsf{nat}.\mathsf{s}\ 0 = \mathsf{s}\ Y \wedge \mathsf{bool} \times \mathsf{Vec}\ Y\ \rangle}}{\vdash \mathsf{fold}\ \langle\ \mathsf{cons} = \mathsf{pack}\ (0,\ (\mathsf{true},\ \mathsf{fold}\ \langle\ \mathsf{nil} = ()\ \rangle\ ))\ \rangle \Leftarrow \mathsf{Vec}\ (\mathsf{s}\ 0)}}$$

Figure 4. Typing derivation for vector `cons (true, nil)` := $\mathsf{fold}\ \langle\ \mathsf{cons} = \mathsf{pack}\ (0,\ (\mathsf{true}, \mathsf{fold}\ \langle\ \mathsf{nil} = ()\ \rangle\ ))\ \rangle$

$\boxed{\Delta; \Gamma \vdash \mathit{pat} \Leftarrow T}$ Pattern checks against type T

$$\dfrac{\Gamma(x) = T}{\Delta; \Gamma \vdash x \Leftarrow T} \quad \dfrac{\Delta \vdash C \Leftarrow U}{\Delta; \Gamma \vdash C \Leftarrow U} \quad \dfrac{}{\Delta; \Gamma \vdash \mathsf{unit} \Leftarrow \mathsf{Unit}} \quad \dfrac{T = \mu Z.\lambda \vec{X}.\, S \quad \Delta; \Gamma \vdash \mathit{pat} \Leftarrow [T/Z][\![\vec{C}/\vec{X}]\!]S}{\Delta; \Gamma \vdash \mathsf{fold}\ \mathit{pat} \Leftarrow T\ \vec{C}} \quad \dfrac{\Delta; \Gamma \vdash \mathit{pat} \Leftarrow T}{\Delta; \Gamma \vdash \mathit{pat} \Leftarrow C = C \wedge T}$$

$$\dfrac{\Delta; \Gamma \vdash \mathit{pat}_i \Leftarrow T_i \quad \text{where } l_i : T_i \in \overrightarrow{l : T}}{\Delta; \Gamma \vdash \langle l_i = \mathit{pat}_i \rangle \Leftarrow \langle \overrightarrow{l : T} \rangle} \quad \dfrac{\Delta \vdash C \Leftarrow U \quad \Delta; \Gamma \vdash \mathit{pat} \Leftarrow [\![C/X]\!]T}{\Delta; \Gamma \vdash \mathsf{pack}\ (C, \mathit{pat}) \Leftarrow \Sigma X{:}U.T} \quad \dfrac{\Delta; \Gamma_1 \vdash \mathit{pat}_1 \Leftarrow T_1 \quad \Delta; \Gamma_2 \vdash \mathit{pat}_2 \Leftarrow T_2}{\Delta; \Gamma_1, \Gamma_2 \vdash (\mathit{pat}_1, \mathit{pat}_2) \Leftarrow T_1 \times T_2}$$

Figure 5. Typing rules for patterns

$\boxed{E\ [\theta\ ;\ \rho] \Downarrow V}$ Expression E in environment $[\theta; \rho]$ evaluates to value V

$$\dfrac{I_1\ [\theta\ ;\ \rho] \Downarrow (\mathsf{fn}\ y.E)\ [\theta_1\ ;\ \rho_1] \quad E_2\ [\theta\ ;\ \rho] \Downarrow V_2 \quad E\ [\theta_1\ ;\ \rho_1, V_2/y] \Downarrow V}{(I_1\ E_2)\ [\theta\ ;\ \rho] \Downarrow V} \quad \dfrac{I\ [\theta\ ;\ \rho] \Downarrow (\lambda X.\, E)\ [\theta_1\ ;\ \rho_1] \quad E\ [\theta_1, [\![\theta]\!]C/X\ ;\ \rho_1] \Downarrow V}{(I\ \ C)\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\dfrac{}{C\ [\theta\ ;\ \rho] \Downarrow [\![\theta]\!]C} \quad \dfrac{}{\mathsf{unit}\ [\theta\ ;\ \rho] \Downarrow \mathsf{unit}} \quad \dfrac{\rho(y) = V}{y\ [\theta\ ;\ \rho] \Downarrow V} \quad \dfrac{\rho(y) = (\mathsf{rec}\ f.E)\ [\theta_1\ ;\ \rho_1] \quad \rho(y) \Downarrow V}{y\ [\theta\ ;\ \rho] \Downarrow V} \quad \dfrac{E\ [\theta\ ;\ \rho] \Downarrow V}{(E : T)\ [\theta\ ;\ \rho] \Downarrow V} \quad \dfrac{E\ [\theta\ ;\ \rho] \Downarrow V}{(\mathsf{fold}\ E)\ [\theta\ ;\ \rho] \Downarrow \mathsf{fold}\ V}$$

$$\dfrac{E\ [\theta\ ;\ \rho] \Downarrow V}{\langle l = E \rangle\ [\theta\ ;\ \rho] \Downarrow \langle l = V \rangle} \quad \dfrac{}{(\mathsf{fn}\ y.E)\ [\theta\ ;\ \rho] \Downarrow (\mathsf{fn}\ y.E)\ [\theta\ ;\ \rho]} \quad \dfrac{}{(\lambda X.\, E)\ [\theta\ ;\ \rho] \Downarrow (\lambda X.\, E)\ [\theta\ ;\ \rho]} \quad \dfrac{E\ [\theta\ ;\ \rho, (\mathsf{rec}\ f.E)\ [\theta\ ;\ \rho]/f] \Downarrow V}{(\mathsf{rec}\ f.E)\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\dfrac{E\ [\theta\ ;\ \rho] \Downarrow V}{\mathsf{pack}\ (C, E)\ [\theta\ ;\ \rho] \Downarrow \mathsf{pack}\ ([\![\theta]\!]C, V)} \quad \dfrac{E_1\ [\theta\ ;\ \rho] \Downarrow V_1 \quad E_2\ [\theta\ ;\ \rho] \Downarrow V_2}{(E_1, E_2)\ [\theta\ ;\ \rho] \Downarrow (V_1, V_2)} \quad \dfrac{\Delta_i \vdash \theta \not\doteq \theta_i \quad \mathsf{case}\ I\ \mathsf{of}\ \vec{B}\ [\theta\ ;\ \rho] \Downarrow V}{\mathsf{case}\ I\ \mathsf{of}\ (\Delta_i; \Gamma_i\, .\, \mathit{pat} : \theta_i \mapsto E_i \mid \vec{B})\ [\theta\ ;\ \rho] \Downarrow V}$$

$$\dfrac{\begin{array}{cc} \Delta_i \vdash \theta \doteq \theta_i/(\theta'; \Delta'_i) & \Delta'_i; [\![\theta']\!]\Gamma_i \vdash V_1 \not\doteq [\![\theta']\!]\mathit{pat} \\ I\ [\theta\ ;\ \rho] \Downarrow V_1 & \mathsf{case}\ I\ \mathsf{of}\ \vec{B}\ [\theta\ ;\ \rho] \Downarrow V \end{array}}{\mathsf{case}\ I\ \mathsf{of}\ (\Delta_i; \Gamma_i\, .\, \mathit{pat} : \theta_i \mapsto E_i \mid \vec{B})\ [\theta\ ;\ \rho] \Downarrow V} \quad \dfrac{\begin{array}{cc} \Delta_i \vdash \theta \doteq \theta_i/(\theta'; \Delta'_i) & \Delta'_i; [\![\theta']\!]\Gamma_i \vdash V_1 \doteq [\![\theta']\!]\mathit{pat}/(\theta''; \rho') \\ I\ [\theta\ ;\ \rho] \Downarrow V_1 & E_i\ [[\![\theta'']\!]\theta'\ ;\ \rho, \rho'] \Downarrow V \end{array}}{\mathsf{case}\ I\ \mathsf{of}\ (\Delta_i; \Gamma_i\, .\, \mathit{pat} : \theta_i \mapsto E_i \mid \vec{B})\ [\theta\ ;\ \rho] \Downarrow V}$$

Figure 7. Big-step semantics

We have two rules for evaluating applications: the first is for a non-dependent application $I_1\ \ E_2$ in an environment $[\theta; \rho]$. We first evaluate I_1 in the given environment obtaining a closure $(\mathsf{fn}\ y.E)\ [\theta_1\ ;\ \rho_1]$. Then we evaluate E_2 to a value V_2 and finally proceed to evaluate E in the extended environment $[\theta_1\ ; \rho_1, V_2/y]$ where the meta-substitution θ_1 remains unchanged. On the other hand, when evaluating a dependent application $I\ C$ in an environment $[\theta\ ;\ \rho]$, we evaluate I to a closure $(\lambda X.\, E)\ [\theta_1\ ;\ \rho_1]$. We now extend the meta-substitution θ with the binding $[\![\theta]\!]C/X$ and evaluate E in the extended environment $[\theta_1, [\![\theta]\!]C/X\ ;\ \rho_1]$ where the computation environment ρ_1 remains unchanged.

The most interesting cases are those for case-expressions. A branch may be skipped if either the type of the scrutinee and the type of the pattern are not compatible, i.e. the current meta-substitution and the refinement substitution in the given branch do not unify, or if the types are compatible then the scrutinee itself may still be incompatible with the pattern of the current branch. Evaluating a case expression, we first evaluate the scrutinee I in the current environment $[\theta\ ;\ \rho]$ to some value V_1. Next, we check that the current meta-substitution θ is unifiable with the refinement substitution θ_i of the given branch. This is written as $\Delta_i \vdash \theta \doteq \theta_i/(\theta'\ ;\ \Delta'_i)$ and θ' is the result of unifying θ with θ_i s.t. $\theta = [\![\theta']\!]\theta_i$ and θ' is a substitution which maps contextual variables from Δ_i to Δ'_i. Unifying the contextual substitutions ensures that the type of the scrutinee and the type of the pattern are compatible. Next, we check that the pattern is compatible with the

$\boxed{V : T}$ Value V has type T

$$\frac{\cdot \vdash \theta \Leftarrow \Delta \quad \rho : [\![\theta]\!]\Gamma \quad \Delta;\Gamma \vdash F \Leftarrow T}{F\ [\theta\ ;\ \rho] : [\![\theta]\!]T} \qquad \frac{\cdot \vdash C \Leftarrow U}{C : U} \qquad \frac{}{\mathsf{unit} : \mathsf{Unit}}$$

$$\frac{V : T_i \quad \text{where } l_i : T_i \in \overrightarrow{l : T}}{\langle l_i = V\rangle : \langle \overrightarrow{l : T}\rangle} \qquad \frac{T = \mu Z.\lambda \vec{X}.\, S \quad V : [T/Z][\![\vec{C}/\vec{X}]\!]S}{\mathsf{fold}\ V : T\ \vec{C}}$$

$$\frac{V : T}{V : C = C \wedge T} \qquad \frac{\cdot \vdash C \Leftarrow U \quad V : [\![C/X]\!]T}{\mathsf{pack}\ (C, V) : \Sigma X{:}U.T} \qquad \frac{V_1 : T_1 \quad V_2 : T_2}{(V_1, V_2) : T_1 \times T_2}$$

$\boxed{L : T}$ Closure L has type T

$$\frac{\cdot \vdash \theta \Leftarrow \Delta \qquad \rho : [\![\theta]\!]\Gamma \qquad \Delta;\Gamma \vdash E \Leftarrow T \text{ or } \Delta;\Gamma \vdash E \Rightarrow T}{E\ [\theta\ ;\ \rho] : [\![\theta]\!]T}$$

$\boxed{\rho : \Gamma}$ Environment ρ has domain Γ

$$\frac{}{\cdot : \cdot} \qquad \frac{\rho : \Gamma \qquad W : T}{(\rho, W/y) : \Gamma, y{:}T}$$

Figure 6. Value and closure typing

value of the scrutinee. Before matching the value of the scrutinee against the the pattern, we apply the contextual substitution θ' to the pattern pat and also to the variables listed Γ_i which occur in the pattern. The result will be a contextual substitution θ'' for the meta-context Δ_i' and a substitution ρ' for actual pattern variables from $[\![\theta']\!]\Gamma_i$. Finally, the body of the branch E_i is evaluated. Recall that if the overall case-expression has type T in a meta-context Δ and computation context Γ, then E_i has type $[\![\theta_i]\!]T$ in a meta-context Δ_i and computation context $[\![\theta_i]\!]\Gamma, \Gamma_i$. Therefore, we will now evaluate E_i in the contextual environment $[\![\theta'']\!]\theta'$ and extend the computation environment ρ with the new bindings in ρ'.

We now proceed to prove subject reduction which guarantees that types are preserved during evaluation.

Theorem 4.2 (Subject reduction)**.** *Let $L : T$. If $L \Downarrow V$ then $V : T$.*

Proof. Structural induction on $L \Downarrow V$. ☐

To prove progress, we follow Cousot and Cousot [1992] and Leroy and Grall [2009] and extend our big-step operational semantics to allow for non-terminating computations. In addition to the judgment $E\ [\theta\ ;\ \rho] \Downarrow V$ we also allow for diverging computation using the judgment $E\ [\theta\ ;\ \rho] \Downarrow^\infty$. For example, the diverging evaluation rules for products are shown below:

$$\frac{E_1\ [\theta\ ;\ \rho] \Downarrow^\infty}{(E_1, E_2)\ [\theta\ ;\ \rho] \Downarrow^\infty} \quad \frac{E_1\ [\theta\ ;\ \rho] \Downarrow V_1 \quad E_2\ [\theta\ ;\ \rho] \Downarrow^\infty}{(E_1, E_2)\ [\theta\ ;\ \rho] \Downarrow^\infty}$$

These rules should be read coinductively. We note that matching described by $\doteq$ and the substitution operation do not lead to non-termination in our operational semantics.

Lemma 4.3 (Canonical Forms)**.**

1. *If $V : T \to S$ then V is of the form:* $(\mathsf{fn}\ y.E)\ [\theta\ ;\ \rho]$
2. *If $V : \Pi X{:}U.T$ then V is of the form:* $(\lambda X.\, E)\ [\theta\ ;\ \rho]$

Proof. By inversion on value typing. ☐

Assuming that patterns cover all cases, we finally can state and prove progress.

Theorem 4.4.
If $L : T$ and for all values V, $\neg L \Downarrow V$ then $L \Downarrow^\infty$

Proof. (Classical) By coinduction and case analysis on the typing derivation, appealing to canonical forms. ☐

Corollary 4.5 (Progress)**.**
If $L : T$ then either $L \Downarrow V$ or $L \Downarrow^\infty$.

5. Mechanization

We have mechanized the proofs of the subject reduction and progress theorems presented in Section 4.4 in the Coq proof assistant. The development is approximately 800 lines of specification and under 500 lines of proof. We refer the interested reader to the supplementary material for this paper for the Coq proofs.

We do not formalize contextual LF. Rather, the proofs are abstract over the domain of meta-terms C and meta-types U. We need only assume that they behave well with meta-substitution and that pattern matching for C is decidable. This demonstrates our point that the computation language forms a general core for dependently typed languages parametric over the domain C.

We use the fresh look approach to binding due to Pouillard and Pottier [2010], which is an abstract interface to well-scoped de Brujin indices. Informally we found that the additional abstraction pushed us to towards a more high-level algebraic approach relying heavily on simultaneous substitutions in place of low level de Bruijn shifting.

A natural question to ask is why we did not choose to formalize the language in a system with built-in support for binding and substitution such as Beluga. One answer is that Beluga is currently lacking the recursive types we propose here. We would argue that the computation language is best represented as a recursive type, since the syntax contains meta-substitutions θ, which are conveniently represented as computational functions `#tm[`ψ`] -> tm[`ϕ`]` which cannot be written in LF. In fact we never perform substitutions on computation level expressions, hence the lack of substitution for free is moot.

6. Related Work

Over the past two decades, programming language researchers have been investigating language-based approaches to design safe and reliable software. Our work draws on and combines two domains: programming with binders and programming with indexed types.

Our work follows the tradition of indexed types [Xi and Pfenning 1999; Zenger 1997] where we separate the index domain of types from the computation-language. This has several known advantages: it is easy to allow state, exceptions, and polymorphism. Moreover, we are not restricted to total functions as in full dependently-typed languages such as Coq [Bertot and Castéran 2004] or Agda [Norell 2007]. However, since we can pattern match on index objects, we cannot naively erase them.

Most closely related to our work is the work by Chen and Xi [2005]; Xi [2004] and Sarkar [2009]. The ATS system designed by Xi and collaborators [Xi 2004] allows programmers to specify formal systems within the logical framework LF and embed LF objects as indices in computation-level types (see [Donnelly and Xi 2005]). The programmer can then supply her own proofs witnessing the equality between two objects when automatic constraint solving in ATS fails. However, the major challenge when manipulating and traversing LF objects is that we will encounter open LF objects, i.e. LF objects which may contain “free” variables. Unfortunately, ATS does not provide support for manipulating such open LF objects. To support certified programming, Sarkar [2009] proposed ML/LF which extended an ML-like language with LF as an index domain. To allow programming with open LF objects, he proposes to reify the dynamic assumptions which arise when traversing a binder and manipulate their representatives explicitly. This requires an extension of LF with Sigma-types and unit to model contexts and their dependencies. In contrast, our work builds on contextual LF [Pientka 2008] and explicitly supports contexts and parameter variables and types for them. This allows us to express strong invariants by for example stating that we map variables from a context to variables to another context which seems difficult in Sarkar's approach, since there is no guarantee by the underlying type system that we are only storing and manipulating variables in the reified context.

Westbrook et al. [2005] also suggests to index types with LF objects in a type-safe functional language to support programming with proofs in the presence of unrestricted recursion and imperative features while retaining decidable type checking. However their work restricts LF to the first-order fragment and explicitly forbids λ-abstractions. As a consequence, encodings based on higher-order abstract syntax are not supported.

In recent years, we have also made substantial advances in programming with binders. We build on the idea of contextual types which is central to Beluga [Pientka 2008]; however so far, Beluga and similar systems such as Delphin [Poswolsky and Schürmann 2008] are limited to only manipulating contextual LF objects. We take it in this paper one step further of allowing contextual objects and contexts as indices to computation-level types. Recently, Licata and Harper [2009] has proposed a system where one can mix computation functions and binding abstractions; this builds on their earlier ideas in Licata et al. [2008]. A prototype based on these ideas is implemented as a library within Agda and has been used to for example implement normalization by evaluation. Structural properties such as weakening or substitution do however not hold in general, but they can be implemented generically. While [Licata and Harper 2009] demonstrate convincingly that their library within Agda elegantly supports programming with binders, it is less clear whether their prototype will scale to support dependent types and meta-reasoning. It is also remarkable that we do not need to fully mix LF function space with the function space for computations to implement normalization-by-evaluation.

Taking a broader view on programming with binders, our work also seems superficially similar to Pouillard and Pottier [2010] where the authors describe an Agda library to support safe programming with names. The fundamental idea is to index terms with a world which names inhabit. When traversing a binding construct, we build up a chain of worlds which is similar to our context. However, it is unclear whether their work scales to support dependent types and hence encoding proofs and meta-reasoning. We hope that the presented work will enable us to shed further light on the relationship between nominal systems and LF-style encodings.

An interesting application of the presented work is its use as a tactic language for an interactive theorem prover. Stampoulis and Shao [2010] for example propose a language VeriML where we write computations about λHOL^{ind}, a higher-order logic with inductive definitions. At this point, VeriML only allows direct pattern matching on λHOL^{ind} objects, but does not allow in general computation-level types to be indexed with λHOL^{ind} objects. As a consequence, recursive types in VeriML cannot be dependently typed. Our approach of adding dependently-typed inductive types to the computation language is directly applicable to their work and would add additional flexibility to their language.

Our work takes inspiration of the work on inductive definitions in proof theory, in particular the work by McDowell and Miller [2002] and more recently Baelde et al. [2010]; Gacek et al. [2008] which targets reasoning about higher-order abstract syntax representations. Our addition of recursive types in Beluga$^{\mu}$ gives us effectively the same power as fixpoint definitions in their work. To facilitate reasoning about HOAS representations, Miller and his collaborators have extended the logic itself with the ∇-quantifier which allows generic quantification. Contexts must be modelled and reasoned about explicitly. In contrast, our work allows us to remain in first-order logic by generalizing the term language to allow for contextual meta-objects and contexts. This allows us to parameterize our recursive types over contexts and provide explicit support for reasoning about contexts. We believe the described work is an important step of understanding the differences and similarities between the approaches based on proof theory on the one hand and the approaches grounded in type theory on the other hand.

7. Conclusion and future work

We presented a type-theoretic foundation for programming with binders and indexed data-types. In particular, we have shown how to add indexed recursive types to the Beluga language and proven the extension to be type safe. We have also mechanized the type safety proof in Coq.

There are however more general lessons: we have streamlined earlier presentations of Beluga by separating meta-objects from computations. This has two important consequence: first, our computation language becomes modular; we can in fact easily replace contextual objects by another decidable index domain. Second, our modular approach lays the groundwork for adding contextual objects to other languages richer than the ML-like computation language we used in this paper, such as Agda.

In this paper, we have concentrated on programming with binders and indexed data-types, however frameworks such as Beluga are also proof development environments. To use the presented language as a core language for a proof assistant, we need to guarantee that the implemented functions are total, i.e. all cases are covered and the functions themselves are terminating. We believe coverage checking can be solved by extending prior work on coverage checking contextual objects [Dunfield and Pientka 2009; Schürmann and Pfenning 2003]. Termination checking requires us to identity a suitable notion of acceptable inductive datatypes, e.g. based on strict positivity as in Coq [Paulin-Mohring 1993], and we plan to adapt sized types as for example in Abel [2007, 2006] in the future.

References

Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transaction on Programming Language Systems*, 15(4):706–734, 1993.

Martin Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 147–160, New York, NY, USA, 1999. ACM Press. doi: http://doi.acm.org/10.1145/292540.292555.

Andreas Abel. Mixed inductive/coinductive types and strong normalization. In Zhong Shao, editor, *5th ASIAN Symposium on Programming Languages and Systems (APLAS'07)*, volume 4807 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2007.

Andreas Abel. Polarized subtyping for sized types. *Mathematical Structures in Computer Science*, 18(5):797–822, 2006. Special issue on subtyping, edited by Healfdene Goguen and Adriana Compagnoni.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In Luke Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer Berlin / Heidelberg, 2010.

David Baelde, Zach Snow, and Dale Miller. Focused inductive theorem proving. In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI), pages 278–292. Springer, 2010.

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Logic in Computer Science*, pages 203–211, 1991.

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

Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In Olivier Danvy and Benjamin C. Pierce, editors, *10th International Conference on Functional Programming*, pages 66–77, 2005.

K. L. Clark. Negation as failure. In J. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, pages 83–94. ACM, 1992.

Kevin Donnelly and Hongwei Xi. Workshop on Mechanized reasoning about languages with variable binding (MERLIN'05). In Randy Pollack, editor, *Combining higher-order abstract syntax with first-order abstract syntax in ATS*, pages 58–63. ACM, 2005.

Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2008.

D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19)*. IEEE Computer Society Press, 2006.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 83–92, 2007.

Robert Harper and Daniel R. Licata. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

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

Xavier Leroy and Herv Grall. Coinductive big-step operational semantics. *Information and Computation*, pages 284–304, 2009.

Daniel R. Licata. *Dependently Typed Programming with Domain-Specific Logics*. PhD thesis, Carnegie Mellon University, 2011.

Daniel R. Licata and Robert Harper. A universe of binding and computation. In Graham Hutton and Andrew P. Tolmach, editors, *14th ACM SIGPLAN International Conference on Functional Programming*, pages 123–134. ACM Press, 2009.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*, pages 241–252. IEEE Computer Society Press, 2008.

Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002. ISSN 1529-3785.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *In Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

Kenji Miyamoto and Atsushi Igarashi. A modal foundation for secure information flow. In A. Sabelfeld, editor, *Workshop on Foundations of Computer Security (FCS'04)*, pages 187–203, 2004.

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008a.

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

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

Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications(TLCA '93), Utrecht, The Netherlands*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. ISBN 3-540-56517-5.

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

Brigitte Pientka. Programming proofs: A novel approach based on contextual types. *submitted*, 2011.

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

Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21, 2010.

Adam Poswolsky and Carsten Schürmann. System description: Delphin— a functional programming language for deductive systems. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 135–141. Elsevier, 2009.

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

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

Nicolas Pouillard and Franois Pottier. A fresh look at programming with names and binders. In *15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 217–228, 2010.

Susmit Sarkar. *A Dependently Typed Programming Language, with applications to Foundational Certified Code Systems*. PhD thesis, Carnegie Mellon University, 2009. CMU-CS-09-128.

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'03)*, volume 2758 of *Lecture Notes in Computer Science*, pages 120–135, Rome, Italy, 2003. Springer.

Tim Sheard. Languages of the future. *SIGPLAN Notices*, 39(12):119–132, 2004.

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

Antonis Stampoulis and Zhong Shao. VeriML: typed computation of logical terms inside a language with effects. In Paul Hudak and Stephanie Weirich, editors, *15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, pages 333–344. ACM, 2010.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'07)*, pages 53–66. ACM, 2007. ISBN 1-59593-393-X.

E. Westbrook, A. Stump, and I. Wehrman. A Language-based Approach to Functionally Correct Imperative Programming. In Olivier Danvy and Benjamin C. Pierce, editors, *10th International Conference on Functional Programming (ICFP05)*, pages 268–279. ACM, 2005.

Hongwei Xi. Applied type system. In *TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2004.

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

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, pages 224–235. ACM Press, 2003.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2): 147–165, 1997.

A. Hereditary substitution

Meta-substitution for terms

$$\begin{array}{lll}
\llbracket C/X \rrbracket_U(\lambda x.M) & = \lambda x.M' & \text{where } \llbracket C/X \rrbracket_U(M) = M' \\
\llbracket C/X \rrbracket_U(u[\sigma]) & = R' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.R/u \rrbracket_{P[\Psi]} \\
 & & \text{and } [\sigma']_\Psi(R) = R' \\
\llbracket C/X \rrbracket_U(u[\sigma]) & = u[\sigma']' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.R/u \rrbracket_{P[\Psi]} \\
\llbracket C/X \rrbracket_U(R\,N) & = R'N' & \text{where } \llbracket C/X \rrbracket_U(R) = R' \\
 & & \text{and } \llbracket C/X \rrbracket_U(N) = N' \\
\llbracket C/X \rrbracket_U(R\,N) & = M'' : B & \\
\multicolumn{3}{r}{\text{if } \llbracket C/X \rrbracket_U(R) = \lambda y.M' : \Pi x{:}A_1.B \text{ where } \Pi x{:}A_1.B \leq U} \\
\multicolumn{3}{r}{\text{and } N' = \llbracket C/X \rrbracket_U(N) \text{ and } M'' = [N'/y]_{A_1}(M')} \\
\llbracket C/X \rrbracket_U(x) & = x & \\
\llbracket C/X \rrbracket_U(\mathbf{c}) & = \mathbf{c} & \\
\llbracket C/X \rrbracket_U(p[\sigma]) & = R' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.R/p \rrbracket_{\#A[\Psi]} \\
 & & \text{and} [\sigma']_\Psi(R) = R' \\
\llbracket C/X \rrbracket_U(p[\sigma]) & = M' : A & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \\
 & & \llbracket C/X \rrbracket_U = \llbracket \hat{\Psi}.R/p \rrbracket_{\#A[\Psi]} \\
 & & \text{and } [\sigma']_\Psi(R) = M' : A \\
\llbracket C/X \rrbracket_U(p[\sigma]) & = p[\sigma'] & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U \neq \llbracket \hat{\Psi}.R/p \rrbracket_{\#A[\Psi]}
\end{array}$$

Meta-substitution for substitutions

$$\begin{array}{lll}
\llbracket C/X \rrbracket_U(\cdot) & = \cdot & \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi) & = \sigma & \text{where } \llbracket \mathcal{C}/X \rrbracket_U = \llbracket \Psi/\psi \rrbracket_G \\
 & & \text{and } \mathsf{id}(\Psi) = \sigma \\
\llbracket C/X \rrbracket_U(\mathsf{id}_\psi) & = \mathsf{id}_\psi & \text{where } \llbracket \mathcal{C}/X \rrbracket_U \neq \llbracket \Psi/\psi \rrbracket_G \\
\llbracket C/X \rrbracket_U(\sigma, M) & = \sigma', M' & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma' \text{ and} \\
 & & \llbracket C/X \rrbracket_U(M) = M' \\
\llbracket C/X \rrbracket_U(\sigma; x) & = \sigma'; x & \text{where } \llbracket C/X \rrbracket_U(\sigma) = \sigma'
\end{array}$$

Meta-substitution for context

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

B. Examples

B.1 Closure conversion

```
tm : type .
lam: (tm → tm) → tm.
app: tm → tm → tm.

schema ctx = tm;


envr : type .
ctm : type .
clam : (envr → ctm) → ctm.
proj : envr → nat → ctm.
close : ctm → envr → ctm.
open : ctm → (env → ctm → ctm) → ctm
nil : envr.
create : envr → ctm.
snoc : envr → ctm → envr.

schema cctx = ctm;


rec addProjs : (ϕ:cctx) (N:nat[]) (M:cexp[ϕ,e:envr])
  → cexp[e:envr] =
λ φ ⇒ λ N ⇒ λ M ⇒ case  φ of
| [] ⇒ [e:envr]. M e
| φ,x:ctm ⇒ addProjs φ (s N) (φ,e. M .. (proj e N) e)
```

```
rec  ctxToEnv : (φ:cctx) envr[φ] =
λ φ ⇒ case  φ of
| [] ⇒ []. nil
| φ,x:ctm ⇒ let  φ. env .. = ctxToEnv φ in
  φ,x:ctm. snoc (env ..) x
```

```
datatype  ctx_rel : ctx → cctx → ctype
| rnil : ctx_rel [] []
| rsnoc : {ψ φ} → ctx_rel ψ φ
  → ctx_rel (ψ,x:tm) (φ,x:ctm)
```

```
rec  conv : {ψ:ctx} (φ:cctx) → ctx_rel ψ φ → tm[ψ]
  → ctm[φ] =
λ φ ⇒ fn  cr  ⇒ fn  m ⇒ case  m of
| ψ',x. x     ⇒ let  rsnoc {φ = φ',x} _ = cr in φ',x. x
| ψ',x. #p .. ⇒ let  rsnoc {φ = φ',x} cr' = cr in
 let  φ'. M .. = conv _ cr' (ψ'. #p ..) in φ',x. M ..
| ψ. lam (λx. M .. x) ⇒
 let  φ, x. M' .. x = conv _ (rsnoc cr) (ψ,x. M .. x) in
 let  [ev:envr]. M'' ev = addProjs (φ,x) z (φ, x:ctm, ev:
      envr. M' .. x) in
 let  φ. Env .. = ctxToEnv φ in
 φ.  close (clam (λev. M'' ev)) (Env ..)
| ψ. app (M ..) (N ..) ⇒
 let  φ. M' .. = conv _ cr (ψ. M ..) in
 let  φ. N' .. = conv _ cr (ψ. N ..) in
 φ. open (M' ..)
         (λenv. λf. capp f (create (snoc env (N' ..))))
```

B.2 Normalization by Evaluation

```
atomic_tp : type .
tp : type .
atomic : atomic_tp → tp.
arr : tp → tp → tp.
```

```
tm : tp → type .
app : tm (arr T S) → tm T → tm S.
lam : (tm T → tm S) → tm (arr T S).
```

```
schema  tctx = some  [T:tp] tm T;
```

```
neut : tp → type .
norm : tp → type .
nlam : (neut T → norm S) → norm (arr T S).
rapp : neut (arr T S) → norm T → neut S.
embed : neut (atomic P) → norm (atomic P).
```

```
schema ctx = some  [T:tp] neut T;
```

```
type  sub ψ φ = {T:tp[]} → #(neut T)[ψ] → #(neut T)[φ]
```

```
datatype  sem : ctx → tp[] → ctype =
| syn : {ψ} {P:atomic_tp[]}
   → (neut (atomic P))[ψ] → sem ψ  (atomic P)
| slam : {ψ A B} ({φ} → sub ψ φ → sem φ  A → sem φ  B)
   → sem ψ  (arr A B)
```

```
rec  subst:{ψ  φ  S} → sub ψ φ → sem ψ  S → sem φ  S =
fn  σ ⇒ fn  e ⇒ case  e of
| syn (ψ. R ..) ⇒ nsubst σ (ψ. R ..)
| slam f ⇒ slam (fn  σ' ⇒ fn  s ⇒ f (σ' ∘ σ) s)
```

```
rec  nsubst : {ψ  φ  S} → sub ψ φ → (neut S)[ψ]
 → (neut S)[φ] =
fn  σ ⇒ fn  e ⇒ case  e of
| ψ. #p .. ⇒ σ (ψ. #p ..)
| ψ. rapp (R ..) (N ..) ⇒
  let  φ. R' .. = nsubst σ (ψ. R ..) in
  let  φ. N' .. = nosubst σ (φ. N ..) in
  φ. rapp (R' ..) (N' ..)
```

```
rec  nosubst : {ψ  φ  S} → sub ψ φ → (norm S)[ψ]
 → (norm S)[φ] =
fn  σ ⇒ fn  e ⇒ case  e of
| ψ. embed (R ..) ⇒
  let  φ. R' .. = nsubst σ (ψ. R ..) in
  φ. embed (R' ..)
| ψ. nlam (λx. N .. x) ⇒
  let  φ,x:neut _. N' .. x = nosubst
    (fn  y ⇒ case  y of
      | ψ,x:neut _. x ⇒ φ,x:neut _. x
      | ψ,x:neut _. #p .. ⇒
         let  φ. #q .. = σ (ψ. #p ..) in
         φ,x:neut _. #q ..
    ) (ψ,x:neut _. N .. x) in
  φ. nlam (λx.  N' .. x)
```

We remark that if we add the type ψ`[`ϕ`]` of substitutions [Pientka 2008] to the domain of contextual objects then the above implementations of `nsubst` and `nosubst` could take advantage of substitution-for-free by first writing a conversion function of type `{ψ φ} → sub ψ φ → ψ[φ]`.

```
rec  reflect : (ψ A) (R:(neut A)[ψ]) → sem ψ A =
λ ψ ⇒ λ A ⇒ λ R ⇒ case  []. A of
| []. atomic P ⇒ syn (ψ. R ..)
| []. arr T S ⇒ slam (λ {φ} ⇒ fn σ ⇒ fn  s ⇒
 let  φ. R' .. = nsubst σ (ψ. R ..) in
 let  φ. N .. = reify _ T s in
 reflect _ S (φ. rapp (R' ..) (N ..)))
```

```
rec  reify : (ψ A) → sem ψ  A → (norm A)[ψ] =
λ ψ ⇒ λ A ⇒ fn  s ⇒ case  []. A of
| []. atomic P ⇒ let  syn (ψ. R ..) = s
 in  ψ. embed (R ..)
| []. arr T S ⇒ let  slam f = s in
 let  ψ,x:tm T. E .. x =
   reify (f weaken (reflect _ T (ψ,x:neut T. x)))
 in  ψ. nlam (λx. E .. x)
```

```
rec  weaken : {ψ:ctx} {S:tp[]} → sub ψ (ψ,x:tm S) =
fn  y ⇒ case  y of
| ψ. #p .. ⇒ ψ,x:neut _. #p ..
```

```
rec  eval : {ψ  φ  S} →
  → ({T} #(tm T)[ψ] → sem φ T)
  → (tm S)[ψ] → sem φ S =
fn  r ⇒ fn  σ ⇒ fn  e ⇒ case  e of
| ψ. #p ..  ⇒ σ (ψ . #p .. )
| ψ. lam (λx. E .. x) ⇒ slam (fn σ' ⇒ fn  s ⇒
   eval (extend ((subst σ') ∘ σ) s) (ψ,x. E .. x)
| ψ. app (E1 ..) (E2 ..) ⇒
   let  slam f = eval σ (ψ. E1 ..) in
   f id (eval σ (ψ. E2 ..))
```

```
rec  extend : {ψ:tctx} {φ:ctx} {S}
  → ({T} → #(tm T)[ψ] → sem φ T) → sem φ S
  → ({T} → #(tm T)[ψ,x:tm S] → sem φ T) =
fn  σ ⇒ fn  s ⇒ fn  y ⇒ case  y of
| ψ,x:tm S. ⇒ s
| ψ,x:tm S. #p .. ⇒  σ (ψ. #p ..)
```

```
rec  nbe : {A} → (tm A)[ ] → (norm A)[ ] =
fn  e ⇒ reify [] A (eval (fn  y ⇒ impossible y) e)
```