Generic programming with binders and scope

Steven Keuchel

MSc Thesis

August 31, 2011

ICA-3719936

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisor:
prof. dr. J.T. Jeuring
Second Supervisor:
dr. A. Löh

Contents

1 Introduction

Meta-programming is a discipline that concerns itself with writing programs like compilers, interpreters, refactorers and theorem provers that process data representing languages. The language in which the program is written is commonly called the *meta-language*, and the language that is being process is called the the *object language*. Many symbolic languages in mathematics or computer science involve the use of variables and variable binding. Significant effort has been put into the development of idioms for programming with and reasoning about syntax that involves binding and many approaches have been proposed with various advantages and disadvantages. However, dealing with binding in the implementation of meta-programs is and remains notoriously difficult and troublesome. To alleviate the pain researchers seek to develop programming languages and libraries that help taking care of this matter. This thesis presents the development of a datatype-generic approach to syntax with binding and a generic programming library that are particularly useful in the context of domain-specific languages.

The main driving force behind this thesis project is the development of a embedded domain-specific language for specifying solutions to program exercises of an intelligent tutoring system by Gerdes [GHJ08] and Gerdes et al. [GJH10]. The object language is a subset of Haskell that includes rich binding forms such as patterns and recursive let expressions. To this end we make it a requirement to model this rich forms in our approach.

1.1 Motivation

Domain-specific languages

Languages that have attracted attention in the past are domain-specific languages (DSLs) and they have been successfully used in software development to overcome semantic gaps between application domains and software implementations. They are tailored to a particular domain, focus on its semantics and provide natural notations. Domain experts that are not necessarily programmers need not be familiar with the underlying software implementation, but simply use the language.

Standalone implementations of DSLs require the development and maintenance of compilers or interpreters and tools like debuggers. They also have to be interfaced with other software components. In order to avoid these tedious issues DSLs have been embedded in general-purpose languages like Haskell. However, this also implies that the DSL is subject to restrictions implied by its host. Semantic and syntactic properties and constraints are not easily expressible and usually left implicit. In particular this applies to scoping rules of languages with binding and domain-specific type systems for DSLs. This impaires writing and maintaining good software.

Abstract syntax representation

Simply using for example strings or a similar datatype to encode variables is known as *first-order abstract syntax*. It is easy to specify and use, however, it is also an inherently unsafe representation especially for use with embedded DSLs where we write abstract syntax terms directly. This representation does not distinguish the use of names as binding sites or reference sites of variables and the scoping rules of the language are not encoded but simply left implicit in the implementation.

For DSLs we not only want to make the scoping explicit but as with normal programs ensure statically that programs we write in DSLs are fully syntactic correct. For example, all variables should have a corresponding binding site or explicitly be free variables. A representation that provides this possibility is called a *well-scoped* representation. Type-systems are a form of lightweight syntactic checks for program correctness. Again we would wish to use these also for DSLs and have syntax representations that are *well-typed*.

Well-scoped or even well-typed representations are not only useful for the correctness of software written in DSLs, but also for the DSL implementation. Embedded interpreters benefit from explicitly available type information, because they free the programmer from the trouble what to do when the impossible happens. Like normal well-typed programs also well-typed DSL programs do not go wrong.

We can coarsely classify approaches to represent syntax into first-order and higher-order approaches. Higher-order approaches stand out because they use the function space of the meta-language to encode variable binding. We refer to other approaches as being first-order.

Higher-order abstract syntax

The higher-order abstract syntax (HOAS) approach uses the function space of the meta-language to model binding structures of the object language, and represents object language variables by meta-language variables. In fact the meta-language lambda-expression $\lambda\ \mathsf{x} \rightarrow \mathsf{e}$ binds the variable x in the expression e. We can use this in the the definition of a recursive datatype for representing the syntax of a language: each term former with variable binding is modeled by a constructor that takes a function as argument. For the untyped λ-calculus we arrive at the following datatype definition:

```
infixl 5 _·_
data Lam : Set where
  abs : (Lam → Lam) → Lam
  _·_ : Lam → Lam → Lam
```

Note that there is no explicit constructor for variables, because they are handled by the function space used in the λ-abstraction case. Using this datatype definition we can specify values representing the I, K and S combinators:

```
I = abs (λ x → x)
K = abs (λ x → abs (λ y → x))
S = abs (λ f → abs (λ g → abs (λ x → f · x · (g · x))))
```

Being able to write terms with names provides a convenient approach to embed domain-specific languages with binding. Moreover, the scoping of the object language terms is determined by the scoping of meta-language λ-expressions and furthermore all variables are bound, i.e. Lam encodes closed λ-terms. Moreover, it is also possible to encode type information in the HOAS representation. However, while HOAS provides a convenient way to write programs in DSLs with binding, it is less suited as a representation used in implementations. Variables are only implicitly represented and thus questions like how often a specific variable occurs in a term cannot be answered easily and thus writing analyses over the syntax is not directly possible.

Generic treatment of syntax

On the implementation side of meta programs, operations on syntax terms appear when analyzing, transforming, interpreting or compiling languages. A huge number of these operations is not specific to a single language, but is common to a class of languages. When dealing with syntax with bindings commonly found functionality are operations like variable substitutions, term weakening and strengthening that are used to perform transformations for optimization. Furthermore, depending on the used representation it might be possible or impossible to perform certain analyses, so that conversions between representations become necessary. Again conversion between similar representations for different languages exhibit common functionality.

When implementing this common functionality for different types the same programming pattern has to be applied over and over again which is tiring and therefore error-prone. The goal is to elide this burden by defining a single generic function that can be instantiated to different concrete cases. This reuse of code not only prevents subtle mistakes, it also avoids redevelopment and therefore speeds-up the creation of new systems and raises their quality while reducing development costs. Besides reusability, generic solutions ensure maintainability and the ability for evolution of embedded DSLs because they easily adapt to changes in the DSL implementation.

1.2 Related work

Datatype-generic programming techniques have been used before to develop libraries of generic functions for syntax based on different representations. Cheney [Che05a] describes the implementation of a Haskell library called *FreshLib* for generic programming based on *nominal abstract syntax* representations. The library provides a single abstraction type constructor that explicitly indicates scoping and it provides capture-avoiding substitutions and free-variable computations that respect the scoping of the abstraction type. However, while the provided functions treat the syntax in a safe way no static guarantees about the safe use of names is given, i.e. the appearing names are not enforced to be bound and the scoping rules are not encoded in the representation but left implicit in the implementation of the generic functions. Moreover, the library only handles single variable bindings generically, but it is possible to non-generically extend the library with binders that bind multiple variables simultaneously.

Oliveira et al. [OLCY11] are working on a datatype-generic framework for the mechanization of formal meta-theory with implementations in Coq and Agda. They cover multiple first-order

approaches for representing syntax with binding and provide generic syntax operations like substitutions including associated *infrastructure lemmas* for use in proofs. They use an enriched version of the universe of regular tree types [MAM06] to make the scoping structure of syntax explicit. However, their goal is to mechanize established ways to formalize meta-theory and thus their focus is on traditional first-order representations and not on well-scoped representations. Moreover, they also limit themselves to the treatment of single variable bindings.

Licata and Harper [LH09] present a universe in Agda that allows the definitions of syntax terms that mix binding and computations. Their representation is based on well-scoped de Bruijn terms where scoping is made explicit in the representation types. They provide substitutions, term weakening and strengthening generically, but they only handle the single variable case.

Weirich, Yorgey and Sheard [WYS11] present a domain-specific language in Haskell for specification of binding structure of languages together with a generic library that provides generic versions of α-equivalence and capture-avoiding substitutions for free. Their focus is on providing an practical library to be used in real-world implementations of meta programs. As such it is particularly expressive because it covers bindings of an arbitrary amount of variables simultaneously and more sophisticated binding forms that include sequential or recursive scoping. Internally they use unsafe first-order representations, but their interface remains abstract and they prove that their implementation handles terms in a safe way.

1.3 Contributions

The main contribution of the thesis is the development of a datatype-generic approach to abstract syntax that lifts the expressivity bar of previous approaches so that it can be used in the development of domain-specific languages. We impose the requirement to specify and handle well-scoped and simply well-typed representations generically. Moreover, we also want to make rich binding forms found in real languages available.

Associated with the approach we provide an implementation of a generic library of syntax operations for use in implementations. Specifically we include generic conversions from a higher-order representation to a first-order representation so that the interface language of a DSL can be generically transformed into an internal language for processing. Furthermore, we include capture-avoiding substitutions, term weakening and strengthening on the first-order representation.

1.4 Approach

The dependently-typed language Agda developed by Norell [Nor07] will be used for the code in this thesis. Agda's powerful type system allows us to implement type level computations easily and specify constraints on our representations. Dependently-typed languages make universe constructions available that will be used to define generic functionality over syntax types. A universe is a collection of types defined by a set of codes $\mathsf{C} : \mathsf{Set}$ and an interpretation function $\mathsf{E} : \mathsf{C} \to \mathsf{Set}$ mapping codes to actual types. A generic function applies common functionality to different datatypes where the functionality only depends on the structure. To define generic

functions for all types in a universe, the relevant structure needs to be reflected in the codes. The generic functions then inspect the code of a specific datatypes to learn about its structure and apply the functionality.

We will develop a universe that forms a specification language for the scoping structure of languages and will provide two interpretations for it. One will be based on de Bruijn representations that are well-scoped and that can be made well-typed for simple type systems. A second interpretation will be based on parametric higher-order approach to abstract syntax as advocated by Washburn and Weirich [WW03] and practically exemplified for DSLs by Atkey, Lindley and Yallop [ALY09].

Abstract syntax definitions for languages make heavy use of mutually recursive datatypes and operations like heterogeneous substitutions need the availability of the recursive structure of types. To this end the approach to datatype-generic programming will be based on fixed points of functors for families of datatypes as described by Yakushev et al. [YHLJ09] which allows the use of the full recursive structure of families of mutually recursive datatypes in the definitions of generic functions. We will extend their definitions to cover the scoping structure of languages and moreover not only use functors between families of simple datatypes but between families of indexed datatypes that model safe syntax representations.

1.5 Overview

We begin with the introduction of well-formed de Bruijn representations in chapter 2, i.e. well-scoped or aditionally well-typed representations. In chapter 3 we will introduce parametric higher-order abstract syntax representations that can be used for interface languages and show a non generic conversion to well-scoped de Bruijn representation for the untyped lambda calculus.

In chapter we will present increasingly expressive universes specifying binding structure of languages up to the point where we can specify rich binding forms with recursive and sequential scoping. On the way we will develop an interpretation based on well-formed de Bruijn representations and at the same time on parametric higher-order abstract syntax and also show generic conversion functions from the latter to the former and exemplify their use.

We begin in section 4.1 with a small universe of syntax types with unary scoping of a single sort corresponding to regular datatypes in Set. Subsequently in section 4.2 we will generalize this universe to families of syntax types with unary scoping. As an intermediate step to show the parametric HOAS representation of multi-variable binders we will inject simple binders into the universe, i.e. we will not reflect their structure in the universe. As a last step we will show a universe of binders in section 4.4 in which types may do both, bind variables and at the same time represent terms that reference variables and we show how recursive and sequential scoping can be expressed in that universe.

In chapter 5 we will develop a generic traversal function for the well-formed de Bruijn interpretation of the most expressive universe of binders and show how to implement term weakening, shifting as well as substitutions using the generic traversal. In section we will show a type-indexed datatype that encodes a predicate for free occurrences of variables in terms and show how to use it to implement strengthening of terms in case a variable does not appear free.

2 Well-formed de Bruijn representations

In this section we introduce essential concepts for dealing with language syntax with binding for well-formed de Bruijn representations. Different forms of scoping and binding structures that are found in many languages are introduced. We will start with a general treatment of scope, look at the untyped λ-calculus as a basic example to which we progressively introduce other, sometimes more exotic, forms of binding.

Scoping is an integral concept of syntax with binding, which explains how to associate variable occurrences with binding sites. The type of scoping we will be most interested in is commonly referred to as *static* or *lexical scoping* which is used in λ-calculi as well as most modern programming languages. In static scoping a variable occurrence refers to a variable binding in a syntactically enclosing binding site. Because it is a syntactic property it can be resolved entirely by looking at a syntax term and taking language specific scoping rules into account. Furthermore in this type of scoping the binding site a variable references does not change, it is fixed. In contrast to that *dynamic scoping* associates variables with binding sites based on run-time information.

We will use the following definitions of scope and context without further formalizing signatures, terms or term positions. The *scope* of a *variable* v in a term t is the set of all positions within t where the variable v can be referenced. We will also use a reversed notation and say that a variable v is in scope at a position p, if p is an element of the scope of v. The context of a term position p of a term t is the set of all variables which are in scope at that position.

2.1 Unary scoping

In the untyped λ-calculus λ-abstraction is the only binding construct and exactly one variable is bound in an abstraction. The scope of the bound variable is the body of the abstraction, which is represented in an abstract syntax tree by a single immediate subtree, where shadowing might remove subtrees. We will call this form of scoping *unary scoping*.

Figure 2.1 shows an abstract syntax tree for the λ-term $(\lambda\mathsf{x}.\mathsf{x})\ (\lambda\mathsf{f}.\lambda\mathsf{x}.\mathsf{f}\ \mathsf{x})$ in context Γ where each node corresponding to a sub-term is annotated with its context. Clearly for every variable occurrence the variable is an element of the context in the annotation. We will call a term t *well-scoped* in context Γ if this property holds for every variable in t. The term in Figure 2.1 is well-scoped in every context because it is *closed*, but the term $(\lambda\mathsf{x}.\mathsf{g}\ \mathsf{x})$ is only well-scoped in contexts containing g.

Well-scopedness is a property we can enforce in the definition of datatypes representing syntax. The general idea is to form an inductive family of datatypes where the index set determines the context and variable occurrences are restricted to variables from the context given by the in-

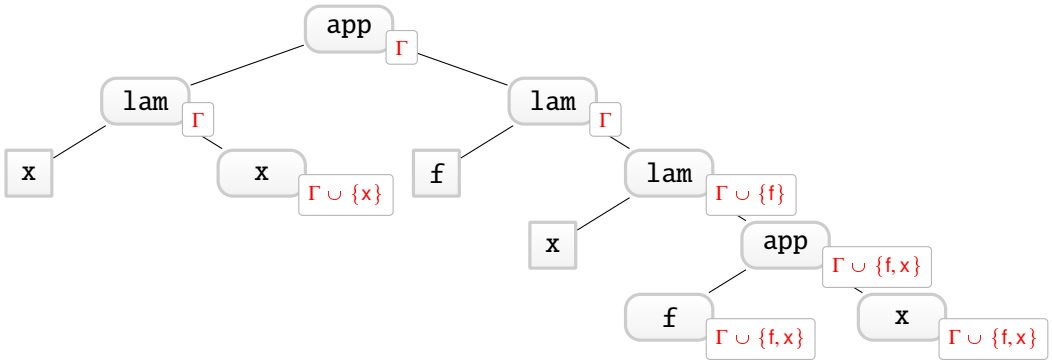

Figure 2.1: Syntax tree for $(\lambda x.x)\ (\lambda f.\lambda x.f\ x)$ annotated with contexts.

dex of the family, i.e. the type of variables is a membership predicate on the context and variables themselves are represented as proofs for this predicate.

For a nameless representation using de Bruijn indices the contexts are determined by natural numbers, which we use as the index set. A natural number determines an upper bound on valid de Bruijn indices. Thus well-scoped de Bruijn indices can be modeled by the inductive family of bounded naturals $\mathsf{Fin\ n} = \{0, ..., n\text{-}1\}, n \in \mathbb{N}$.

```
data Fin : ℕ → Set where
  zero : {n : ℕ}          → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

Now we can define an inductive family for untyped λ-terms using a well-scoped de Bruijn representation. Note that in case of a λ-abstraction the context has to be extended, which is achieved by incrementing the index.

```
data Lam (n : ℕ) : Set where
  var : Fin n         → Lam n
  abs : Lam (1 + n)   → Lam n
  app : Lam n → Lam n → Lam n
```

2.2 Multi-sorted scopes

Object languages with polymorphism like System F deal with two syntactic sorts with variables: terms and types. The representation for well-scoped object language types can use the scheme from the previous section. To achieve a well-scoped term representation we index the datatype family with two copies of the naturals. One determines upper bounds on indices for term variables and the other upper bounds on indices for type variables.

```
data Type (ty : ℕ) : Set where
  var : Fin ty → Type ty
```

```
  for    : Type (1 + ty) → Type ty
  arr    : Type ty → Type ty → Type ty
data Lam (tm : ℕ) (ty : ℕ) : Set where
  var    : Fin tm                      → Lam tm ty
  abs    : Type ty → Lam (1 + tm) ty → Lam tm ty
  app    : Lam tm ty → Lam tm ty     → Lam tm ty
  τabs   : Lam tm (1 + ty)           → Lam tm ty
  τapp   : Lam tm ty → Type ty       → Lam tm ty
```

Another option is to use a single context type for both term and type variables. To prevent the unsafe situation where an index for a type variable is used in a position of a term variable we have to extend the indices with information about the sort of variables they are representing. In this scheme the context is a list of sorts and an index for a specific sort s is then a position within that list where the sort s appears. Natural numbers as upper bounds for de Bruijn indices allow a similar interpretation: a unary representation of a natural number can be viewed as a list of a single sort.

To avoid confusion in the presentation we do not use the normal list datatype, but define a new Ctx type specifically representing contexts and an inductive family _∋_ representing indices for contexts from Ctx.

```
module Context (Sort : Set) where
  infixl 5 _▷_
  data Ctx : Set where
    ϵ     : Ctx
    _▷_ : (Γ : Ctx) (s : Sort) → Ctx
  infix 4 _∋_
  data _∋_ : (Γ : Ctx) → Sort → Set where
    vz : {Γ : Ctx} {s  : Sort}          → Γ ▷ s ∋ s
    vs : {Γ : Ctx} {s t : Sort} → Γ ∋ s → Γ ▷ t ∋ s
```

Note the similarity in the definition of the bounded naturals Fin and the context indices _∋_. Both have one base constructor and one constructor for the successor and are bounded by their context index. In that sense _∋_ still represents de Bruijn indices but all sorts share a common namespace and the sort indexing guarantees that we don't mix variables of different sorts.

The System F example with two natural number bounds can be recast in this scheme. A two-valued datatype is used as a representation of the two sorts.

```
data Sort : Set where
  type : Sort
  term : Sort
open Context Sort
```

In the case of a type binder, such as the universal quantifier for types or the type abstraction in terms, we extend the context with the sort value representing types and in the case of the λ-abstraction we extend the context with the sort value representing terms.

```
data Type (Γ : Ctx) : Set where
  var : Γ ∋ type           → Type Γ
  for : Type (Γ ▷ type)    → Type Γ
  arr : Type Γ → Type Γ → Type Γ
data Lam (Γ : Ctx) : Set where
  var  : Γ ∋ term                    → Lam Γ
  abs  : Type Γ → Lam (Γ ▷ term)     → Lam Γ
  app  : Lam Γ → Lam Γ               → Lam Γ
  τabs : Lam (Γ ▷ type)              → Lam Γ
  τapp : Lam Γ → Type Γ              → Lam Γ
```

Both approaches can be used with an arbitrary but finite number $\mathsf{n} \in \mathbb{N}$ of different syntactic sorts: by using $\mathbb{N}^{\mathsf{n}}$ respectively Ctx (Fin n) for the context. But only the second approach scales to situations with an infinite number of "sorts" like the simply typed λ-calculus of the next section.

2.3 Simple well-typed terms

In the previous section we enriched the context with information about the syntactic sort a variable is representing. This idea can be taken further to include type information about variables in the context and give a definition of a well-typed representation of the simply-typed λ-calculus.

We consider simple types with one base type unit.

```
infixr 6 _↠_
data Ty : Set where
  unit : Ty
  _↠_ : (τ₁ τ₂ : Ty) → Ty
open Context Ty
```

The well-scoped and well-typed de Bruijn representation is given by the following family which is indexed over two sets: one for contexts and one for types of terms. The type $\Gamma \vdash \tau$ denotes well-typed object language terms of object language type τ in context Γ. Note that the indices in the data declaration correspond exactly to the typing rules of the simply-typed lambda calculus (see for example Pierce [Pie02, p. 99 ff.]).

```
infix  2 _⊢_
infixl 10 _·_
data _⊢_ (Γ : Ctx) : Ty → Set where
  var : ∀ {τ}     (x : Γ ∋ τ)                       → Γ ⊢ τ
  _·_ : ∀ {τ₁ τ₂} (f : Γ ⊢ τ₁ ↠ τ₂) (a : Γ ⊢ τ₁)    → Γ ⊢ τ₂
  ƛ   : ∀ {τ₁ τ₂} (b : Γ ▷ τ₁ ⊢ τ₂)                 → Γ ⊢ τ₁ ↠ τ₂
  tt  :                                               Γ ⊢ unit
```

2.4 Binders

Modern functional programming languages like Haskell and ML let the user define algebraic datatypes and provide mechanisms to pattern match on values of those types. In an abstract syntax representation we will have besides a usual datatype for expressions a new datatype describing patterns including the variables that will be bound by patterns. That is, a value that specifies variables to be bound. We will calls such a value a *binder*. An abstraction then consists of a binder that binds a certain set of variables and a term in which those variables are considered bound.

In a nameless de Bruijn representation of patterns names are just replaced by a constructor that indicates a binding site of a variable. Furthermore, we need to define for every pattern value the order in which those bindings are added to the context. In a well-scoped representation this order is made explicit.

There are two possibilities to provide a well-scoped de Bruijn representation of a language with patterns. We can think of a pattern as a relation between contexts. The context outside of the abstraction is related to the bigger inner context, where the variables of the pattern are considered to be bound. Alternatively we can think of a pattern as binding a list of distinct variables and the inner context is the outer context extended by this list. We will look at both possibilities for an untyped λ-calculus with products and multi-level pattern matching on products.

2.4.1 Relation view

The following code shows a representation of patterns in the relation view. It is a datatype indexed by two contexts where a pattern variable $\bullet$ relates any context with itself extended by one term variable. A pattern for pairs $\_,\_$ consists of two sub-patterns and the resulting relation is the composition of the relations of the sub-patterns, i.e. we accumulate the variables bound by both sub-patterns. The definition explicitly enforces that the variables of the left sub-pattern are bound first. A pattern-match alternative in context Γ consists of a PatR Γ Γ^+ and a term Lam Γ^+ in the inner context.

```
data PatR (Γ : Ctx) : Ctx → Set where
  •   : PatR Γ (Γ ▷ term)
  _,_ : ∀ {Γ₂ Γ₃} → PatR Γ Γ₂ → PatR Γ₂ Γ₃ → PatR Γ Γ₃
data LamR (Γ : Ctx) : Set where
  var : Γ ∋ term → LamR Γ
  abs : ∀ {Γ⁺} → PatR Γ Γ⁺ → LamR Γ⁺ → LamR Γ
  app : LamR Γ → LamR Γ → LamR Γ
  _,_ : LamR Γ → LamR Γ → LamR Γ
```

As an example consider the swap4 combinator that pattern matches on a product of products and reverses the order of the four components.

```
swap4 : LamR ϵ
swap4 = abs
```

```
((•, •), (•, •))
((v 3, v 2), (v 1, v 0))
```

Note that the inner context Γ^+ in the definition of a λ-abstraction is existentially quantified, i.e. the concrete inner context is hidden from the outside. We will call this a *binding abstraction* or simply *abstraction* and the relation specified by a binder will be called *binder relation*.

2.4.2 Extension view

An extension of a context by a list of new variables, i.e. another context, is simply concatenation of the two lists.

```
_⊳⊳_ : Ctx → Ctx → Ctx
Γ ⊳⊳ ϵ = Γ
Γ ⊳⊳ (Δ ⊳ τ) = Γ ⊳⊳ Δ ⊳ τ
```

The pattern datatype is indexed by the list of variables it abstracts from. We will call this list the *binder context* and the context by which terms are indexed the term context. The binder contexts of sub-patterns are concatenated together, i.e. the variables that will be bound are accumulated in the binder context. Again the order is defined explicitly. A pattern-match alternative in term context Γ is an abstraction, which abstracts from Δ variables introduced by a pattern PatE Δ in a term LamE $(\Gamma \rhd\rhd \Delta)$.

```
data PatE : Ctx → Set where
  •   :                                          PatE (ϵ ⊳ term)
  _,_ : ∀ {Δ₁ Δ₂} → PatE Δ₁ → PatE Δ₂ →          PatE (Δ₁ ⊳⊳ Δ₂)
data LamE (Γ : Ctx) : Set where
  var : Γ ∋ term                                 → LamE Γ
  abs : ∀ {Δ} → PatE Δ → LamE (Γ ⊳⊳ Δ)           → LamE Γ
  app : LamE Γ → LamE Γ                          → LamE Γ
  _,_ : LamE Γ → LamE Γ                          → LamE Γ
swap4E : LamE ϵ
swap4E = abs
           ((•, •), (•, •))
           ((v 3, v 2), (v 1, v 0))
  where
    family : Ctx → Sort → Set
    family Γ type = ⊤
    family Γ term = LamE Γ
    var′ : ∀ {Γ s} → Γ ∋ s → family Γ s
    var′ {Γ} {type} x = tt
    var′ {Γ} {term} x = var x
    open Indices _≟Sort_ var′
```

Note that in this case the context extension is existentially quantified in the abstraction.

2.5 Declarations

Commonly found in languages are declarations or definitions where one wishes to associate names with expressions, with type signatures or other kinds of annotation. Functional programming languages often provide let-expressions. Consider a non-recursive let expression of the following form.

```
let
  x₁ = e₁
  ...
  xₙ = eₙ
in eₙ₊₁
```

The scope of the variables $x_1, \ldots, x_n$ is the expression e_{n+1}. They are considered to be bound simultaneously. The corresponding expressions are in the outer context. To represent the declaration lists in let-expressions adequately it is best to group related things together. To this end a declaration must now be seen as both a value that contains expressions referencing variables and thus needs to be indexed by a term context and also a value that binds a variable and thus needs a binder context. In the extension view this leads to the following Decl and Decls datatypes, which are indexed by two contexts: a binder context which keeps track of the variable introduced in declarations and a term context specifying the context of embedded expressions.

```
mutual
  data Decl : Ctx → Ctx → Set where
    decl  : ∀ {Γ} → Lam Γ → Decl (ϵ ▷ term) Γ
  data Decls : Ctx → Ctx → Set where
    dnil  : ∀ {Γ}          →                                  Decls ϵ Γ
    dcons : ∀ {Γ Δ₁ Δ₂} → Decl Δ₁ Γ → Decls Δ₂ Γ → Decls (Δ₁ ▷▷ Δ₂) Γ
```

Note that the binder context in the dcons case of a declaration list is analogous to the product case for patterns: the concatenation of the sub-binder contexts accumulates the variable bindings from both components. The term context is simply distributed to all embedded expressions. A let expression abstracts variables of a declaration list in a λ-term.

```
data Lam (Γ : Ctx) : Set where
  var  : Γ ∋ term                          → Lam Γ
  abs  : Lam (Γ ▷ term)                    → Lam Γ
  app  : Lam Γ → Lam Γ                     → Lam Γ
  let′ : ∀ {Δ} → Decls Δ Γ → Lam (Γ ▷▷ Δ) → Lam Γ
```

Trying to split the concerns of binding and referencing variables in declarations is possible for example by using a list of variables for the binder and a list of expressions for the right-hand-sides and the body in a single abstraction. Working with this representation is awkward because we need to ensure that we have n+1 expressions for n variables or allow an unsafe representation by not enforcing this constraint. To retrieve the information for one declaration we would need to traverse both lists in this representation.

2.5.1 Recursive scoping

In a mutually recursive let expression the scope of the variables $x_1, \ldots, x_n$ are all the right hand sides $e_1, \ldots, e_n$ and the body e_{n+1}. The well-formed de Bruijn representation of a recursive let is just as easy as a normal one: the embedded expressions are simply in the inner context $\Gamma \rhd\!\rhd \Delta$ of the declaration list.

$$\mathsf{letrec} \;:\; \forall\,\{\Delta\} \to \mathsf{Decls}\;\Delta\;(\Gamma \rhd\!\rhd \Delta) \to \mathsf{Lam}\;(\Gamma \rhd\!\rhd \Delta) \to \mathsf{Lam}\;\Gamma$$

2.5.2 Sequential scoping

As a third variant of let expressions consider the case where a variable scopes only over all subsequent declarations and the body, i.e. x_i scopes over all e_j with $i < j \leqslant n+1$. This kind of scoping can for example be found in the let* form of the Scheme programming language (Sperber [SDF+10]). In the dcons case the binder context again accumulates the variable bindings from the sub-components, but additionally the context of the embedded expression in a declaration is changed.

```
mutual
  data Decl : Ctx → Ctx → Set where
    decl    : ∀ {Γ} → Lam Γ → Decl (ϵ ▷ term) Γ
  data Decls : Ctx → Ctx → Set where
    dnil    : ∀ {Γ}          → Decls ϵ Γ
    dcons   : ∀ {Δ₁ Δ₂ Γ} → Decl Δ₁ Γ → Decls Δ₂ (Γ ▷▷ Δ₁) → Decls (Δ₁ ▷▷ Δ₂) Γ
  data Lam (Γ : Ctx) : Set where
    var     : Γ ∋ term                          → Lam Γ
    abs     : Lam (Γ ▷ term)                    → Lam Γ
    app     : Lam Γ → Lam Γ                     → Lam Γ
    letseq  : ∀ {Δ} → Decls Δ Γ → Lam (Δ ▷▷ Γ) → Lam Γ
```

An important example of sequential scoping are *telescopes* that were introduced by de Bruijn [dB91] to model dependently-typed λ-calculi and are found in the specification and implementation of dependently-typed languages such as Agda or Epigram. A telescope is a list of variables together with type signatures

$$x_1 : \tau_1, \ldots, x_n : \tau_n$$

such that each of the variables $x_1, \ldots, x_n$ scopes over all subsequent type expressions. They are used amongst others to describe the types of dependent functions where the types of parameters may reference previous parameter variables. Weirich et al. [WYS11] describe this example in more detail and provide a representation using their type combinator library.

2.6 Discussion and conclusion

Both, the relation and the extension view for binders exhibit advantages and disadvantages when writing functions over well-formed syntax terms which will be discussed in this section. Here-

after we will consistently take the extension view on datatypes. But both views are equally expressive and it should be possible to carry our definitions and functions over to the relation view. Pouillard and Pottier [PP10] for example take a relation view on nameful syntax with binders and define traversals like weakening, strengthening and capture avoiding substitution with commutations, although they don't provide datatype generic version of them.

2.6.1 Associativity of context extensions

First of all note that the context extension function _⊳⊳_ is associative, which we can easily prove

```
⊳⊳-assoc : ∀ Γ₁ Γ₂ Γ₃ → (Γ₁ ⊳⊳ Γ₂) ⊳⊳ Γ₃ ≡ Γ₁ ⊳⊳ (Γ₂ ⊳⊳ Γ₃)
⊳⊳-assoc Γ₁ Γ₂ ϵ       = refl
⊳⊳-assoc Γ₁ Γ₂ (Γ ⊳ s) = cong (λ Γ → Γ ⊳ s) (⊳⊳-assoc Γ₁ Γ₂ Γ)
```

Unfortunately we also have to reason about this associativity when writing some functions. Consider for example the case of a traversal with a context indexed applicative functor [MP08]. Here F is a context indexed applicative functor, where the App : RawIApplicative F record provides the applicative functions for F. The value vareffect is an applicative effect that we want to execute at every variable binding site for the patterns of type PatE from section 2.4.2. We are only interested in the effect, so the result type ⊤ is used. The _⊛>_ combinator evaluates two computations for their effects and returns the result of the second.

```
module Traverse {F : Ctx → Ctx → Set → Set}
                (App : RawIApplicative F)
                (vareffect : {Γ : Ctx} → F Γ (Γ ⊳ term) ⊤) where
  open RawIApplicative App
  traverseE : ∀ {Γ Δ} → PatE Δ → F Γ (Γ ⊳⊳ Δ) ⊤
  traverseE •                 = vareffect
  traverseE {Γ} (_ , _ {Δ₁} {Δ₂} p q) =
    cast
      (cong (λ Δ → F Γ Δ ⊤) (⊳⊳-assoc Γ Δ₁ Δ₂))
      (traverseE p ⊛> traverseE q)
    where cast : {A B : Set} → A ≡ B → A → B
          cast refl x = x
```

In the case of a product pattern p, q with p : PatE Δ_1 and q : PatE Δ_2 the expected type of the applicative computation on the right-hand side is F Γ (Γ ⊳⊳ (Δ_1 ⊳⊳ Δ_2)) ⊤. The traversals of the sub-patterns evaluate to computations of type F Γ (Γ ⊳⊳ Δ_1) ⊤ and F (Γ ⊳⊳ Δ_1) ((Γ ⊳⊳ Δ_1) ⊳⊳ Δ_2) ⊤ which when taken together form a computation of type F Γ ((Γ ⊳⊳ Δ_1) ⊳⊳ Δ_2) ⊤. Thus we need to rewrite the type of the result value with the associativity proof. In the case of the relation view and the PatR family from section 2.4.1 we get the associativity for free by construction from the associativity of the composition of relations.

```
traverseR : ∀ {Γ Δ} → PatR Γ Δ → F Γ Δ ⊤
traverseR •       = vareffect
traverseR (p, q)  = traverseR p ⊛> traverseR q
```

2.6.2 Commutations during traversals

But also the relation view of binders bears disadvantages. When dealing with other relation-like structures we need to commute them with the binder relation. Consider for example how one would define a simultaneous renaming of all variables contained in a term. A renaming of variables is simply a function mapping variables from one context Γ to variables in another context Δ

```
infix 3 _⇒_
_⇒_ : Ctx → Ctx → Set
Γ ⇒ Δ  =  ∀ {s} → Γ ∋ s → Δ ∋ s
```

To apply a renaming to a term we traverse it to the variable positions and apply the renaming function. In the case of an abstraction we have to extend the renaming function to account for the newly introduced variables. For a single variable this is done by the following $\_\uparrow$ function.

```
_↑ : ∀ {Γ Δ t} → Γ ⇒ Δ → Γ ▷ t ⇒ Δ ▷ t
(f ↑) vz      = vz
(f ↑) (vs y)  = vs (f y)
```

For the renaming of a λ-abstraction consider the case of a pattern $\mathsf{p} : \mathsf{PatR}\ \Gamma_1\ \Gamma_2$ and a renaming $\mathsf{f} : \Gamma_1 \Rightarrow \Delta_1$ for the outer context. We need to construct a new renaming $\mathsf{f}' : \Gamma_2 \Rightarrow \Delta_2$ for the inner context and moreover we also need a new pattern $\mathsf{p}' : \mathsf{PatR}\ \Delta_1\ \Delta_2$ for the renamed contexts, i.e. we need f' and p' such that the following diagram commutes:

$$\begin{array}{ccc} \Gamma_1 & \xRightarrow{\mathsf{f}} & \Delta_1 \\ {\scriptstyle \mathsf{p}}\big\downarrow & & \big\downarrow{\scriptstyle \mathsf{p}'} \\ \Gamma_2 & \xRightarrow{\mathsf{f}'} & \Delta_2 \end{array}$$

We know that there exists some context Γ such that $\Gamma_2 \equiv \Gamma_1 \rhd\!\rhd \Gamma$. Furthermore the equality $\Delta_2 \equiv \Delta_1 \rhd\!\rhd \Gamma$ will hold for Δ_2. To obtain the commuted renaming and pattern we traverse p. At every variable binding site we extend the renaming. The new pattern is simply created by using the same constructors, but which will result in a type of the inductive family of patterns with different contexts. The function commute implements[1] this idea.

```
commute : ∀ {Γ1 Γ2 Δ1} → Γ1 ⇒ Δ1 → PatR Γ1 Γ2 →
          ∃ λ Δ2 → PatR Δ1 Δ2 × Γ2 ⇒ Δ2
commute f •  =  _, •, (λ x → (f ↑) x)
```

[1] It is more convenient to implement commute using continuation-passing style

```
commute f (p, q) with commute f p
commute f (p, q) | _, p′, f′ with commute f′ q
commute f (p, q) | _, p′, f′ | _, q′, f′′ = _, (p′, q′), λ x → f′′ x
```

And we can finally define renameLamR that traverses a LamR value and commutes the contained patterns with the renaming function.

```
renameLamR : ∀ {Γ Δ} → Γ ⇒ Δ → LamR Γ → LamR Δ
renameLamR f (var x)              = var (f x)
renameLamR f (abs p x) with commute f p
renameLamR f (abs p x) | _, p′, f′ = abs p′ (renameLamR f′ x)
renameLamR f (app x y)            = app (renameLamR f x) (renameLamR f y)
renameLamR f (x, y)               = renameLamR f x, renameLamR f y
```

For the lifting of a function in the extension view we do not need to inspect the pattern, it is enough to know the binder context. We can then lift the function by iterating the single variable lifting.

```
_↑⋆_ : {Γ Δ : Ctx} → Γ ⇒ Δ → (E : Ctx) → Γ ▷▷ E ⇒ Δ ▷▷ E
f ↑⋆ ϵ       = f
f ↑⋆ (Γ ▷ τ) = (f ↑⋆ Γ) ↑
renameLamE : ∀ {Γ Δ} → Γ ⇒ Δ → LamE Γ → LamE Δ
renameLamE f (var x)          = var (f x)
renameLamE f (abs {Δ} p x)    = abs p (renameLamE (f ↑⋆ Δ) x)
renameLamE f (app x y)        = app (renameLamE f x) (renameLamE f y)
renameLamE f (x, y)           = renameLamE f x, renameLamE f y
```

Similar operations like simultaneous or single variable substitutions require similar commutations.

2.6.3 Hidden term contexts

In chapter 4 we will develop a universe for syntax representations and give two interpretations, one based on well-scoped de Bruijn indices and one on parametric higher-order abstract syntax and we will develop a generic conversion function from the latter to the former. In the parametric HOAS representation the term context will be handled entirely at the meta-level and will thus not be available to definitions of binder relations. To still be able to use the relation view, one could first let binders relate the empty context ϵ to some other context Δ and then shift this to Γ and $\Gamma \rhd\rhd \Delta$ once the term context Γ is fixed in the conversion. The extension view does not suffer from this problem, because the binder context is constructed independently of any term context.

3 Parametric higher-order abstract syntax

Higher-order abstract syntax uses the meta-language function space to model the binding forms of an object language. A notable advantage over de Bruijn representations is that terms can be directly written with names for variables. At the same time it also ensures that object language terms are well-scoped and in this sense exhibits the same advantage as the well-scoped de Bruijn representations over first-order abstract syntax.

Recall the definition of a HOAS representation of the untyped lambda calculus from the introduction.

```
data Lam : Set where
  abs : (Lam → Lam) → Lam
  app : Lam → Lam → Lam
```

A major benefit of higher-order syntax representation is that they inherit capture-avoiding substitution operations for free from the meta-level: substituting a term for a variable amounts to applying the function representing the variable to the term. As an example consider the following reduce function, that performs a β-reduction if it finds a β-redex at the outermost level. The function from the abstraction is applied to the right-hand side term of the application.

```
reduce : Lam → Lam
reduce (app (abs f) a) = f a
reduce x               = x
```

It is also possible to model multiple *syntactic sorts* using HOAS representations. The following datatype definitions encode the syntax of System F.

```
data Type : Set where
  for  : (Type → Type) → Type
  arr  : Type → Type → Type
data Term : Set where
  abs  : Type → (Term → Term) → Term
  app  : Term → Term → Term
  τabs : (Type → Term) → Term
  τapp : Term → Type → Term
```

Different syntactic sorts are modeled by distinct representation types and variable binding is again represented using the function space, where the domain of a function corresponds to the syntactic sort of a variable. The typesystem of the meta-language ensures that variables of a sort only appear at positions where that sort is expected.

While the above HOAS encodings clearly represents abstract syntax terms, they come with their own set of drawbacks which make them awkward or inconvenient to use in some situations. Modern functional programming languages allow users to define functions by pattern matching on the arguments. This can also be done with values of the HOAS representation types as shown in the definition of the reduce function above. But clearly, wrapping reduce in the abs constructor

```
exotic : Lam
exotic = abs reduce
```

gives us a value of type Lam. Those values are called *exotic terms* because they do not correspond to syntax terms with variable binding. For a Lam value to represent a syntax term we need to ensure that all functions contained in an abs do not case-analyse their argument, i.e. they are parametric functions. Restricting oneself to parametric functions is desirable for another reason: writing catamorphisms over datatypes becomes easier. Paterson and Meijer and Hutton show how to write catamorphisms over general datatypes with contravariant recursive occurrences, but they require at the same time the definition of an anamorphism which forms an inverse for the catamorphism. Building upon that work, Fegaras and Sheard [FS96] present a way to construct catamorphisms, without the requirement to specify a corresponding anamorphism, if all embedded function types are restricted to parametric functions.

Parametricity of functions can be guaranteed statically by using type systems. Fegaras and Sheard [FS96] use type annotations to distinguish between parametric and non-parametric functions and show how to write folds for types with embedded parametric functions types. Another elegant way was shown by Washburn and Weirich [WW03]. They use parametric polymorphism found in type systems of languages such as Haskell, to ensure parametricity of functions used in HOAS representation.

Another disadvantage of HOAS representations is that variables are only represented implicitly. Consider for example a shrinking reduction optimization for the lambda calculus where a β-redex is reduced in a term iff the bound variable is used at most once. Unfortunately the HOAS encoding does not allow this intensional analysis of the body of an abstraction.

To overcome this issue of HOAS, researchers have presented conversions to and from first-order representations such as representations using de Bruijn indices. Atkey et al. [ALY09] specifically address this conversion in the context of domain-specific languages and show how to convert a parametric well-typed HOAS representation of the simply-typed λ-calculus to a well-typed de Bruijn representation in Haskell.

A practical example of the use of such a conversion is the Haskell accelerate library developed by Chakravarty et al. [CKL$^+$11] that implements an embedded language for array computations. The user can write programs using a convenient HOAS interface which are converted to a de Bruijn representation for code generation. Subsequently they can be run on GPUs.

3.1 Church encodings

In this section we will introduce the parametric higher-order abstract syntax (HOAS) representation proposed by Atkey et al. [ALY09]. Their representation is based on a Church encoding

for the term formers of the syntax of a language syntax comparable to a Church encoding of inductive datatypes. Consider the following System F type that is a Church encoding of the naturals

$$C_{\text{nat}} = \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha.$$

A value of C_{nat} is a function that expects some type α and one value per constructor of the naturals, a value of type α for the zero and a function of type $\alpha \to \alpha$ for the successor, and will construct another value of type α. Due to parametricity the C_{nat} value can only use the given constructors. This means that a value of type C_{nat} represents a natural number by its fold operator. The type it gets is the carrier of an algebra and the arguments are the algebra functions for that carrier.

Similarly, we can read the System F type below as a Church encoding of the untyped λ-calculus.

$$C_\lambda = \forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha.$$

Again, a value of this type gets an abstract type α and functions for each of the term formers, one for λ-abstractions and one for applications. In HOAS representations variables are handled by the function space at the meta-level, so there is no constructor for them. The noteworthy difference between C_λ and Church encodings of inductive datatypes is the negative appearance of α in the type $(\alpha \to \alpha) \to \alpha$ of the constructor of λ-abstractions. This is exactly where the meta-level binding is introduced: a λ-abstraction introduces one term variable that is used in the definition of another term. As for C_{nat} the values of C_λ encode a fold operator. This can be used to define computations over parametric HOAS terms by providing a suitable algebra.

In Agda we write Church encodings more practically by grouping the algebra functions in a parameterized record, so that the field names can be used for the constructors. The LamAlgebra record collects algebra functions for an algebra of the untyped λ-calculus. The record also contains the syntax macro `syntax lam (λ x → e) = ƛ x ⇒ e` that reduces the syntactic overhead introduced by the meta-level λ-abstractions.

The C_λ type can then equivalenty be defined as a function that maps an abstract algebra to a value of its carrier.

```
record LamAlgebra (A : Set) : Set where
  infixl 10 _·_
  field
    lam : (A → A) → A
    _·_ : A → A → A
  syntax lam (λ x → e) = ƛ x ⇒ e

Cλ : Set₁
Cλ = {A : Set} (alg : LamAlgebra A) → A
```

To write values of C_λ in Agda it is most useful to abstract from a LamAlgebra using a parameterized module. Inside the module we can simply use the fields of the abstract record. We give terms for the I, K and S combinators where I is defined without the syntax macro, but K and S are defined with it. The num function produces for every natural number the associated Church numeral.

```
module Examples {Exp : Set} (alg : LamAlgebra Exp) where
  open LamAlgebra alg
  I = lam λ x → x
  K = ƛ x ⇒ ƛ y ⇒ x
  S = ƛ f ⇒ ƛ g ⇒ ƛ x ⇒ f · x · (g · x)
  num : ℕ → Exp
  num n = ƛ s ⇒ ƛ z ⇒ body s z n
    where body : Exp → Exp → ℕ → Exp
          body s z zero    = z
          body s z (suc n) = s · body s z n
```

Writing computations as folds over parametric HOAS terms is done by providing an instance of LamAlgebra. The sizeAlg algebra below calculates the size of C_λ terms. The algebra function for applications simply sums the sizes of the subterms and increments the result. For abstraction we get a function f that expects the value that will replace occurrences of the bound variable. We give every variable occurrence the size 1 and also add 1 for the λ-abstraction itself. The size of a C_λ term can then simply be calculated by applying it to sizeAlg.

```
module SizeExample where
  open Examples
  sizeAlg : LamAlgebra ℕ
  sizeAlg = record {lam = λ f → 1 + f 1; _·_ = λ m n → m + n + 1}
  size : C_λ → ℕ
  size t = t sizeAlg
```

3.2 Conversion to de Bruijn terms

In this section a conversion function from the parametric HOAS representation of the untyped λ-calculus to a well-scoped de Bruijn representation is developed. The de Bruijn representation below will be used. Again we have a single sort denoted by tm.

```
data Lam (Γ : Ctx) : Set where
  var : (i : Γ ∋ tm)          → Lam Γ
  abs : (t : Lam (Γ ▷ tm))    → Lam Γ
  app : (s t : Lam Γ)         → Lam Γ
```

The conversion will be written as a fold over HOAS terms by constructing a suitable LamAlgebra. Instead of using the Lam syntax type directly the following Exp type is used as the carrier of the conversion algebra. The idea is that applying a value f of type Exp to a context Γ will construct a term which is closed in Γ. Abstracting from the context will allow us to instantiate a term in the correct context.

```
Exp : Set
Exp = (Γ : Ctx) → Lam Γ
```

```
makeExpVar : (Γ : Ctx) → Exp
makeExpVar Γ Δ = ...
convAlg : LamAlgebra Exp
convAlg = record
            {lam = λ f   Γ → abs (f (makeExpVar Γ) (Γ ▷ tm))
            ; _·_ = λ a b Γ → app (a Γ) (b Γ)
            }
```

To construct a de Bruijn term of an application we can simply pass the current context through to the subterms. This will produce the left and right hand side de Bruijn terms of the application which are then applied to the de Bruijn term application constructor app. In the case of an object langugage λ-abstraction the body is represented by a meta-language function f. The function makeExpVar will create an Exp value representing the bound variable which is passed to f. The body of the λ-abstraction will then be constructed in a bigger context: Γ extended by a new variable for the sort tm.

The value produced by makeExpVar Γ will eventually be applied to some inner context Δ, which is a strict super-context of Γ because recursing under λ-abstractions adds variables to the context and variables are never removed. Because of this property the introduced variable has an index in Δ. We would like to create this index by using a proof of the sub-context relation between Γ and Δ, but as Atkey et al. [ALY09] point out, Agda's type system unfortunately does not provide us with enough information to obtain such a proof. Thus a completely well-typed implementation of makeExpVar is not possible. However, Atkey [Atk09] provide a meta-theoretical proof of this property for an encoding in System F by using parametricity results. We will define an almost well-typed makeExpVar function at the end of the following section.

Using the conversion algebra with the K and S combinator examples produces the following results. We make use of a function v : {Γ : Ctx} (n : ℕ) {_ : ...} → Lam Γ as a convenient way to write de Bruijn indices using natural numbers, i.e. v 2 stands for var (vs (vs vz)).

```
K ϵ ≡ abs (abs (v 1))
S ϵ ≡ abs (abs (abs (app (app (v 2) (v 0)) (app (v 1) (v 0)))))
```

3.3 Prefix relation

To escape from the situation we will develop a predicate for the sub-context relation and implement a deciding function for it. In the positive case we get a proof term for the relation and in the negative case we *postulate* the result. As this case will not occur the computation will not get stuck during the conversion.

In this section we will examine the general case of contexts of arbitrary sorts and not restrict ourselves to a single sort, because in later chapters we will deal with syntax representations of languages with multiple sorts. We assume that we operate on a language with syntactic sorts given by Sort : Set and that we have a function _$\stackrel{?}{=}$_ which decides equality on Sort. The result of _$\stackrel{?}{=}$_ is a value of the Dec type, that carries either a proof of the predicate, or a proof of its negation.

```
data Dec (P : Set) : Set where
  yes : (p : P) → Dec P
  no : (¬p : ¬ P) → Dec P

module Prefix {Sort : Set} (_≟S_ : ∀ (x y : Sort) → Dec (x ≡ y)) where
  open Context Sort
```

For the conversion we will need to relate a context Γ outside of an abstraction to contexts inside of the abstraction. Consider the case of a single variable abstraction where a new variable for the sort s is introduced to an outer context Γ. As we only append new variables to a context, clearly the context $\Gamma \rhd s$ will be a prefix of any context Δ inside the abstraction. The following predicate $\_\sqsubseteq\_$ encodes the prefix relation between contexts. Furthermore this predicate is decidable and $\_\sqsubseteq?\_$ denotes a deciding function. The implementation of the deciding function is omitted.

```
  infix 4 _⊑_
  data _⊑_ (Γ : Ctx) : Ctx → Set where
    refl : Γ ⊑ Γ
    _▷_ : {Δ : Ctx} (Γ⊑Δ : Γ ⊑ Δ) (s : Sort) → Γ ⊑ Δ ▷ s
  infix 4 _⊑?_
  _⊑?_ : Decidable _⊑_
```

Given a proof for $\Gamma \rhd s \sqsubseteq \Delta$ we can create an index in Δ for the variable introduced to Γ.

```
  makeVar : {Γ Δ : Ctx} {s : Sort} → Γ ▷ s ⊑ Δ → Δ ∋ s
  makeVar refl        = vz
  makeVar (Γ⊑Δ ▷ _)  = vs (makeVar Γ⊑Δ)
```

We can now finish the definition of the makeExpVar function from the last section. As mentioned before we have to postulate the result in the negative case.

```
makeExpVar : (Γ : Ctx) → Exp
makeExpVar Γ Δ with Γ ▷ tm ⊑? Δ
makeExpVar Γ Δ | yes Γ▷tm⊑Δ = var (makeVar Γ▷tm⊑Δ)
makeExpVar Γ Δ | no ¬p       = whatever
  where postulate whatever : _
```

4 Universes of syntax types

We begin this chapter with a small universe of regular syntax types with unary scoping of a single sort and arrive at a universe of mutually recursive syntax types with rich binding forms by the end of the chapter. For the universes we will provide two interpretations, one based on well-formed de Bruijn terms and one on parametric higher-order abstract syntax. Our first generic function that we will implement for the universes in this chapter is a generic conversion from the parametric HOAS interpretation to the de Bruijn interpretation.

4.1 Regular syntax types

As mentioned above, in this section we are interested in abstract syntax for a language of a single sort that can be represented by a single directly recursive datatype. We begin by looking at a well-scoped de Bruijn representation of the untyped λ-calculus and look at the modeling of this representation as the fixed point of a functor.

Well-scoped de Bruijn representations of a single sort have type $\mathsf{Ctx} \to \mathsf{Set}$ and we are interested in functors between syntax types of this type. A functor is a function for which we can define a functorial mapping. We will use $\_\rightsquigarrow\_$ to denote morphisms between syntax types, i.e. a function that is natural in the context.

```
Syntax : Set₁
Syntax = Ctx → Set

_⇝_ : Syntax → Syntax → Set
Φ ⇝ Ψ = {Γ : Ctx} → Φ Γ → Ψ Γ
```

For the well-scoped untyped λ-calculus representation with the single sort tm

```
data Lam : Syntax where
  var : {Γ : Ctx} → Γ ∋ tm          → Lam Γ
  abs : {Γ : Ctx} → Lam (Γ ▷ tm)    → Lam Γ
  app : {Γ : Ctx} → Lam Γ → Lam Γ   → Lam Γ
```

we get the following pattern functor, by abstracting from the recursive positions with a new parameter Φ.

```
data LamF (Φ : Syntax) : Syntax where
  var : {Γ : Ctx} → Γ ∋ tm       → LamF Φ Γ
  abs : {Γ : Ctx} → Φ (Γ ▷ tm)   → LamF Φ Γ
  app : {Γ : Ctx} → Φ Γ → Φ Γ    → LamF Φ Γ
```

It is straigthforward to verify that LamF indeed is a functor between syntax types by providing a functorial mapping

```
map-lamf : {Φ Ψ : Syntax} → (Φ ⇝ Ψ) → LamF Φ ⇝ LamF Ψ
map-lamf f (var x)   = var x
map-lamf f (abs s)   = abs (f s)
map-lamf f (app s t) = app (f s) (f t)
```

Furthermore we can define LamF-algebra and LamF-coalgebra functions with Lam as carrier.

```
to : LamF Lam ⇝ Lam
to (var x)   = var x
to (abs s)   = abs s
to (app s t) = app s t

from : Lam ⇝ LamF Lam
from (var x)   = var x
from (abs s)   = abs s
from (app s t) = app s t
```

It is easy to see that to and from form inverses of each other and so Lam and LamF Lam are isomorphic. In that sense Lam can be seen as a fixed point of the functor LamF and LamF Lam describes a one-level unfolding of the fixed point.

4.1.1 A universe of syntax functors

Our universe will describe the structure of pattern functors like LamF. We define a small set of codes that we will interpret as primitive functors and functor combinators from which we can systematically build functors for regular syntax types. We closely follow the structure exhibited by data declarations of pattern functors and define two types of codes: ProdCode, that describes the structure of a single constructor declaration and DataCode, that describes the structure of a complete data declaration.

Unit constants are described by the one code. The rec code marks a position that references the parameter of the functor and _⊗_ represents the product structure of a constructor. A code abs c denotes an abstraction over one new term variable in all the parameter references contained in the code c. The term constructor for variables will be treated differently as will be described below. A DataCode is a list of ProdCodes with the intention that a DataCode is a sum of the ProdCodes in the list and as such denotes choice between different constructors.

```
data ProdCode : Set where
  one  : ProdCode
  rec  : ProdCode
  _⊗_  : (c d : ProdCode) → ProdCode
  abs  : (c : ProdCode) → ProdCode

DataCode : Set
DataCode = List ProdCode
```

The well-scoped representation of the untyped λ-calculus has the following DataCode describing the λ-abstraction and application constructors

```
LamD : DataCode
LamD = abs rec :: rec ⊗ rec :: [ ]
```

We complete the definition of our universe by interpreting codes as functors. We use a parameterized module Interpretation to add a parameter Φ : Syntax to all the interpretations. The parameter is referenced in the interpretation of the rec code. Note furthermore the context extension in the recursive interpretation of an abs c code.

```
module Interpretation (Φ : Syntax) where
  data P⟦_⟧ : ProdCode → Syntax where
    one  : ∀ {Γ}       →                          P⟦ one   ⟧ Γ
    rec  : ∀ {Γ}       → Φ Γ                   → P⟦ rec   ⟧ Γ
    _⊗_  : ∀ {Γ c d}   → P⟦ c ⟧ Γ → P⟦ d ⟧ Γ   → P⟦ c ⊗ d ⟧ Γ
    abs  : ∀ {Γ c}     → P⟦ c ⟧ (Γ ▷ tm)       → P⟦ abs c ⟧ Γ
```

For DataCodes we provide two interpretations. The inductive family D⟦_⟧ interprets the sum structure of a datatype. There are two possibilities to construct the interpretation of a DataCode: either we have an interpretation of the ProdCode at the head, or we have an interpretation of the tail. The inductive family ⟦_⟧ adds another choice for variables to the D⟦_⟧ interpretation of DataCodes.

```
  data D⟦_⟧ : DataCode → Syntax where
    top  : ∀ {Γ c cs} → P⟦ c  ⟧ Γ → D⟦ c :: cs ⟧ Γ
    pop  : ∀ {Γ c cs} → D⟦ cs ⟧ Γ → D⟦ c :: cs ⟧ Γ
  data ⟦_⟧ (c : DataCode) : Syntax where
    ⟨_⟩ : ∀ {Γ} → D⟦ c ⟧ Γ → ⟦ c ⟧ Γ
    var : ∀ {Γ} → Γ ∋ tm   → ⟦ c ⟧ Γ
```

To show that the interpretation of LamD really describes a pattern functor for Lam we provide conversions from Lam to the one-level unfolding and back. Note that we open the Interpretation module with Lam as an argument, so the parameter of the interpreted functors is already set to Lam.

```
open Interpretation Lam

from : Lam ⇝ ⟦ LamD ⟧
from (var i)     = (var i)
from (abs t)     = ⟨ top       (abs (rec t))     ⟩
from (app s t)   = ⟨ pop (top  (rec s ⊗ rec t)) ⟩

to : ⟦ LamD ⟧ ⇝ Lam
to (var i)                      = var i
to ⟨ top       (abs (rec t))    ⟩ = abs t
```

```
to ⟨ pop (top  (rec s ⊗ rec t)) ⟩ = app s t
to ⟨ pop (pop ())               ⟩
```

The record type Regular joins a syntax type with a code representing its pattern functor and conversions between the syntax type and the one-level unfolding of the fixed point.

```
record Regular : Set₁ where
  field
    syntaxType : Syntax
    code       : DataCode
  open Interpretation syntaxType
  field
    from       : syntaxType ⇝ ⟦ code ⟧
    to         : ⟦ code ⟧ ⇝ syntaxType
```

4.1.2 Parametric HOAS interpretation of regular syntax types

We begin the development of a generic conversion from parametric HOAS to well-formed de Bruijn representations. To this end we first specify the Church encodings of the HOAS representations in terms of DataCodes and later define one generic algebra on the structure given by DataCodes that performs the conversion.

Church encodings

The Church encodings are abstracted over algebra carrier types for which we will use a module parameterized by A : Set. The interpretation functions inside this module will calculate the type of algebras with carrier A. As in the non-generic conversion example we define an algebra for a parametric HOAS representation by giving one algebra function per non-variable constructor. Generally an algebra function with carrier A has the form F A → A where F is a functor. The function Arg computes the functor F we want to use in the argument from the ProdCode of a constructor.

```
module HOAS (A : Set) where
  Arg : ProdCode → Set
  Arg one       = ⊤
  Arg rec       = A
  Arg (c ⊗ d)   = Arg c × Arg d
  Arg (abs c)   = A → Arg c
```

In the case of an abstraction, a function type with one argument of type A is introduced providing us with one meta-language variable. As a variation of Arg we define the function AlgProd which computes the type of an algebra function in a curried style. It additionally takes a result type R which we allow to be different from the parameter A of the functor, i.e. we calculate F A → R in a curried style. Interesting is the case for products: the resulting algebra

function should first take the arguments from the left code and then the arguments from the right code. To achieve this we pass AlgProd c_2 R as the result type to AlgProd c_1.

Note that in case of an abstraction we still need to use Arg to calculate the result type of the function, i.e. for the body of the abstraction.

```
AlgProd : ProdCode → Set → Set
AlgProd one        R = R
AlgProd rec        R = A → R
AlgProd (c₁ ⊗ c₂)  R = AlgProd c₁ (AlgProd c₂ R)
AlgProd (abs c)    R = (A → Arg c) → R
```

An algebra for a datatype is a product of the algebra functions of its constructors, which is computed by the AlgData function.

```
AlgData : DataCode → Set → Set
AlgData [ ]        R = ⊤
AlgData (c :: cs)  R = AlgProd c R × AlgData cs R
```

The AlgProd and AlgData functions are also functors in the result type. The function map-prod respectively map-data encodes a functorial mapping for AlgProd respectively AlgData, that will be used in the conversion algebra.

```
map-prod : (c : ProdCode) {R S : Set} →
           (R → S) → AlgProd c R → AlgProd c S
map-prod one     f x = f x
map-prod rec     f x = f ∘ x
map-prod (c ⊗ d) f x = map-prod c (map-prod d f) x
map-prod (abs c) f x = f ∘ x

map-data : (d : DataCode) {R S : Set} →
           (R → S) → AlgData d R → AlgData d S
map-data [ ]       f x      = tt
map-data (c :: cs) f (x, y) = map-prod c f x, map-data cs f y
```

The AlgebraF function computes the algebra type for a constructor given by a ProdCode. Finally Algebra computes the product algebra for a whole datatype. Both simply use the carrier type as the result type.

```
AlgebraF : ProdCode → Set
AlgebraF c = AlgProd c A

Algebra : DataCode → Set
Algebra c = AlgData c A
```

Conversion algebra

As in the non-generic example, we specify a datatype Exp as the carrier of the conversion algebra which abstracts from the context a de Bruijn syntax term is instantiated in. Furthermore, we specify related datatypes ExpP and ExpD where our regular syntax type is wrapped in interpretations of ProdCodes or DataCodes. The intention is that ExpP and ExpD represent partial constructions of the one-level unfolding of the syntax type.

```
module Conversion (regular : Regular) where
  open Regular regular
  open Interpretation syntaxType

  Exp : Set
  Exp = (Γ : Ctx) → syntaxType Γ

  ExpP : ProdCode → Set
  ExpP c = (Γ : Ctx) → P⟦ c ⟧ Γ

  ExpD : DataCode → Set
  ExpD c = (Γ : Ctx) → D⟦ c ⟧ Γ

  open HOAS Exp
```

conv-arg takes a value of the HOAS interpretation Arg c of a ProdCode c and produces an Exp value for the well-scoped de Bruijn representation. The context Γ is propagated to the recursive positions. At the recursive positions we have Exp values which are instantiated in the given context. The case of an unary abstraction is handled by the conv-arg-abs helper function. It creates a de Bruijn index for the variable and applies the function representing the binding to it. The body is then created in a context extended by a new variable.

```
  mutual
    conv-arg : (c : ProdCode) → Arg c → ExpP c
    conv-arg one       tt       Γ = one
    conv-arg rec       x        Γ = rec (x Γ)
    conv-arg (c ⊗ d) (x , y)    Γ = conv-arg c x Γ ⊗ conv-arg d y Γ
    conv-arg (abs c) x          Γ = conv-arg-abs c x Γ

    conv-arg-abs : (c : ProdCode) → (Exp → Arg c) → ExpP (abs c)
    conv-arg-abs c x Γ = abs (conv-arg c (x (makeExpVar Γ)) (Γ ▷ tm))
```

The same functionality for the curried variant is taken care of the conv-prod function. The functorial mappings of AlgProd c are used to apply the constructors of the de Bruijn structure representation types. conv-data recurses over a DataCode and applies corresponding top or pop constructors to the result values, to select the correct summand for a ProdCode.

```
  conv-prod : (c : ProdCode) → AlgProd c (ExpP c)
  conv-prod one       = λ Γ → one
  conv-prod rec       = λ x Γ → rec (x Γ)
  conv-prod (c ⊗ d)   = map-prod c (λ x →
```

```
                                map-prod d
                                  (λ y Γ → x Γ ⊗ y Γ)
                                (conv-prod d))
                                (conv-prod c)
conv-prod (abs c)    =  conv-arg-abs c

conv-data : (c : DataCode) → AlgData c (ExpD c)
conv-data [ ]        =  tt
conv-data (c :: cs)  =  map-prod c  (_∘_ top) (conv-prod c),
                        map-data cs (_∘_ pop) (conv-data cs)
```

Finally, conversionAlgebra takes care of a one-level folding of the fixed point at the end.

```
conversionAlgebra : Algebra code
conversionAlgebra =  map-data code
                       (λ x Γ → to ⟨ x Γ ⟩)
                       (conv-data code)
```

4.1.3 Example: Regular tree types

We will exemplify the usage of the conversion algebra by working through a small example language: *μ-expressions* for the representation of *regular tree types*. Regular tree types are the class of datatypes closed under empty, unit, products, coproducts and least fixed points. Morris et al. [MAM06] give a well-scoped de Bruijn representation of μ-expressions and use this representation as the type of codes for a universe construction of regular tree types and show how to program generically over this universe. We will define a parametric higher-order abstract syntax interface for μ-expressions and instantiate the generic conversion to well-scoped de Bruijn representations to this particular case.

Let V be a set of variables. Then the language L of μ-expressions is defined by the following grammar

$$L ::= 0 \mid 1 \mid \mu V.L \mid L + L \mid L \times L \mid V \quad .$$

The grammar specifies polynomial expressions in the set of variables V extended by an operator μ that represents the least fixed point of a μ-expression and abstracts over one variable. To add another binding form to the language we extend it by *let* expressions which bind exactly one variable. They are introduced by the following additional production

$$L ::= \text{let } V = L \text{ in } L \quad .$$

The variable is only considered bound in the right subexpression.

The Reg datatype gives a well-scoped de Bruijn representation for μ-expressions. The zero, one, $\_\boxplus\_$, $\_\boxtimes\_$ and var constructor represent polynomial expressions. The mu constructor abstracts over one variable and the let′ constructor takes two subexpression, one in the outer context and one in the outer context extended by one variable, that is considered bound to the first subexpression.

```
infixr 4 _⊞_
infixr 6 _⊠_
data Reg (Γ : Ctx) : Set where
  zero : Reg Γ
  one  : Reg Γ
  _⊞_  : (r s : Reg Γ) → Reg Γ
  _⊠_  : (r s : Reg Γ) → Reg Γ
  mu   : (f   : Reg (Γ ▷ tm)) → Reg Γ
  let′ : (a   : Reg Γ) (f : Reg (Γ ▷ tm)) → Reg Γ
  var  : (i   : Γ ∋ tm) → Reg Γ
```

For the Reg datatype we get the following ProdCodes for the constructors and a DataCode for the whole datatype. Conversion functions from and to the well-scoped de Bruijn interpretation of the DataCode are straightforward to write.

```
reg-zero = one
reg-one  = one
reg-⊞    = rec ⊗ rec
reg-⊠    = rec ⊗ rec
reg-mu   = abs rec
reg-let  = rec ⊗ abs rec
reg-code = reg-zero :: reg-one :: reg-⊞
         :: reg-⊠   :: reg-mu  :: reg-let
         :: []
```

The parametric HOAS interface is given by an algebra type which is again grouped in a record type. The types of the record fields are calculated by the AlgebraF function from the ProdCodes. The normalized types are also shown and subsequently we will only provide those. As convenience we specify syntax macros to hide the overhead of meta-level λ-abstractions in manually written HOAS terms.

```
record RegAlgebra (A : Set) : Set where
  field
    0    : AlgebraF A reg-zero   -- A
    1    : AlgebraF A reg-one    -- A
    _+_  : AlgebraF A reg-⊞      -- A → A → A
    _×_  : AlgebraF A reg-⊠      -- A → A → A
    mu   : AlgebraF A reg-mu     -- (A → A) → A
    let′ : AlgebraF A reg-let    -- A → (A → A) → A
  infixr 4 _+_
  infixr 6 _×_
  syntax mu   (λ x → e) = ν x ⇒ e
  syntax let′ e (λ x → f) = let′ x ≔ e in′ f
```

The Algebra type for the DataCode of the μ-expressions and the RepAlgebra type are trivially isomorphic. Converting from the former to the latter is done by convert.

```
convert : {A : Set} → Algebra A reg-code → RegAlgebra A
convert (a, b, c, d, e, f, tt) =
  record {0    = a; 1    = b
         ; _+_ = c; _×_  = d
         ; mu  = e; let′ = f
         }
```

We give some examples using the HOAS interface. It is straightforward to define the representation nat for natural numbers, or tree for binary trees.

```
module Examples {Exp : Set} (alg : RegAlgebra Exp) where
  open RegAlgebra alg
  nat  = ν n ⇒ 1 + n
  tree = ν t ⇒ 1 + t × t
```

Parameterized datatypes are encoded with free variables for the parameters, which amounts to using the function space in the HOAS representation. A parameterized list representation is given by list, and a binary tree where all nodes are annotated by a parameter type is given by bintree. Substituting a free variable is done by function application as can be seen in the definition of rosetree, a representation for finitely branching trees, also called rose trees.

```
list : Exp → Exp
list a = ν as ⇒ 1 + a × as

bintree : Exp → Exp
bintree a = ν b ⇒ a + b × a × b

rosetree : Exp
rosetree = ν t ⇒ list t
```

A representation for binary trees annotated by naturals can be obtained by applying bintree to nat. As an alternative we can first bind nat to a new variable using a let declaration and then apply bintree to that variable.

```
natbintree  = bintree nat
natbintree′ = let′ n ≔ nat in′ bintree n
```

Here are the resulting well-scoped de Bruijn terms for the closed HOAS terms. Note specifically that the converted term for nat appears twice in the converted term for natbintree, because it was substituted for the two occurrences of the free variable. In natbintree′ the let-bound variable appears twice and the term for nat just once.

```
nat ϵ      ≡ mu (one ⊞ v 0)
tree ϵ     ≡ mu (one ⊞ v 0 ⊠ v 0)
```

```
rosetree ϵ    ≡ mu (mu (one ⨥ v 1 ⊠ v 0))
natbintree ϵ  ≡ mu (mu (one ⨥ v 0) ⨥ v 0 ⊠ mu (one ⨥ v 0) ⊠ v 0)
natbintree′ ϵ ≡ let′ (mu (one ⨥ v 0)) (mu (v 1 ⨥ v 0 ⊠ v 1 ⊠ v 0))
```

4.2 Families of syntax types

In the previous section we looked at a universe of directly recursive regular syntax types with unary scoping. We will now generalize this to families of arbitrary mutually recursive syntax types but we will retain the unary scoping restriction. In this universe we can model languages with multiple syntactic sorts like term and type expressions. Not all syntactic sorts need to have variables. We can have for example type expressions without variables.

4.2.1 Universe codes

Our universe is parameterized over two index sets. I_u contains the indices for the syntactic sorts without variables and I_v the indices for sorts with variables. Their sum I_x is the index set for the family of all syntax types of a language.

```
module Universe (Iu Iv : Set) where
  data Ix : Set where
    inju : Iu → Ix
    injv : Iv → Ix
```

The objects we will look at in this section are families of syntax types indexed by the set I_x, i.e. they are of the following Family type. The morphisms of the syntax type families in Family are functions that are quantified over both the index set of the family and the term context. Instead of viewing a single syntax type as the fixed point of a functor, we will view complete Familys as fixed point of functors of the correct type.

```
Family : Set1
Family = Ix → Ctx → Set
```

The codes for our universe are now given by three types: ProdCode, QuantCode and DataCode. The ProdCode type again describes the product structure of a syntax type constructor. At a recursive position a specific type from the family is referenced, which is designated by the index argument of rec. An abstraction abs now additionally takes an index s : I_v of the sort of the variable to be bound. In that sense a primitive abstraction can be considered to denote a special product of a value that specifies a variable to be bound and a value in which that variable is bound.

```
data ProdCode : Set where
  one :                    ProdCode
  rec : (i : Ix)           → ProdCode
```

```
_⊗_ : (c₁ c₂ : ProdCode)       → ProdCode
abs : (s : Iᵥ) (c : ProdCode) → ProdCode
```

The QuantCode type is a new kind of codes that represents a constructor. It pairs the product structure given by a ProdCode together with an index designating the result type in the family of this constructor. We will say that _▷_ tags a code with an *output index*.

```
data QuantCode : Set where
  _▷_ : (c : ProdCode) (i : Iₓ) → QuantCode
```

A DataCode is now a sum of QuantCodes instead of ProdCodes.

```
DataCode : Set
DataCode = List QuantCode
```

As a convenience in examples we define variants of rec and _▷_ which have I_u respectively I_v as argument type instead of I_x.

```
recᵤ = rec ∘ injᵤ
recᵥ = rec ∘ injᵥ
_▷ᵤ_ : ProdCode → Iᵤ → QuantCode
c ▷ᵤ i = c ▷ injᵤ i
_▷ᵥ_ : ProdCode → Iᵥ → QuantCode
c ▷ᵥ i = c ▷ injᵥ i
```

4.2.2 Interpretation

The interpretation of our codes is parameterized by a family Φ_x : Family of syntax types which is used for the recursive positions. Effectively, codes are interpreted as functors on objects of type Family. The definition of the inductive families interpreting codes have a term context of type Ctx as before, and now additionally an index of type I_x, the *output index*.

In the interpretation of a ProdCode the index argument of rec is used to select a specific type from the input family Φ_x.

```
module Interpretation (Φₓ : Family) where
  data P⟦_⟧ : ProdCode → Family where
    one : ∀ {o Γ}       →                                  P⟦ one     ⟧ o Γ
    rec : ∀ {o Γ i}     → Φₓ i Γ                        → P⟦ rec i   ⟧ o Γ
    _⊗_ : ∀ {o Γ c₁ c₂} → P⟦ c₁ ⟧ o Γ → P⟦ c₂ ⟧ o Γ → P⟦ c₁ ⊗ c₂ ⟧ o Γ
    abs : ∀ {o Γ s c}   → P⟦ c ⟧ o (Γ ▷ s)          → P⟦ abs s c ⟧ o Γ
```

For a tagged code c ▷ o the interpretation forces the output index to coincide with o. Selecting any other output index from the interpretation will result in an uninhabited type, since there is no way to construct a value of this type.

```
data Q⟦_⟧ : QuantCode → Family where
  tag  : ∀ {o Γ c}      → P⟦ c ⟧ o Γ              → Q⟦ c ▷ o  ⟧ o Γ
```

As before a DataCode is interpreted in two steps. The interpretation function D⟦_⟧ covers the sum structure of a DataCode and the function ⟦_⟧ adds variables to exactly those types in the family described by the index subset I_v of the complete index set I_x.

```
data D⟦_⟧ : DataCode → Family where
  top  : ∀ {o Γ c cs} → Q⟦ c ⟧ o Γ              → D⟦ c :: cs  ⟧ o Γ
  pop  : ∀ {o Γ c cs} → D⟦ cs ⟧ o Γ              → D⟦ c :: cs  ⟧ o Γ
data ⟦_⟧ : DataCode → Family where
  ⟨_⟩  : ∀ {o Γ c}    → D⟦ c ⟧ o Γ              → ⟦ c ⟧ o Γ
  var  : ∀ {o Γ c}    → Γ ∋ o                   → ⟦ c ⟧ (inj_v o) Γ
```

4.2.3 Example: First-order predicate logic of the naturals

As an example of a language with a family of syntax types with binding we use a first-order predicate logic of the natural numbers with addition and equality. Natural number expressions are the only sort with variables. The second sort are formulas which do not have variables. Thus we use the following type I_v respectively I_u as index sets for the sorts with respectively without variables.

```
data I_v : Set where
  expr    : I_v
data I_u : Set where
  formula : I_u
open Context I_v
open Universe I_u I_v
```

The term formers for natural number expressions are the zero, a local let definition binding one variable and the addition and successor operations. In formulas we allow conjunction, disjunction, negation, equality of naturals as well as universal and existential quantification over naturals. Both quantifications bind exactly one natural number variable. Below is a well-scoped de Bruijn representation of this language.

```
data Expr (Γ : Ctx) : Set where
  var   : Γ ∋ expr                    → Expr Γ
  let′  : Expr Γ → Expr (Γ ▷ expr)    → Expr Γ
  _+_   : Expr Γ → Expr Γ             → Expr Γ
  zero  :                               Expr Γ
  succ  : Expr Γ                      → Expr Γ
data Formula (Γ : Ctx) : Set where
  _==_  : Expr Γ → Expr Γ             → Formula Γ
```

```
_∧_    : Formula Γ → Formula Γ → Formula Γ
_∨_    : Formula Γ → Formula Γ → Formula Γ
¬_     : Formula Γ             → Formula Γ
forall′ : Formula (Γ ▷ expr)   → Formula Γ
exists′ : Formula (Γ ▷ expr)   → Formula Γ
```

The PLNat family is a mapping from the index sets to the syntax types of the language.

```
PLNat : Family
PLNat (inj_u formula) = Formula
PLNat (inj_v expr)    = Expr
```

The language can be encoded in the universe using the following QuantCodes which we collect together in the plnat-code DataCode.

```
expr-let     = rec_v expr ⊗ abs expr (rec_v expr) ▷_v expr
expr-plus    = rec_v expr ⊗ rec_v expr            ▷_v expr
expr-zero    = one                                ▷_v expr
expr-succ    = rec_v expr                         ▷_v expr

formula-equal  = rec_v expr ⊗ rec_v expr          ▷_u formula
formula-and    = rec_u formula ⊗ rec_u formula    ▷_u formula
formula-or     = rec_u formula ⊗ rec_u formula    ▷_u formula
formula-neg    = rec_u formula                    ▷_u formula
formula-forall = abs expr (rec_u formula)         ▷_u formula
formula-exists = abs expr (rec_u formula)         ▷_u formula

plnat-code = expr-let    :: expr-plus    :: expr-zero
           :: expr-succ  :: formula-equal :: formula-and
           :: formula-or :: formula-neg   :: formula-forall
           :: formula-exists :: [ ]
```

Part of the conversion from the user defined syntax types to the one-level unfolding of the fixed point is shown below. The conversion back is omitted.

```
open Interpretation PLNat
from : PLNat ⇝ ⟦ plnat-code ⟧
from {inj_v expr} (var x)   = var x
from {inj_v expr} (let′ x y) = ⟨ top (tag (rec x ⊗ abs (rec y)) ) ⟩
from {inj_v expr} (x + y)    = ⟨ pop (top (tag (rec x ⊗ rec y) )) ⟩
from {inj_v expr} zero       = ⟨ pop (pop (top (tag one ))) ⟩
from {inj_v expr} (succ x)   = ⟨ pop (pop (pop (top (tag (rec x) )))) ⟩
from {inj_u formula} ...
```

4.2.4 Quantified constructors

As discussed in section 2.3 a type system can be seen as a refinement of syntactic sorts of a language, by viewing for example expressions of a particular type as one sort. This influences the behavior of term constructors because they have to work for multiple types. In the well-typed de Bruijn representation of the simply-typed λ-calculus with one base type from section 2.3 for example,

```
data Ty : Set where
  unit : Ty
  _↠_ : (τ₁ τ₂ : Ty) → Ty
open Context Ty
data _⊢_ (Γ : Ctx) : Ty → Set where
  var  : ∀ {τ}     (x : Γ ∋ τ)                      → Γ ⊢ τ
  _·_  : ∀ {τ₁ τ₂} (f : Γ ⊢ τ₁ ↠ τ₂) (a : Γ ⊢ τ₁)   → Γ ⊢ τ₂
  ƛ    : ∀ {τ₁ τ₂} (b : Γ ▷ τ₁ ⊢ τ₂)               → Γ ⊢ τ₁ ↠ τ₂
  tt   :                                           Γ ⊢ unit
```

the application and abstraction constructors are quantified over two types. We will introduce this kind of quantification of constructors in our universe by augmenting the previous definition of QuantCode with a new alternative. The quantification will be over the object language types, which form some subset of the index set I_x. Usually it will be the set I_v or I_x, but we do not want to restrict ourselves to those cases, so we allow quantification over an arbitrary set A. The σ code takes as argument a function of type A → QuantCode that represents the quantification of a constructor.

```
data QuantCode : Set where
  _▷_ : (c : ProdCode) (i : Iₓ)              → QuantCode
  σ   : {A : Set} (cf : A → QuantCode)       → QuantCode

data Q⟦_⟧ : QuantCode → Family where
  tag  : ∀ {o Γ c}                                  → P⟦ c ⟧ o Γ   → Q⟦ c ▷ o ⟧ o Γ
  some : ∀ {o Γ A a} {cf : A → QuantCode} → Q⟦ cf a ⟧ o Γ → Q⟦ σ cf ⟧ o Γ
```

4.2.5 Example: Simply-typed lambda calculus

For the simply-typed lambda calculus with one base type we get the following codes in the universe.

```
tm-abs  = σ λ α → σ λ β → abs α (recᵥ β)          ▷ᵥ α ↠ β
tm-app  = σ λ α → σ λ β → recᵥ (α ↠ β) ⊗ recᵥ α   ▷ᵥ β
tm-unit = one                                     ▷ᵥ unit

stlc-code : DataCode
stlc-code = tm-app :: tm-abs :: tm-unit :: []
```

The data declaration already defines a family of syntax types. All expressions of a fixed simple object language type form one syntax type in the family.

```
Stlc : Family
Stlc (inj_u ()) Γ
Stlc (inj_v α) Γ = Γ ⊢ α

open Interpretation Stlc

from : Stlc ⇝ ⟦ stlc-code ⟧
from {inj_v τ} (var x) = var x
from {inj_v τ} (f · a) = ⟨ top (some (some (tag (rec f ⊗ rec a)))) ⟩
from {inj_v ._} (ƛ b)  = ⟨ pop (top (some (some (tag (abs (rec b)))))) ⟩
from {inj_v ._} tt     = ⟨ pop (pop (top (tag one))) ⟩
from {inj_u ()} x
```

4.3 Families with simple binders

In this section we will extend the universe of families of syntax types with a family of simple binders, i.e. binders that do not contain embedded terms such as patterns. This is an intermediate step to show the parametric HOAS encoding of multi-variable binders before we model them in our universe in the next section. In this section the codes of the universe will not reflect the inner structure of binders, but they are injected into the intepretations by an additional family and referenced by indices. An abstraction will not bind a single variable of some sort, but will take a binder with some binder context Δ and abstract from all variables in Δ simultaneously.

The universe is now parameterized over three index sets I_a, I_u and I_v. Like for families with unary binding the index set I_u is for syntax types without variables, while I_v is for sorts with variables. The third set I_a is an index set for a family of binders of type $I_a \to \mathsf{Ctx} \to \mathsf{Set}$, that will be given as input to the interpretations. The Ctx represents the binder context in this case. For our purposes, we only need to know the amount and sorts of variables a binder binds for use in abstractions. Their specific structure is immaterial, so we do not model it in the universe.

```
module Universe (I_a I_u I_v : Set) where
  data I_x : Set where
    inj_u : I_u → I_x
    inj_v : I_v → I_x
  Family : Set
  Family = I_x → Ctx → Set
  Binders : Set
  Binders = I_a → Ctx → Set
```

Only the definition of ProdCode changes in comparison with the universe of families with unary binding. So we will only outline the differences. An abstraction now takes the index of a binder instead of an index of the sort of a single variable.

```
data ProdCode : Set where
  one  : ProdCode
  rec  : Iₓ → ProdCode
  _⊗_  : (c d : ProdCode)        → ProdCode
  abs  : (i : Iₐ) (c : ProdCode) → ProdCode
```

The interpretation of our codes is parameterized by a family $\Psi : I_a \to Ctx \to Set$ of binders as discussed above and by a family Φ : Family for the recursive positions. The abstraction case selects the binder from Ψ given by the index from the abs argument and extends the term context by the binder context.

```
module Interpretation (Ψ : Iₐ → Ctx → Set) (Φ : Family) where
  data P⟦_⟧ : ProdCode → Family where
    one  : ∀ {o Γ}         →                                  P⟦ one     ⟧ o Γ
    rec  : ∀ {o Γ i}       → Φ i Γ                          → P⟦ rec i   ⟧ o Γ
    _⊗_  : ∀ {o Γ c₁ c₂}   → P⟦ c₁ ⟧ o Γ → P⟦ c₂ ⟧ o Γ      → P⟦ c₁ ⊗ c₂ ⟧ o Γ
    abs  : ∀ {o Γ Δ c iₐ}  → Ψ iₐ Δ → P⟦ c ⟧ o (Γ ▷▷ Δ)     → P⟦ abs iₐ c ⟧ o Γ
```

The record type Family groups a family of syntax types together with its binder family, its DataCode and morphisms for the one-level folding and unfolding.

```
_⇝_ : Family → Family → Set
Φ ⇝ Ψ = {i : Iₓ} {Γ : Ctx} → Φ i Γ → Ψ i Γ

record SyntaxFamily : Set where
  field
    family  : Family
    code    : DataCode
    binders : Binders
  open Interpretation binders family
  field
    from    : family ⇝ ⟦ code ⟧
    to      : ⟦ code ⟧ ⇝ family
```

Curch

Church encoding

The higher-order encoding of an abstraction that abstracts from all variables of a context Δ simultaneously can be thought to be of the form

```
Env D Δ → R
```

where $D : I_v \to Set$ is the domain of the values in the environment. That means we abstract from an environment, that holds a value for every variable in Δ, in a term of type R. Of course it is more convenient to write these functions in a curried style. Curried calculates the corresponding

curried function type for a context Δ and uncurry converts a curried function back to its uncurried counterpart.

```
Curried : (D : I_v → Set) (Δ : Ctx) → Set → Set
Curried D ϵ r = r
Curried D (Δ ▷ x) r = Curried D Γ (D x → r)
uncurry : {D : I_v → Set} {Δ : Ctx} {r : Set} → Curried D Δ r → Env D Δ → r
uncurry f ϵ = f
uncurry f (xs ▷ x) = uncurry f xs x
```

We will use the curried functions to provide Church encodings for a parametric HOAS interpretation of the universe. An algebra for the HOAS terms in the universe has a carrier of type $I_x \to$ Set. Like for the regular case, the Arg functor provides the argument type of an algebra function for a ProdCode. The interesting case is the abstraction, where a binder context is existentially quantified in a product of a binder of that context and a meta-level binding as a curried function for that context. The domain of the arguments of the meta-level binding is the sub-family of the abstract carrier belonging to the sorts with variables. Instead of Instead of repeating this selection all the time we provide a variant of Curried called Curried_x.

```
module HOAS (A : I_x → Set) where
  Curried_x : Ctx → Set → Set
  Curried_x Δ R = Curried (A ∘ inj_v) Δ R
  Arg : ProdCode → Set
  Arg one       = ⊤
  Arg (rec i)   = A i
  Arg (c ⊗ d)   = Arg c × Arg d
  Arg (abs b c) = Σ Ctx λ Δ → binders b Δ × Curried_x Δ (Arg c)
```

In AlgProd we curry the algebra function types again. The dependent product of Arg becomes a dependent function type and we make the binder context implicit, so that we are only left with the binder and the meta-level binding as explicit arguments. A ProdCode represents the product structure of a constructor for a particular syntax type in the family that is already fixed, i.e. the output index in the family has already been selected. So the result argument R of AlgProd has type Set. In the interpretation AlgQuant of a QuantCode the result type is not fixed until we reach a tagged ProdCode, so the result argument is a family of type $I_x \to$ Set. The interpretation of a tagged ProdCode fixes the result type to be the one with the output index. A quantified constructor is interpreted as a dependent function type, where we universally quantify over an implicit argument a from the type A.

```
AlgProd : ProdCode → (R : Set) → Set
AlgProd one       R = R
AlgProd (rec i)   R = A i → R
AlgProd (c ⊗ d)   R = AlgProd c (AlgProd d R)
AlgProd (abs b c) R = {Δ : Ctx} → binders b Δ → Curried_x Δ (Arg c) → R
```

```
AlgQuant : (c : QuantCode) (R : Iₓ → Set) → Set
AlgQuant (c ▷ i)      R = AlgProd c (R i)
AlgQuant (σ {A} cf) R = {a : A} → AlgQuant (cf a) R
```

The AlgData, AlgebraF and Algebra functions are similar to the ones for the regular syntax types.

```
AlgData : (c : DataCode) (R : Iₓ → Set) → Set
AlgData [ ]       R = ⊤
AlgData (c :: cs) R = AlgQuant c R × AlgData cs R

AlgebraF : QuantCode → Set
AlgebraF c = AlgQuant c A

Algebra : Set
Algebra = AlgData code A
```

Again AlgProd, AlgQuant and AlgData are functorial in the result argument types. The implementations are straightforward, so we omit them.

```
map-prod : (c : ProdCode) {r s : Set} →
           (f : r → s) → AlgProd c r → AlgProd c s
map-quant : (c : QuantCode) {R S : Iₓ → Set} →
            (f : {i : Iₓ} → R i → S i) → AlgQuant c R → AlgQuant c S
map-data : (c : DataCode) {R S : Iₓ → Set} →
           (f : {i : Iₓ} → R i → S i) → AlgData c R → AlgData c S
```

Conversion algebra

For the conversion algebra we need to produce a complete environment of variables. The functions makeExpVar and makeExpVars take care of this. In makeExpVars we need to ensure that each new variable τ of the binder context $\Delta \rhd \tau$ is introduced into the context $\Gamma \rhd\rhd \Delta$ where all previous variables have already been added.

```
makeExpVar : (Γ : Ctx) → (s : Iᵥ) → (Δ : Ctx) → family (injᵥ s) Δ
makeExpVar Γ s Δ with Γ ▷ s ⊑? Δ
makeExpVar Γ s Δ | yes Γ▷s⊑Δ = to (var (makeVar Γ▷s⊑Δ))
makeExpVar Γ s Δ | no ¬p      = whatever
  where postulate whatever : _

makeExpVars : (Γ Δ : Ctx) → Env (λ s → (Δ : Ctx) → family (injᵥ s) Δ) Δ
makeExpVars Γ ε = ε
makeExpVars Γ (Δ ▷ τ) = makeExpVars Γ Δ ▷ makeExpVar (Γ ▷▷ Δ) τ
```

The carrier of the conversion algebra is the parameter family where we abstract from the term context a term is instantiated in. Like for the regular case we also define types for a partially constructed one-level unfolding.

```
Exp : Iₓ → Set
Exp i = (Γ : Ctx) → family i Γ
ExpP : ProdCode → Iₓ → Set
ExpP c i = (Γ : Ctx) → P⟦ c ⟧ i Γ
ExpQ : QuantCode → Iₓ → Set
ExpQ c i = (Γ : Ctx) → Q⟦ c ⟧ i Γ
ExpD : DataCode → Iₓ → Set
ExpD c i = (Γ : Ctx) → D⟦ c ⟧ i Γ
open HOAS Exp
```

We only look at the interesting case of an abstraction for the conversion algebra. We create an environment for the binder context Δ with values that will instantiate indices for variables and subsequently apply the meta-level binding given by the curried function to that environment. The result is the body of the abstraction that will be instantiated in the extended context $\Gamma \rhd\rhd \Delta$.

```
mutual
  conv-arg : (c : ProdCode) {i : Iₓ} → Arg c → ExpP c i
  conv-arg one       x       Γ = one
  conv-arg (rec i)   x       Γ = rec (x Γ)
  conv-arg (c ⊗ d)   (x , y) Γ = conv-arg c x Γ ⊗ conv-arg d y Γ
  conv-arg (abs b c) (Δ , bd , x) Γ = conv-arg-abs c Δ bd x Γ
  conv-arg-abs : (c : ProdCode) {i : Iₓ} {b : Iₐ} (Δ : Ctx) (bd : binders b Δ) →
                 Curriedₓ Δ (Arg c) → ExpP (abs b c) i
  conv-arg-abs c Δ bd f Γ = abs bd (conv-arg c
                                    (uncurry f (makeExpVars Γ Δ))
                                    (Γ ⊳⊳ Δ))
```

The remaining functions for the conversion algebra take care of applying the constructors of the primitive well-formed de Bruijn representation types and do a one-level folding of the fixed point of the parameter family.

```
conv-prod  : (c : ProdCode) {i : Iₓ} → AlgProd c (ExpP c i)
conv-quant : (c : QuantCode)         → AlgQuant c (ExpQ c)
conv-data  : (c : DataCode)          → AlgData c (ExpD c)
conversionAlgebra : Algebra
```

4.3.1 Example: Simply-typed λ-calculus with patterns

In this section we will show an extended example: the simply-typed lambda calculus with patterns. It also serves as the first example of a family of syntax types with quantified constructors. We will provide a definition of well-scoped and well-typed de Bruijn terms. The language will support products, coproducts and list types and provide pattern matching in λ-abstractions and in a separate case-split similar to the constructs found in Haskell. We will look at an encoding in the universe with simple binders and the associated parametric HOAS interpretation.

The object language types are represented by the following type Ty.

```
data Ty : Set where
  unit : Ty
  _↠_ : (τ₁ τ₂ : Ty) → Ty
  _⊠_ : (τ₁ τ₂ : Ty) → Ty
  _⊞_ : (τ₁ τ₂ : Ty) → Ty
  list : (α : Ty)    → Ty
```

The patterns are the only binders of our language. They are indexed by a binder context and by the type the pattern matches on. The pattern constructor • represents pattern variable positions which bind exactly one variable. The binder context of • thus consists of a single variable of some type τ and the type of the patterns as given by the Ty index is also τ. The other pattern constructors represent the object language's data constructors. All of them accumulate the binder context of sub-patterns. • is the only pattern constructor that introduces new variable bindings.

```
data _⊩⊢_ : (Γ : Ctx) (τ : Ty) → Set where
  •   : ∀ {τ}              → ϵ ▷ τ ⊩⊢ τ
  tt  :                     ϵ ⊩⊢ unit
  _,_ : ∀ {Γ₁ Γ₂ τ₁ τ₂}    → Γ₁ ⊩⊢ τ₁ → Γ₂ ⊩⊢ τ₂ → Γ₁ ▷▷ Γ₂ ⊩⊢ τ₁ ⊠ τ₂
  inl : ∀ {Γ τ₁ τ₂}        → Γ ⊩⊢ τ₁ → Γ ⊩⊢ τ₁ ⊞ τ₂
  inr : ∀ {Γ τ₁ τ₂}        → Γ ⊩⊢ τ₂ → Γ ⊩⊢ τ₁ ⊞ τ₂
  [ ] : ∀ {τ}              → ϵ ⊩⊢ list τ
  _::_ : ∀ {Γ₁ Γ₂ τ}       → Γ₁ ⊩⊢ τ → Γ₂ ⊩⊢ list τ → Γ₁ ▷▷ Γ₂ ⊩⊢ list τ
```

Next to patterns our language has three more sorts: expressions, case alternatives and list of case alternatives which are defined mutually recursively. The λ-expressions are indexed by the term context and the type of the expression. An alternative for a case distinction is indexed by a term context and by two types. One type indicates the type of the value the alternative matches on and the second type is the type of the right hand side of the alternative. The list of alternatives is also indexed by two types, where all alternatives in the list are indexed by the same type.

The expressions are the only sort that has variables. We have expression constructors for variables, application, λ-abstraction, data type constructors as well as an expression constructor fix introducing a fixed-point combinator into the language and case_of_ that introduces a case distinction. case_of_ takes an expression of type τ_1 and a list of alternatives that pattern match on a value of type τ_1. The type of the complete case distinction is the type of the right hand sides of the list of case alternatives.

```
mutual
  data _⊢_ (Γ : Ctx) : Ty → Set where
    var  : ∀ {τ}       (x : Γ ∋ τ)                    → Γ ⊢ τ
    _·_  : ∀ {τ₁ τ₂}   (f : Γ ⊢ τ₁ ↠ τ₂) (a : Γ ⊢ τ₁)  → Γ ⊢ τ₂
    abs  : ∀ {τ₁ τ₂ Δ} (p : Δ ⊩⊢ τ₁) (b : Γ ▷▷ Δ ⊢ τ₂) → Γ ⊢ τ₁ ↠ τ₂
    tt   :                                             Γ ⊢ unit
```

```
_,_       : ∀ {τ1 τ2}  (a : Γ ⊢ τ1) (b : Γ ⊢ τ2)          → Γ ⊢ τ1 ⊠ τ2
inl       : ∀ {τ1 τ2}  (t : Γ ⊢ τ1)                       → Γ ⊢ τ1 ⊞ τ2
inr       : ∀ {τ1 τ2}  (t : Γ ⊢ τ2)                       → Γ ⊢ τ1 ⊞ τ2
[]        : ∀ {τ}                                         → Γ ⊢ list τ
_::_      : ∀ {τ}      (t : Γ ⊢ τ) (ts : Γ ⊢ list τ)      → Γ ⊢ list τ
case_of_  : ∀ {τ1 τ2}  (t : Γ ⊢ τ1) (cs : Alts Γ τ1 τ2)  → Γ ⊢ τ2
fix       : ∀ {τ}      (t : Γ ⊢ τ ↠ τ)                  → Γ ⊢ τ
```

```
data Alt (Γ : Ctx) (τ1 τ2 : Ty) : Set where
  alt : {Δ : Ctx} (p : Δ ⊩ τ1) (t : (Γ ▷▷ Δ) ⊢ τ2) → Alt Γ τ1 τ2

data Alts (Γ : Ctx) (τ1 τ2 : Ty) : Set where
  []   : Alts Γ τ1 τ2
  _::_ : Alt Γ τ1 τ2 → Alts Γ τ1 τ2 → Alts Γ τ1 τ2
```

A λ-abstraction or a case alternative in context Γ take a pattern with some binder context Δ and abstract from the Δ variables in an expression of the extended context $\Gamma \rhd\rhd \Delta$. These two are the only forms of abstraction in the language.

We turn now to the structure description of the simply-typed lambda calculus. The sorts of the language are case alternatives and list of case alternatives which are both index by two simple types. Thus we take the following type $\mathsf{I_u}$ as the index set for the non-variable sub-family.

```
data Iu : Set where
  alt  : Ty → Ty → Iu
  alts : Ty → Ty → Iu
```

The patterns are the only binders and the λ-expressions are the only syntactic sorts with variables. Both are refined by exactly one simple type. Thus we use Ty for the index sets $\mathsf{I_a}$ and $\mathsf{I_v}$. StlcF is then the family of syntax type of our language.

```
open Base Ty Iu Ty public

StlcF : Family
StlcF (inju (alt τ1 τ2)) Γ = Alt Γ τ1 τ2
StlcF (inju (alts τ1 τ2)) Γ = Alts Γ τ1 τ2
StlcF (injv τ)            Γ = Γ ⊢ τ
```

Several of the term formers of the simply typed λ-calculus are quantified over one or two types. Quantification is introduced by σ codes which gives us variables that we can use in recursive positions or in the output index of a tag. Note also that we can apply arbitrary constructors of the index sets in between. The ProdCodes for the constructors follow easily from the corresponding constructor declarations.

```
lam-abs   = σ λ α → σ λ β → abs α (recv β)          ▻v (α ↠ β)
lam-app   = σ λ α → σ λ β → recv (α ↠ β) ⊗ recv α    ▻v β
lam-tt    =                 one                     ▻v unit
lam-_,_   = σ λ α → σ λ β → recv α ⊗ recv β          ▻v (α ⊠ β)
```

```
lam-inl    = σ λ α → σ λ β → rec_v α                          ▷_v (α ⊞ β)
lam-inr    = σ λ α → σ λ β → rec_v β                          ▷_v (α ⊞ β)
lam-case   = σ λ α → σ λ β → rec_v α ⊗ rec_u (alts α β)       ▷_v β
lam-fix    = σ λ α →          rec_v (α ↠ α)                  ▷_v α
lam-[]     = σ λ α →          one                            ▷_v list α
lam-::     = σ λ α →          rec_v α ⊗ rec_v (list α)        ▷_v list α
alt-alt    = σ λ α → σ λ β → abs α (rec_v β)                 ▷_u alt α β
alts-[]    = σ λ α → σ λ β → one                             ▷_u alts α β
alts-::    = σ λ α → σ λ β → rec_u (alt α β) ⊗ rec_u (alts α β) ▷_u alts α β
```

We collect the algebra functions for the parametric HOAS representation of the simply typed λ-calculus in a record. The carrier is a family A of type $\mathsf{I}_x \to \mathsf{Set}$. The term context is handled at the meta-level and not encoded in the types. The sub-families Lam, Alt and Alts from A are selected for convenience. The fields of the record are the algebra functions. The given types are equivalent to the result of the AlgebraF function to the QuantCodes.

```
record STLCAlgebra (A : I_x → Set) : Set where
  open HOAS A
  Lam : Ty → Set
  Lam τ = A (inj_v τ)
  Alt : Ty → Ty → Set
  Alt τ₁ τ₂ = A (inj_u (alt τ₁ τ₂))
  Alts : Ty → Ty → Set
  Alts τ₁ τ₂ = A (inj_u (alts τ₁ τ₂))
  field
    lam      : {α β : Ty} {Δ : Ctx} → Δ ⊩ α → Curried_x Δ (Lam β) → Lam (α ↠ β)
    _·_      : {α β : Ty} →           Lam (α ↠ β) → Lam α → Lam β
    tt       :                        Lam unit
    _,′_     : {α β : Ty} →           Lam α → Lam β → Lam (α ⊠ β)
    inl′     : {α β : Ty} →           Lam α → Lam (α ⊞ β)
    inr′     : {α β : Ty} →           Lam β → Lam (α ⊞ β)
    case_of_ : {α β : Ty} →           Lam α → Alts α β → Lam β
    fix      : {α   : Ty} →           Lam (α ↠ α) → Lam α
    []′      : {α   : Ty} →           Lam (list α)
    _::′_    : {α   : Ty} →           Lam α → Lam (list α) → Lam (list α)
    _⟶_      : {α β : Ty} {Δ : Ctx} → Δ ⊩ α → Curried_x Δ (Lam β) → Alt α β
    ∅        : {α β : Ty} →           Alts α β
    _|_      : {α β : Ty} →           Alt α β → Alts α β → Alts α β
```

We present some example terms of the parametric HOAS representation. swapPair represents a function that pattern matches on a product with object language type $\alpha \boxtimes \beta$ in a λ-abstraction and creates a new product with swapped components. The pattern is given by $(\bullet, \bullet)$ which has two pattern variables. The λ-abstraction thus gets a meta-level function with two parameters of type Lam α and Lam β, that encodes the variable binding of the two pattern variables.

The swapPair function is polymorphically parameterized by two object language types α, β. This polymorphism only exists at the meta-level. For each choice of types α, β we get a simply-typed object language function, not an object language function that is polymorphic.

foldr is an object language representation of the foldr function. It shows the use of the case_of_ construct and of case alternatives, that are created by _⟶_. The left hand side is a pattern. In the above foldr function we have the patterns $(\bullet :: \bullet)$, pattern matching on a cons constructor of lists with two pattern variables, and $[\,]$ pattern matching on a nil constructor of lists without pattern variables. The right hand sides are curried functions with the corresponding number of arguments, one argument per pattern variable.

```
module Examples {A : Iₓ → Set} (alg : STLCAlgebra A) where
  open STLCAlgebra alg
  swapPair : (α β : Ty) → Lam (α ⊠ β ↠ β ⊠ α)
  swapPair α β = lam (•, •) (λ x y → y, ′ x)
  foldr : (α β : Ty) → Lam ((α ↠ β ↠ β) ↠ β ↠ list α ↠ β)
  foldr α β = fix (lam • λ foldr → lam • λ cons → lam • λ nil → lam • λ xs →
    case xs of
      ( (• :: •) ⟶ (λ y ys → cons · y · (foldr · cons · nil · ys))
      | [ ]      ⟶ nil
      | ∅))
```

The converted terms of the de Bruijn representation are shown below.

```
foldr unit unit ϵ
  ≡ fix (abs • (abs • (abs • (abs •
    (case v 0 of
      ( alt (• :: •) (v 4 · v 1 · (v 5 · v 4 · v 3 · v 0))
      :: alt [ ]     (v 1)
      :: [ ])))))))
```

4.4 Binders with embedded terms, sequential and recursive scoping

In the previous universes of syntax types we only modeled the structure of terms that reference variables but did not specifically model binders, i.e. values that specify a set of variables to be bound in an abstraction. In this section we will generalize the universe of families from section 4.2 to include binders. We will allow values of any syntactic sort to bind variables. Thus we will not distinguish between sorts which are used solely as binders, like patterns, and sorts which are not used as binders at all, like λ-expressions. The generalization will allow us to mix both categories and have values that bind variables and at the same time contain variables, like declarations. Furthermore we extend the universe by new structure descriptions for richer forms of binding like sequential and recursive scoping.

Families in this universe are again indexed by two sets I_u and I_v. The syntax types in the families are now indexed by two contexts, a binder context and the usual term context.

```
data I_x : Set where
  inj_u : I_u → I_x
  inj_v : I_v → I_x
Fam : Set
Fam = I_x → (Δ Γ : Ctx) → Set
```

We introduce sng as a new alternative for ProdCode. It represents a value that binds exactly one new variable of the given sort. The bseq code describes sequential scoping. The variables of the first code are intended to scope over the second. Recursive scoping is introduced by brec. The variables bound by values of the given code scope over the value itself. As a new special product we add balt which forces values of the given codes to have equal binder and term contexts. This introduces a form of choice between different alternatives of binders that bind the same variables.

```
data ProdCode : Set where
  one  :                          ProdCode
  sng  : I_v                    → ProdCode
  rec  : I_x                    → ProdCode
  _⊗_  : (c_1 c_2 : ProdCode)   → ProdCode
  abs  : (c_1 c_2 : ProdCode)   → ProdCode
  bseq : (c_1 c_2 : ProdCode)   → ProdCode
  balt : (c_1 c_2 : ProdCode)   → ProdCode
  brec : (c : ProdCode)         → ProdCode
```

The definition of QuantCode and DataCode remain the same as in the previous section.

```
data QuantCode : Set where
  _⊳_ : (c : ProdCode) (ix : I_x) → QuantCode
  σ   : {A : Set} (cf : A → QuantCode) → QuantCode
DataCode : Set
DataCode = List QuantCode
```

The interpretations are now functors over syntax families with two contexts. The one code does not bind any variables, so the binder context is empty. sng s has a binder context consisting of exactly one variable of sort s. The binder context of a product is the concatenation of the binder contexts Δ_1 and Δ_2 of the components. In the case of an abs the variables of the first ProdCode scope over the second, i.e. the binder context Δ_1 of the first interpretation is appended to the term context of the second, thus the variable bindings are accumulated. The binder context Δ_1 of the first code is hidden in the abstraction by existential quantification and the binder context of the result is the binder context Δ_2 of the second ProdCode, i.e. the binder context of an abstraction is the binder context of the value inside the abstraction. Similarly for bseq the

variables of the first code scope over the second, but the binder context of the result is the concatenation of the binders of the sub-components like for the product case. The interpretation for balt forces both binder and term context of the sub-components to be equal. For recursive scoping via brec the binder context Δ is appended to the term context of the interpretation itself, thus the sub-component is in the context $\Gamma \rhd\!\rhd \Delta$.

```
module Interpretation (Φ : Fam) where
  data P⟦_⟧ : ProdCode → Fam where
    one  : ∀ {o Γ}              → P⟦ one   ⟧ o ϵ Γ
    sng  : ∀ {s o Γ}            → P⟦ sng s ⟧ o (ϵ ▷ s) Γ
    rec  : ∀ {o i Γ Δ}          → Φ i Δ Γ → P⟦ rec i ⟧ o Δ Γ
    _⊗_  : ∀ {Δ₁ Δ₂ c₁ c₂ o Γ} → P⟦ c₁ ⟧ o Δ₁ Γ → P⟦ c₂ ⟧ o Δ₂ Γ
                                → P⟦ c₁ ⊗ c₂ ⟧ o (Δ₁ ▷▷ Δ₂) Γ
    abs  : ∀ {Δ₁ Δ₂ c₁ c₂ o Γ} → P⟦ c₁ ⟧ o Δ₁ Γ → P⟦ c₂ ⟧ o Δ₂ (Γ ▷▷ Δ₁)
                                → P⟦ abs c₁ c₂ ⟧ o Δ₂ Γ
    bseq : ∀ {Δ₁ Δ₂ c₁ c₂ o Γ} → P⟦ c₁ ⟧ o Δ₁ Γ → P⟦ c₂ ⟧ o Δ₂ (Γ ▷▷ Δ₁)
                                → P⟦ bseq c₁ c₂ ⟧ o (Δ₁ ▷▷ Δ₂) Γ
    balt : ∀ {Δ c₁ c₂ o Γ}      → P⟦ c₁ ⟧ o Δ Γ → P⟦ c₂ ⟧ o Δ Γ
                                → P⟦ balt c₁ c₂ ⟧ o Δ Γ
    brec : ∀ {Δ c o Γ}          → P⟦ c ⟧ o Δ (Γ ▷▷ Δ) → P⟦ brec c ⟧ o Δ Γ
```

The interpretations of QuantCode and DataCode are extended by an argument for the binder context. It is passed through to the ProdCode interpretations.

```
  data Q⟦_⟧ : QuantCode → Fam where
    tag  : ∀ {Γ Δ c o}                          → P⟦ c    ⟧ o Δ Γ → Q⟦ c ▷ o ⟧ o Δ Γ
    some : ∀ {Γ Δ A a o} {cf : A → QuantCode} → Q⟦ cf a ⟧ o Δ Γ → Q⟦ σ cf   ⟧ o Δ Γ
  data D⟦_⟧ : DataCode → Fam where
    top  : ∀ {o Γ Δ c cs} → Q⟦ c  ⟧ o Δ Γ → D⟦ c :: cs ⟧ o Δ Γ
    pop  : ∀ {o Γ Δ c cs} → D⟦ cs ⟧ o Δ Γ → D⟦ c :: cs ⟧ o Δ Γ
  data ⟦_⟧ (c : DataCode) : Fam where
    ⟨_⟩  : ∀ {o Γ Δ} → D⟦ c ⟧ o Δ Γ → ⟦ c ⟧ o Δ Γ
    var  : ∀ {o Γ}   → Γ ∋ o        → ⟦ c ⟧ (inj_v o) ϵ Γ
```

4.4.1 Parametric HOAS types

In section 4.3 we showed a universe with simple binders, whose structure was not modeled in the universe itself, but which were injected into the interpretations. The same binders have been used in the parametric HOAS representation. The binder context was accumulated in an index before it was finally used in an abstraction. In the same way we will index the values in the parametric HOAS interpretation of the binders universe with a binder context. The term context is still handled at the meta-level and thus hidden in the representation. The definitions below reside in a module parameterized by an abstract algebra carrier.

The parameter types of meta-level functions representing the variable bindings will always lie in the family of the abstract carrier. A variable does not bind any other variables, so the binder context of the parameters is always chosen to be the empty context ϵ.

```
module HOAS (A : Ix → Ctx → Set) where
  Curriedx : Ctx → Set → Set
  Curriedx Δ r = Curried (λ i → A (injv i) ϵ) Δ r
```

We start with the argument functor for argument types of algebra functions. These are now functions between sets indexed by binder contexts, and need to be natural in it, i.e. we have the situation $\forall\ \{\Gamma\} \to \mathsf{Arg}\ \mathsf{c}\ \Gamma \to \mathsf{r}\ \Gamma$ for some ProdCode c.

For single variable binders sng s respectively unit binders one the binder context needs to consists of a single variable respectively needs to be empty. We enforce this restriction with an equality constraint. For a recursive position we take the syntax type with the given index and binder context from the carrier family. For products and sequentially scoped products we need to split the given binder context into two binder contexts for the sub-components. To this end we use existentially quantified contexts Γ_1, Γ_2 and force Γ to be their concatenation by an equality constraint. In the case of an abstraction we take a value of the first code and use its existentially quantified binder context for a curried function resulting in a value of the second code. The binder context of the second code is the context parameter Γ of Arg, thus the binder context of an abstraction is the binder context of the value inside the abstraction. For bseq the situation is similar to both the product and abstraction case. The binder context is split into two sub-contexts for the two codes like for products. The value of the second code is then inside a curried function of the first binder context like for abstractions.

```
Arg : ProdCode → Ctx → Set
Arg one          Γ = Γ ≡ ϵ
Arg (sng s)      Γ = Γ ≡ ϵ ▷ s
Arg (rec i)      Γ = A i Γ
Arg (c1 ⊗ c2)    Γ = Σ Ctx λ Γ1 → Σ Ctx λ Γ2 →
                     Γ ≡ Γ1 ▷▷ Γ2 × Arg c1 Γ1 × Arg c2 Γ2
Arg (abs c1 c2)  Γ = Σ Ctx λ Γ+ → Arg c1 Γ+ × Curriedx Γ+ (Arg c2 Γ)
Arg (bseq c1 c2) Γ = Σ Ctx λ Γ1 → Σ Ctx λ Γ2 →
                     Γ ≡ Γ1 ▷▷ Γ2 × Arg c1 Γ1 × Curriedx Γ1 (Arg c2 Γ2)
Arg (brec c)     Γ = Curriedx Γ (Arg c Γ)
Arg (balt c1 c2) Γ = Arg c1 Γ × Arg c2 Γ
```

In the definition of the curried variant AlgProd we can calculate the binder context of the algebra type from the binder contexts of the components instead of the other way round. For one the algebra type is the result type with an empty context and for sng it is the result type with a single variable context. For a recursive position the algebra function needs to map for every binder context the syntax type with the given index to the result type. For convenience we make the binder context an implicit argument of the dependent function. In the case of a product or a bseq the result type has as binder context the concatenation of the binder contexts of the

components. Further for bseq we have a meta-level binding given by a curried function with the binder context from the left code.

```
AlgProd : ProdCode → (R : Ctx → Set) → Set
AlgProd one          R = R ϵ
AlgProd (sng i)      R = R (ϵ ▷ i)
AlgProd (rec i)      R = {Δ : Ctx} → A i Δ → R Δ
AlgProd (c₁ ⊗ c₂)    R = AlgProd c₁ (λ Γ₁ → AlgProd c₂ (λ Γ₂ → R (Γ₁ ▷▷ Γ₂)))
AlgProd (abs c₁ c₂)  R = AlgProd c₁ (λ Γ₁ →
                           {Γ₂ : Ctx} → Curried_x Γ₁ (Arg c₂ Γ₂) → R Γ₂)
AlgProd (bseq c₁ c₂) R = AlgProd c₁ (λ Γ₁ →
                           {Γ₂ : Ctx} → Curried_x Γ₁ (Arg c₂ Γ₂) → R (Γ₁ ▷▷ Γ₂))
AlgProd (brec c)     R = {Δ : Ctx} → Curried_x Δ (Arg c Δ) → R Δ
AlgProd (balt c₁ c₂) R = AlgProd c₁ (λ Δ → Arg c₂ Δ → R Δ)
```

The definition AlgQuant fixes the output index of the result type R for a tagged ProdCode and AlgData collects the resulting types in a product. Again the types are functorial in the result type and allow the definition of a functiorial mapping. The implementation of these is omitted.

```
AlgQuant : (c : QuantCode) (R : I_x → Ctx → Set) → Set
AlgQuant (c ▷ i) R = AlgProd c (R i)
AlgQuant (σ cf)  R = ∀ {a} → AlgQuant (cf a) R

AlgData : (c : DataCode) (R : I_x → Ctx → Set) → Set
AlgData [ ]       R = ⊤
AlgData (c :: cs) R = AlgQuant c R × AlgData cs R
```

The carrier of the conversion algebra is a family indexed by the set I_x and a binder context that abstracts from the term context a de Bruijn term is instantiated in.

```
Exp : I_x → Ctx → Set
Exp i Δ = (Γ : Ctx) → fam i Δ Γ

ExpP : ProdCode → I_x → Ctx → Set
ExpP c i Δ = (Γ : Ctx) → P⟦ c ⟧ i Δ Γ
open HOAS Exp


mutual
  conv-arg : (c : ProdCode) {i : I_x} {Δ : Ctx} → Arg c Δ → ExpP c i Δ
  conv-arg one          refl              Γ = one
  conv-arg (sng y)      refl              Γ = sng
  conv-arg (rec y)      x                 Γ = rec (x Γ)
  conv-arg (c₁ ⊗ c₂)    (Δ₁ , Δ₂ , refl , x , y) Γ = conv-arg c₁ x Γ ⊗ conv-arg c₂ y Γ
  conv-arg (abs c₁ c₂)  (Δ₁ , x , y)      Γ = abs
                                                (conv-arg c₁ x Γ)
                                                (conv-arg-abs c₂ Δ₁ Γ y)
```

```
conv-arg (bseq c₁ c₂) (Δ₁, Δ₂, refl, x, y) Γ  =  bseq
                                                   (conv-arg c₁ x Γ)
                                                   (conv-arg-abs c₂ Δ₁ Γ y)
conv-arg (balt c₁ c₂)  (x, y)                 Γ  =  balt
                                                   (conv-arg c₁ x Γ)
                                                   (conv-arg c₂ y Γ)
conv-arg (brec c) {Δ = Δ} x                   Γ  =  brec
                                                   (conv-arg-abs c Δ Γ x)
conv-arg-abs : (c : ProdCode) {i : Iₓ} {Δ₂ : Ctx} → (Δ₁ Γ : Ctx) →
                 Curriedₓ Δ₁ (Arg c Δ₂) → P⟦ c ⟧ i Δ₂ (Γ ▷▷ Δ₁)
conv-arg-abs c Δ Γ f  =  conv-arg c (uncurry f (makeExpVars Γ Δ)) (Γ ▷▷ Δ)
```

4.4.2 Example: Let bindings

We will give smaller examples of the parametric HOAS interpretation for small λ-calculi with various forms of let-bindings. We start with non-sequential non-recursive let-bindings and later look at the more powerful variants. Our languages have three sorts: λ-expressions with variables as well as declarations and list of declarations as sorts without variables. We thus get the following index sets.

```
data Iᵥ : Set where
  lam  : Iᵥ
data Iᵤ : Set where
  decl  : Iᵤ
  decls : Iᵤ
```

Below are types for a well-scoped de Bruijn representation of the language and corresponding codes in the universe. We also index λ-expression with a binder context to match the uniform handling of syntax types in the universe.

```
mutual
  data Decl : Ctx → Ctx → Set where
    decl  : ∀ {Γ Δ} → Lam Δ Γ → Decl (Δ ▷ lam) Γ
  data Decls : Ctx → Ctx → Set where
    dnil  : ∀ {Γ}          →                                  Decls ϵ Γ
    dcons : ∀ {Γ Δ₁ Δ₂} → Decl Δ₁ Γ → Decls Δ₂ Γ → Decls (Δ₁ ▷▷ Δ₂) Γ
  data Lam : Ctx → Ctx → Set where
    var   : ∀ {Γ}          → Γ ∋ lam                  → Lam ϵ Γ
    abs   : ∀ {Γ Δ}        → Lam Δ (Γ ▷ lam)          → Lam Δ Γ
    app   : ∀ {Γ Δ₁ Δ₂} → Lam Δ₁ Γ → Lam Δ₂ Γ          → Lam (Δ₁ ▷▷ Δ₂) Γ
    let′  : ∀ {Γ Γ⁺ Δ} → Decls Γ⁺ Γ → Lam Δ (Γ ▷▷ Γ⁺) → Lam Δ Γ
```

For the above types we have the following universe codes.

```
lam-abs   = abs (sng lam) (rec_v lam)   ▷_v lam
lam-app   = rec_v lam ⊗ rec_v lam       ▷_v lam
lam-let   = abs (rec_u decls) (rec_v lam) ▷_v lam
decl-decl = rec_v lam ⊗ sng lam         ▷_u decl
decls-nil = one                         ▷_u decls
decls-cons = rec_u decl ⊗ rec_u decls   ▷_u decls
```

We come to parametric HOAS types for our language. A λ-abstraction is a single variable binding represented by a single argument function. The calculated HOAS algebra types consider the case that potentially any value can bind variables, even for sorts like λ-expressions which do no bind variables. For those values the implicit binder context will always be empty. Thus the interface includes unnecessary binder context calculations.

```
record LamAlgebra (A : I_x → Ctx → Set) : Set where
  open HOAS A
  Lam   = A (inj_v lam)
  Decl  = A (inj_u decl)
  Decls = A (inj_u decls)
  infixr 5 _◁′_
  infixl 2 _·_
  field
    lam′     : {Δ : Ctx} → (Lam ϵ → Lam Δ) → Lam Δ
    _·_      : {Δ₁ : Ctx} → Lam Δ₁ →
               {Δ₂ : Ctx} → Lam Δ₂ → Lam (Δ₁ ▷▷ Δ₂)
    let′_in′_ : {Δ₁ : Ctx} → Decls Δ₁ →
               {Δ₂ : Ctx} → Curried_x Δ₁ (Lam Δ₂) → Lam Δ₂
    decl′    : {Δ : Ctx} → Lam Δ → Decl (Δ ▷ lam)
    ϵ′       : Decls ϵ
    dcons′   : {Δ₁ : Ctx} → Decl Δ₁ →
               {Δ₂ : Ctx} → Decls Δ₂ → Decls (Δ₁ ▷▷ Δ₂)
```

For convenience we use a syntax macro for λ-expressions and a helper function adding a λ-expression to a list of declarations.

```
syntax lam′ (λ x → e) = ƛ x ⇒ e
_◁′_ : {Δ : Ctx} → Lam ϵ → Decls Δ → Decls (ϵ ▷ lam ▷▷ Δ)
e ◁′ ds = dcons′ (decl′ e) ds
```

As example terms we define the I, K and S combinators in a let and use them in the body. We have three declarations binding one variable each, so the body is represented by a three argument function type.

```
module Examples {A} (alg : LamAlgebra A) where
  open LamAlgebra alg
  test : Lam ϵ
```

```
test = let′
         (ƛ x ⇒ x)                          ◁′
         (ƛ x ⇒ ƛ y ⇒ x)                    ◁′
         (ƛ f ⇒ ƛ g ⇒ ƛ x ⇒ f · x · (g · x)) ◁′ ϵ′
       in′ λ I K S → I · K
```

A downside of the HOAS representation of let bindings is that the variable names are physically separated from the expressions of the declarations in the source code. The above term converted to the well-scoped de Bruijn representation is shown below.

```
let′
  (dcons (decl (abs (v 0)))
  (dcons (decl (abs (abs (v 1))))
  (dcons (decl (abs (abs (abs (app (app (v 2) (v 0))
                                   (app (v 1) (v 0)))))))
    dnil)))
  (app (v 2) (v 1))
```

4.4.3 Example: Sequential let bindings

We modify the previous example by using sequential let-bindings instead. Each declaration in a declaration list binds a variable that scopes over all subsequent declarations. The definition of Decls thus becomes

```
data Decls : Ctx → Ctx → Set where
  dnil  : ∀ {Γ}          →                                  Decls ϵ           Γ
  dcons : ∀ {Γ Δ₁ Δ₂} → Decl Δ₁ Γ → Decls Δ₂ (Γ ▻▻ Δ₁) → Decls (Δ₁ ▻▻ Δ₂) Γ
```

The corresponding code in the universe is

```
decls-cons = bseq (rec_u decl) (rec_u decls) ▻_u decls


record LamAlgebra (A : I_x → Ctx → Set) : Set where
  open HOAS A
  Lam   = A (inj_v lam)
  Decl  = A (inj_u decl)
  Decls = A (inj_u decls)
  infixr 5 _◁′_
  infixl 2 _·_
  field
    lam′  : {Δ : Ctx} → (Lam ϵ → Lam Δ) → Lam Δ
    _·_   : {Δ₁ : Ctx} → Lam Δ₁ →
            {Δ₂ : Ctx} → Lam Δ₂ → Lam (Δ₁ ▻▻ Δ₂)
```

```
  let′_in′ : {Δ1 : Ctx} → Decls Δ1 →
             {Δ2 : Ctx} → Curried_x Δ1 (Lam Δ2) → Lam Δ2
  decl′    : ({Δ : Ctx} → Lam Δ → Decl (Δ ▷ lam))
  ϵ′       : Decls ϵ
  dcons′ : {Δ1 : Ctx} → Decl Δ1 →
           {Δ2 : Ctx} → Curried_x Δ1 (Decls Δ2) → Decls (Δ1 ▷▷ Δ2)
syntax lam′ (λ x → e) = ƛ x ⇒ e
_◁′_ : {Δ : Ctx} → Lam ϵ →
       Curried_x (ϵ ▷ lam) (Decls Δ) → Decls (ϵ ▷ lam ▷▷ Δ)
e ◁′ ds = dcons′ (decl′ e) ds
```

Below is an example of a sequential let. We first declare one combinator t which is then used in the second declaration. Note that scoping occurs at every declaration so for each declaration there is immediately a meta-level function scoping over the subsequent declarations. In the end the abstraction of the let also adds a binding of all variables simultaneously for scoping over the body. Unfortunately this duplicates variable names and does not enforce equal names for variables in the declaration list and the body.

```
module Examples {A} (alg : LamAlgebra A) where
  open LamAlgebra alg
  test : Lam ϵ
  test = let′
           (ƛ x ⇒ x · x) ◁′ (λ t →
           (t · t)       ◁′ λ ω → ϵ′)
         in′
           (λ t ω → ω)
```

The translated term of the above is

```
test ϵ ≡ let′ (dcons (decl (abs (app (v 0) (v 0))))
              (dcons (decl (app (v 0) (v 0)))
              dnil))
             (v 0)
```

Another awkwardness in the chosen HOAS type for bseq is that every bseq introduces a function for the complete accumulated binder context of its left code. We have chosen cons lists for the let declarations in our example, so the head introduces a single variable that scopes over the tail. Choosing a snoc list will scope all the variables of the tail over the head.

4.4.4 Example: Recursive let bindings

For recursive lets we have the following well-scoped de Bruijn representation for λ-expressions. The binder context of a let group is reintroduced in the term context of the declarations.

```
data Lam : Ctx → Ctx → Set where
  var    : ∀ {Γ}       → Γ ∋ lam
                       → Lam ϵ Γ
  abs    : ∀ {Γ Δ}     → Lam Δ (Γ ▷ lam)
                       → Lam Δ Γ
  app    : ∀ {Γ Δ₁ Δ₂} → Lam Δ₁ Γ → Lam Δ₂ Γ
                       → Lam (Δ₁ ▷▷ Δ₂) Γ
  let′   : ∀ {Γ Γ⁺ Δ}   → Decls Γ⁺ (Γ ▷▷ Γ⁺)
                       → Lam Δ (Γ ▷▷ Γ⁺) → Lam Δ Γ

lam-let = abs (brec (recᵤ decls)) (recᵥ lam) ▷ᵥ lam
```

We get the following type for the algebra function of recursive lets. We have a curried function over a context which is at the same time the binder context of the result value of the function.

```
letrec : {Δ : Ctx} →
           Curriedₓ A Δ (Decls Δ) →
           {Γ₂ : Ctx} → Curriedₓ A Δ (Lam Γ₂) → Lam Γ₂
```

The following is an example of a recursive let with two mutually recursive functions. Note that in general it is not possible to determine the binder context automatically. We have to provide it explicitly to the algebra function of let. Moreover we have duplication of variables as for the sequential let bindings. One function that introduces all the variables for the declaration and one function for the body.

```
module Examples {A} (alg : LamAlgebra A) where
  open LamAlgebra alg
  test : Lam ϵ
  test = letrec {ϵ ▷ lam ▷ lam}
           (λ a b →
             (ƛ x ⇒ b)     ◁′
             (ƛ x ⇒ x · a) ◁′ ϵ′)
           (λ a b → b · a)
```

Again names are not enforced to coincide. In this case it is possible to add another constructor absrec c_1 c_2 to ProdCode, that is equivalent to abs (rec c_1) c_2, i.e. variables from the first code scope recursively and over values of the second code and interpret it with a HOAS type that lifts both, the declarations and the let body into the body of the meta-level function. The conversion of the above let expression is shown below.

```
test ϵ ≡ let′
           (dcons (decl (abs (v 1)))
           (dcons (decl (abs (app (v 0) (v 2))))
           dnil))
           (app (v 0) (v 1))
```

5 Generic syntax operations

5.1 Motivation example

In this section we describe generic traversals over syntax types. We will specifically look at simultaneous renaming and shifting of variables as well as simultaneous and single variable substitution. McBride [McB05] describes a simultaneous substitution function for a well-typed de Bruijn representation of the simply-typed λ-calculus in Epigram. Noticing the similarities between renamings and substitutions McBride provides a single traversal function that is first instantiated to simultaneous renamings and later to simultaneous substitutions.

The Agda standard library [AT11] includes the module Data.Fin.Substitution that builds on McBride's [McB05] ideas. It provides a general definition of renaming and substitution for well-scoped de Bruijn representations with natural numbers as the context type. The user can instantiate the definitions to a concrete datatype by providing a traversal function for it. Furthermore a wealth of lemmas and theorems about the given substitutions can be imported once a certain key lemma about the traversal function is provided.

Building upon the definitions from the Agda standard library we will develop a datatype generic traversal function for syntax operations for the binders universe of the last section. It abstracts from the concrete operation that is performed even further so that it also allows the definition of shifting and single variable substitutions.

We will start with renaming and substitution specific to a de Bruijn representation of the simply-typed λ-calculus and building on that abstract from the operation that is performed and define a generic traversal over the universe. We use the following de Bruijn representation of the simply-typed λ-calculus in the first part.

```
data Ty : Set where
  unit : Ty
  _↠_ : (τ₁ τ₂ : Ty) → Ty
open Context Ty
data _⊢_ (Γ : Ctx) : Ty → Set where
  var  : ∀ {τ}      (x : Γ ∋ τ)                          → Γ ⊢ τ
  _·_  : ∀ {τ₁ τ₂} (f : Γ ⊢ τ₁ ↠ τ₂) (a : Γ ⊢ τ₁)  → Γ ⊢ τ₂
  ƛ    : ∀ {τ₁ τ₂} (b : Γ ▷ τ₁ ⊢ τ₂)                  → Γ ⊢ τ₁ ↠ τ₂
  tt   :                                                 Γ ⊢ unit
```

Simultaneous variable renamings

The first operation we will look at is simultaneous renaming of variables of a context. Here a variable renaming is a function that maps all variables from one context to variables in another context.

```
module Renaming where
  infix 3 _⇒_
  _⇒_ : Ctx → Ctx → Set
  Γ ⇒ Δ = {σ : Ty} → Γ ∋ σ → Δ ∋ σ
```

A renaming $\rho : \Gamma \Rightarrow \Delta$ can be extended to a renaming $\Gamma \rhd \tau \Rightarrow \Delta \rhd \tau$ between bigger contexts. Note that _▷_ binds more tightly than _⇒_. The least de Bruijn index of $\Gamma \rhd \tau$ is mapped to itself and for the Γ indices we apply ρ, which will result in an index in Δ that we weaken to an index in $\Delta \rhd \tau$ using vs. To highlight a more general structure we use the identity function in the vz case. Later we will abstract from the function located at that position.

```
  infix 10 _↑
  _↑ : {Γ Δ : Ctx} {τ : Ty} → Γ ⇒ Δ → Γ ▷ τ ⇒ Δ ▷ τ
  (ρ ↑) vz     = id vz
  (ρ ↑) (vs x) = vs (ρ x)
```

As an example consider the renaming ρ_1 defined below. Extending it by a new variable of sort τ will result in a renaming that is pointwise equal to ρ_2. The τ variable with index 0 is mapped to itself while the other indices on the right-hand side are incremented to account for the new variable.

```
  ρ₁ : ∀ {σ₁ σ₂} → ε ▷ σ₁ ▷ σ₂ ▷ σ₁ ⇒ ε ▷ σ₁ ▷ σ₂
  ρ₁ (vs (vs (vs ())))
  ρ₁ (vs (vs vz))     = v 1
  ρ₁ (vs vz)          = v 0
  ρ₁ vz               = v 1
  ρ₂ : ∀ {σ₁ σ₂ τ} → ε ▷ σ₁ ▷ σ₂ ▷ σ₁ ▷ τ ⇒ ε ▷ σ₁ ▷ σ₂ ▷ τ
  ρ₂ (vs (vs (vs (vs ()))))
  ρ₂ (vs (vs (vs vz))) = v 2
  ρ₂ (vs (vs vz))     = v 1
  ρ₂ (vs vz)          = v 2
  ρ₂ vz               = v 0
```

To apply a renaming to a term we traverse it to the variable positions and apply the renaming function there. Whenever a λ-abstraction is encountered the renaming is extended with _↑. The function application in the var case is made explicit. We will abstract from the operation performed there and also from the var constructor which is applied to the result to produce a term.

For the definition of substitutions we will need weakening of terms that can be defined by renaming with the variable weakening vs. Only the indices corresponding to free variables are increased.

```
infix 8 _/_
_/_ : {Γ Δ : Ctx} {σ : Ty} → Γ ⊢ σ → Γ ⇒ Δ → Δ ⊢ σ
var x   / ρ = var (ρ $ x)
t₁ · t₂ / ρ = (t₁ / ρ) · (t₂ / ρ)
ƛ t     / ρ = ƛ (t / ρ ↑)
tt      / ρ = tt

weaken : ∀ {Γ τ σ} → Γ ⊢ τ → Γ ▷ σ ⊢ τ
weaken t = t / vs
```

As an example consider the term t with three free variables corresponding to the λ-expression $(\lambda x.(\lambda y.yxa)c)b$. Applying the renaming ρ_1 from above to t will result in the term u corresponding to $(\lambda x.(\lambda y.yxa)c)a$. Weakening t will give use w where the indices for the free variables a, b and c have been increased.

```
t : ε ▷ unit ▷ (unit ↠ unit ↠ unit) ▷ unit ⊢ unit
t = ƛ (ƛ (v 0 · v 1 · v 4) · v 2) · v 0
u : ε ▷ unit ▷ (unit ↠ unit ↠ unit) ⊢ unit
u = ƛ (ƛ (v 0 · v 1 · v 3) · v 1) · v 1
w : ε ▷ unit ▷ (unit ↠ unit ↠ unit) ▷ unit ▷ unit ⊢ unit
w = ƛ (ƛ (v 0 · v 1 · v 5) · v 3) · v 1
```

Simultaneous substitutions

Simultaneous substitution of terms for variables is similar to renaming. Variables from one context are mapped to terms in another context.

```
module Substitution where
  open Renaming using (weaken)
  infix 3 _⇒_
  _⇒_ : Ctx → Ctx → Set
  Γ ⇒ Δ = {σ : Ty} → Γ ∋ σ → Δ ⊢ σ
```

Extending a substitution to a bigger context is achieved by mapping the lowest index to itself. For all other variables we use ρ and weaken the result term.

```
_↑ : {Γ Δ : Ctx} {τ : Ty} → Γ ⇒ Δ → Γ ▷ τ ⇒ Δ ▷ τ
(ρ ↑) vz     = var vz
(ρ ↑) (vs x) = weaken (ρ x)
```

At variable positions during traversal we can apply the substitution function to the variable to produce a result term.

```
infix 8 _/_
_/_ : ∀ {Γ₁ Γ₂ τ} → Γ₁ ⊢ τ → Γ₁ ⇒ Γ₂ → Γ₂ ⊢ τ
var x   / ρ = id (ρ $ x)
t₁ · t₂ / ρ = (t₁ / ρ) · (t₂ / ρ)
ƛ t     / ρ = ƛ (t / ρ ↑)
tt      / ρ = tt
```

A substitution of the smallest index for a term is defined by sub. For the smallest index the given term is returned and all other indices are strengthened. Combining this function with the substitution traversal above gives us a function that substitutes one term for the smallest context index in another term.

```
sub : ∀ {Γ σ} → Γ ⊢ σ → Γ ▷ σ ⇒ Γ
sub t vz     = t
sub t (vs x) = var x
_[_] : ∀ {Γ σ τ} → Γ ▷ σ ⊢ τ → Γ ⊢ σ → Γ ⊢ τ
t [ s ] = t / sub s
```

5.2 Transformation definitions

Both simultaneous renaming and substitution of variables recurse the terms to the variable position and apply the function they are carrying, effectively lifting an operation on variables to an operation on terms. The difference between both traversals lies in the result type of the functions that are carried around. A renaming maps variables to variables and a substitution maps variables to terms. McBride [McB05] abstracts from this result type and later instantiates it first to variables for renamings and then terms for substitutions. We will subsequently call this type the *contents type* of an operation.

Real functions have been used in the previous section to map variables to contents types. They can also be given by another piece of data like for example an environment. TransformationData collects this idea in a record for a specific contents type _◆_ and a specific transformation. It includes a type _⇒_ holding the information about the transformation function, the auxiliary function _↑ for recursing under binders, that extends the transformation function to a bigger context, as well as app that forms post application of the transformation function to variables. For the traversal of binders we will need to recurse under abstractions that bind more than one variable, hence we provide a helper function _↑⋆_ that extends the transformation by a binder context.

```
Syntax : Set₁
Syntax = Ctx → Ty → Set
record TransformationData (_◆_ : Syntax) : Set₁ where
  infix 3 _⇒_
  infix 10 _↑
```

```
  infixl 10 _↑⋆_
  field
    _⇒_ : Ctx → Ctx → Set
    _↑   : {Γ Δ : Ctx} {τ : Ty} → Γ ⇒ Δ → Γ ▷ τ ⇒ Δ ▷ τ
    app  : {Γ Δ : Ctx} {τ : Ty} → Γ ∋ τ → Γ ⇒ Δ → Δ ◆ τ
  _↑⋆_ : {Γ₁ Γ₂ : Ctx} → Γ₁ ⇒ Γ₂ → (Δ : Ctx) → Γ₁ ▷▷ Δ ⇒ Γ₂ ▷▷ Δ
  ρ ↑⋆ ϵ       = ρ
  ρ ↑⋆ (Γ ▷ τ) = (ρ ↑⋆ Γ) ↑
```

The definition of the _↑ function for renaming and substitution required the availability of a two functions, a weaken function for the contents type and a function var mapping variables to the contents type. The definition for variables as contents type is straightforward.

```
record Simple (_◆_ : Syntax) : Set where
  field
    var    : ∀ {Γ τ}   → (x : Γ ∋ τ) → Γ ◆ τ
    weaken : ∀ {Γ τ σ} → (x : Γ ◆ τ) → (Γ ▷ σ) ◆ τ
varSimple : Simple _∋_
varSimple = record {var = id; weaken = vs}
```

Abstracting from the contents type used in TransformationData gives us a general definition of a transformation for all syntax types.

```
Transformation : Set₁
Transformation = {_◆_ : Syntax} → Simple _◆_ → TransformationData _◆_
```

Simultaneous renamings and substitutions

The renaming and substitution operations from the last section can now be made instances of the more general definition of transformation above and in fact they can be defined using a single definition. We now use an environment rather than a function to carry the contents type. The lookup function will look up a value in the environment for a given variable. Note that the parameters of the following module match the parameters of Transformation. Using varSimple as the module parameter will give us the definition of simultaneous variable renamings of the previous section.

```
module SubstitutionData {_◆_ : Syntax} (simple : Simple _◆_) where
  open Simple simple
  infix 3 _⇒_
  _⇒_ : Ctx → Ctx → Set
  Γ ⇒ Δ = Env (_◆_ Δ) Γ
  _↑ : {Γ Δ : Ctx} {τ : Ty} → Γ ⇒ Δ → Γ ▷ τ ⇒ Δ ▷ τ
  _↑ ρ = map (λ τ → weaken) ρ ▷ var vz
```

```
transformation : TransformationData _◆_
transformation = record {_⇒_ = _⇒_; _↑ = _↑; app = lookup}
```

Below are some standard substitutions defined. wk⇒ weakens the result values of a given substitution, id⇒ is the identity substitution and wk is a substitution that weakens terms by one context variable.

```
wk⇒ : ∀ {Γ Δ τ} → Γ ⇒ Δ → Γ ⇒ Δ ▷ τ
wk⇒ = map (λ τ → weaken)

id⇒ : ∀ {Γ} → Γ ⇒ Γ
id⇒ {ε} = ε
id⇒ {Γ ▷ τ} = id⇒ ↑

wk : ∀ {Γ τ} → Γ ⇒ Γ ▷ τ
wk = wk⇒ id⇒
```

5.3 Generic traversal function for binders

Variables themselves do not bind any other variables. This implies that they cannot be substituted by terms that bind any variables, because the binder context of the result would depend on the number of occurrences of that variable. Thus the types that we allow in the syntax transformations need to be from the subfamily indexed by I_v and have an empty binder context.

```
infix 2 _◆_
_◆_ : Ctx → Iv → Set
Γ ◆ i = fam (injv i) ∈ Γ
```

At variable positions during the traversal the result of the application of the transformation function will produce a contents type value. This value has to be lifted to the term datatype that is being traversed. If the contents type are variables _∋_ then the var constructor needs to be applied, in the case of terms _◆_ the lifting is done by the identity function. The following record Lift describes such a lifting of contents types. We can already define the lifting of variables at this point. The lifting of terms requires weakening of terms which we can only define after using the term traversal.

```
record Lift (_◆′_ : Syntax) : Set where
  field
    simple : Simple _◆′_
    lift : ∀ {Γ τ} → Γ ◆′ τ → Γ ◆ τ
  open Simple simple public

varLift : Lift _∋_
varLift = record {simple = varSimple; lift = to ∘ var}
```

The traversal function for a family in the universe is now abstracted over two things. The contents type that is used during traversal with a lifting record and the transformation that is

being applied. The traversal itself is described by a set of mutually recursive traversal functions. One traversal function for each interpretation function of the universe and one traversal function _/_ that takes care of the one-level folding and unfolding of the functor. The traversals push the transformation function ρ through the constructors of the interpretations. In the case of scoping in the abs, bseq and brec case the transformation function is extended to the bigger context by $\_\uparrow^\star\_$.

```
module Traversal {_◆′_ : Syntax} (I : Lift _◆′_)
                 (transformation : Transformation) where
  open Lift I hiding (var)
  open TransformationData (transformation simple)
  mutual
    infix 8 _P/_ _Q/_ _D/_ _//_ _/_
    _P/_ : ∀ {c i Δ Γ₁ Γ₂} → P⟦ c ⟧ i Δ Γ₁ → Γ₁ ⇒ Γ₂ → P⟦ c ⟧ i Δ Γ₂
    one          P/ ρ = one
    sng          P/ ρ = sng
    rec x        P/ ρ = rec (x / ρ)
    (x ⊗ y)      P/ ρ = (x P/ ρ) ⊗ (y P/ ρ)
    abs {Δ} x y  P/ ρ = abs (x P/ ρ) (y P/ (ρ ↑⋆ Δ))
    bseq {Δ} x y P/ ρ = bseq (x P/ ρ) (y P/ (ρ ↑⋆ Δ))
    balt x y     P/ ρ = balt (x P/ ρ) (y P/ ρ)
    brec {Δ} x   P/ ρ = brec (x P/ (ρ ↑⋆ Δ))
    _Q/_ : ∀ {c i Δ Γ₁ Γ₂} → Q⟦ c ⟧ i Δ Γ₁ → Γ₁ ⇒ Γ₂ → Q⟦ c ⟧ i Δ Γ₂
    tag x    Q/ ρ = tag (x P/ ρ)
    some x Q/ ρ = some (x Q/ ρ)
    _D/_ : ∀ {c i Δ Γ₁ Γ₂} → D⟦ c ⟧ i Δ Γ₁ → Γ₁ ⇒ Γ₂ → D⟦ c ⟧ i Δ Γ₂
    top y  D/ ρ = top (y Q/ ρ)
    pop y  D/ ρ = pop (y D/ ρ)
    _//_ : ∀ {i Δ Γ₁ Γ₂} → ⟦ code ⟧ i Δ Γ₁ → Γ₁ ⇒ Γ₂ → ⟦ code ⟧ i Δ Γ₂
    ⟨ x ⟩ // ρ = ⟨ x D/ ρ ⟩
    var x // ρ = from (lift (app x ρ))
    _/_ : ∀ {i Δ Γ₁ Γ₂} → fam i Δ Γ₁ → Γ₁ ⇒ Γ₂ → fam i Δ Γ₂
    x / ρ = to (from x // ρ)
```

With the traversal function we can now define term lifting for the family. The weakening of terms is defined by traversing with a variable renaming. That is we use the substitution transformation with variables as the contents type and use the wk substitution.

```
termSimple : Simple _◆_
termSimple = record
  {var    = to ∘ var
  ;weaken = λ t → t / SubstitutionData.wk varSimple}
  where open Traversal varLift SubstitutionData.transformation
```

```
termLift : Lift _◆_
termLift = record {simple = termSimple; lift = id}
```

5.4 Syntax operations

In this section we define more syntax operations and derive some standard functions that provide a more convenient interface to the operations. To define substitutions we use a traversal with terms as the contents type and use the substitution transformation we defined earlier. Usually one only wants to substitute the least de Bruijn index for some term, i.e. when performing β-reduction in a λ-calculus. This is implemented by the $\_\,[\_]$ function.

```
module Substitution where
  open Traversal termLift SubstitutionData.transformation public
  open SubstitutionData termSimple public hiding (transformation)
  _ [_] : ∀ {i Δ Γ j} → fam i Δ (Γ ▻ j) → fam (inj_v j) ∈ Γ → fam i Δ Γ
  t [s] = t / (id⇒ ▻ s)
```

Shifting

Another operation that we will use in the next section is shifting of variables and terms. It is usually used to define weakening of terms as that coincides with shifting of the zero index. Shifting increments all indices above a cutoff c, or put differently it introduces the index c into a context. For well-typed shifting we use the following approach: reintroduce an index into a context from which it has been removed. The removal of an index from a context is defined below.

```
infixl 5 _-_
_-_ : ∀ {s} (Γ : Ctx) → (Γ ∋ s) → Ctx
∈      - ()
Γ ▻ s - vz   = Γ
Γ ▻ s - vs c = Γ - c ▻ s
```

Given the context removal function we can give a definition of the shifting of variables. Note that the index x is increased if and only if it is greater or equal to c when read as a natural number.

```
shift∋ : ∀ {Γ s t} (c : Γ ∋ s) → (Γ - c) ∋ t → Γ ∋ t
shift∋ vz     x      = vs x
shift∋ (vs c) vz     = vz
shift∋ (vs c) (vs x) = vs (shift∋ c x)
```

The definition of the shifting for terms uses the following transformation definition. The transformation function contains the cutoff index that is being introduced. Encapsulating the

cutoff index in the datatype _⇒_ ensures that the context removal is hidden in the indices during traversal. Recursing under a binder means we have to increase the cutoff index. The shifting module defines a generic shift function that can be used with any type in the family and a cutoff index.

```
module ShiftingData {_◆_ : Syntax} (simple : Simple _◆_) where
  open Simple simple
  infix 3 _⇒_
  data _⇒_ : Ctx → Ctx → Set where
    ⟨_⟩ : ∀ {Γ σ} (c : Γ ∋ σ) → Γ - c ⇒ Γ
  infix 10 _↑
  _↑ : ∀ {Γ Δ τ} → Γ ⇒ Δ → (Γ ▻ τ) ⇒ (Δ ▻ τ)
  ⟨ c ⟩ ↑ = ⟨ vs c ⟩
  shiftVar : ∀ {Γ Δ τ} → Γ ∋ τ → Γ ⇒ Δ → Δ ◆ τ
  shiftVar y ⟨ c ⟩ = Simple.var simple (shift∋ c y)
  transformation : TransformationData _◆_
  transformation = record {_⇒_ = _⇒_; _↑ = _↑; app = shiftVar}
module Shifting where
  open Traversal termLift ShiftingData.transformation public
  open ShiftingData termSimple public hiding (transformation)
  shift : ∀ {Γ Δ σ τ} → (x : Γ ∋ σ) → fam τ Δ (Γ - x) → fam τ Δ Γ
  shift x t = t / ⟨ x ⟩
```

Single variable substitution

The last syntax operation we present is substituting a single variable by a term. To define this operation we first have to look at equality of de Bruijn indices. We reuse definitions and terminology from Keller and Altenkirch [KA10] for this. The predicate _≡∋_ specifies if two indices x, y in a common context represent the same variable. The indices represent the same variable iff they are equal which is indicated by the same constructor. If however x and y do not represent the same variable then the variable of y has an index z in Γ - x and consequently y ≡ shift∋ x z. The function _$\overset{?}{=}$∋_ decides _≡∋_ for any two indices.

```
data _≡∋_ {Γ σ} (x : Γ ∋ σ) : ∀ {τ} → Γ ∋ τ → Set where
  same :                    x ≡∋ x
  diff : ∀ {τ} (z : Γ - x ∋ τ) → x ≡∋ shift∋ x z
_≟∋_ : ∀ {Γ σ τ} (x : Γ ∋ σ) (y : Γ ∋ τ) → x ≡∋ y
vz   ≟∋ vz   = same
vz   ≟∋ vs y = diff y
vs x ≟∋ vz   = diff vz
vs x ≟∋ vs y with x ≟∋ y
```

```
vs x ≟∋ vs .x                | same = same
vs x ≟∋ vs ∘ (shift∋ x p)    | diff p = diff (vs p)
```

Note in particular that the sorts of the indices need not be the same for deciding equality of them. In case the indices are equal, we also learn that the sorts must be equal. This is essential for the definition of a heterogeneous single variable substitution, because when substitution a term for a variable of some sort in a value of another sort at a recursive positions that is a variable we need to know if the types corresponding to the sorts are equal or not.

For the transformation function of single variable substitutions we need two pieces of information: the index x of the variable and the term s that is to be substituted for the variable. Recursing under a binder means increasing the index and weakening the term. For applying a single variable substitution to a variable with index v we unpack the index x of the variable to be substituted and compare it to v. If they are the same we use the term s else we get an index v′ for v in the smaller context Γ - x which is used as the result.

```
module SingleSubstitutionData {_◆_ : Syntax} (simple : Simple _◆_) where
  open Simple simple
  infix 3 _⇒_23
  data _⇒_ (Γ : Ctx) : Ctx → Set where
    ⟨_,_⟩ : ∀ {σ} (x : Γ ∋ σ) → (Γ - x) ◆ σ → Γ ⇒ Γ - x
  _↑ : ∀ {Γ Δ τ} → Γ ⇒ Δ → (Γ ▻ τ) ⇒ (Δ ▻ τ)
  ⟨ x, s ⟩↑ = ⟨ vs x, weaken s ⟩
  app : ∀ {Γ Δ τ} → Γ ∋ τ → Γ ⇒ Δ → Δ ◆ τ
  app v ⟨ x , s ⟩ with x ≟∋ v
  app v ⟨ .v, s ⟩ | same = s
  app ._ ⟨ x , s ⟩ | diff v′ = Simple.var simple v′
  transformation : TransformationData _◆_
  transformation = record {_⇒_ = _⇒_; _↑ = _↑; app = app}
```

The SingleSubstitution module provides a function with a convenient interface to the operation.

```
module SingleSubstitution where
  open Traversal termLift SingleSubstitutionData.transformation public
  open SingleSubstitutionData termSimple public hiding (app; transformation)
  _ [_/_] : ∀ {i j Δ Γ} (t : fam j Δ Γ) (x : Γ ∋ i)
            (s : fam (inj_v i) ϵ (Γ - x)) → fam j Δ (Γ - x)
  t [x / s] = t / ⟨ x, s ⟩
```

5.5 Decidable predicate for free variables

In this section we want to look at free variables of terms and strengthening. A term can be strengthened, i.e. remove variables from its context, iff the removed variables do no appear free

in the term. We develop a generic decidable predicate for free variables appearing in terms. If the variable given by an index x appears free in the term t then the position of an occurrence of x within t is a proof for this. The position of a variable in a term is an example of a type-indexed datatype that is defined similar to one-hole contexts for zippers. If however the variable x does not appear free in the term t, then there exists an equivalent term t' in the strengthened context Γ - x, i.e. the context from which the variable x has been removed. In this case we have $t \equiv \mathsf{shift}\ x\ t'$.

We will give a definition of the predicate as well as a deciding function for the universe of families of syntax types from section 4.2. For this we use a direct implementation of the shift function given by the set of mutually functions shown below. The implementations of shiftD and shiftQ are straightforward and therefore omitted.

```
mutual
  shift : ∀ {Γ σ τ} (x : Γ ∋ σ) → family τ (Γ - x) → family τ Γ
  shift x t = to (shift′ x (from t))

  shift′ : ∀ {Γ σ τ cs} (x : Γ ∋ σ) → ⟦ cs ⟧ τ (Γ - x) → ⟦ cs ⟧ τ Γ
  shift′ x ⟨ s ⟩     = ⟨ shiftD x s ⟩
  shift′ x (var v)   = var (shift∋ x v)

  shiftP : ∀ {Γ s τ c} (x : Γ ∋ s) → P⟦ c ⟧ τ (Γ - x) → P⟦ c ⟧ τ Γ
  shiftP x one       = one
  shiftP x (rec t)   = rec (shift x t)
  shiftP x (s ⊗ t)   = shiftP x s ⊗ shiftP x t
  shiftP x (abs t)   = abs (shiftP (vs x) t)
```

We will first develop the predicate that a variable appears free in a term. It is given by a set of mutually recursive predicates; one predicate per interpretation of the universe as well as one predicate for the family itself. A variable appears free in a term if it appears free in the one-level unfolding, i.e. we have $x \in\mathsf{fv}'\ t$ if $x \in\mathsf{fv}\ \mathsf{to}\ t$ for every $t : ⟦\ \mathsf{code}\ ⟧\ \tau\ \Gamma$. Equivalently we could have used $x \in\mathsf{fv}'\ \mathsf{from}\ s$ iff $x \in\mathsf{fv}\ s$. The predicate for the immediate one-level unfolding handles the base case for variables, i.e. we have $x \in\mathsf{fv}'\ \mathsf{var}\ x$ for every variable x.

```
mutual
  data _∈fv_ {Γ s} (x : Γ ∋ s) : ∀ {τ} → family τ Γ → Set where
    to′ : ∀ {τ} {t : ⟦ code ⟧ τ Γ} → x ∈fv′ t → x ∈fv to t
  data _∈fv′_ {Γ s} (x : Γ ∋ s) : ∀ {o} → ⟦ code ⟧ o Γ → Set where
    var :                                  x ∈fv′ var x
    ⟨_⟩ : ∀ {τ} {t : D⟦ code ⟧ τ Γ} → x D∈fv t → x ∈fv′ ⟨ t ⟩
```

Note that for ProdCodes in the case of a product we have two possibilities, the variable can appear in either component or both. The $,_1$ and $,_2$ take care of this. For a unary abstraction we need to increase the index for the body. In all other cases the variable appears free in a value if it appears free in a sub-component. The definitions of the predicates for DataCodes and QuantCodes are straightforward and omitted.

```
data _P∈fv_ {Γ s} {o : I_x} (x : Γ ∋ s) : ∀ {c} → P⟦ c ⟧ o Γ → Set where
  rec : ∀ {i}      {t : family i Γ}                  → x    ∈fv t   → x P∈fv rec t
  ⊗₁  : ∀ {c₁ c₂} {s : P⟦ c₁ ⟧ o Γ} {t : P⟦ c₂ ⟧ o Γ} → x    P∈fv s → x P∈fv (s ⊗ t)
  ⊗₂  : ∀ {c₁ c₂} {s : P⟦ c₁ ⟧ o Γ} {t : P⟦ c₂ ⟧ o Γ} → x    P∈fv t → x P∈fv (s ⊗ t)
  abs : ∀ {c τ}    {t : P⟦ c ⟧ o (Γ ▻ τ)}           → vs x P∈fv t → x P∈fv abs t
```

As explained above, deciding whether a variable given by index x appears free in a term t will either give us a position of the variable or a term t′ such that t ≡ shift x t′. The following datatypes represent those result for each the predicates for the family and the one-level unfolding. Similar datatypes can be defined for the ProdCodes, DataCodes and QuantCodes. In the positive case we get a proof for the corresponding free variable predicate and in the negative case a pre-image of the corresponding shift function.

```
data Dec_∈fv_ {Γ s} (x : Γ ∋ s) : ∀ {τ} → family τ Γ → Set where
  yes : ∀ {α} {t : family α Γ} (p : x ∈fv t) → Dec x ∈fv t
  no  : ∀ {α} (t : family α (Γ - x))         → Dec x ∈fv shift x t
data Dec_∈fv′_ {Γ s} (x : Γ ∋ s) : ∀ {τ} → ⟦ code ⟧ τ Γ → Set where
  yes : ∀ {α} {t : ⟦ code ⟧ α Γ} (p : x ∈fv′ t) → Dec x ∈fv′ t
  no  : ∀ {α} (t : ⟦ code ⟧ α (Γ - x))         → Dec x ∈fv′ shift′ x t
```

Below are the deciding functions for the predicates, which are defined mutually recursively. In the base case of a variable in the deciding function _∈fv′?_ for the one-level unfolding we make use of the variable equality defined in section 5.4. To go from the one-level unfolding to the family of syntax types itself we need to prove a lemma to-shift that states that the to function commutes with shifting. Note that in the code below we also need proofs to∘from≡id and from∘to≡id that to and from are inverses of each other.

```
mutual
  _∈fv′?_ : ∀ {Γ s τ} → (x : Γ ∋ s) (t : ⟦ code ⟧ τ Γ) → Dec x ∈fv′ t
  x ∈fv′? var y                with x ≟∋ y
  x ∈fv′? var .x               | same = yes var
  x ∈fv′? var ∘ (shift∋ x y′)  | diff y′ = no (var y′)
  x ∈fv′? ⟨ t ⟩                with x D∈fv? t
  x ∈fv′? ⟨ t ⟩                | yes p = yes ⟨ p ⟩
  x ∈fv′? ⟨ ∘ (shiftD x t′) ⟩  | no t′ = no ⟨ t′ ⟩
  _∈fv?_ : ∀ {Γ s τ} → (x : Γ ∋ s) (t : family τ Γ) → Dec x ∈fv t
  x ∈fv? t = cast (cong (λ t' → Dec x ∈fv t') (to∘from≡id t)) (lem (x ∈fv′? from t))
    where
      cast : ∀ {A B} → A ≡ B → A → B
      cast refl x = x
      to-shift : ∀ {Γ σ τ} (x : Γ ∋ σ) (t : ⟦ code ⟧ τ (Γ - x)) →
                 to (shift′ x t) ≡ shift x (to t)
      to-shift x t = cong (to ∘ shift′ x) (sym (from∘to≡id t))
      lem : ∀ {Γ s τ} {x : Γ ∋ s} {t : ⟦ code ⟧ τ Γ} →
```

```
        Dec x ∈fv′ t → Dec x ∈fv to t
lem (yes p)  =  yes (to′ p)
lem {x  =  x} (no t) rewrite to-shift x t  =  no (to t)
```

5.5.1 Example: Reductions in the simply-typed lambda calculus

As an example for substitutions and strengthening we will look at $\beta\eta$-normalization of simply-typed lambda terms. In the case of an application we check for a β-redex, i.e. the left-hand side of the application is a λ-abstraction. In this case we perform a β-reduction by performing the single variable substitution from section 5.4 and substituting left-hand side of the application for the bound variable that has index vz. We recursively normalize the result, because the β-reduction could have created new redexes. In the case of a λ-abstraction we check for an η-redex. If the body of the abstraction is an application and the right-hand side is the variable from the abstraction with index vz and it does not appear free in the left-hand side we can perform an η-reduction. In this case the strengthened left-hand side is the result.

```
red : ∀ {Γ σ} → Γ ⊢ σ → Γ ⊢ σ
red (var x)                                  =  var x
red (f · a) with red f | red a
red (f · a) | ƛ b      | a′                  =  red (b [vz / a′])
red (f · a) | f′       | a′                  =  f′ · a′
red (ƛ b) with red b
red (ƛ b) | f · var vz              with vz ∈fv? f
red (ƛ b) | f · var vz               | yes y  =  ƛ (f · var vz)
red (ƛ b) | ∘ (shift vz t) · var vz  | no t   =  t
red (ƛ b) | b′                               =  ƛ b′
red tt  =  tt
```

5.6 Discussion

Restricting ourselves to the unary binding case makes the definition of the predicate and deciding function easier. Defining a similar predicate for the universe of binders from section 4.4 causes problems due to the simultaneous binding of an arbitrary amount of variables in the primitive types.

Consider the case of a value abs s t of the interpretation of an abstraction abs c_1 c_2 in the context Γ-x for some variable x : $\Gamma \ni \sigma$ and let Δ_1 be the binder context the of s. Then t has the term context $(\Gamma\text{-x}) \rhd\!\rhd \Delta_1$. In the definition of the shiftP function the goal type for the term inside an abstraction is P⟦ c_2 ⟧ τ Δ $(\Gamma \rhd\!\rhd \Delta_1)$ which we can produce by shifting a term of type $(\Gamma \rhd\!\rhd \Delta_1)$ - y for some variable y : $\Gamma \rhd\!\rhd \Delta_1 \ni \sigma'$.

```
shiftP : ∀ {Γ σ c τ Δ} (x : Γ ∋ σ) → P⟦ c ⟧ τ Δ (Γ - x) → P⟦ c ⟧ τ Δ Γ
shiftP x (abs {Δ₁} s t)  =  abs (shiftP x s) {!!}
shiftP x ...
```

Of course we want that term to be t and y shall be the weakening of x by the context Δ_1 as computed by the following vs⋆ function.

```
vs⋆ : ∀ {Γ τ} (x : Γ ∋ τ) (Δ₁ : Ctx) → Γ ▻▻ Δ₁ ∋ τ
vs⋆ x ϵ       = x
vs⋆ x (Δ ▻ s) = vs (vs⋆ x Δ₁)
```

However, the normalized context expressions $(\Gamma$ - x$)$ $\rhd\rhd$ Δ_1 and $(\Gamma$ $\rhd\rhd$ $\Delta_1)$ - vs⋆ x Δ_1 are not equal and thus Agda can not determine that the types are equal and it becomes necessary to reason about type equality and rewrite the goal types or use type castings. For a single variable binding this is not a problem because the expression $(\Gamma \rhd \tau)$ - vs x normalizes to $(\Gamma$ - x$) \rhd \tau$.

It is possible to overcome this problem for binders by modeling the contexts as follows. A binder context BCtx is a list of sorts as the contexts before, but a term context is a list of binder contexts. Essentially we are not working anymore with a list of single variable bindings but with a list of multi-variable bindings.

```
data BCtx : Set where
  ϵ    : BCtx
  _▻_ : (Δ : BCtx) (s : Sort) → BCtx
data TCtx : Set where
  ϵ     : TCtx
  _▻▻_ : (Γ : TCtx) (Δ : BCtx) → TCtx
```

Variables are then of course described by two indices, one indexing into the term context to select the proper binder context, and one indexing into that binder context. The essential difference is that for this modeling the expression $(\Gamma$ $\rhd\rhd$ $\Delta_1)$ - vs⋆ x Δ_1 will normalize to $(\Gamma$ - x$)$ $\rhd\rhd$ Δ_1 and thus the type-checker will see that the types are equal.

6 Conclusion

6.1 Related work

A datatype-generic treatment of syntax with binding has been addressed before in the literature. We will look at some of these works in more detail and address how variables are handled in the approach and how are the scoping rules of the language specified. Moreover, when defining heterogeneous substitutions is necessary to get type equality information. That is when substitution a term t for a variable x of equal sorts σ in some other term s of a different sort τ we need to know during a generic traversal when we reach positions corresponding to the sort σ. For our single variable substitutions from section 5.4 we got this information from the context of the de Bruijn indices. We provided a function that decides equality for indices, and equality of indices in the same context implies equality of the sorts. We will specifically look how this problem is handled in the other implementations.

6.1.1 Scrap your nameplate

Cheney [Che05a] describes the implementation of a Haskell library called *FreshLib* for generic programming with abstract syntax. He applies a nameful approach similar to FreshML's support for nominal abstract syntax and combines it with generic programming techniques available in Haskell.

Key ingredients are a Name datatype and an abstraction type constructor Abs a b which are provided by the library and are expected to be used in user-defined datatypes. A datatype representing untyped lambda-terms can be defined as follows

```
data Lam = Var Name
         | App Lam Lam
         | Lam (Abs Name Lam)
```

Fresh name generation is handled through the use of freshness monads. Nominal operations like name swapping, name permutations, freshness checks and α-equivalency checks as well as syntax operations like capture avoiding substitution and free variable calculation are defined in several type classes. Special care has to be taken in the instances for Name and Abs a b, which are provided by the library. Instances of user-defined datatypes are simple recursion steps, that means the corresponding function is called on all subterms using polymorphic recursion, and can thus be handled generically. Cheney describes automatic derivations of those using *derivable type classes* and support for modular generic traversal provided by the *scrap your boilerplate with class* library.

Datatypes with variables are made instances of the type class HasVar

```
class HasVar e where
  is_var :: e → Maybe Name
  var    :: Name → e
```

which allows recognition of term variables, extraction of names as well as construction from names. The supported substitutions and free variable calculations are heterogeneous

```
class Subst e t where
  subst :: FreshMonad m ⇒ Name → e → t → m t
class FreeVars e t where
  fvs   :: e → t → [Name]
```

subst performs a substitution of a term of type e for the variable represented by the given name in a value of type t. This relies on dynamic type equality and casting to recognize for a given position within the t value if a value of type e is located there. The same holds for free variable calculations via fvs.

FreshLib only provides one Name datatype which is used for all atom abstractions. This results in name clashes when names are used for variables of multiple syntactic sorts at the same time, like term and type variables. An atom abstraction used for binding variables of one sort can capture a variable of a different sort (Cheney [Che05a]). Cheney also describes an extensions of the library where Name is parameterized over a specific name representation.

```
data Name n = Name n (Maybe Int)
```

This allows the specification of different kinds of names for different syntactic sorts simply by using different datatypes, like different copies of the String type. Unfortunately the nominal operations are not parametric so that dynamic type casting becomes necessary in these cases too.

Finally, Cheney [Che05a] describes an extension with a general class of binders that can be used in the left hand side of an abstraction allowing user-defined bindable forms. He provides examples of let-bindings and pattern-match cases, which bind names simultaneously in a term but he does not delve into a datatype-generic treatment of binders.

6.1.2 GMeta

Oliveira et al. [OLCY11] are working on a datatype-generic framework for the mechanization of formal meta-theory of first order representations with implementations in Coq and Agda. They make use of a universe construction to represent abstract syntax types with binding. The universe itself does not depend on a particular first-order representation but can be instantiated to different ones. The extension is parameterized by two datatypes which describe the necessary data in binders and variables occurrences. GMeta includes libraries that provide generic syntax operations and lemmas for locally nameless and de Bruijn index representations, but it is also possible to extend it to other first-order representations like a nominal or a locally named approach.

Oliveira et al. [OLCY11] describe the use of a simplified version of regular tree types [MAM06] in which only a single top-level recursive binder is allowed. The simplification removes the complexity of telescopic sums in the interpretation of the full universe of regular tree types for presentation purposes, at the expense of not being able to represent mutually recursive datatypes. Experimental implementations in Coq and Agda which cover the whole universe of regular tree types are provided. The universe is extended by constructs to model binding and variables occurrences. In Agda the simplified universe with name binding extensions looks like this

```
module Rep (Q : Set) (V : Set) where
  data Rep : Set where
      -- simplified regular tree type codes
    unit : Rep
    k    : Rep → Rep
    _⊕_  : Rep → Rep → Rep
    _⊗_  : Rep → Rep → Rep
    rec  : Rep
      -- binding codes
    bnd  : Rep → Rep → Rep
    emb  : Rep → Rep
  mutual
    ⟦_⟧ : Rep → Rep → Set
    ⟦ unit   ⟧ r = ⊤
    ⟦ k s    ⟧ r = μ s
    ⟦ a ⊕ b  ⟧ r = ⟦ a ⟧ r ⊎ ⟦ b ⟧ r
    ⟦ a ⊗ b  ⟧ r = ⟦ a ⟧ r × ⟦ b ⟧ r
    ⟦ rec    ⟧ r = μ r
    ⟦ emb s  ⟧ r = μ s
    ⟦ bnd s t ⟧ r = Q × ⟦ t ⟧ r
    data μ (s : Rep) : Set where
      ⟨_⟩ : ⟦ s ⟧ s → μ s
      var : V       → μ s
```

The unit, _⊕_ and _⊗_ codes provide the usual sums-of-products view and k allows the inclusion of constants given a type representation. Recursive occurrences of the single recursive type being defined are marked with the rec code. Binders are represented using the bnd constructor where the intended meaning of bnd s t is a binding, which abstracts over one variable for a value of a type represented by the code s in a term of a type represented by t. The module parameter Q describes the data to be put in binders, like for example a name in a nominal representation. Embedded terms of a different sort are included with emb. Variables are introduced as an alternative to close the fixed point and are represented by a value of the module parameter V.

The library provides heterogeneous substitutions via subst

$$\begin{aligned}&\mathsf{subst} &&: \{\mathsf{r}\ \mathsf{r0}\ :\ \mathsf{Rep}\} \to \mathsf{V} \to \mu\ \mathsf{r0} \to \mu\ \mathsf{r} \to \mu\ \mathsf{r}\\ &\mathsf{substRep} &&: \{\mathsf{r}\ \mathsf{r0}\ \mathsf{s}\ :\ \mathsf{Rep}\} \to \mathsf{V} \to \mu\ \mathsf{r0} \to [\![\ \mathsf{s}\ ]\!]\ \mathsf{r} \to [\![\ \mathsf{s}\ ]\!]\ \mathsf{r}\end{aligned}$$

where r0 denotes the type whose variables are to be substituted in a term of type r. subst uses equality defined on Rep to check for the important case when r0 equals r. Furthermore subst is defined mutually with substRep which recurses over "open" values.

6.1.3 Binders unbound

Weirich, Yorgey and Sheard [WYS11] present a domain-specific language UNBOUND and associated generic programming library in Haskell for the specification of binding structure of languages. It covers bindings of an arbitrary amount of variables simultaneously and more sophisticated binding forms that include sequential or recursive scoping and as such is as expressive as the binders universe of section 4.4. In fact they provide a set of type combinators very similar to the primitive codes of the binders. However, they distinguish types of binder values that bind variables P from term values that reference variables T. Using P and T as kinds they have the following combinators.

$$\begin{aligned}&\mathsf{Bind} &&:: \mathsf{P} \to \mathsf{T} \to \mathsf{T}\\ &\mathsf{Rebind} &&:: \mathsf{P} \to \mathsf{P} \to \mathsf{P}\\ &\mathsf{Rec} &&:: \mathsf{P} \to \mathsf{P}\\ &\mathsf{Embed} &&:: \mathsf{T} \to \mathsf{P}\end{aligned}$$

Bind corresponds to the primitive abstraction code in the universes, Rebind is used for sequential scoping for binders and Rec for recursive scoping. The Embed combinator allows the embedding of terms within binders, so that definitions of declarations are possible. Furthermore the interface specifies a single Name T type that is used for both, marking binding sites of variables as well as reference sites. Internally a locally nameless style is used where free variables are represented by names and bound variables by de Bruijn indices. The de Bruijn indices are not well-scoped, but the interface is completely abstract so that de Bruijn indices do not leak to client code and only the implementors of the library need to ensure correct handling of indices. Binding and unbinding of variables is only available through smart constructors and destructors

$$\begin{aligned}&\mathsf{bind} &&:: \mathsf{P} \to \mathsf{T} \to \mathsf{Bind\ P\ T}\\ &\mathsf{unbind} &&:: \mathsf{Fresh\ m} \Rightarrow \mathsf{Bind\ P\ T} \to \mathsf{m}\ (\mathsf{P}, \mathsf{T})\end{aligned}$$

where m is a monad providing fresh variable names. Name T is indexed by a term datatype that determines the sort of the represented variable. For example the untyped lambda calculus can be represented like this

$$\begin{aligned}&\mathsf{type\ N} &&= \mathsf{Name\ E}\\ &\mathbf{data}\ \mathsf{E} &&= \mathsf{Var\ N}\\ &&&\mid \mathsf{Lam\ (Bind\ N\ E)}\\ &&&\mid \mathsf{App\ E\ E}\end{aligned}$$

The UNBOUND library provides generic versions of operations like name swapping, α-equivalence checks and capture-avoiding substitutions for free. It is based on the RepLib generic programming library by Weirich [Wei06]. Heterogeneous substitutions are provides by a type class. subst substitutes a term of type b for a variable name of sort b in a value of type a.

```
class ... ⇒ Subst b a where
  isvar :: a → Maybe (SubstName a b)
  subst :: Name b → b → a → a
data SubstName a b where
  SubstName :: Name a → SubstName a a
```

A value of type b is traversed with a reified dictionary for Subst b a. At recursive positions the isvar function is checked to see if b is a variable. Moreover in this case the equality constraint of SubstName will tell us that a and b are indeed equal such that the given term can be substituted if both variables are equal.

6.2 Future work

There are ample opportunities for future work of using datatype-generic proramming to deal with syntax with binding.

Automatic derivations

Most datatype-generic programming libraries for Haskell allow automatic derivation of the structure representations from datatype declarations and corresponding conversion functions. Doing this for well-scoped de Bruijn representations is surely desirable but recognizing the scoping structure from the changes to the term and binder contexts is difficult if not in general impossible. Certain heuristics will surely exist but one might also try a generative approach. Instead of recognizing the structure from existing datatypes, the universe codes can be used as a specification language from which the datatypes and conversion functions can be generated. Of course for this the codes should be augmented by information like constructor and datatype names.

Haskell implementation

While Agda proved itself invaluable for the development of the universes and the implementation of the type computations and generic functions it is not as widely used as other functional programming languages like for example Haskell. Porting most of the implementation to Haskell and releasing it as a library is practical and useful. While certainly some dependently-typed programming can be faked in Haskell using extensions like GADTs and type families, Haskell does not provide the flexibility we have in Agda. For example it is not possible to have polymorphic types in the right-hand sides of type family instances. The solution is to use higher-rank type constructors wrapping those cases or use data families to begin with. However, this leads to constructor pollution in the HOAS types and would render them unusable. Clean HOAS types can

be written in Haskell, but they cannot be calculated generically. Also here a generative approach helps that provides clean HOAS types and a conversion function to a HOAS interpretation using data families.

Generic proofs

Meta-theoretical formalizations of languages need to reason about variable binding and associated operations like substitutions. Again this is repetitive, burdensome and usually the biggest parts of the formalizations. Researches thus seek to mechanize this process as much as possible. The GMETA framework by Oliveira et al. [OLCY11] tries exactly to provide these so called infrastructure lemmas for classical first-order representations generically. However, the universes in this thesis are more expressive because they cover well-scoped or even simply well-typed representations and richer binding forms from which meta-theoretical formalizations can greatly benefit. To this end, we plan to implement generic proofs of lemmas for the generic operations in this thesis.

Separate namespaces

The types in our universe are indexed by a single context but in section 2.2 we presented a well-scoped definition of System F where types are indexed by a context for type variables and terms are indexed by two contexts, one for types and one for terms. Types will never reference term variables so it is not necessary to index types by a term context or use a common namespace for types and terms. It is interesting to explore the possibility of having every type in the family individually indexed by multiple namespaces. This is another dimension of genericity, namely arity-genericity, and has for example been examined by Weirich and Casinghino [WC10].

Well-typed representations of polymorphic languages

One motivation for separating namespaces is an approach to representing well-typed representations of polymorphic languages. Benton et al. [BHKM] for example present a well-typed representation of System F in which types and terms are idnexed by a type context. Terms are furthermore indexed by a term context that depends on the type context. It is a list of System F types that can reference variables from the type context. It is worth examining the possibilities to extend our universes model this more generalform of indexing. However, polymorphic representations are not as well behaved as simply-typed ones. System F type abstraction needs to weaken all the types contained in the term context and any operation over System F terms needs to commute with this weakening. Benton et al. [BHKM] observe this and need to resort to excessive usage of type-equalities and casting or switch to heterogeneous equality to deal with type-equalities.

Telescopic contexts

Another possibility to model polymorphic languages, but also a first step into providing support for dependently-typed representations, is to make contexts telescopic, i.e. the type of a

variable may refer to earlier variables in the context. Representations of dependently-typed languages have for example been covered by Danielsson [Dan07], Chapman [Cha09] and McBride [McB10]. However, they require a definition of a universe for the object type system using induction-recursion. See for example Dybjer and Setzer [DS01] for a treatment of induction-recursion. So far no attempt has been made to model inductive-recursive definitions generically and is not clear what the possibilities are.

Other datatype-generic approaches

The approach to datatype-generic programming we have used is based on work by Yakushev et al. for modeling families of mutually recursive types by a single fixed point of a functor. However, there exists an abundance of different libraries and approaches to datatype-generic programming. If we give up on modeling the recursive structure of well-scoped de Bruijn representations we can get simpler definitions that model even richer kinds of datatypes, but in return allow less definitions of generic functions. Using those approaches it might be easier to model for example multiple namespaces or allow parametrization of types and gain modularity. It can be expected that it is not possible to provide a parametric HOAS interpretation with generic conversions if we give up modeling the recursive structure, but most of the other definitions presented in this thesis should be transferrable to most other approaches to datatype-generic programming.

More generic functionality

Of course there are more operations on syntax terms that can be made generic. For example zippers are used to navigate values datatypes and have been treated generically before. Similar definitions are possible for syntax types. Pouillard and Pottier [PP10] for example provide the definition of a one-hole context type for an untyped lambda calculus with lets. There exists other functionality which can be made generic and uses variables, but not necessarily variable binding. Term rewriting has been covered generically by Van Noort et al. [NRH+08] and similarly it should also be possible to treat unification generically.

Acknowledgements

First and foremost, I would like to thank you, dear reader, for your interest in the topic of my thesis and the time you take to read it. It means a lot to me that my work is also seen by others and that it might prove itself useful for them.

I would like to thank my supervisors Johan Jeuring and Andres Löh for all their critical notes and suggestions. Johan's calmness, patience and constant output of ideas has helped me a great deal to write my thesis in its current form. I thank Andres for suggesting this thesis topic to me, but also warning me about the crazy world of name-binding and for introducing me to various aspects of functional programming in his courses.

Moreover I thank the participants of the reading club for the interesting discussions and inspirations during the meetings. It has surely broadened my mind and also gave me the opportunity to discuss my topic with others and get early feedback. Especially José Pedro Magalhães has been very helpful in answering my questions about nifty programming details in Haskell.

Last but not least, many thanks to my family and friends for their love and support. It was a fairly turbulent time for me, but despite physical distances I always had someone to catch me when I fall. Without all of you, I would not be where I am today. You are the greatest, never forget that.

Bibliography

[ALY09] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2009.

[AT11] The Agda Team, 2011. http://wiki.portal.chalmers.se/agda/.

[Atk09] Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin / Heidelberg, 2009.

[BHKM] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, pages 1–19.

[Cha09] James Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, January 2009.

[Che05a] James Cheney. Scrap your nameplate: (functional pearl). In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 180–191, New York, NY, USA, 2005. ACM.

[Che05b] James Cheney. Toward a general theory of names: binding and scope. In *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, MERLIN '05, pages 33–40, New York, NY, USA, 2005. ACM.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[Dan07] Nils Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin / Heidelberg, 2007.

[dB91] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189 – 204, 1991.

[DS01] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In *Proceedings of the International Seminar on Proof Theory in Computer Science*, PTCS '01, pages 93–113, London, UK, UK, 2001. Springer-Verlag.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*,

POPL '96, pages 284–294, New York, NY, USA, 1996. ACM.

[GHJ08] A. Gerdes, B.J. Heeren, and J.T. Jeuring. Constructing strategies for programming. 2008.

[GJH10] A. Gerdes, J.T. Jeuring, and B.J. Heeren. Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 441–445. ACM, 2010.

[HJLR06] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer Berlin / Heidelberg, 2006.

[KA10] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM.

[LH09] Daniel R. Licata and Robert Harper. A universe of binding and computation. *SIGPLAN Not.*, 44:123–134, August 2009.

[MAM06] Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer Berlin / Heidelberg, 2006.

[McB05] Conor McBride. Type-preserving renaming and substitution. 2005.

[McB10] Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM.

[MP08] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18:1–13, January 2008.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Chalmers University of Technology, 2007.

[NRH+08] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM.

[OLCY11] Bruno C. d. S. Oliveira, Gyesik Lee, Sungkeun Cho, and Kwangkeun Yi. Gmeta: A generic formal metatheory framework for first-order representations. Draft paper, 2011.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PP10] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 217–228, New York, NY, USA, 2010. ACM.

[SDF+10] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Find-

ler, and Jacob Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[WC10] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 15–26, New York, NY, USA, 2010. ACM.

[Wei06] Stephanie Weirich. Replib: a library for derivable type classes. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 1–12, New York, NY, USA, 2006. ACM.

[WW03] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. *SIGPLAN Not.*, 38:249–262, August 2003.

[WYS11] S. Weirich, B. Yorgey, and T. Sheard. Binders unbound. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, New York, NY, USA, 2011. ACM.

[YHLJ09] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 233–244, New York, NY, USA, 2009. ACM.